THE DESIGN AND IMPLEMENTATION

OF

A TRULY INTEGRATED GIS

USING THE PERSISTENT PROGRAMMING LANGUAGE NAPIER88

BY

YING JEAN KUO

VOLUME I

A Thesis Submitted for the Degree of Doctor of Philosophy (Ph.D.)
of the Faculty of Science at the University of Glasgow

Department of Geography & Topographic Science

Department of Computing Science

June 1995

ACKNOWLEDGEMENTS

The author wishes to express his sincere gratitude to his supervisor, Professor G. Petrie, for suggesting this topic, for his continuous advice and supervision and for supplying the materials relevant to the IGIS research and development. Also, his assistance in the writing up of this thesis is greatly appreciated. Without all this help, this research would never have come to its present form.

The author also wishes to thank his other supervisor, Dr. R. Cooper, for his advice and numerous constructive discussions, for providing material about the Napier88 system and other persistent programming languages and for all his other help.

Thanks and gratitude are also due to Professor M. Atkinson, the initiator of persistent programming systems, for the enlightenment of the author's knowledge of Napier88 through his informative lectures and for supplying recent research papers about spatial indexing techniques.

The author also wishes to acknowledge the assistance received from the research and support staff of the Department of Computing Science who were always ready to help. In particular, special thanks are due to Mr. P. Philbrow for his technical support and the immediate assistance which he gave when problems were encountered using the Napier88 system. Also the assistance given by Mr. D. MacFarlane in making network connections between the Department of Geography and Topographic Science and the Department of Computing Science must be acknowledged.

Sincere thanks are also due to:
Mr. T. Ibbs, for his valuable comments on Chapters 1 and 2 of this thesis;
Dr. Q. Cutts and Dr. G. Kirby, members of the Napier88 system development group at the University of St. Andrews, for their explanations and clarifications of several specific features of the Napier88 system; and
Mr. B. Shannon of the CAD-X company for loaning the Pericom X-200 terminal and the Sun SparcStation 1+ workstation for the X-window tests carried out in this research.

The author also wishes to extend his gratitude to Dr. J. Briggs (Head of the Department of Geography & Topographic Science) and to the Topographic Science staff members for their help throughout the research period.

Finally, the author wishes to express sincere thanks to his sponsor, the Ministry of Education of the Taiwan government, for providing the scholarship during the period of his study.

ABSTRACT

This thesis is concerned with the design and development of an integrated geographical information system (IGIS) based on the use of a persistent programming language called Napier88. It reports on the research carried out to implement a wholly new approach to deal with the problems of constructing a truly integrated GIS.

The main aspects discussed within the context of this thesis are: -

* an overview of the current status and trends in IGIS development;
* the characteristics and functions of the persistent programming language Napier88;
* the design considerations and the definition of the system architecture of an IGIS;
* the integration of vector map data and raster image data in a persistent store;
* the multiple data modelling of geographical data;
* the superimposition and cross indexing of vector maps and raster images;
* the spatial indexing and querying of geographical data;
* the management of geographical data in a persistent database environment; and
* the implementation of a prototype IGIS.

This thesis concludes that the Napier88 language can provide a sound framework for the construction of a truly integrated GIS, although some current deficiencies in the language need to be overcome. Since persistent programming languages are still in the stage of research and development, more research is necessary to investigate other features that they could provide which may be beneficial to the development of a truly integrated GIS.

Table of Contents

VOLUME I

VOLUME II

CHAPTER 1 : INTRODUCTION

1.1 The Importance of the Development of an IGIS

The emergence of **G**eographical **I**nformation **S**ystems (**GIS**) began in the 1960's with a computer-based technology that allowed the storage, analysis and display of geo-referenced data. Since then, GIS software packages have usually been developed as being either vector-based or raster-based, since the vector and raster data formats are the two fundamental representations of the spatial component of geographical features. Both data formats have their pros and cons in GIS applications. For example, the vector format suits those applications involving network analysis, geometric measurement and high-quality cartographic production, while the raster format is better suited to applications such as overlay operations, proximity analysis and feature classification [Star and Estes, 1990; Maguire *et al.*, 1991; Maguire and Dangermond, 1991]. Despite of their equal importance, a GIS software package requires all data to be stored either in the one format or the other, since both formats are by their nature incompatible in a database. Quite apart from this, the specific requirements of digital cartography and the limitations of computer processing and storage during the early development of GIS packages have resulted in vector-based GISs being dominant in most application fields.

With the rapid development in both computer and remote sensing technology, the uses of raster image data have come to the fore in the recent years and are potentially highly useful in a number of areas. Many researchers feel that the integration of a vector-based GIS and a raster-based GIS (or image processing system) would lead to important advances in many kinds of applications [Aronoff, 1989; Ehlers *et al.*, 1989; Fisher and Lindenberg, 1989; Davis *et al.*, 1991; Estes, 1992]. Such an integration involves bringing together diverse information from a variety of sources, including maps, field surveys, photogrammetry and remote sensing, within a single system. Hence, an **I**ntegrated **G**eographical **I**nformation **S**ystem (**IGIS**) can be seen as the synthesis of all kinds of geographical information in a computer system which integrates vector and raster data and technology within the same working environment [Aybet, 1990; Piwowar and LeDrew, 1990; Shepherd, 1991].

As in any GIS, an IGIS comprises both software and a database. The database component consists of locational (vector and raster type) and non-locational (attribute type) information to represent geographical features. The software is made up of various modules that provide specific functions. There are five essential functions that should be provided in an IGIS: data acquisition, pre-processing, data manipulation and analysis, data management and product generation [Star and Estes, 1990].

Apart from generating products for particular applications, an IGIS communicates and interacts with its surroundings - the real world and other systems. On the one hand, information on the locations and characteristics of geographical features is collected by a variety of data acquisition devices which produce either vector or raster types of locational data together with associated attribute data which describes non-locational information. Vector data, which is in the form of geometric coordinates, is collected by devices such as total stations, stereo-photogrammetric instruments, cartographic digitisers, global positioning systems and other instruments, usually with the aid of an operator's interpretation of the geographical features concerned. By contrast, raster data, which is in the form of cells or pixels, is generated by devices such as map scanners, or airborne and spaceborne sensors, all of which are based on an automatic sensing mechanism with an appropriate sampling technique that detects and records the location and other information about geographical features. An IGIS must be able to handle both types of data seamlessly and should be able to communicate and exchange information with other systems such as CAD systems, digital mapping systems, image processing systems, or another GIS. An overview of an IGIS with its surroundings is illustrated in Fig. 1.1.

The use of an IGIS has many advantages. A detailed account of the benefits that follow from the creation of an IGIS is given in Shepherd [1991]. The most significant advantages can be summarised as: -

1. An IGIS provides a broader range of operations on integrated information than on disparate sets of data.
2. An IGIS allows users to work in a single information environment without needing to consider differences in data sources, information types, storage devices, computer platforms, *etc.*
3. An IGIS eliminates duplicate data collection and conversion activities, and resolves the data inconsistency problem.

An IGIS which combines the strengths of both vector and raster technologies is able to facilitate and deliver more benefits for certain applications. Some gains have already been achieved by using an IGIS, especially in the areas of DEM (**D**igital **E**levation **M**odel) generation and terrain visualisation [Kraak, 1993]; change detection and map revision [Derenyi and Pollock, 1990; Jensen *et al.*, 1994]; the production of digital orthophoto maps [Grenzdörffer and Bill, 1994] and using vector information as an aid to image classification [Mason *et al.*, 1988; Janssen *et al.*, 1990; Davis and Simonett, 1991]. These are the most successful areas in each of which, vector maps have been interwoven with raster images for analysis and presentation. New applications of an IGIS are emerging such as three-dimensional spatial modelling for geological exploration, spatial-temporal dynamic modelling for environmental monitoring, *etc.* [Fabbri, 1992].

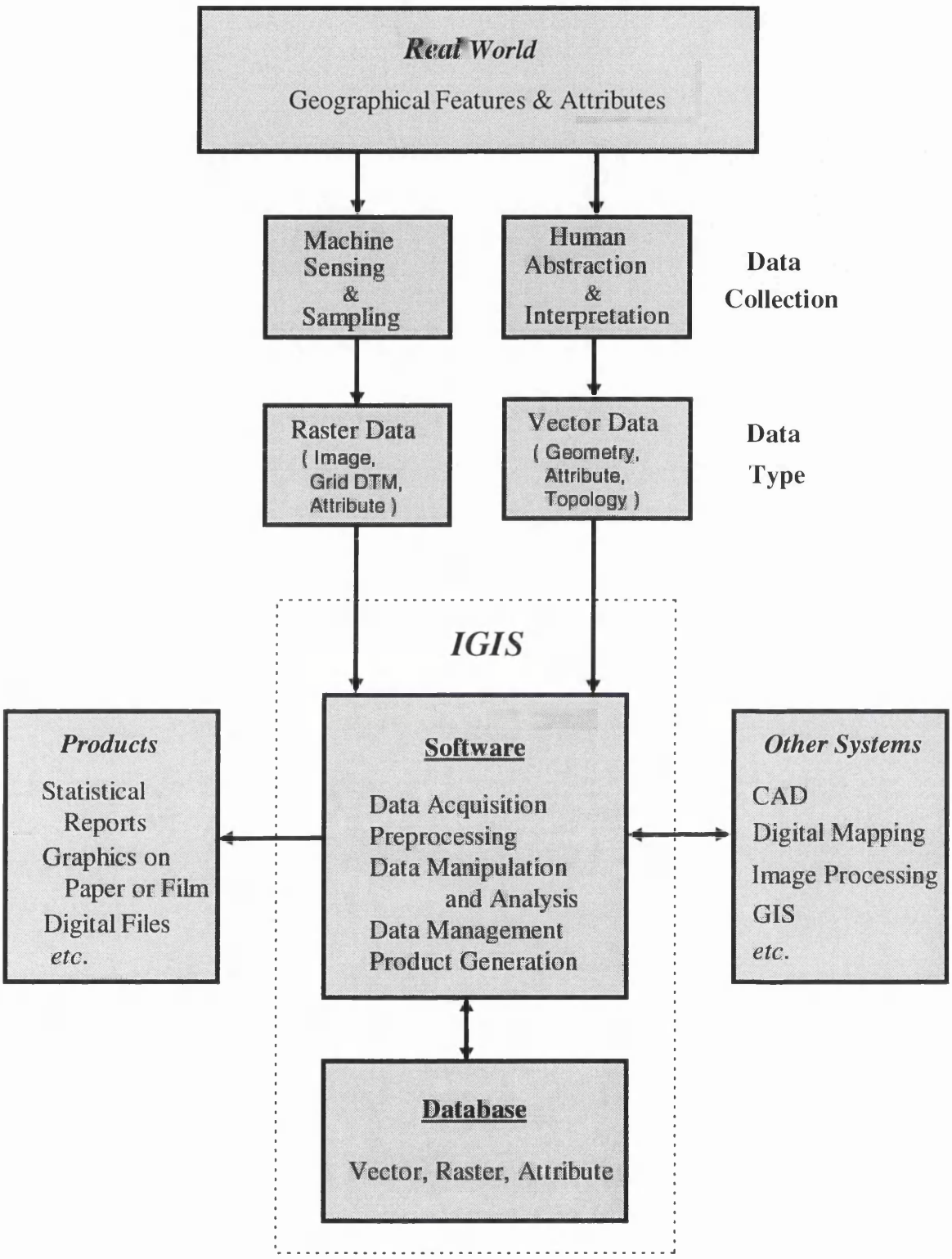


Figure 1.1 An overview of an IGIS with its surrounding environment

The integration of geographical information gives rise to various issues, including the removal of data inconsistency, spatial data standards, the decentralising and networking of spatial databases, generalisation, database management, integrating modelling functions, integration of multi-media technologies and GIS, linkage between GPS and GIS, and others [Ehlers, *et al.*, 1989; Petrie, 1989a, 1989b; Flowerdew, 1991; Coleman and McLaughlin, 1992; Newton, *et al.*, 1992; Arnaud, *et al.*, 1993; Oosterhoff, 1993]. Many papers have been published on the development of an IGIS since the late 1980's. Among them, the integration of vector and raster data with their operating functions into a single system is seen as a key issue in the fundamental design of an IGIS [Jackson and Mason, 1986; Ehlers, *et al.*, 1989; Newell and Theriault, 1989]. The success and effectiveness of an IGIS mainly depends on the strategy and methodology adopted in merging these two heterogeneous data formats into a single system. There are deficiencies or problems in the existing "so-called" IGISs. The full integration of vector and raster data within a single database is still a challenging task for GIS researchers to tackle before a truly IGIS can be created.

1.2 The Problems of Implementation of an IGIS

Many attempts have been made to create an IGIS. The most commonly used method is that where a uni-format based GIS extends its capabilities to deal with the format of its counterpart, *i.e.*, a vector-based GIS has facilities added for the storage and processing of data in raster format, or *vice versa.* As a result, many GIS vendors claim that their software package is able to support dual-format data. However, the level of integration may be quite different to that which a user would expect from such a claim.

1.2.1 The Degree of IGIS Integration

An IGIS may achieve different degrees of integration as far as the function of handling vector and raster data is concerned. The degree of integration can be categorised as being at one of three levels [Ehlers *et al.*, 1989; Oosterhoff, 1993; Kuo, 1994a] - either at display level, process level or storage level.

1. **Display level** : This is the basic function that an IGIS should provide. In this category, a GIS is composed of two subsystems which process vector and raster data independently and is able to superimpose the results in a common window. However, only one data type can be processed or analysed, while the other data type is mainly used as a backdrop or an overlay for the purpose of feature identification. A dedicated data conversion or interface program is required to join vector and raster GIS packages together due to the fact that there is no direct linkage between them. Figure 1.2(a) illustrates the view of an IGIS with this **display** level of integration.

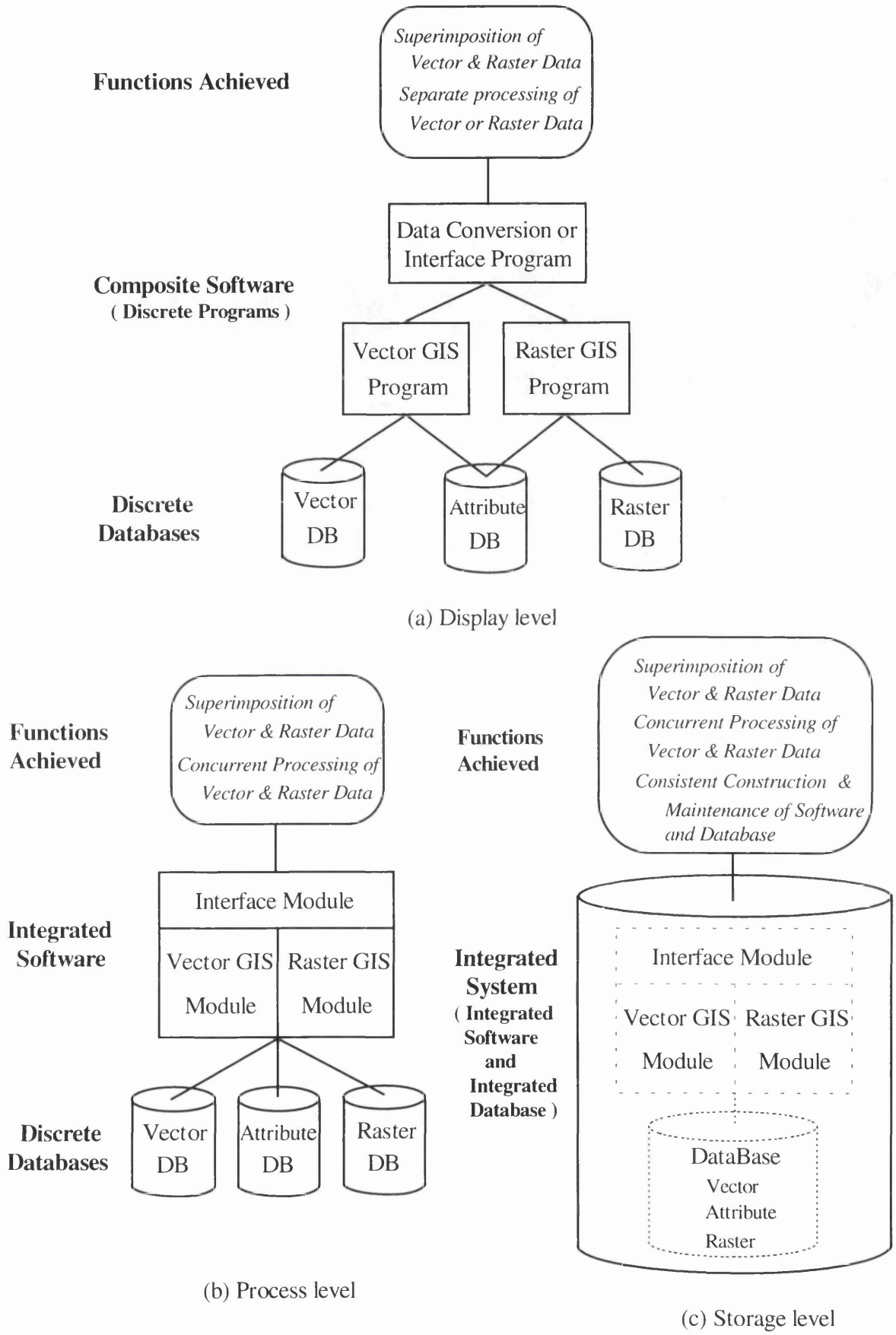


Figure 1.2 The degree or level of IGIS integration

2. **Process level** : This is the intermediate function that an IGIS may provide. Unlike the previous category, a system is developed as having vector, raster and interface modules which can be integrated in a single software package. Because all modules in a GIS software are closely linked, this kind of IGIS allows both vector and raster data to be processed concurrently in addition to the capability of superimposition of vector and raster data. Although the GIS software itself is an integrated unit, different types of data (vector, raster and attribute) are stored separately in different databases. Figure 1.2(b) shows the view of an IGIS with this **process** level of integration.

3. **Storage level** : This is the ultimate set of functions that an IGIS will provide. Not only are software modules combined as an integrated unit, as has already been described for the **process** level, but also discrete databases are formed as an integrated unit. Furthermore, both software and database are united as a single system. This type of integration provides an integrated database environment which 'hides' the details of programs and databases from the user, so that it gives the highest degree of integration. Figure 1.2(c) illustrates the view of an IGIS with this **storage** level of integration.

In general, the power and sophistication of an IGIS increases as the degree of integration extends or rises from the **display** level to the **process** level, and finally to the **storage** level

1.2.2 The Methods of Integration

In order to achieve the goal of integration, there are three methods that could be used in the development of an IGIS.

1. **The composite method** is to produce a composite system from two existing GISs, one of which is vector-based, while the other is raster-based.
2. **The extended method** is to extend an existing vector-based or raster-based GIS to handle both vector and raster data.
3. **The complete method** is to develop a GIS that incorporates both vector and raster capabilities.

The **composite** method can be implemented by developing a data conversion or interface program to join a vector-based GIS and a raster-based GIS (or an image processing system) together. These two independent software packages may be running on the same or different hardware with 'links' between them. Using this method, the degree of integration is often limited to the **display** level since a data conversion program which is commonly used for data exchange has no explicit linkage between two systems. Very few systems have reached the **process** level since the development of a dedicated interface program demands close collaboration between the vendors of the two systems being integrated. For example,

the ARC/INFO GIS and the ERDAS image processing system have been integrated through a 'Live Link' interface program that permits the dynamic exchange and display of data from both systems [Stow *et al.*, 1990; Rado *et al.*, 1991].

The **extended** method is to modify a uni-format GIS to accommodate the co-existence of vector and raster data. Developers of existing GISs, which basically are either vector-based or raster-based, have tried to add a variety of raster/vector processing capabilities within a single software package. Such a system has a common user interface and a simultaneous display of maps and images, so that not only the **display** level but also the **process** level of integration can be realised. However, the vector and raster data are kept in different databases, and quite often, an existing proprietary **D**ata**B**ase **M**anagement **S**ystem (**DBMS**) such as Oracle, Ingres, Informix, *etc.* is used to store the corresponding attribute data. Therefore, the **storage** level of integration cannot be achieved by this method. Examples of systems using the **extended** method are ESRI's ARC/INFO [Menon *et al.*, 1991], USL's CARIS [Derenyi and Pollok, 1990; Derenyi, 1991], *etc.*

The **complete** method is to develop a GIS providing both vector and raster capabilities in the system design from scratch. The need to create a complete GIS is usually driven by the realisation of the drawbacks of existing systems. The development of a complete system may either use conventional database technology but with a dual-format system architecture or it may adopt a new computer technique or an advanced database technology such as an **E**xtended **R**elational **D**ata**B**ase **M**anagement **S**ystem (**ERDBMS**), an **O**bject-**O**riented **P**rogramming **L**anguage (**OOPL**), an **O**bject-**O**riented **D**ata**B**ase **M**anagement **S**ystem (**OODBMS**), or a **P**ersistent **P**rogramming **L**anguages (**PPL**). For example, GIS software packages such as Intergraph MGE [Intergraph, 1990], Siemens SICAD-HYGRIS [Siemens, 1989; Kaehler and Theissing, 1989], Genasys Genamap [Genasys II, 1993], and Tydac SPANS [Intera Tydac, 1993] have been designed to support both vector and raster capabilities, but are based on a conventional database technology. Some software packages such as Intergraph's TIGRIS [Herring, 1987], the Smallworld GIS [Chance *et al.*, 1990] and Laser-Scan's prototype IGIS [Hartnall, 1993a] have implemented the OOPL technique in their system design. Some of these systems will be discussed in more detail in Chapter 2. The **complete** method can normally achieve a far better integration of the **display** and **process** levels. However, if the data storage is similar to that used in the **extended** method, then the **storage** level of integration cannot be achieved. With the new advanced database technology, the degree of integration which can be accomplished should reach the **storage** level as well as the **display** and **process** levels, depending on which technique is used and how it is implemented. Nevertheless, only very few systems do actually provide a certain degree of integration at the **storage** level, and indeed the devising and implementation of a truly IGIS with complete levels of integration is still a matter of intensive research and development.

It should be noted that there is also a so-called **hybrid GIS**. This term appears quite often in GIS literature whenever information integration is discussed. However, there is no clear definition about what is meant by such a hybrid GIS. A hybrid GIS may refer to a composite system which is made up of two independent GISs or it may comprise an independent system where spatial data is held separately from aspatial data. Since almost always confusion results when a GIS software package is described or referred to as a hybrid system, therefore, for the sake of clarity, this term will not be used in this thesis.

In order to distinguish differences in IGIS products and their development methods, the following new terms have been coined to be used in the context of this thesis. An IGIS which is able fully to achieve all three levels of integration will be called a **F**ully **IGIS** (**FIGIS**). If this has only been achieved partially, it will be described as a **P**artially **IGIS** (**PIGIS**). Accordingly, the two approaches of producing a FIGIS and a PIGIS will be named as the fully integrated approach and the partially integrated approach respectively. These integration approaches, development methods and their degree of integration for IGISs are summarised in Table 1.1.

Integrated System	*Approach*	*Development Method*	*Degree of Integration*
PIGIS	Partially-Integrated	Composite	display process
		Extended	display process
		Complete	display process storage
FIGIS	Fully-Integrated	Complete	display process storage

* The gray levels indicate the degrees of integration which may be achieved in each level.

: weak : intermediate : strong

Table 1.1 Integration approaches, development methods and degree of integration for IGISs

1.2.3 Problems Occurring in the Partially Integrated Approach

Traditionally, the partial integrated approach has been used in the development of an IGIS. Software developers of existing GISs who have tried to take the compatibility and maintenance of their existing systems into account will almost certainly adopt either the **composite** or **extended** method, whereas a new system developer will consider taking advantage of the powerful features supported by an advanced technology. In spite of the considerable effort put into the development of IGISs in the past few years, a number of problems exist and remain unsolved. These may be described for each method as follows: -

The **composite** method has to use an interfacing program to join two distinct packages in order to achieve a higher level of integration. In fact, each software package will need to

have a user interface module built into it. The interface program bridges and communicates with these two interface modules. In order to process both forms of data in one operation, data conversion to a single format, *i.e.* either vector-to-raster or raster-to-vector format conversion, must take place [Peuquet 1981a, 1981b; Davis and Simonett, 1991]. The principal advantage of using the **composite** method is its relative ease of implementation. However, there are many disadvantages; the major problems are the following [Piwowar and LeDrew, 1990; Piwowar *et al.*, 1990]:

- The development of an interfacing program requires close collaboration between the system developers of two GISs because the internal data structure of a software package is usually proprietary and a commercial secret.

- Data conversion between vector and raster format decreases processing efficiency, and leads to some generalisation and loss of accuracy.

- Because the choice of a vector or raster function for an application depends on which format will give the better performance, a composite GIS may end up with having both vector and raster data for an area. Not only will this increase the use of disk storage, but it is also very difficult to ensure that both forms of data are always consistent.

- The upgrade of one software package also has to keep the interface program updated, *i.e.*, system maintenance is difficult and expensive.

The **extended** method is the most common means used to overcome these problems. Existing GISs or image processing software will have been extended to allow, for example, the simultaneous display of vector maps and raster images and to provide processing of both forms of data in tandem. The extension of existing GISs and image processing software requires a system to be redesigned to include the capabilities of providing storage and processing for the additional data format. Although an IGIS developed by this method provides a more integrated environment than that of the **composite** method, there are still some problems [Ehlers *et al.*, 1989; Piwowar *et al.*, 1990]. These are as follows: -

- Existing software and databases have to be restructured in order to add linkages between vector and raster data. This implies that incompatibilities between upgraded software and existing databases do indeed exist, so that the task of converting existing databases or files to suit the updated software version can hardly be avoided.

- The programming language which was used to develop existing systems may not be able to supply or has limitations to provide the features necessary for this development.

- Vector, raster and attribute data are kept in discrete databases or files, thus the effort needed to keep these databases in a consistent state is laborious and costly.

As for the **complete** method, it is normally able to create an even better integrated software package if an appropriate programming language has been used for the system development. In fact, it is also possible to create an integrated database with the novel features supported by the advanced database technology which will be described briefly in Section 1.3.1. Nevertheless, if the conventional database technology is explicitly or implicitly used for building the databases of an IGIS, the complete system can by no means be termed a FIGIS. For example, object-oriented programming technology has been recognised as an effective tool to reduce the large costs involved in the implementation and customisation of complex integrated software [Chance *et al.*, 1990]. However, they may still use conventional database technology or file systems to store geographical data, and not all the data is stored in a single database. Furthermore, the conventional database technology does not support the data models used in an object-oriented program design. The structured data needs to be translated (or mapped) as they move from the program domain to the database domain, and *vice versa.* In summary, the main problems encountered in the current development of IGISs that use a software integration strategy are the following:

- The **complete** method through which a software integration technique is applied allows software developers the possibility to develop an integrated software package at a comparatively low cost. Nevertheless, all data are not kept in an integrated database. From a user's perspective, the maintenance of separate databases is still rather tedious and expensive.

- The capability of an IGIS is limited by the constraints of conventional database technology. For example, a traditional relational DBMS does not support the storage and manipulation of complex objects. Software developers have to keep the mapping between software programs and databases consistent, in which case, the construction and maintenance of the software is both tedious and costly.

From the above discussion, one can conclude that, with the partial integration approach, the software component of an IGIS can be developed as a composite or integrated unit, but the database component cannot be constructed as an integrated unit due to the deficiencies of the functions available in the conventional database technology.

1.3 The Trend Towards a Fully Integrated GIS

From what has been described and discussed so far, it is clear that the central problem in the design of a FIGIS is the creation of an integrated database which is suitable for holding all forms of spatial and non-spatial data. The supply of such an integrated database obviously requires the support of an advanced database technology. In other words, the development of a FIGIS is totally dependent on the availability of a suitable database technology.

1.3.1 The New Development Tool - Database Programming Languages

Recently, the development of database programming languages which implement the integration of programming languages and database facilities has advanced to a level where they can provide a single database environment for storing different types of data with various data structures. These database programming languages may be generally categorised as **O**bject-**O**riented **D**ata**B**ase **P**rogramming Languages (**OODBPL**) or **P**ersistent **P**rogramming **L**anguages (**PPL**), depending on the architecture and the data model implemented in the integration of programming and database facilities. An OODBPL, which is the meeting of object-oriented programming and database management, provides both the benefits of the capabilities of modelling and representing the real world as closely as possible and the traditional facilities of DBMSs such as data persistence, transaction, recovery, concurrency, *etc.* [Atkinson *et al.*, 1989; Hamon and Créhange, 1991; Cooper, 1993]. Examples of OODBPLs include O_2, ObjectStore, Objectivity/DB, ONTOS, GemStone, and ITASCA. Most of these programming languages have been integrated with either the C++ or Smalltalk languages or an object-oriented LISP derivative [Cattell, 1991]. By contrast, a PPL is a programming language which treats persistence as an orthogonal property of data, and this provides an integrated environment for a consistent treatment of the data used in both programs and database [Atkinson and Buneman, 1987; Morrison *et al.*, 1993b]. Examples of PPLs include PS-algol, Napier88, Galileo, Amber, Poly, STAPLE and FAD [Atkinson and Buneman, 1987; Kirby, 1992; Morrison *et al.*, 1993a].

Both OODBPLs and PPLs provide similar capabilities of programming and database management in an integrated system environment, including support for the persistence of complex objects in object-oriented databases and powerful mechanisms for the representation of real world entities in the form of a high degree of object abstraction [Morrison *et al.*, 1987; Hamon and Créhange, 1991; Cooper, 1993]. However, there are different features present in these two language systems. For example, in a PPL, the persistence of data is independent of data types, whereas in an OODBPL, it normally requires explicit organisation of, or even mention of, data movement by the programmers [Atkinson *et al.*, 1989; Cattell, 1991]. PPLs are usually developed as a completely new system and need not be dependent on the features supported by other programming

languages, while OODBPLs are generally created by adding database capabilities to existing object-oriented programming languages. Furthermore, PPLs are program (or process) centred which means the programmer tends to write procedures which operate over data rather than treating functions as properties of data. By contrast, OODBPLs tend to be data centred where the computational aspects of the system have become attributes of the data [Dearle *et al.*, 1989; Morrison *et al.*, 1989]. It is always worth mentioning that PPLs do not force users to adhere to an OO programming style; however they allow users to do if they wish. By contrast, all OOPLs force users to follow the OO programming style.

1.3.2 Current Status of Research and Development

Considerable research has been devoted to the development of application software based on database programming languages. Typical examples of such systems are computer-aided design (CAD) packages, office information systems (OIS) and computer-aided software engineering (CASE) tools. These applications require databases that can manage very complex data in an effective way [Atkinson *et al.*, 1989; Morrison *et al.*, 1993a]. Building a GIS around DBPL technology is another new emerging application in the last few years. Quite a lot of experimental work is under way, especially the development of object-oriented geographical databases based on OODBPLs. For example, the French Institut Géographique National (IGN) has developed a prototype GIS called GeO_2 which provides a geographical DBMS that is based on a commercially available OODBMS O_2 from O_2 Technology [David *et al.*, 1993]. The CSIRO Division of Information Technology, Centre for Spatial Information Systems in Australia has developed a prototype GIS with a object-oriented geographical database which is built on another commercial OODBMS product, ONTOS [Milne *et al.*, 1993]. The Politecnico di Milano in Italy has also developed a prototype object-oriented GIS based on the OODBMS OpenDB developed by the Hewlett-Packard Company [Sacchi and Sbattella, 1994]. With regard to the development of a GIS using a PPL, very little effort has been made till now because almost all PPLs are themselves still in the stage of research and development and are not available commercially. Despite this, an analysis of the characteristics of PPLs shows that they are very promising in terms of the development of an IGIS.

The benefits of building persistent application systems using a PPL have been described extensively in the literature [Atkinson, 1992a; Kirby, 1992]. The principal advantages provided by persistent systems are:

- Persistent systems only require a single mapping from the data model of the real world to the internal storage structure of a database.

- The same mechanisms operate on both short-term and long-term data, avoiding the traditional need for separate systems to control access to data having different degrees of longevity.

- Programs such as procedures and modules can be represented by first class values which reside in the persistent store.

- Type checking protection mechanisms operate over the whole environment to ensure type correctness in the systems.

These favourable characteristics of PPLs provide an ideal model for building applications in an integrated data-intensive system, especially those dealing with large amounts of long-lived data. Given the various advantages listed above, it was felt that the use of a PPL should be advantageous for the design and development of an integrated GIS.

Abdallah [1990] was probably the first person who has attempted to design and build a prototype GIS based on a persistent programming language. Abdallah experimented with the persistent programming language PS-algol, which is a predecessor of Napier88, and developed a purely vector-based GIS. His research work showed that a persistent programming language has great potential for developing a GIS. The primary advantages of using a PPL for GIS development can be summarised as follows: -

1. Data type completeness. Procedure is a first class type, so it is very easy to design a modular system.
2. It allows the programmer to declare and use names in the same way anywhere in a program. All aspects of the programs are all written in a consistent style.
3. The availability of 'picture' and 'image' (vector and raster) construction.
4. Data persistence.

Abdallah concentrated his research into the capability of vector processing in a GIS, especially for its applications in cartographic production. The investigation of the potential of the tandem processing of both vector and raster data in a GIS is an obvious consequence of this original research by Abdallah and motivated this follow-up research with the persistent programming language Napier88.

Napier88 has been developed at the University of St. Andrews in the UK. The Napier88 system provides some novel features that normally cannot be seen or encountered in traditional programming languages [Morrison *et al.* 1993b]. These features, which will be described in more detail in Chapter 3, provide the capabilities to meet the basic requirement for developing a FIGIS - the creation of an integrated software and database. Thus this new

database technology provides researchers with a quite different tool to explore the design and implementation of an IGIS. Since Napier88 has significantly enriched the persistent environment provided by PS-algol [Morrison *et al.*, 1993b], it should be able to provide a still better environment for the incorporation of various types of geographical data into a persistent store, which is deemed to be of the essence of constructing an IGIS.

1.4 Research Objectives

Having described the importance, problems and possibilities of developing a FIGIS, this thesis identifies the main objectives of this research into such a possibility. The intention of the research is to explore the feasibility of developing a FIGIS based on the persistent programming language Napier88. The purpose of this research is to build a framework and to provide the foundation for the development of a fully integrated GIS software package. In this context, the main research objectives can be defined as follows: -

1. ***The integration of vector, raster and attribute data within a single database, and the concurrent processing of all of them within a single workspace.***
2. ***The implementation of different kinds of data models within a GIS.***
3. ***The construction of an object-oriented geographical database.***

The first two objectives are the key items in this research agenda, and, if they can be satisfied, then they provide the basis for a FIGIS. If the third objective can be satisfied, then it will also be possible to implement object-oriented data management which tends to be a requirement in current GIS development. In more specific terms, the following tasks will be carried out in the course of this research.

1. Carrying out the design of the system architecture for an IGIS.
   ⇒ The system design includes design consideration, design criteria, functional design, database design and overall system configuration.

2. Implementing multiple data modelling of geographical data in an IGIS.
   ⇒ Each type of geographical data can be structured using an appropriate data model for specific applications, and various different data models which represent multi-scale map data and multi-resolution image data, can all be accommodated within the same database environment.

3. Organising geographical data in a persistent store.
   ⇒ A persistent store is used to comprehensively accommodate geographical information in a single database environment, no matter what types of data are

being stored, and all the data is constructed in the manner required for an object-oriented organisation.

4. Superimposition and interrelation of vector maps and raster images.
   ⇒ Vector maps and raster images are superimposed in a single display window, and are geographically cross-referenced for the purpose of tandem processing.

5. Spatial indexing and querying of geographical information.
   ⇒ A spatial indexing technique is chosen to index the point, line and polygon entities of geographical features, and different query methods are designed to examine the computational capabilities of Napier88.

6. Implementation of a prototype IGIS for the evaluation of system capabilities and performance.
   ⇒ Based on the design and framework of the above items, a prototype IGIS is implemented for the studies of various functions, such as response time of queries, database updates, system management and others.

1.5 Outline of the Thesis

The thesis is devised as follows; Chapter 2 presents an overview of current IGIS development with a review of some existing well-known IGISs which are representative systems based on conventional database technology. It also conducts a discussion on a number of relevant research programmes which are being developed on the basis of advanced database technology. Chapter 3 gives the basic concepts concerning persistent programming languages and provides an introduction to Napier88, the tool used in the present project to develop a prototype IGIS. The Napier88 system's functions and their suitability for IGIS development are also discussed. Chapter 4 describes the IGIS system architecture devised for this project. It includes a detailed description of its software modules and database design. Chapter 5 through to Chapter 7 deal with the methodology employed in the key areas of the system, namely: Geographical Data Modelling and Organisation, Vector and Raster Superimposition and Interrelation, and Spatial Indexing and Queries. Next, Chapter 8 is concerned with the implementation of the prototype IGIS. Large volumes of real geographical data are used to experiment with and demonstrate capabilities and performance of the prototype IGIS. Finally, Chapter 9 summarises the results of the research work and gives recommendations for future work.

CHAPTER 2 : AN OVERVIEW OF IGIS DEVELOPMENT

2.1 Introduction

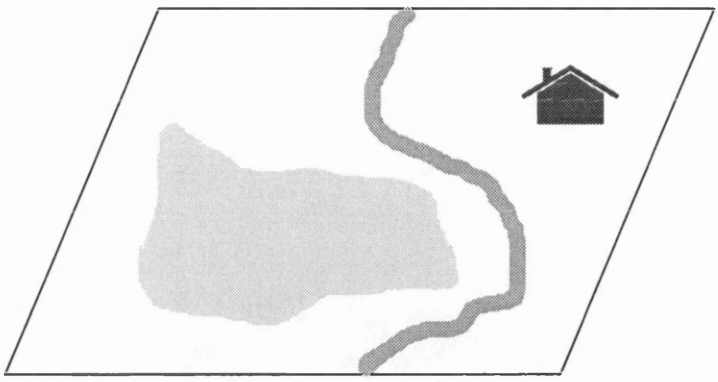
In the previous introductory chapter, the importance and some of the problems of developing a fully integrated GIS (FIGIS) have been discussed. A short introduction explaining the main trends in the current development of a FIGIS has also been given. In this chapter, some further discussion of the different types of information system which are related to IGISs will be conducted and this will be followed by a review of representative IGIS systems which are currently available on the market. Finally a brief account of a number of research programmes utilising database programming languages will also be presented. Because the contents and structures of the database is the most important single aspect of an IGIS, all the discussions and system overview will centre around the database issue, including data models, data structures, system architecture, *etc.*

2.2 Digital Representation of Geographical Data

Digital representation of geographical data is concerned with modelling the real world and structuring the modelled data. Numerous data models have been developed for representing geographical data. These data models can be classified into the two basic types of vector and raster representation [Aronoff, 1989; Laurini and Thompson, 1992]. The vector representation perceives the real world as being made up of points, lines, polygons which are defined by coordinate values, whereas the raster representation uses an ordered matrix of uniform-sized cells.

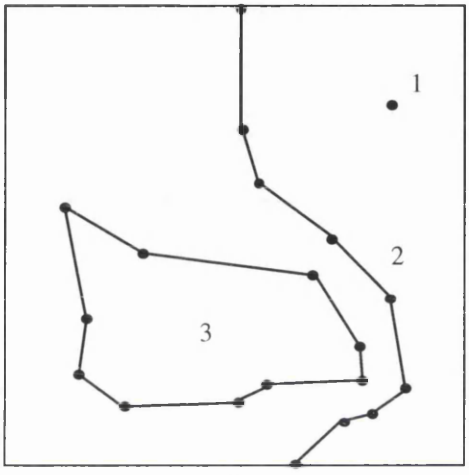
In the vector representation, the raw geographical data of the real world can be acquired by means of field survey including GPS; by analogue, analytical and digital photogrammetry; and by digitising existing maps. This process is known as data collection or data acquisition. The raw vector data is then transformed into clean vector files through data pre-processing including error detection and editing, transformation into a specific map projection, edge matching, topologic integrity checks, restructuring of data, *etc.* [Star and Estes, 1990; Cromley, 1992]. Thereafter, the clean vector files may be organised and constructed in the form of a graphical data base with appropriate data structures suited for specific applications. The basic concept of the vector representation of geographical data is illustrated in Fig. 2.1.

The raster representation has a data flow similar to that of the vector representation, but quite different data acquisition devices and data manipulation are required due to the



Real World

Data Collection



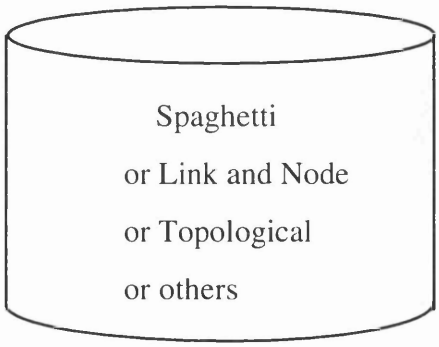
Representing Features in Vector Format (Raw Data)

Data Pre-processing

Clean Vector Files

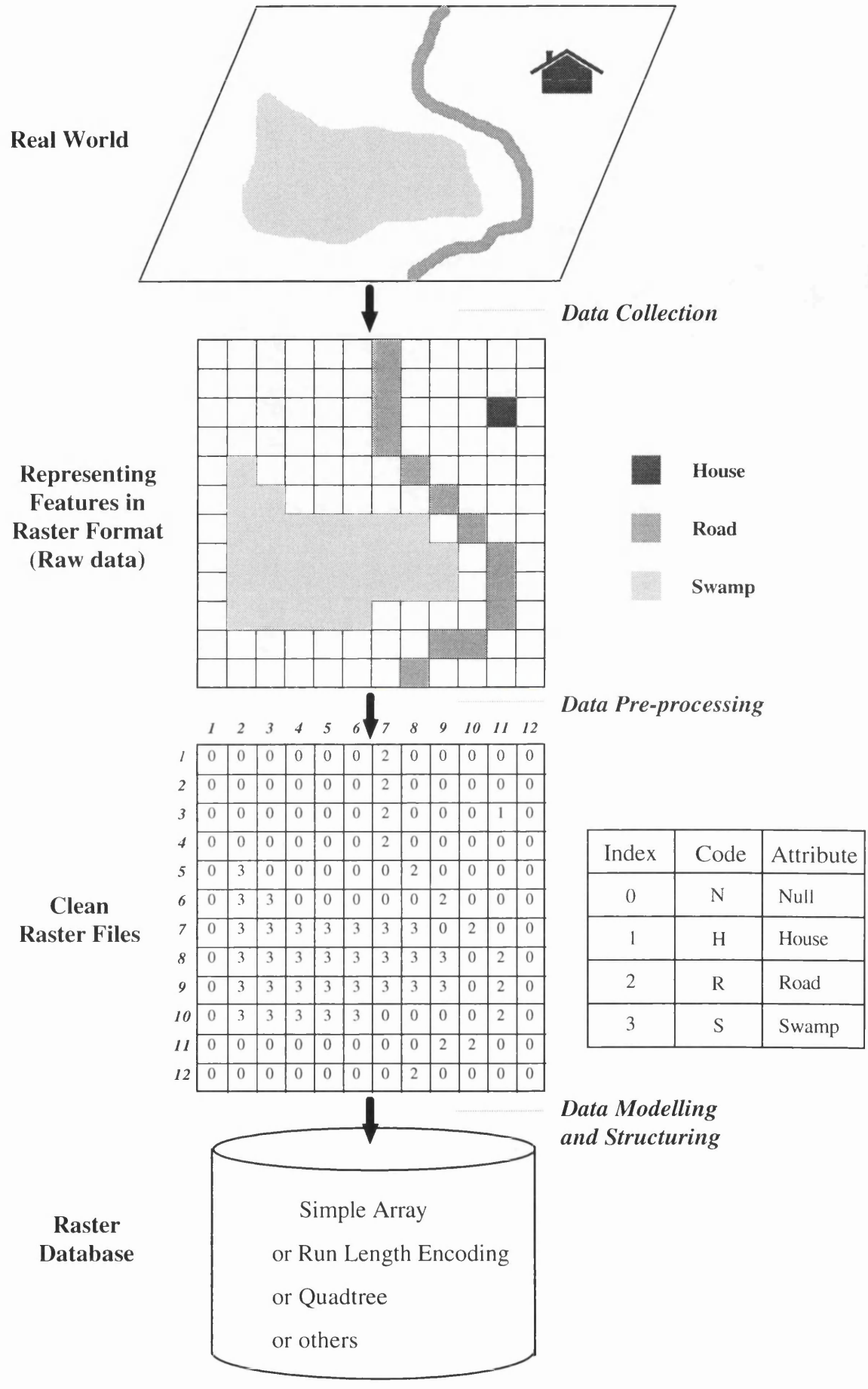
Feature ID	Entity Type	Feature Code	Coordinates	Attribute
1	Point	H	X, Y	House
2	Line	R	$X_1, Y_1, \ldots X_n, Y_n$	Road
3	Polygon	S	$X_1, Y_1, \ldots X_1, Y_1$	Swamp

Data Modelling and Structuring



Vector Database

Figure 2.1 Vector representation of geographical data



	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	2	0	0	0	0	0
2	0	0	0	0	0	0	2	0	0	0	0	0
3	0	0	0	0	0	0	2	0	0	0	1	0
4	0	0	0	0	0	0	2	0	0	0	0	0
5	0	3	0	0	0	0	0	2	0	0	0	0
6	0	3	3	0	0	0	0	0	2	0	0	0
7	0	3	3	3	3	3	3	3	0	2	0	0
8	0	3	3	3	3	3	3	3	3	0	2	0
9	0	3	3	3	3	3	3	3	3	0	2	0
10	0	3	3	3	3	3	0	0	0	0	2	0
11	0	0	0	0	0	0	0	0	2	2	0	0
12	0	0	0	0	0	0	0	2	0	0	0	0

Index	Code	Attribute
0	N	Null
1	H	House
2	R	Road
3	S	Swamp

Figure 2.2 Raster representation of geographical data

disparate characteristics of vector and raster data. The raster data is obtained by means of airborne or satellite remote sensing, or by scanning aerial photographs and scanning existing maps. The data pre-processing involves the removal of distortions (geometric and/or radiometric), ground control registration, image enhancement, image classification, and so on [Star and Estes, 1990; Cromley, 1992]. The basic concept of the raster representation of geographical data is shown in Fig. 2.2.

The following discussion describes the information systems dedicated to handle digital geographical data in either a single (vector or raster) or a dual (vector and raster) representation.

2.3 Digital Mapping and Vector GIS

Both digital mapping systems and vector GISs are vector-based systems used to capture, store, manipulate and display geographical vector data. A digital mapping system is a replacement for the traditional manual cartographic process and is primarily designed for the production of topographic and thematic maps, whereas a vector GIS is a coordinate-oriented system in which geographical data is spatially indexed and upon which a set of procedures operate to answer queries about spatial entities in order to support problem-solving or decision making [Cowen, 1988; Maguire, 1991]. However, each of them has distinctly different characteristics and applications.

The development of digital mapping systems is usually based on CAD technology. Data in a digital mapping system is organised by layers which are used to classify map features by theme such as buildings, communications, land cover, and hydrography, or by essential elements of cartography such as line weight, line style, colour, texture, and symbols. Map production can be easily carried out by accessing the relevant layers required for the production of a general or specific purpose map. Digital mapping systems are so flexible for map production and so efficient for map editing that they have been playing a major role in cartographic industry for more than two decades. However, digital mapping systems are not suited for querying or analysing geographical data because the data structures employed in these systems are often quite simple, *e.g.* a spaghetti structure may be used. Furthermore, spatial relationships among geographical features, known as topology, are not explicitly defined and stored in the dataset of the digital mapping system. Thus special processing is required to construct these relationships before any geographical analysis can be performed. Furthermore, a digital mapping system normally does not provide the capabilities of a DBMS nor does it support a linkage to a DBMS for the storage of non-graphical data [Cowen, 1988; Newell and Sancha, 1990; Korte, 1994]. As a result, digital mapping systems are not capable of efficiently handling the analysis of geographical data.

With regard to vector GISs, the emphasis is placed on the data storage aspects and the system's capabilities of analysing geographical data in vector format [Theriault, 1989]. In order to allow interactive analytical operations, a more complex data structure such as a topological structure is required for the efficient storage and manipulation of geographical data. Hence, in a vector GIS, the geometries of geographical features as well as the topology existing between them are stored in a graphical database. Apart from this, a vector GIS also contains attribute data which provides further descriptive information about graphical entities. Attribute information is normally held in a conventional DBMS such as a hierarchical or relational DBMS. Both the graphical and non-graphical databases are cross-linked by identification keys. In vector GISs, geographical information is partitioned into features according to some coding schemes which normally have a hierarchical classification. Each feature has a distinguishing code so that a user is able to search or analyse specified graphical data and relate it to attribute data and *vice versa*. Both graphical and associated attribute information can be analysed and displayed at the same time [Aronoff, 1989; Newell and Sancha, 1990; Star and Estes, 1990]. This is a capability that a digital mapping system normally cannot provide without additional customised programming.

It should be noted that there are other vector-based geo-related information systems which lie between digital mapping systems and vector GISs, namely, **A**utomated **M**apping / **F**acility **M**anagement (**AM/FM**) and **L**and **I**nformation **S**ystems (**LIS**). AM/FM systems are specifically designed to manage the spatial data associated with utility systems such as electricity, telephone, gas, water and sewage networks, whereas LISs are specifically designed to deal with the parcel-based data associated with land ownership, land valuation and land use. Usually, an AM/FM system is also based on CAD or GIS technology, but because of the networks of pipes and cables which form the greatest part of the data, it normally employs a network-like data structure to deal with graphical data, and a DBMS for handling non-graphical data [Aronoff, 1989; Abdallah, 1990; Korte, 1994]. Regarding a LIS, the basic concept is essentially similar to that of a vector GIS except that a LIS is more oriented towards areas rather than the lines of the vector GIS or AM/FM systems, so it often use a polygon-like data structure to organise graphical data.

2.4 Digital Image Processing and Raster GIS

Digital image processing is concerned with the various operations, including image rectification, quantization, enhancement, filtering and others, that can be applied to digital image data. A digital image processing system may be used in a variety of disciplines such as medicine, criminology, engineering and so on. In respect of GIS applications, the digital image processing is oriented or specialised towards a particular aspect - the processing and analysis of geographical image data.

Both image processing systems and raster GISs are raster-based systems used to store, manipulate and display geographical data held in a raster format. However, within this general area, image processing systems are designed mainly for the production of orthophotos, image maps and thematic maps or the revision of digital maps from remote sensing or scanned photographic data, whereas raster GISs are primarily designed for the analysis and display of classified image data or scanned map data with the ability to link with the attribute information stored in a DBMS. In doing so, image processing systems place most emphasis on the manipulation of raw data produced by remotely sensed imaging systems or by scanning aerial photographs, while raster GISs lay stress on the storage and analysis of the processed data. Therefore, an image processing system often acts as the front end processor of a raster GIS. However, this distinction becomes obscure since the capabilities which are offered by either system may be extended by including some or all of those implemented by the other system. In fact, this is one aspect of system integration and is discussed further in the next section.

Despite their rather divergent development in their early age, the data structures employed in both types of system are quite similar. The simplest form is that where the image data is stored as a two-dimensional matrix of uniform (usually square) grid cells or pixels. Each pixel is assigned a value which represents the attributes of that pixel such as its soil type, land use, or slope. The quality and clarity of the raster image depends on the size of the grid cell, known as its pixel size or image resolution. The smaller the pixel size, the more exact the information, but the larger the storage space required. A number of techniques have been developed in order to overcome this storage problem. Run-length encoding and the quadtree data structure are the two most common forms currently being used in the structuring and storage of image data [Gunston, 1993].

One excellent feature of raster image systems is that it is very easy to perform Boolean operations on raster images and raster data structures. For example, overlaying two raster images is simply a matter of adding or subtracting the values assigned to each individual pixel. Apart from this, the data capture of raster data is often much easier than that of vector data [Aronoff, 1989; Star and Estes, 1990]. Nevertheless, the development of raster image systems has been hindered by the problems of storing and handling vast quantities of image data and the high demand on computational power. For example, each SPOT panchromatic scene of 6,000 x 6,000 pixels contains about 36 Mbytes of data; an aerial photograph of 23 cm x 23 cm format scanned with a 20 μm pixel size generates more than 100 Mbytes of data, and a rasterized map of 1m x 1m contains about 400 Mbytes of data if scanned using a 0.05 mm pixel size. Due to the problems associated with these large data sets, the development of raster image systems has been much slower than that of vector graphic systems. As a result, till now, raster GISs have not so been popular as vector GISs.

Nowadays, the rapid advances in computing power are beginning to be able to solve the above-mentioned problems. High performance graphical workstations along with Gigabytes of storage capacity have become quite commonplace. During the last few years, digital photogrammetric workstations have probably been the most noteworthy development in digital image processing. A digital photogrammetric workstation (DPW) is an analytical plotter which uses computer technology to exploit digital imagery rather than the mechanical stages, servomotors, and optics used to exploit hardcopy film imagery in the classic analytical plotter [Kaiser, 1991]. A review of commercially available digital photogrammetric systems is well described by Matambanadzo [1992]. Digital photogrammetric workstations can be classified into two categories - the monoscopic (2D) DPW and the stereoscopic (3D) DPW - depending on whether single or overlapping images are being utilised.

The monoscopic DPW is mainly designed for the purpose of map revision, updating digital cartographic information from scanned imagery, whereas the stereoscopic DPW is generally designed for multi-purposed applications including the measurements of the stereo images used for producing or updating maps and the production of digital elevation models (DEM's) or ortho-images. The implementation of a monoscopic DPW often employs the digital monoplotting approach because it corrects both the relief and tilt displacements and gives a better accuracy in the final result. Apart from the development of stand-alone monoscopic DPWs, the digital monoplotting system has also being integrated into many GISs and provides basic capabilities for the registration of vector maps and raster images, and for screen-based digitising [Petrie, 1994].

On the other hand, at present, stereoscopic DPWs such as those developed by Intergraph, Leica-Helava and Zeiss have proven to be very efficient both for the automated production of orthophoto and digital terrain model data and for the manual stereo-plotting of vector data. Also, the image processing suppliers are beginning to produce such products, *e.g.* ERDAS's OrthoMax package. The whole production process is so highly automated that a digital photogrammetric workstation can be utilised as a front end processor for a raster GIS [Dowman, 1991]. Another important characteristic of the stereoscopic DPW is that it is equipped with a stereoscopic display device allowing users to visualise, interpret and measure digital imagery in 3D [Cogan *et al.*, 1991; Kaiser, 1991]. This is a very useful feature for the measurement of the geographical features required in 3D GIS applications. Obviously, the development of digital photogrammetric workstations will make a significant impact on the role of raster GISs. Especially in the context of remotely sensed imagery, these raster-based image systems will certainly become more important than ever before.

2.5 *Convergence to IGIS*

From the above description, it can be seen that the systems designed to handle or deal with geographical data are divergently developed because the two representations of the real world are fundamentally very different in their nature and in the approaches that can be taken when manipulating geographical data. As a result, the use of either a vector or a raster GIS has been limited to a very specific processing environment by its capabilities. The trade-offs between the different vector and raster systems chosen for GIS applications have been comprehensively discussed in the literature [Burrough, 1986; Wallace and Clark, 1988; Aronoff, 1989; Star and Estes, 1990; Davis and Simonett, 1991; Peuquet, 1991; Gunston, 1993]. The characteristics and the respective advantages and disadvantages of a vector GIS and a raster GIS are summarised in Table 2.1.

Characteristics/Functions	Vector GIS	Raster GIS
Data Capture	Slow	Fast
Data Volumes	Small	Large
Geometric Accuracy	High	Low
Data Structure	Complex	Simple
Distance and Area Measurement	Good	Poor
Buffer Zone Analysis	Poor	Good
Network Analysis	Good	Poor
Overlay Analysis	Poor	Good
Data Display	Slow	Fast
Data Aggregation / Segregation	Complex	Simple
Data Generalisation	Complex	Simple
Data Semantics	Good	Poor

Table 2.1 Advantages and disadvantages of a vector GIS versus a raster GIS

It should be noted that these comparisons are based on the intrinsic characteristics of the two basic representations and may not apply to certain specific systems which have utilised a dedicated hardware (such as a graphic accelerator) or special software (*e.g.* a spatial indexing scheme) that may result in one data format outperforming the other.

The decision as to whether to use a vector or a raster GIS is largely dependent on the application requirements and the available data sources. For example, if the data has been collected primarily by means of analytical photogrammetry and the main application is path queries concerning emergency vehicle routing which demands network analysis [Laurini and

Thompson, 1992], then a vector GIS will naturally be the better choice. In contrast to this, if remotely sensed imagery is used for the monitoring of hazardous waste sites for environmental conservation planning - a task which needs buffer zone and overlay analysis, then it is advantageous to use a raster GIS. However, most GIS applications often require a number of thematic layers for a sensible analysis. In fact, a GIS project usually involves the use of both vector and raster data types in the same area. For instance, following the above examples, satellite imagery covering a disaster area caused by widespread flooding may be acquired quickly to help define the area affected and to facilitate the task of urgently routing rescue vehicles or vessels. Similarly, the locations of waste sites and their ambient geographical features could be directly extracted from existing digital topographical maps rather than be identified and digitised via satellite imagery. However, in each case, it would be necessary to perform a series of data format conversions in order to import data from these different sources into a uni-format system.

As matter of fact, data format conversion can be seen as the most primitive way of data implementing integration although the singular nature of the data model tends to limit the applications to which a particular uni-format system may be put [Piwowar and LeDrew, 1990]. Data format conversion may involve either of the following two operations: -

I. When the foreign data and the system data are in different forms.
   One data file is in vector format, the other is in raster format. The conversion process deals with changing the data from vector to raster form (rasterization) or raster to vector form (vectorization) [Peuquet, 1981a, 1981b].

II. When the foreign data and the system data are in the same generic form.
   Both data files are in either in vector or in raster formats. The conversion process is primarily concerned with translating the data structure of foreign data to that of system data.

Almost all GISs provide import/export modules to convert some of the well-known exchange formats such as DXF, SIF, IGES, DLG, DIME, TIGER, *etc.* commonly used for vector data, and TIFF, GIF, PCX, ERDAS, GRASS, PCI, Landsat TM, SPOT, *etc.* used for raster data. However, these industrial *de facto* standards are not able to completely and accurately convey geographical information between two systems. For example, DXF, SIF and IGES all lack the capability of explicitly transferring topologic information, whereas DLG, DIME and TIGER allow the user to do so but are not able to carry object semantics. Furthermore, none of these formats is able to deliver the inherent data structures employed within the systems. Hence, a substantial effort is now being made in many countries to overcome these difficulties through the development of national standards. Recently, some digital data interchange standards such as SDTS (Spatial Data Transfer Standard) [Fegeas

et al., 1992], NTF (National Transfer Format) [BSI, 1992], and others [Williams, 1993] have been developed. Nevertheless, there is no clear indication that any one of these national standards will be widely accepted in the international community in the short term. There are also two standardisation efforts in progress. First of all, the CEN TC (Technical Committee) 287 is producing a European standard for the exchange of geographic information, which is aimed at the transfer of both vector and raster data, rather than the purely vector GIS or mapping data covered by existing standards. Initial parts of this composite standard should be published within the next year. Secondly, ISO has just begun to consider standardisation procedures in the same field. The standardisation of geographical interchange format involves various issues, including feature definition and classification, feature data encoding (both vector and raster formats), relationships between features, geographic referencing, data quality, symbology, *etc.* [Evangelatos, 1991]. A great effort will have to be made to achieve a broad consensus on these issues. Thus it seems unlikely that a truly universal standard that allows all kinds of geographical information to be precisely transferred and flexibly shared will become available in the near future. Therefore, integrating data from different sources into a uni-format GIS can be tedious and error-prone due to the problems of data conversion. Particularly the Type I (*i.e.*, vector-to-raster and raster-to-vector) data conversion tends to cause some generalisation and loss of accuracy, while the demands of the vector-to-raster conversion normally requires specialised hardware ranging from add-in boards to special parallel processor boxes.

An alternative approach to data integration is to accommodate both vector and raster data within a system. The concept of this dual-format approach is the coexistence of both data formats in the same working environment and the ability to operate functions such as those listed in Table 2.1 with their appropriate data formats. This dual-format approach removes the need for the Type I data conversion required in the uni-format approach. A significant benefit arising from the adoption of this approach is that it minimises any loss of data quality since the data would be left in their native form and in the manner best suited to their intended use [Piwowar and LeDrew, 1990]. There are three methods of integration (**composite, extended, complete**) which could be used in the dual-format approach, and different degrees of integration could be achieved at the different levels (*i.e.* at the **display, process, storage** levels) which have been described in Sections 1.2.2 and 1.2.1 respectively. In fact, there is an emerging trend in recent developments which points to the fact that digital mapping systems, vector GISs, image processing systems and raster GISs are starting to converge into IGISs. With the advances in computing power accompanied by the evolving of standards for operating systems, network protocols, graphical interfaces, programming languages and database management systems, the boundaries between digital mapping systems and vector GISs are gradually becoming fuzzy. A digital mapping system tends to provide some of the analytical capabilities of a vector GIS. On the other hand, a vector GIS is inclined to encompass some of the cartographic functions of a digital mapping

system. Consequently, a generic vector-based system can be formed to provide the features of the both systems. The same holds true for the integration of image processing systems and raster GISs into a generic raster-based system [Ehlers and Blesius, 1991]. On the other hand, gradually a vector-based system tends to supply some basic raster capabilities, *e.g.* in the form of a raster backdrop, and conversely a raster-based system also starts to provide some essentially vector capabilities. Although inevitably the evolution of the system development will have to take place over some time, it is already possible to conceive of a situation where all of these systems will eventually be merged together and fused as a multi-purpose IGIS. This evolution of IGIS development is conceptualised in Fig. 2.3.

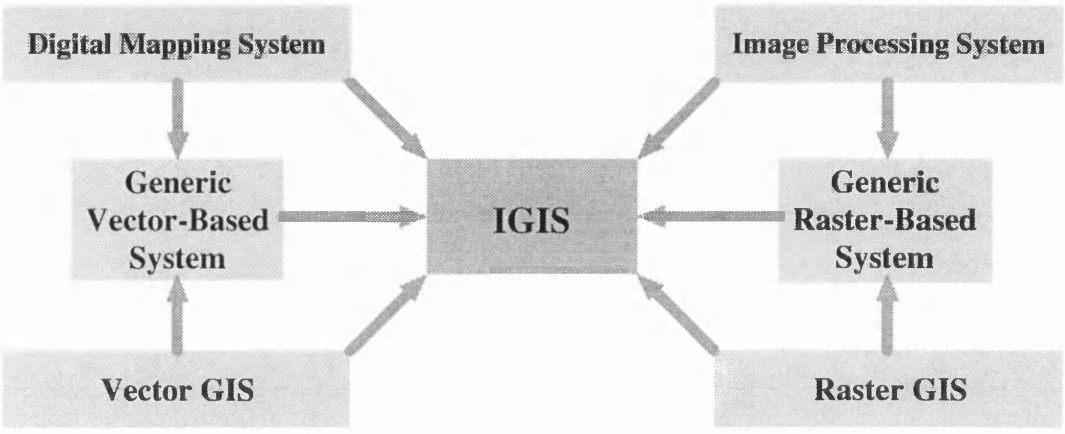


Figure 2.3 The possible evolution of IGIS development

The following two sections will review some representative GIS systems already available on the market and several prototype systems implementing advanced database technology. Each of these reviewed systems has achieved some degree of integration and has shown at least some of the indications of the tendency towards an IGIS outlined above.

2.6 Representative Commercial IGIS Products

As has been discussed in Section 1.2.3, both the **extended** and the **complete** method are commonly used in current IGIS development. The products developed by the **composite** method will not be reviewed here because they are being eliminated through market competition due to their very low level of integration and the high cost of software investment relative to their actual performance. Therefore, this section reviews certain IGIS products which are representative of the systems developed by either the **extended** or the **complete** method. Six software packages are reviewed: Intergraph MGE, ESRI ARC/INFO, Genasys Genamap, Tydac SPANS, Smallworld GIS, and Laser-Scan IGIS. Although there are dozens of systems on the market, these six selected products are sufficient to give a clear profile of the development approaches currently employed in commercially available products.

The reasons for choosing them as the representative systems are as follows: -

- Both the Intergraph and ESRI companies were founded in the 1960s and have the two largest shares of the GIS market [Korte, 1994]. Furthermore both the Intergraph MGE and the ESRI ARC/INFO systems employ conventional database technology. However, the Intergraph MGE system adopts the **complete** method but is built on a CAD system, whereas the ESRI ARC/INFO system utilises the **extended** method to add raster capabilities.

- Both the Genasys and Tydac companies were founded in the 1980s and have become well-known GIS vendors over the last decade. The Genasys Genamap and Tydac SPANS systems both use the **complete** method as well as the conventional database technology. Nevertheless, their database approaches to the achievement of the required integration are quite different.

- Both the Smallworld GIS and the Laser-Scan IGIS use the **complete** method and have implemented their solutions using the object-oriented technique. Furthermore, both Smallworld Systems and Laser-Scan have developed their own object-oriented application development environment and toolkits, namely **Magik** and **Gothic** respectively, to achieve a very high degree of the integration of geographical information contained in their GISs.

It should be noted that it is not always easy to give a detailed account of the underlying database architecture for a specific system because most commercial GISs are closed systems. So detailed information on their database architecture and data structures is difficult to obtain. Therefore the following review is based on the material published in the literature, including software overview, conference and journal papers, and technical reports.

2.6.1 Intergraph MGE

Intergraph is recognised as a major player in digital mapping and GIS technology. Before the release of the Modular GIS Environment (MGE) product in 1989, Intergraph's IGDS (Interactive Graphics Design Software) and DMRS (Data Management and Retrieval System) have been widely used for digital mapping, AM/FM, and LIS for more than 20 years. IGDS is an interactive graphics system which handles graphical data stored in Intergraph proprietary design file format (DGN) that essentially has a spaghetti data structure, whereas DMRS is a hierarchical DBMS which deals with attribute data. IGDS and DMRS work together to provide an interactive graphical-oriented management information system capable of supporting a large number of map-related applications [Abdallah, 1990; Korte, 1994]

MGE is a family of products which provide GIS management, processing and analysis. The system architecture of MGE has a quite different design to that of IGDS/DMRS in order to support efficient data structures for spatial queries and analysis. MGE is built on the top of the MicroStation CAD program and the so-called Relational Interface System (RIS) which link to graphical files and to a relational database respectively. MicroStation runs on both Unix platforms (as MicroStation 32) and PCs (as MicroStation PC) and is a well-known CAD system used for graphic data collection, editing and output, whereas RIS uses ANSI-standard **S**tructured **Q**uery **L**anguage (**SQL**) to provide a direct link to contain commonly available relational DBMSs for managing non-graphical data. Specifically, by using RIS, MGE can link to commercially available RDBMSs including Oracle, Informix, Ingres and DB2. The particular type of database actually in use is transparent to the user. Apart from this, MGE and MicroStation can also access graphical files directly or through the relational database using RIS. It should be noted that unlike the other GIS systems which also will be discussed in this review, MGE does not have either a vector database or a raster database. Instead all the graphical vector data and the raster data are held purely as a set of files. The MGE system architecture which illustrates the relationships between the software modules, relational databases and graphical files is shown in Fig. 2.4 [Intergraph, 1989a; 1989b; Korte, 1994].

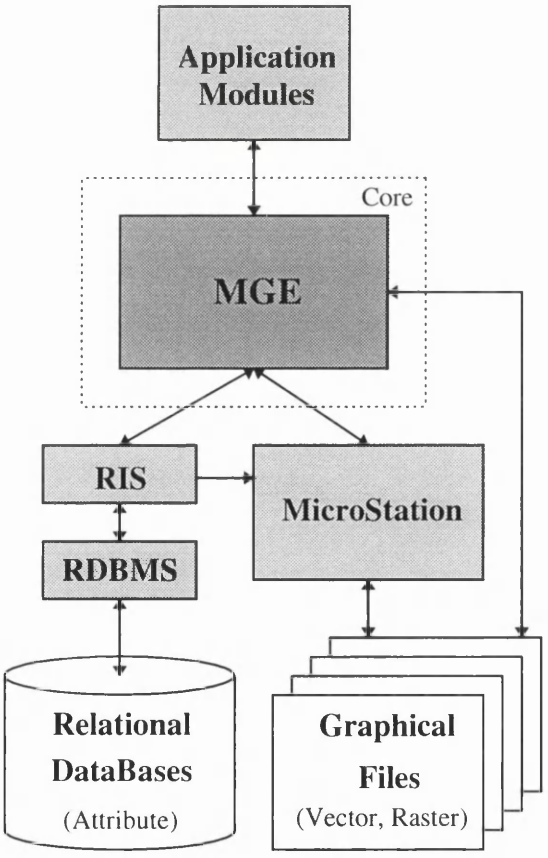


Figure 2.4 MGE system architecture

The graphical data in MicroStation is stored in DGN format which is the same as that used in IGDS. Moreover, MGE also uses the DGN format for its general graphics operations unless the data has to be restructured for specific geographical applications. For example, in order to perform geographical analysis, the graphical data is restructured into a topological file structure using the MGE Analyst module. This arrangement has an advantage of keeping most graphical files compatible across various packages and different platforms while tuning them to meet specific requirements by using specialised modules.

The foundation of MGE products is MGE/SX for Unix systems or MGE/PC-1 for PCs. The former runs both an Intergraph's own proprietary Clipper-based workstations and on Sun's SparcStations, while the latter runs either on an Intergraph's own brand of PC or on a non-proprietary PC. In each case, this is the core environment for setting up, controlling, and manipulating data, and serves as a central integrating core for all of the GIS/Mapping application modules. Each specialised GIS/Mapping application is designed as a module which can be slotted into the overall MGE system to meet specific user needs. The MGE system overview is illustrated in Fig. 2.5 [Intergraph, 1990].

The functions of the major GIS/Mapping application modules used in the MGE system can be summarised as follows [Intergraph, 1990]: -

* **Analyst** (MGA): Executes the creation, query, analysis, and display of topologically structured geographical data.
* **Network Analyst** (MGNA): Generates and manages network data for route planning and analysis.
* **Grid Analyst** (MGGA): Performs overlay, cost surface/optimal path, and statistical analyses, vector/raster conversions, and zone proximity generation of grid data.
* **Imager** (MSI): Carries out digital enhancement and multispectral analysis of remotely sensed raster image data integrated with a vector data set.
* **Terrain Modeller** (MSM): Creates triangle and grid files for use in slope, aspect, elevation, and intervisibility analyses.
* **Map Finisher** (MGFN): Feature-based map composition and symbolisation for screen displays and colour plots.
* **Map Publisher** (MAPPUB): Converts MicroStation graphics files into screened, composited raster files for display and the plotting of the colour-separated film transparencies required for plate making prior to offset litho printing.
* **ETI** : Transfers field survey the data from data collectors used with total stations and converts the data into an MGE compatible format.
* **Stereoplotter I/F Mechanical** (SPI/M): Allows the digitisation of data from a mechanical analogue stereoplotter.
* **Network File Manager** (NFM): Organises and manages information transparently without knowing network location, operating system, or network protocol.

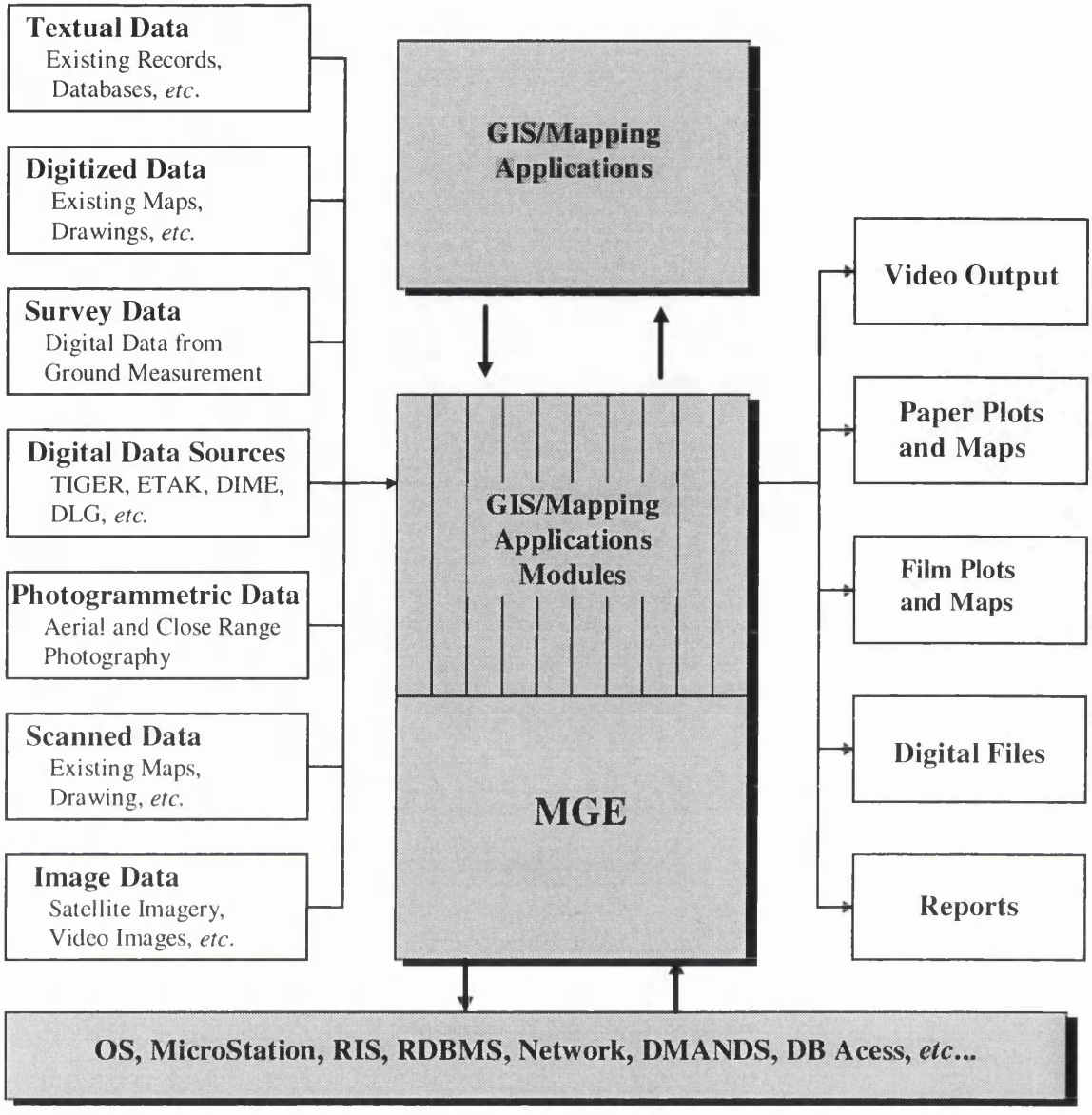


Figure 2.5 MGE system overview [Intergraph, 1990]

* **DB Access**: Performs queries to multiple databases through data-driven displays. GeoDatabase Locate (GDL) allows users to locate features based on specific attributes or characteristics in the associated database. GeoIndex Locate (GIL) allows users to select a working area from a generalised map of the project area.
* **Drawing Management and Distribution System** (DMANDS): For the management, maintenance, and distribution of large-volume drawing and binary images.

It is worth mentioning that Intergraph provides not only software packages but also supplies a wide range of hardware products relevant to data inputs to GIS. In particular, Intergraph's scanners produced by its subsidiary companies, ANAtech and Optronics, are provided to scan existing maps, photographs, films and documents into raster data. The raster data can then be converted into vector data in the form of lines, text and symbols

using its automatic vectorization (I/VEC MS) and symbol/character recognition (I/SCR MS) tools. For very high resolution scanning of aerial photographs, Intergraph, in collaboration with Zeiss, offer the PS-1 which produces raster image data for input to the company's Image Station and its Imager modules.

The emergence of the MGE system has shown the trend of the integration of digital mapping and vector GIS in IGIS development as described in the Section 2.5. The overall approach of the Intergraph MGE system is therefore based on the graphical capabilities of MicroStation CAD software; and sets out to provide a set of integrated tools capable of integrating the diverse representations of geographical information in a common environment. In particular, it supports the multiple data modelling of graphical data by way of restructuring data when needed in specific applications. The MGE system integrates the vector data used by MicroStation and other application modules as well as the raster data used by the Imager module which provides image processing capabilities. Hence, the degree of integration has reached the display and the process levels.

2.6.2 ESRI ARC/INFO

The ARC/INFO system is a very well-known GIS package produced by the Environmental System Research Institute (ESRI). The system runs on a very wide range of platforms from mainframes through mini-computers to graphic workstations and PCs. Thus it has been ported to run on a great variety of operating systems. The ARC/INFO system is based on the tool kit approach which means that it utilises application oriented tools operating on objects. In ARC/INFO, the objects are the geographically locational and non-locational data, while the operators are geo-processing commands employed for the editing, analysis and display of these objects [Morehouse, 1989]. Initially, ARC/INFO was developed as a vector-based GIS and consisted of two main software components: ARC, which was used to manage vector geographical data, and INFO, used for managing the attributes of the geographical features. The ARC component stores graphical data in a topological structure, while the INFO component stores attribute data in relational tables. These two components are linked together by feature identifiers and are also linked individually to the software tools used for various applications. Since then, the system has undergone considerable development. The later software releases also support a Relational Data Base Interface (RDBI) to provide an external link to a second commercial RDBMS such as Oracle, Ingres, Informix and Sybase directly with the ARC part of the system [ESRI, 1992]. The ARC/INFO system architecture is illustrated in Fig 2.6.

The ARC/INFO family of products is built upon a modular software system. The core of the system provides the essential features for basic GIS operations including map capture

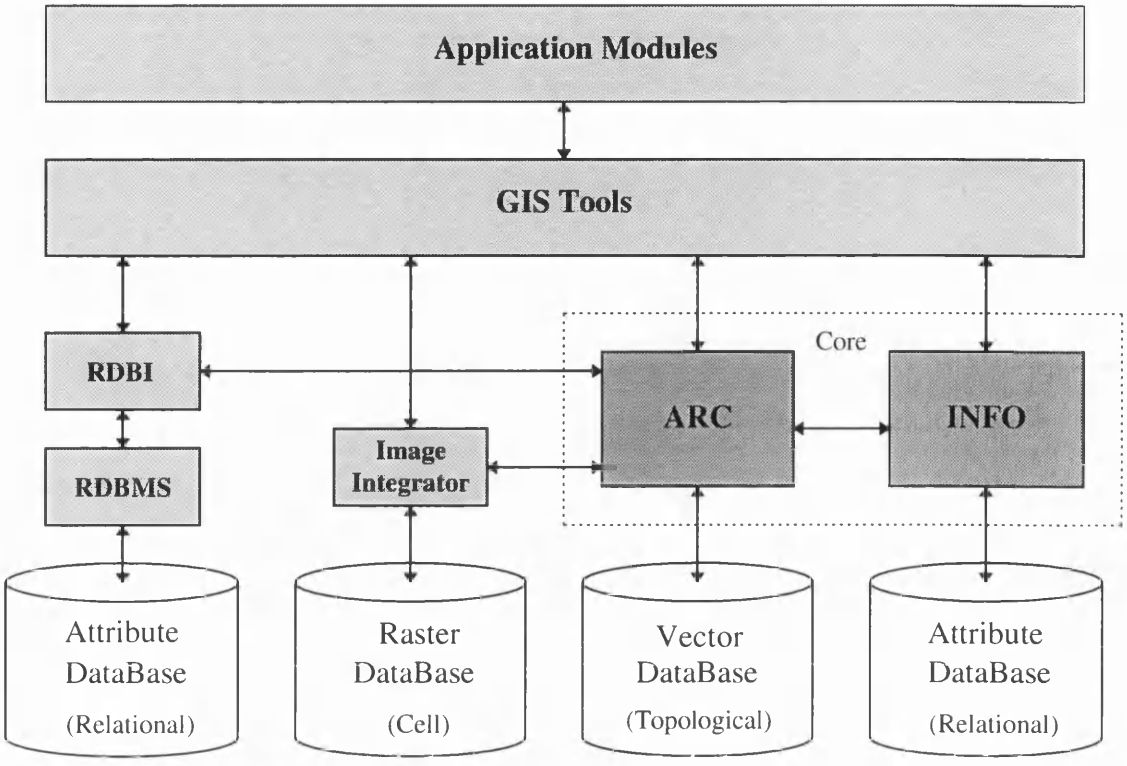


Figure 2.6 ARC/INFO system architecture

and editing, database creation, coordinate transformation and projection, simple map display, data transfer and communication, *etc*. Application specific modules are then built around the core module. As noted above, ARC/INFO has been ported to a great variety of hardware platforms ranging from mainframes to microcomputers. Depending on the type of computer platform, ARC/INFO provides different extended modules for specific applications. For example, the PC STARTER KIT is the core module for IBM PC compatible computers. Other modules can be integrated with this core module. These include ARCEDIT for interactive database creation, update, and management; ARCPLOT for graphic query, display, and cartographic output; OVERLAY for geographical information modelling and analysis; NETWORK for addressing geocoding and network analysis; and so on.

One area of considerable interest in the recent development of the system is the introduction of ArcCAD which allows the user to directly construct a vector database for the ARC/INFO system from the digitisation of existing maps on the widely used AutoCAD systems. The concept of ArcCAD is that it creates an explicit link between the AutoCAD data model and the ARC/INFO data model in order to provide additional GIS functionality. With this development, ARC/INFO GIS databases can be linked tightly with the geometry of the AutoCAD drawing database. This approach is quite different from the commonly used method which builds a GIS on top of a CAD system. Therefore, geographical data

created in ArcCAD is identical to that used by ARC/INFO, and the translation of data between the two systems through exchange file formats such as DXF or IGES is no longer needed [Artz, 1991]. This development has again shown the trend of the integration of digital mapping and vector GIS in IGIS development.

Since 1990, Workstation ARC/INFO has offered the IMAGE INTEGRATOR module for users to integrate raster images with vector maps. The IMAGE INTEGRATOR allows the use of images in conjunction with the ARCEDIT or ARCPLOT modules for the registration of raster data to map coverages so that the raster image can be displayed as the background to a map edit or query session. This feature extends the capabilities of ARC/INFO to provide and to utilise backdrop coverages based on the use of satellite images, scanned maps or raster GIS files imported from other systems. Also it converts raster images into vector representations using raster-to-vector conversion. The process of integrating raster images with maps is illustrated in Fig. 2.7 [Nordstrand, 1990].

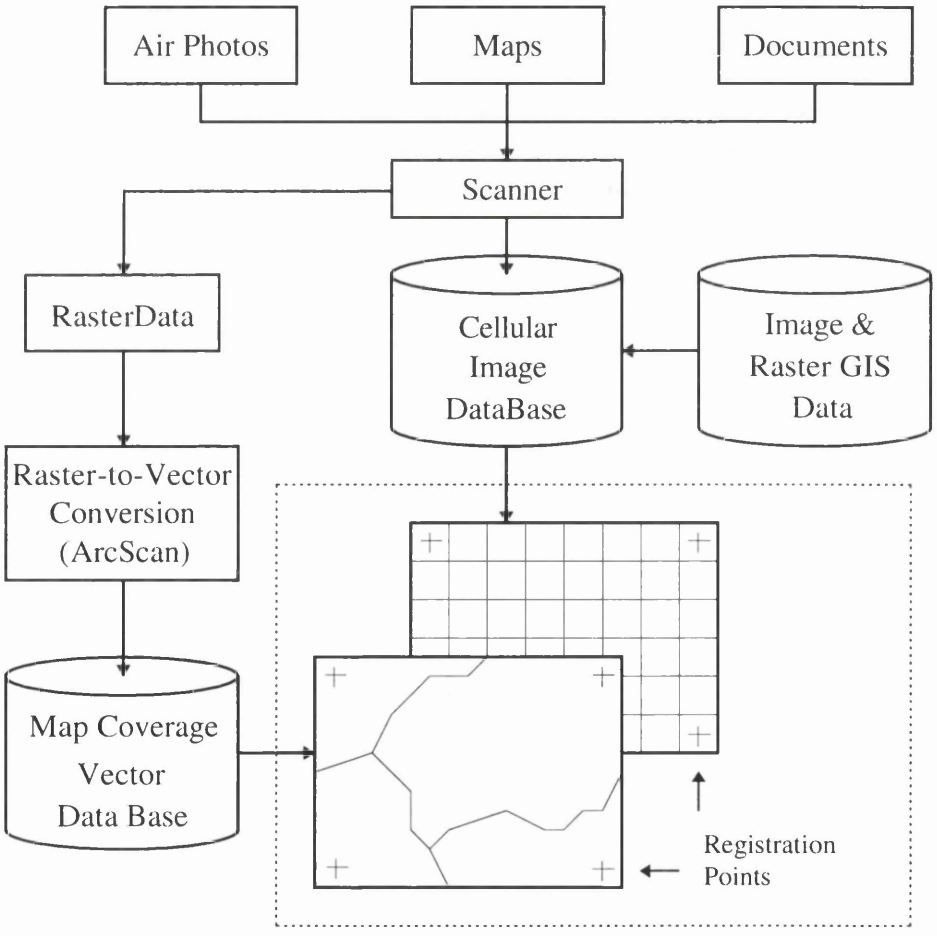


Figure 2.7 Integration process of images and maps in ARC/INFO

More recently, ESRI, in collaboration with Erdas Inc. has also developed the ARC/INFO - Erdas "Live Link" package to combine the ARC/INFO and the Erdas raster image

processing into a form of IGIS. Using the "Live Link" software, the user can process, overlay or display raster images with vector maps. For example, a land use layer in the ARC/INFO system can be graphically overlaid on a remotely sensed geocoded image in the Erdas system for correcting and updating land use data.

The ARC/INFO system was developed as a generic GIS that could be applied to any geo-processing task. The general capabilities of the ARC/INFO system are summarised as follows [Korte, 1994]: -

* Database generation and management.
* Database query.
* Graphic display and report generation.
* Map overlay analysis.
* Network analysis.
* Map sheet manipulation.
* RDBMS integration (DATABASE INTEGRATOR).
* Network database management (Arc Storage Manager).
* Image integration (IMAGE INTEGRATOR).
* Digital terrain modelling (TIN).
* Raster-to-vector conversion and raster editing (ArcScan)
* The entry of survey data directly into the graphics database(COGO).
* Creation of topological databases from AutoCAD (ArcCAD).
* Visualise databases with a uniform user interface (ArcView).

In summary, the ARC/INFO system is centred on a topological vector model for handling graphical data and a relational model for managing attribute data. The strength of the system is generally held to be in its rich variety of analysis tools. The system has recently been extended to support the integration of raster images, and has been combined with AutoCAD via the ArcCAD to take advantage of the power provided by CAD technology.

It can be seen from Fig 2.6 that the ARC/INFO system uses two relational DBMSs - an internal (INFO) and an external (Oracle or Ingres, or other) - for the storage of attribute data. As a result, the duplication of some information in the databases can hardly be avoided. Hence, based on the intrinsic characteristics of the system architecture, it is hard to achieve a high level or degree of data integration. In ARC/INFO, the raster images are mainly used as a backdrop for editing and updating vector maps. The system itself does not support image processing capabilities. Therefore, the ARC/INFO system can only reach the display level of integration. In order to support concurrent processing of vector and raster data, the ARC/INFO system provides an interface module 'Live-Link' which links to the ERDAS image processing system to achieve the process level of integration using the composite method (see Section 1.2.2).

2.6.3 Genasys Genamap

Genamap (formerly Deltamap) was the first commercially available GIS to be offered under the Unix operating system. Genamap is the core product of the Genasys GIS family. Genamap was designed specifically to operate in the Unix environment and operates across a diversity of platforms, including Sun, IBM RS6000, HP, Bull, Data General, Silicon Graphics and PCs under SCO UNIX. Wherever possible, Genasys uses recognised and *de facto* industry standards for software development. For example, it utilises the X Window system based on OSF/Motif for graphics display; TCP/IP and NFS for communications; Relational DBMS and SQL for database management. This has allowed the developers to work with a single copy of the source code for all platforms and the users to choose any Unix-based hardware on which to run the system. In fact, this is one of the key factors resulting in Genamap having soon become a popular GIS product ever since Genasys introduced it in 1986 [Genasys II, 1991; 1994].

Genamap has implemented the concept of spatial views to facilitate spatial queries. A spatial view is physically implemented as a set of optimised index structures to the main database and, as such, is a reference to the query and not the data. A spatial view, which is analogous to a view in an DBMS can be a spatial query using a *Select* operation on the databases or can be the result of other spatial or non-spatial data analysis. The view can be zoomed to, interrogated, queried, reported, tabled, plotted, used for graphical editing and used for any subsequent analytical operation. Since the views are implemented as direct indexes into the map data, this eliminates data duplication and irrelevant intermediate operations when carrying out geographical analysis [Genasys II, 1994]. The mechanism of spatial views provides fast access to databases and consumes little overhead, as well as maintaining the integrity of the databases.

Genamap is the core product of the Genasys GIS family. Basically Genamap is a vector GIS based on a topological data structure. Internally, it provides a spatial (vector/attribute) database and it can also link to external SQL-based RDBMSs, including Oracle, Ingres, Informix, Sybase and others. The spatial database, which is continuous without breaking into map tiles, and can grow or shrink dynamically as storage requirements change. Because of the implementation of spatial views and the efficient use of the Unix file system, Genamap can efficiently handle large databases.

Raster functionality is implemented by another complementary product called Genacell. Genacell uses exactly the same interfaces and command structures as Genamap and can be accessed from within Genamap with no need to move between modules. This feature allows the user to seamlessly combine and analyse both vector and raster data at the same time. Genacell provides a full set of functions for the storage, retrieval, analysis, modelling, and

display of raster data. The integration of Genamap and Genacell forms an IGIS. Application packages may also integrate with Genamap for specific operations. For example, Genacivil is designed for civil engineering applications; Genascan and Genarave are used for the scanning, cleanup, and semi-automatic vectorization of raster information; Genindex is a document management system for various kinds of digital information; and so on. The system architecture of the Genasys GIS is illustrated in Fig. 2.8.

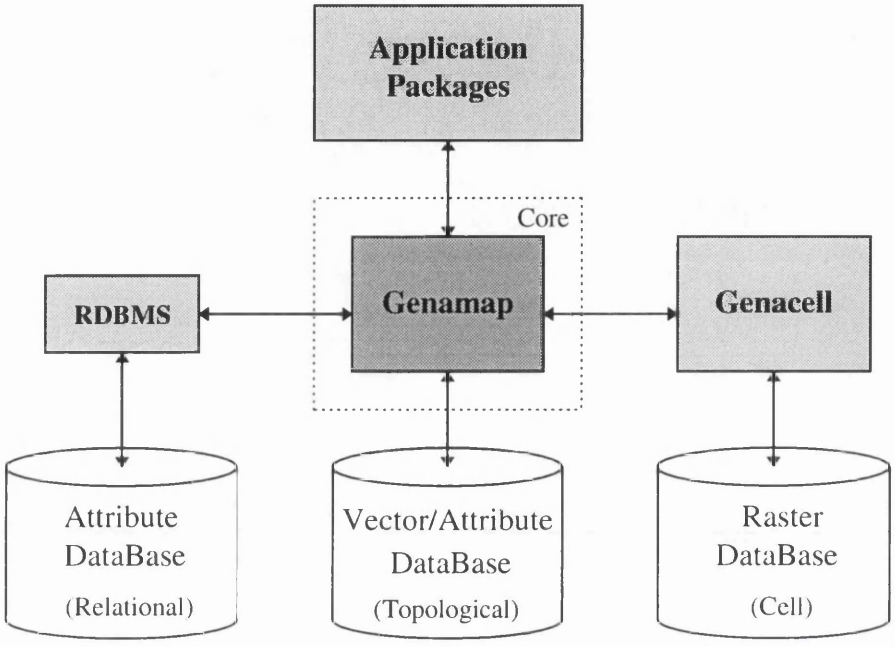


Figure 2.8 Genasys GIS system architecture

Some of the more important functions of Genamap and Genacell can be summarised as follows [Genasys II, 1991; 1994]: -

* Network analysis: Performs districting, shortest path, zones, *etc.*
* Spatial analysis: Includes overlay, buffering, point-in-polygon, proximity, reclassification, the generation of Thiessen polygons, *etc.*
* Raster capabilities: Include slope analysis, visibility, profiling and shading, proximity, neighbour operations, smoothing, surface generation, *etc.*
* Format independence: Spatial data can be stored in any projection system, coordinate system and units. The transformation between the source and the destination formats is handled dynamically and automatically.
* Symbology independence: The display of information can use the raw topology or any combination of point, line and area symbologies. Symbologies can be scale sensitive to automatically adjust to user defined constraints or they may be fixed to certain scales.
* A friendly graphical user interface (GENIUS): uses OSF/Motif "Look and Feel" as a standard interface. It allows the user to build complete interfaces easily without needing to have a knowledge of the OSF/Motif GUI and the X-windows systems.

* A seamless topological database, no space pre-allocation or tiling is required.
* The use of spatial views to query both graphical and attribute data for geographical analysis.
* Supports context (session) control: Allows the user to store the working environment such as current data sets, search criteria, spatial views, *etc.* at any time and resume from any stored environment.
* Hardware and device independence.
* Based on Open Systems concepts and adherence to *de facto* industrial standards..

The overall approach of Genasys GIS is based on the use of a topologically structured database for managing vector data, and of a cellular structured database for storing raster data. Attribute data can either internally be tightly coupled to the spatial features in the vector database or be externally linked to commercially available relational databases. The system provides full database support allowing for both vector and attribute data to be integrated into a single database. This novel feature allows users to make fully use of vector and attribute data in an efficient and fast-access form. In other words, the system has reached the **storage** level of integration for vector and attribute data. As far as the integration of vector and raster data is concerned, the **display** and **process** level of integration can be easily be achieved because both data formats have been included in the initial system design and use the same user interface. However, both types of data are kept in different databases and therefore do not integrate as a single unit at the storage level of integration.

2.6.4 Tydac SPANS

The SPatial ANalysis System (SPANS) is a modular raster-based GIS produced by Intera Tydac Technologies. SPANS runs on microcomputer and graphic workstations under the DOS, OS/2 and Unix operating systems. The graphical user interfaces implemented in different operating systems do not use the same *de facto* industrial standard. For example, the OS/2 version uses Presentation Manager, whereas the Unix version uses OSF/Motif [Tydac, 1989; 1991]. In its promotional literature, Tydac places an emphasis on SPANS's capabilities for data integration, analysis and modelling. Essentially, SPANS was designed to address four major tasks [Tydac, 1990]: -

1. Building and integrating spatial data sets.
2. Exploring the relationship between spatial data sets.
3. Querying and identifying suitable locations.
4. Accessing the impacts of decisions through predictive modelling.

The SPANS approach to data integration is to employ quadtrees to organise and index spatial and attribute data. SPANS provides a wide variety of conversion routines for all the major types of data exchange formats which translate all vector and raster data into the consistent format of the quadtree structure. The quadtree structure enables layers of data to be stored and manipulated at different levels of resolution, and results in very fast overlay times. Apart from this, the quadtree structure is an efficient method of organising and indexing spatial data for the optimisation of data storage and the fast search and retrieval of data.

The quadtree-structured format is employed with the primary data used for performing analytical operations and modelling data. SPANS can also store data in both topologically-structured vector format and cell-structured raster format. Each format has its own inherent advantages for various types of analysis. SPANS uses the particular format (structure) appropriate for the specific analytical task. For example, the vector format is used to compute path lengths within road networks; the cellular raster format is used for rate of spread calculations; and the quadtree-structured data format is used for fast overlay and data compression. These different data formats are integrated within SPANS so that they can be combined in a single analysis operation. For instance, a vector format depicting a proposed road network could be overlaid on a set of raster elevation data to get the average slope of each segment of the road. SPANS also can convert a quadtree file into either a flat raster file or a vector file so that SPANS can be linked as an analytical workstation to other systems [Tydac, 1990; Intera Tydac, 1993]. The system architecture of SPANS can be viewed in Fig 2.9.

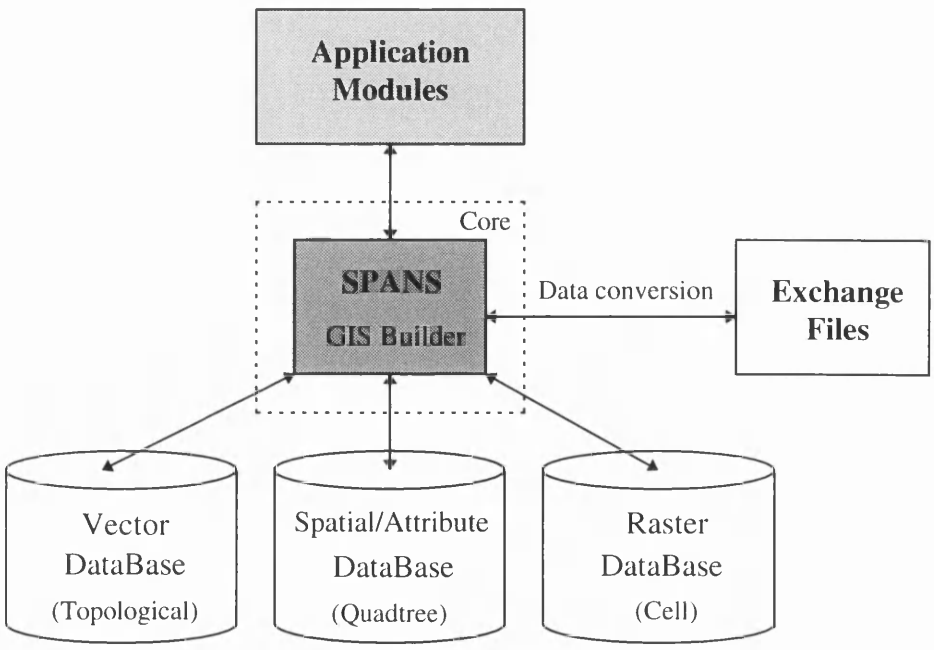


Figure 2.9 SPANS GIS system architecture

The core module of the SPANS GIS system is GIS Builder which is a complete set of basic GIS tools for building databases, constructing analytical models and implementing the visualisation and querying of data. Additional application modules for analytical functions are set on the top of GIS Builder. The major features of the SPANS GIS may be summarised as: -

* It employs quadtrees to organise and index spatial and attribute data.
* It provides the arc-node method for data acquisition and stores it in a fully topological vector format.
* Network and Overlay analysis tools are provided.
* Neighbourhood analysis, *e.g.* data smoothing, edge enhancement, texture analysis, *etc.* can be carried out.
* The topological relationships between areas (adjacency, containment and boundary length) may be analysed.
* It provides a variety of modelling and analysis procedures such interaction, multi-criteria, *etc.*
* DTM contouring and visualisation can be carried out using the TIN method, *i.e.* it generates a perspective 3D view model from surface data.

Apart from these GIS analytical functions, Intera Tydac also developed two extended modules - SPANS MAP and SPANS IMAGE - for the applications of digital mapping and image processing respectively. Recently a separate software package called SPANS Explorer has been developed to deal with the integration of vector and raster data under Microsoft Windows 3.1. SPANS Explorer can transform all the major geographical data types created by other software products, including those produced by ARC/INFO, EASI/PACE, ERDAS, MapInfo, AltaGIS, ArcView2, Lotus 1-2-3 and DBASE III+/IV into the SPANS quadtree-structured format. The Explorer package can be used to view, query, update and manage geographical data as well as performing some general analyses, such as proximity analysis and multiple map overlaying [Tydac, 1994]. This development again helps to confirm that the integration of GIS, digital mapping and image processing systems into an IGIS is an emerging trend.

It is worth mentioning that Tydac Technologies has recently been purchased by PCI Enterprises which produces the well-known image processing software EASI/PACE. As a result, PCI are committed to an integration of the EASI/PACE and SPANS packages. Already the exchange of data between EASI/PACE and SPANS has been made increasingly efficient and compatible. Data derived from remotely-sensed images via EASI/PACE can be exported to SPANS for further spatial analysis. On the other hand, SPANS format files (both vector and raster data) can be imported into EASI/PACE and transformed into the PCIDSK format files. The PCIDSK format was modified from the UNIDSK format which was originally developed by the Canadian Centre for Remote Sensing (CCRS). In this

format, all image data and all statistical data (termed "segments", *e.g.* signatures, training areas, look-up tables, *etc.*) are contained in a single file rather than each segment being stored in a separate file. Thus it is easy to keep track of information relevant to an image and this arrangement also simplifies the operations of data copying, backup and deletion required for data management [PCI, 1993]. It can be seen that, using the PCIDSK format, EASI/PACE has achieved a certain degree of integration for raster data at storage level. Furthermore, EASI/PACE has the ability to create links to non-PCIDSK image files using the LINK program. This creates a PCIDSK file header describing the database structural information and provides pointers to the disk for the raster image being used in other software packages. The advantage is that the raster data stays in one format and is not duplicated in another. This feature allows EASI/PACE to "live link" with the SPANS raster data. However, the "live link" is not a two-way connection, neither does it provide a linkage between vector and raster data.

The overall approach of Tydac SPANS is that the spatial and attribute data acquired from a variety of sources can be integrated into quadtree files. As discussed above, a wide range of import and export routines for vector and raster data are used to perform the data conversion. The quadtree database approach is able to give a full degree of integration. However, the data conversion from vector or raster to quadtree format will normally lead to some generalisation and loss of accuracy. Although SPANS also provides the capabilities of storing uncompressed forms of data in either vector or raster format, this device has however resulted in the disadvantages of having to manage disparate databases. Therefore, it may be said that the SPANS GIS system achieves a full degree of integration at the expense of data accuracy and the details which can be shown on the display. However, SPANS cannot be regarded as a FIGIS because it converts all of its data to the quadtree format instead of actually integrating the vector, raster and attribute data.

2.6.5 Smallword GIS

The Smallworld GIS was launched originally in 1990 and since then has become one of the leading GIS software packages. At the present time, it dominates a significant part of the European utility market including electricity, water, gas and cable companies. In Scotland, it is also being used in land and property information systems. The most distinctive feature of the Smallworld GIS is that the system is based on an object-oriented data model which allows real world semantics to be embodied in the user defined objects at a very high level of abstraction of phenomena, and to be operated in the object-oriented paradigm [Green, 1992; Smallworld, 1992]. The Smallworld GIS also adopts an open systems architecture which adheres to the system standards accepted by the computer industry. Hence, the Smallworld GIS can run on a variety of platforms under the Unix or VMS operating systems as well as accessing data held in different databases (internal and external) using a

single consistent user interface. The overview system architecture of Smallworld GIS is illustrated in Fig. 2.10 [Newell, 1992; Yearsley, *et al.*, 1994].

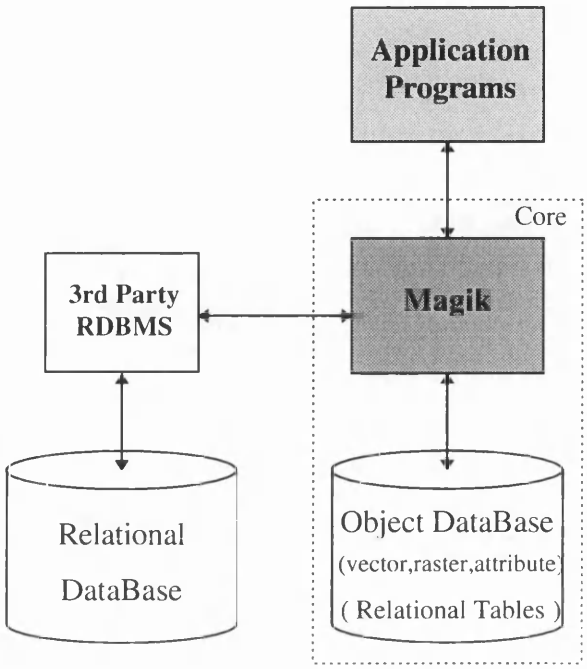
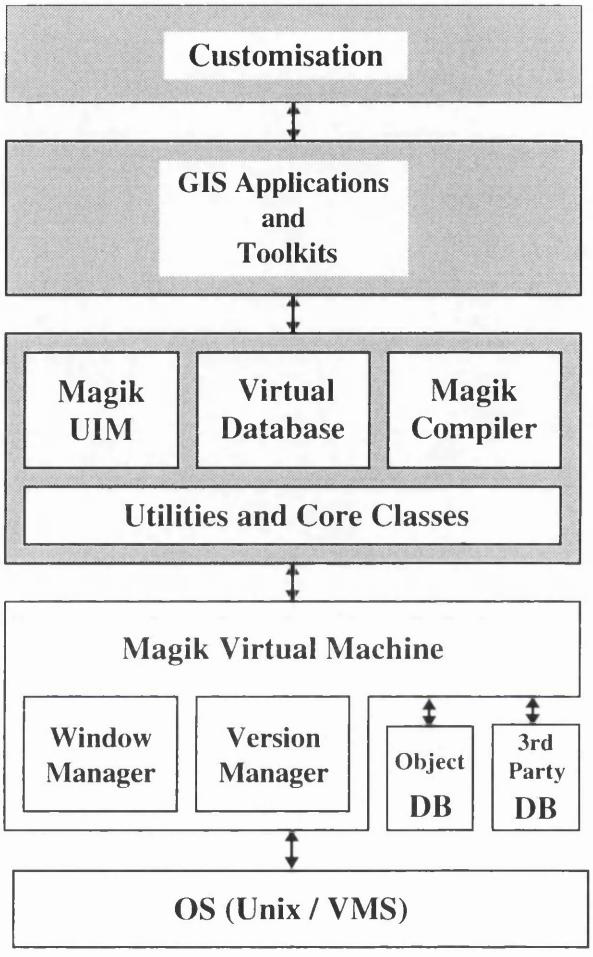


Figure 2.10 Smallworld GIS system architecture

The overall database approach of the Smallworld GIS is to implement an object-oriented database across many platforms by combining an interactive object-oriented programming language, Smallworld **Magik**, with relational database technology. A low-level interface between the object-oriented and relational worlds is built to map a table onto an object, a record onto an instance and a field onto a slot or attribute. The Smallworld GIS provides a proprietary database which allows three kinds (vector, raster and attribute) of data to be stored together internally within the system, and provides external linkages to third-party databases. The fundamental persistence storage is tabular, but using the principle of encapsulation. The table/record structure is made to look like an object data structure. At the lowest level, a table looks like an indexed collection, a record looks like a slotted object and a field is a slot (or attribute). All persistent data is front ended in the object-oriented environment by a "virtual database". This virtual database handles all aspects of data management including data dictionary, integrity, access to the various data sets, and versioning so that the application environment is insulated from the complexities of external data structures and interfaces [Chance *et al.*, 1990; Newell, 1992].

The Smallworld GIS is built using Smallworld **Magik**, which supports both encapsulation and multiple inheritance. The **Magik** language provides a seamless environment in which systems programming, application development, system integration, and customisation are all written in the same language [Chance *et al.*, 1989]. The **Magik** object-oriented

environment is underpinned by a "virtual machine", written in C, which contains a full and extensible set of primitives for handling graphics and interaction with the database. Also they provide remote access to "alien processes", such as external database systems and analysis systems. The **Magik** architecture of Smallworld systems is illustrated in Fig. 2.11 [Newell, 1992].



Figuer 2.11 Smallworld Magik architecture

In Fig. 2.11, the shaded areas represent the **Magik** image which is copied into main memory when the system is initialised. The **Magik** image consists of a set of object classes and associated methods including utilities and core classes, GIS class libraries, *etc.* The database, which is based on a relational data model, contains system tables and application-specific tables. The system tables define the data dictionary and the topological objects, whereas the application-specific tables define the database schema of the particular application. This object-oriented approach enables the system to display and process graphic data in mixed raster/vector format, and virtually stores both maps and images in a single database.

The major features of the Smallword GIS can be summarised as follows: -

* A virtual database integrates vector, raster and attribute data.

* It defines each geographical feature as a manageable object which is comprised of topologically structured geometry with semantic classifiers.
* It supports the creation of user data models as objects and allows these objects to have methods contained within them.
* Analysis functions include network analysis, proximity analysis, buffer zone generation, polygon overlay, *etc.*
* Single query environment for a range of existing databases.
* Seamless mapbase with spatial indexing mechanisms.
* It provides version management to handle updates.
* It uses a single programming language for system, application and customisation development.
* The object-oriented environment provides a comprehensive library of object classes and methods.
* The object-oriented data model allows the system to be configured to meet the needs of customisation.

The prominent feature of the Smallworld GIS is that there are two foundations on which to build an open architecture: (i) a virtual database and (ii) an interactive object-oriented programming language. The virtual database controls the protocols and communications which connect objects to their physical representation on the disk. As a result, the Smallworld GIS is able to access many disparate databases which may be physically distributed over data servers on a heterogeneous network. The interactive object-oriented programming language provides a seamless environment for both system development and user customisation.

Although various data are physically stored in different databases, the Smallworld GIS has achieved a high degree of integration, *i.e.*, it can truly or virtually reach the storage level depending on whether either the internal object store or external databases are being used. However, in order to link to external databases, the software developers have to keep the mapping between programs and various databases consistent. In principle, the construction and maintenance of the software must be quite expensive. On the other hand, Smallworld **Magik** is a weakly typed language. Inevitably, there is a loss in performance and the absence of the compile-time checking available in strongly typed systems. Nevertheless, Smallworld believes that this loss is a price worth paying for the considerable additional flexibility gained by run-time message evaluation in a polymorphic system [Newell, 1992].

The Smallworld GIS system places an emphasis on vector data processing and management incorporated with the capability of on-screen digitising or selective vectorization from a raster image backdrop. Although the system currently does not provide image processing facilities, they may be easily integrated into the system. It can be seen that the system has

achieved a full degree of the display and the process levels of integration as well as a certain degree of the storage level of integration.

2.6.6 Laser-Scan IGIS

Laser-Scan, founded in 1969, started with the development of digital mapping systems, but only later became active in producing geographical information systems. Laser-Scan's products such as LITES, VTRAK, LAMPS have been used in a wide range of mapping applications [Laser-Scan, 1989; 1990; 1991]. Among them, the VTRAK system is recognised as one of the leading software packages specialised in data acquisition. It followed on from the earlier very successful Fastrak and Lasertrak semi-automatic line-following hardware and software system. However VTRAK is designed for interactive and automated vector data capture from scanned raster images, particularly from existing maps. The VTRAK system has been widely used in many national mapping agencies around the world for creating a digital map database [Laser-Scan, 1992]. Laser-Scan also developed the Horizon and Metropolis GIS systems based on the use of the VAX/VMS platform. This took a fairly conventional approach with the use of a Laser-Scan developed vector database and display capability allied to a commercially produced relational database such as Oracle or Ingres used for attribute data. In recent years, Laser-Scan has been involved in developing a prototype IGIS for British National Space Centre (BNSC) and this has resulted in the release of a new product - the IGIS [Hartnall, 1993c].

The Laser-Scan IGIS system has been developed to provide an integrated environment for handling spatial and non-spatial data, advanced data structures for handling geographical information and a range of functions allowing data analysis [Hartnall, 1993a]. The IGIS system is designed to run on a number of Unix-based workstations, including DECstation, DEC Alpha, Sun SparcStation, IBM RS6000 and others.

The Laser-Scan IGIS system architecture is illustrated in Fig 2.12 [Hartnall, 1993b]. The system has been built using the application components provided by the **Gothic** environment. **Gothic** is an Open Systems **A**pplication **D**evelopment **E**nvironment (**ADE**) designed for building information systems that use and process geographical data. The **Gothic ADE** is based on object-oriented database management and programming techniques, and has five layers in its architecture. The functions of each of these five layers are summarised as follows [Laser-Scan, 1993a; 1994]: -

1. **The Operating System** supports the hardware and its associated systems.
2. **The Operating System Interface** holds all operating system dependent functions in service modules to achieve multi-platform interoperability.

3. **The Gothic Toolkit** contains the object database manager and all the spatial and non-spatial analysis, display, user interaction and manipulation tools required by GIS developers.
4. **The Applications Programmer's Interface (API)** is a programming language which is independent of the functionality of the system, and provides capabilities for future extensibility. The API language can access both the toolkit and the data.
5. **Application Programs** are user programs written using the compiled, structured API language, ensuring that applications can be ported without recompilation across different platforms.

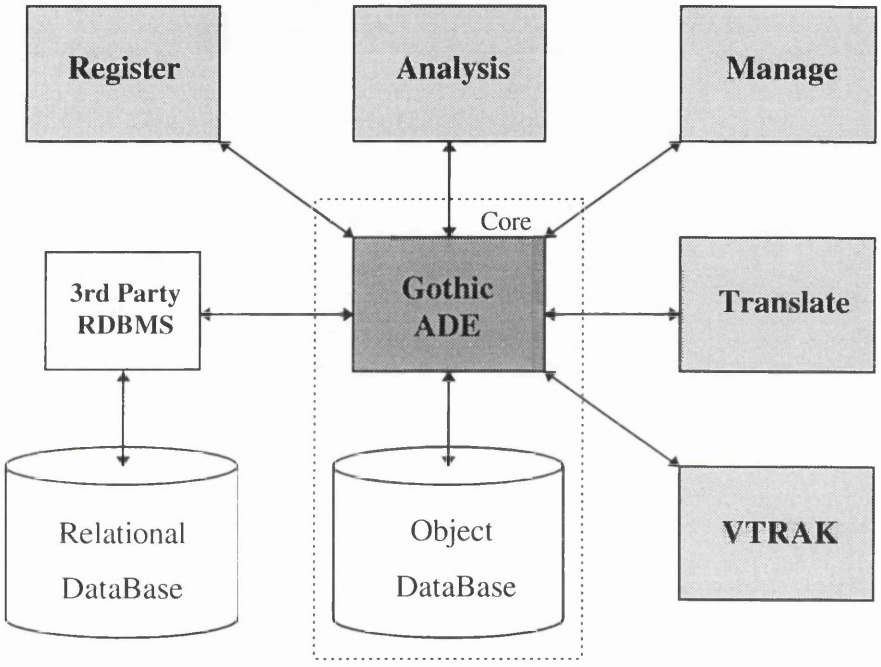


Figure 2.12 Laser-Scan IGIS system architecture

In **Gothic ADE**, an object-oriented database is provided for the storage of both spatial and non-spatial data. The object database allows the modelled data of real world objects to be stored in an intuitive way. As a result, collections of objects can be combined into complex structures which can be used to mimic those found in the real world. The **Gothic** object database provides the following key features: -

* A wide variety of data types can be held in an object, including integers, reals, strings, dates, raster data, vector geometry, *etc.*
* "References" are used to enable objects to point at one another and this provides a mechanism for rapid access between related objects.
* It supports encapsulation and multiple inheritance.
* Version management provides multiple version capability with only the modified data stored in a new version.

* Supports continuous vector maps and raster images.

The **Gothic ADE** also provides gateways to access an SQL-based RDBMS such as Oracle and Ingres. In addition, an object database manager is used for the administration of all user access to data. The **Gothic ADE** architecture supporting the IGIS system is illustrated in Fig. 2.13 [Laser-Scan, 1993a].

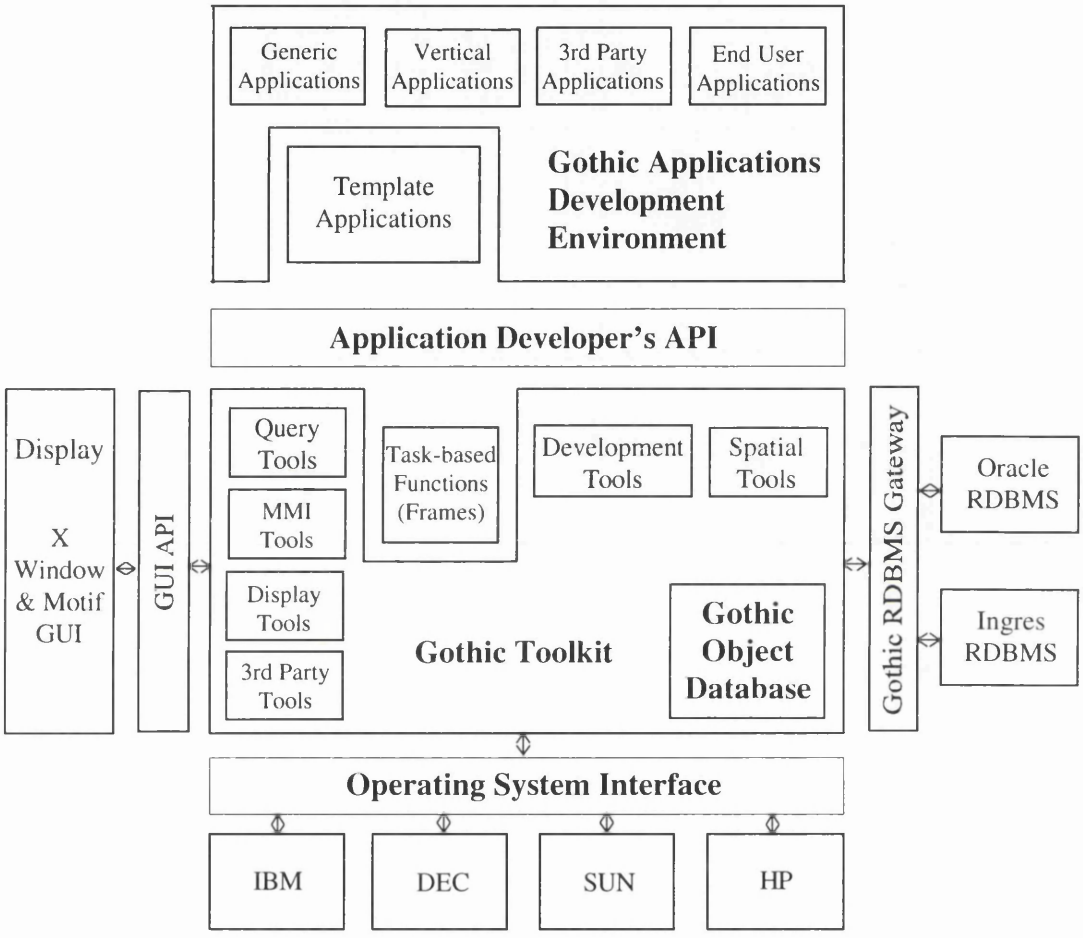


Figure 2.13 Gothic ADE architecture

The Laser-Scan IGIS system has four main core components: these are the **Translate**, **Register**, **Manage** and **Analysis** modules. The **Translate** module provides the means of importing and exporting graphical data and associated attributes. The **Register** module is used to define transformation parameters between different spaces to enable the geometric correction and rubber-sheeting of raster image data to be implemented. The **Manage** module provides facilities for managing the object-oriented database including user access control, versioning, and backtracking. The **Analysis** module provides all the facilities needed for session management, display, manipulation, analytical queries and the production of hard copy maps and reports. The IGIS system can also integrates VTRAK for automated or semi-automated data capture.

The major features of the IGIS system can be summarised as follows [Hartnall, 1993b; 1993c; Laser-Scan, 1993b; Laser-Scan, 1994]: -

* The integration of vector and raster map data, terrain data, satellite and other digital imagery and geo-referenced statistical and attribute data.
* A versioned object database with an integral roll-back recovery mechanism.
* Translation facilities for a wide variety of raster and vector formats, resolving data from various sources and map projections.
* Image classification, filtering and band combination facilities, as provided in image processing systems used with remotely sensed data.
* Image/Map analysis and manipulation capabilities.
* Continuous mapping and cartographic quality output facilities.
* Analysis functions include raster combinations, DTM processing, slope and aspect map generation, intervisibility map generation, hill shading, *etc.*
* All the applications of the IGIS system adopt the same Man Machine Interface (MMI) based on OSF/Motif and the X-window system which provides a Graphical User Interface (GUI).

The Laser-Scan IGIS system fully integrates vector and raster data in an object-oriented database. Within **Gothic ADE**, both vector and raster data are interwoven for concurrent processing and analysis. It is clear that the Laser-Scan IGIS system has achieved a full degree of the display, the process and the storage levels of integration. The system design emphasises the integration of remotely sensed data into the GIS analysis process. Therefore, IGIS provides extensive image processing capabilities in addition to its vector and raster GIS functions. This development again demonstrates the convergence of vector GIS and image processing systems.

2.7 Summary of the Review

The previous section (2.6) of this chapter has reviewed the current situation in the development of commercial IGIS software, taking six well-known GIS vendors' products as representative examples to illustrate the different approaches which have been taken to implement the integration of vector, raster and attribute data. The various approaches to data modelling, database design, system architecture and integration have been the major concerns of this review. Other features which are closely related to the database management issue have also been discussed briefly. In fact, the manner in which many of these features have been implemented is often regarded as a major factor in the success (or otherwise) of a system. Common features to all or most of these systems are summarised as follows: -

- Vector data is topologically structured, raster data is cellular structured, and attribute data is organised in relational tables.

- For those systems that use the conventional database technology, vector data and raster data are stored in disparate databases (or files), while the associated attribute data is stored either along with vector data (*e.g.* as in Genamap and SPANS) or in commercial RDBMSs (*e.g.* as is done in MGE, ARC/INFO and Genamap).

- For those systems that employ the object-oriented approach, comprising an object-oriented programming language together with application development tools (*e.g.* Smallworld Magik, Laser-Scan Gothic), an integrated environment is developed to provide for the handling of both spatial and non-spatial data. A proprietary object database supported by the object-oriented language is used to store the modelled real world objects, for which the vector geometry, raster image geometry, attribute description and methods that can apply to them are regarded as properties. An interface to existing RDBMSs can also be considered as an important function or provision in such systems.

- System design based on an Open Systems architecture and adherence to recognised or *de facto* industrial standards are being widely adopted in the most recent system development.

It is apparent that those systems which are based on conventional database technology cannot provide a full degree of integration. A conventional data model, especially the commonly used relational model, is really too simple for the modelling of complex geographical data. In addition, conventional database systems do not support the complex data types found in programming languages so that they cannot hold the graphical elements of a GIS. As a result, a spatial database has to be constructed separately and this means that a linkage to the attribute database has also to be provided. Furthermore, conventional programming languages do not support a feature that allows both vector and raster data to be coexistent (*i.e.* encapsulated) within an object data type. Once again, this drawback results in the separation of vector and raster databases.

Some vendors have succeeded in implementing an IGIS with an object-oriented approach. The two examples (Smallworld GIS and Laser-Scan IGIS) described in the previous section have shown a full degree of data integration. The Smallworld GIS system employs an object-oriented programming language (Magik) and an object-oriented database to achieve the required data integration. Both of these have been developed in-house. The Laser-Scan IGIS database is also object-oriented. However, it has been developed using an object-oriented approach based on Laser-Scan's Gothic ADE which is not actually a OOPL. Quite

apart from the GIS vendors' interest in the use of an OOPL, laboratory researchers are also tending to adopt this approach. Some typical examples of GIS systems based on the OOPL approach are: GEOSTAR from Wuhan Technical University [Gong, 1994]; the prototype GIS from University of Leiden [Oosterom, 1993], *etc.* However, the persistent data in these object databases cannot be shared by other systems since the use of a database is system specific. However, currently this approach is quite pragmatic with regard to providing both object-oriented handling and an interface to existing RDBMSs. Still it may only be regarded as a temporary solution to provide a FIGIS. In order to have a shareable database, it is very desirable to adapt or to apply an advanced DBMS which provides more flexible capabilities for handling complex geographical data.

Recently, some advanced DBMSs, including extended RDBMSs, OODBMS, and others, have begun to appear in the marketplace. However, unlike the standard RDBMS, there is no consensus in the computing community regarding the formal definition of the data models, query languages, *etc.* which should be used in such systems. Despite this disadvantage, a number of organisations have attempted to develop a prototype GIS based on what is currently regarded as advanced DBMS technology. These pioneer projects have given some indications of the capabilities and limitations of this advanced DBMS technology. Hence, a further review of research work in this field is given in the following section.

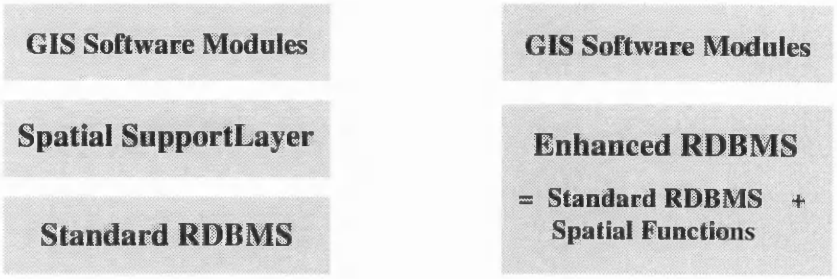
2.8 Recent Progress Made in Advanced DBMS Technology

Standard SQL-based RDBMSs are well suited for handling attribute data, but they are inappropriate for the storage of graphical data. So the most common database design for GISs which are designed specifically to employ commercial RDBMSs is to store the graphical data in a proprietary file management system. A complex pointer mechanism is then required to provide a link between the graphical and the attribute data. As a consequence, the overall system performance is reduced and the integrity constraints can be violated. Recently, two approaches based on advanced DBMS technology have been investigated to overcome these drawbacks. One approach is to extend the capabilities of a standard DBMS to handle graphical entities, whereas another approach is to implement an innovative OODBMS in which the spatial extension is completely embedded in the DBMS.

2.8.1 The Extended RDBMS Approach

The extended RDBMS approach is to provide spatial database facilities by adding spatial data types and associated functions to a standard RDBMS. It can be carried out either by external attachment or through the built-in method. The external attachment method is to develop a spatial support layer (or shell is called) above the standard RDBMS, whereas the

built-in method is to embed the additional features into the standard RDBMS [Sinha and Waugh, 1988; van Oosterom, 1993]. The GIS architectures based on these two variants of the extended RDBMS are illustrated in Fig. 2.14.



(a) External attachment method (b) Built-in method

Figure 2.14 The two possible GIS architectures based on the extended RDBMS approach

The basic concept of the external attachment method is that the spatial support layer translates geographical queries into standard SQL-queries and also provides spatial indexes which are implemented by means of auxiliary relations. The external attachment method does not get involved in the internal structures of existing RDBMSs - and the tremendous development costs which would be involved in altering these - so it has been used in some research laboratory experiments. Examples of prototypes GISs based on the extended RDBMS and developed by the external attachment method are GEOVIEW from the University of Edinburgh [Waugh and Healey, 1987; Sinha and Waugh, 1988] and SIRO-DBMS from CSIRO in Australia [Abel, 1989]. A commercial product, Geo/SQL from Generation 5 Technology, which is a well-known vector-based GIS in North America, has also been developed by this method and is built on the framework of standard RDBMSs and AutoCAD graphical software [Korte, 1994].

The external attachment method is rather inefficient because geographical data types have to undergo the mapping from the program domain to the relational tables via the spatial shell interpreter. Furthermore, the data access system is also inefficient because geographic queries have to be translated into standard SQL-queries and perform operations of relational joins. The concept of the built-in method is to remove the need for a spatial support layer by incorporating spatial data types and access methods into a standard RDBMS. However, it will not be possible to implement this method in a standard DBMS unless its internal structure is open. Therefore, several GIS software vendors have developed their own proprietary RDBMSs which extend spatial capabilities. For example, the vector-based System 9 GIS from Computer Vision (originally developed by Wild) utilises a proprietary RDBMS called Empress to manage both graphic and non-graphic data [van Eck and Uffer, 1989; Abdallah, 1990]. On the other hand, some commercial RDBMS

vendors have also developed general-purpose extended RDBMSs which add facilities such as complex data types, abstract data types, spatial access methods, object data management, and others to standard RDBMSs. Examples of these extended RDBMSs are Ingres 6.3, Sybase and Xidak Orion, all of which have already incorporated limited object data management extensions, including user-defined types, rules, and programming-language procedures stored in database attributes [Cattell, 1991]. Perhaps the most powerful extended RDBMS is the research prototype system Postgres from the University of California at Berkeley. Postgres is the successor to Ingres. It provides some geometric types along with user-defined types that allow the users to define geographical data types as system types, and supports different spatial indexing mechanisms such as B-tree and R-tree. One research-based GIS development is based on this built-in type of extended DBMS. This is a prototype GIS called GEO++ from the University of Leiden which has been developed using Postgres. The GEO++ system provides an extensible DBMS for users to incorporate the specific functionality which they require, such as raster data types or 3D data types [Vijlbrief and van Oosterom, 1992; van Oosterom, 1993].

The advantage of the extended RDBMS approach is that the relational RDBMS already provides well-developed facilities such as the Entity/Relationship (E/R) formalism, the non-procedural language SQL, data transaction and so on for the management of the attribute component of the database. Nevertheless, the extended RDBMSs are still deficient in several aspects. In the first place, the data model is too simple for the representation of complex geographical data. Furthermore the semantic links are lost because the coherent geographic entities have to be broken into multiple parts in order to fit the data into the relational tables. Finally the data access is inefficient because mapping between the user data model and the relational tables is required [Cattell, 1991; Cooper, 1993; van Oosterom, 1993]. Consequently, in general, the use of an extended RDBMS is not regarded as a good tool for developing an IGIS.

2.8.2 The OODBMS Approach

As the name suggests, OODBMSs are the database management systems which are based on the object-oriented data model. These systems have an architecture which is based on the use of a database programming language. Therefore, OODBMSs are sometimes called object-oriented database programming languages (OODBPLs). In this thesis, these two terms (OODBMS and OODBPL) are used interchangeably, depending on which particular aspect - database management or object-oriented programming - is being emphasised. In an OODBMS, the data used by the database maps directly onto that used by the programming language. Consequently the considerable data transformation and data construction/decomposition operations involved in RDBMSs will be eliminated. Apart from the advantage of improving efficiency, the data model representing the real world can also

be made to harmonise well with the representation of complex objects in an OODBMS. An OODBMS provides a query language, object types with inheritance, the caching of objects in main memory, transaction management and remote data access [The Committee for Advanced DBMS Functions, 1990; Helokunnas, 1994]. Quite a number of OODBMSs are commercially available, *e.g.* some well-known systems such as GemStone from the Servio Corporation; ObjectStore from Object Design; ITASCA (originally called Orion) from Itasca Systems; O_2 from O_2 Technology; and ONTOS from Ontos Inc. [Cattell, 1991].

Although OODBMSs suffer from a lack of standards to allow the portability of systems developed using them [Cooper, 1993], these systems have provided a popular approach to the implementation of object data management in CAD and CASE software packages. Recently, OODBMSs have gained much attention from GIS researchers. In particular, the OODBMS approach is being used in experiments into the development of an object-oriented GIS. Two representative prototype systems - IGN's GeO_2 and CSIRO's ONTOS - have been selected for reviewing this particular development. Originally both IGN and CSIRO developed their prototype GISs based on the extended RDBMS approach. These were GéoTropics [Bennis *et al.* 1990] and SIRO DBMS [Abel, 1989] respectively. Afterwards, their system development was transferred to the OODBMS approach; however each uses a different commercial OODBMS product as the basis of the system. Currently, both GeO_2 and ONTOS are vector-based systems. However, they can be easily extended to become IGISs because OODBMSs are capable of handling large attributes, or BLOBs, in which case, an image may be stored as a single attribute value. The short review of these two systems conducted below is intended to give a general idea of current GIS research and development based on the OODBMS approach.

2.8.2.1 IGN GeO_2

The French Institut Géographique National (IGN) has developed its prototype GIS system GeO_2 using an OODBMS called O_2. The O_2 DBMS was originally developed at GIP Altair, in Le Chesnay, France and has been transformed into a commercial product marketed by O_2 Technology. Also O_2 provides a database whose values can be described in the context of a class hierarchy of objects. Any object or value of any type can be stored in the database. O_2 allows a variety of database schemata to be created, and can store many databases against each of these schemata. O_2 can be interfaced with a variety of programming languages such as CO_2 and C++. Apart from its basic database management facilities, O_2 also provides the following important features: a generic graphical interface, O_2Look; a set of system classes, provided as the schemata O_2Kit; an *ad-hoc* query language, O_2SQL; and the incorporation of a network facility into the OODBMS [Cattell, 1991; Cooper 1993]. On the whole, O_2 is a very powerful OODBMS.

The objective of IGN's GeO_2 is to produce a GIS that is capable of handling different data structures, and induces independence between these data structures and the GIS operations. In this way, the GIS operations have to be programmed only once and need not be modified, even when they will be utilising different data structures. GeO_2 takes advantage of the object-oriented aspect, in particular of the inheritance mechanism, of the O_2 DBMS to achieve the concept of independence. The conceptual data model used for the GeO_2 database was based on the data model of the interchange format DIGEST-VPF (DIgital Geographic Exchange STandard - Vector Product Format) which was developed by the DGIWG (Digital Geographic Information Working Group) [DGIWG, 1992] for the exchange of military geographical and topographical data. In DIGEST-VPF, four topological levels are described and each one defines a way of structuring and managing the geographical information. These four levels are: Cartographic Spaghetti; Non Planar Graph; Planar Graph; and Full Topology [Williams, 1993]. Using this data model, three types of data structures: Spaghetti, Network (Non Planar Graph and Planar Graph) and Map (Full Topology) are defined in GeO_2. A hierarchy between these three vector data structures is constructed. Using the inheritance mechanism provided by the O_2 DBMS, a unique internal data model is built to provide users with anyone of these three structures in a transparent way. Hence, it is only necessary to have one implementation of spatial operators for the unique internal data structure rather than one for each of the different data structures. [David *et al.*, 1993]. In other words, GeoO_2 has realised a high-level conceptual data model which points towards a universal user data model by using the OODBMS approach.

GeO_2 was written in CO_2 and has been implemented with the R* tree spatial access method. GeO_2 has been operated with a large volume of real geographical data containing about 40 Mbytes. GeO_2 has also been used as a testbed for evaluating the performance of different spatial access methods [Peloux *et al.*, 1993]. The system is considered as being efficient in performance and being extensible in terms of the management of geographical information. The advantage of the OODBMS approach used in modelling complex geographical data is very obvious. However, some drawbacks related to the O_2 DBMS have also been discovered in the course of developing the GeO_2 system. The O_2 DBMS lacks an array constructor which is important to manage intermediate points in polylines and polygons and an association constructor which is needed to handle inverse references between two objects. Furthermore, the O_2 DBMS is not able to define a spatial clustering of objects [David *et al.*, 1993].

In order to take advantage of the features supported by O_2 DBMS to develop GeO_2, IGN has added the functions necessary to extend the O_2 system for the specific purpose of managing geographical data. These include the provision of geometrical data types (point, line, polygon) to its basic structure; the associated functions (union, distance, rectangle, *etc.*); and the predicates (cross, adjacency, overlapping, *etc.*) which are provided for query

languages. Although a lot of efforts have had to be made to provide these extensions and there still remain some of the disadvantages mentioned above, still the overall features of the GeO_2 system have shown that the OODBMS approach is very promising for development of an IGIS.

2.8.2.2 CSIRO ONTOS

CSIRO ONTOS is a prototype GIS developed by the collaboration of the Commonwealth Scientific and Industrial Research Organisation (CSIRO) and the Defense Science and Technology Organisation in Australia. The CSIRO ONTOS is based on a commercially available OODBMS called ONTOS [Milne *et al.*, 1993]. The ONTOS system is a multi-user, distributed OODBMS with a C++ class library interface. ONTOS provides a persistent object store for the storage and retrieval of objects denoted by the C++ programs. The persistent store uses the class hierarchy to organise the objects. A library of classes and functions is provided in ONTOS to aid the mapping of objects between the C++ programs and the persistent store. However, the programmer has to specify explicitly the persistence of objects. In addition, ONTOS also has the following important features: a variant of SQL as a query language; inverse-attribute pairs that represent a single relationship are automatically maintained; multiple versions of objects are supported; and the provision of administrative tools for managing the schema evolution and physical data clustering [Cattell, 1991; Cooper, 1993]. The advantage of ONTOS is that it works with standard C++ compilers without modification. However, the ONTOS DBMS is generally regarded as a low level kind of OODBMS due to the fact that the persistence of objects needs to be explicitly activated by the programmer [Cooper, 1993].

The objective of CSIRO ONTOS is to test the performance of the ONTOS DBMS when applied to geographical data handling. A contour layer extracted from digital maps was used for such an experiment. The test data contains 2,568 contours, with a total of 266,029 coordinate points. The conceptual data model used for the ONTOS database design was based on the data model of the Spatial Data Transfer Standard (SDTS) developed by the U.S. Geological Survey [Fegeas, 1992; Williams, 1993]. SDTS defines a set of primitive spatial objects that may be used either individually or in aggregate form to represent any spatial phenomenon. In the ONTOS database, the chain object type, which consists of a string of point-coordinates, is used to define a contour class. The same test data was also used to create two different databases for the standard Oracle RDBMS and the SIRO-DBMS which is of the external attachment type - as explained in Section 2.8.1. The comparison of the three DBMSs focused on the capability of each system to handle composite and complex objects. The results show that the ONTOS system gives the best overall performance and has the fastest access of objects after the first retrieval resulting in the advantage of object caching [Milne *et al.*, 1993].

CSIRO concludes that the development of GIS software based on the OODBMS approach requires less time than that needed when using the standard RDBMS and the extended RDBMS approaches. The two principal advantages provided by the OODBMS approach are: (i) no programming effort is required to translate objects delivered at the database interface; and (ii) it is also likely that previously developed modules can be reused. Nevertheless, several disadvantages were also found within the ONTOS DBMS. The object-SQL available in ONTOS does not permit the full power of the relational join to be realised. Thus some queries have to be implemented by procedural object interaction. In addition, there was difficulty in implementing the system due to inadequate documentation and incompatibility between releases. Furthermore, it is a considerable challenge for the average software engineer to learn the concepts in both C++ and ONTOS and become productive with these tools.

This prototype GIS system developed by CSIRO has placed an emphasis on the capability of the ONTOS DBMS to support the definition, creation, update and retrieval of a geographical database. Although many other characteristics of ONTOS DBMS have not been exploited or tested in the CSIRO ONTOS, again this research project has demonstrated that the OODBMS approach has potential for IGIS development.

2.9 Discussion

As mentioned before, the backbone of an IGIS is the database management system. The database management system should provide a database which is either logically or physically integrated, allowing it to store various types of geographical information, and to supply the facilities needed to handle all aspects of data management. The degree of data integration, to either the display, process, or storage level, depends on the capability of the database technology selected and used for the system development. In principle, an IGIS demands a database technology that can provide a fully integrated database environment and can manage complex geographical data in a very efficient way.

As has been discussed above, most commercial IGIS products are based on the use of conventional database technology whereby vector and raster data are held separately in different databases, and attribute data is often stored in a relational database. The conventional database approach can only provide a certain degree of integration depending on the specific methodology applied in the implementation, namely the use of either the composite, extended, or complete method. Few IGIS products have been developed using an object-oriented programming language. Such an approach utilises an object-oriented data model and supports the persistence of objects. The object-oriented programming approach is able to achieve a full degree of integration because vector, raster and attribute

data can be regarded as descriptive information about objects. On the other hand, till now, approaches based on an advanced database technology have only been used in research prototype systems, notably the extended RDBMS and the OODBMS approaches. The extended RDBMS approach takes the available resources of an existing RDBMS into consideration. Several extended RDBMSs may be able to build an IGIS, but the overall performance of such systems is generally poor in terms of their data manipulation. With regard to the OODBMS approach, it appears to be a technique that potentially could take the place of the current conventional database technology. The OODBMS approach employs an object-oriented database to model the real world as closely as possible. In addition, the mapping between the programming language and the database management system may also be removed. As a result, an OODBMS can not only offer an integrated database, but it may also perform quite well on geographical queries. Therefore, the implementation of an IGIS utilising an OODBMS can be considered as a viable approach as long as the relevant standards of the OODBMS are well established.

From the above discussion, it can be seen that IGIS development will benefit from the integration of the programming language and the database management system, particularly if both are object-oriented. Through this approach, a programming language is extended with database capabilities, while the database management system has the capabilities of a programming language(s) added to it. The evolution of such an integration results in the emergence of database programming languages. Analogous to the different levels of integration achievable in an IGIS, a database programming language may reach different degrees of integration. In principle, the stronger the integration, the better the working environment it will provide for an IGIS. There are many existing database programming languages which may be classified into different categories. A particular category called persistent programming languages has not yet been exploited or explored regarding its potential for the development of an IGIS. These persistent programming languages appear to offer a sound environment for the integration of programs and the database. In theory, the persistent programming languages should provide a better integrated environment than other database programming languages. However, building an IGIS involves many other requirements apart from the need for an integrated database environment. It has also to be proved that requirements such as multiple data modelling, the handling of multi-scale maps; the management of multi-resolution images; and the graphical display capability, are viable using such language systems. Therefore, the major objective of this research is to explore and exploit the capabilities of persistent programming languages in terms of furthering the development of an IGIS. Napier88, which is one of the main persistent programming languages, has been chosen as the development tool in this research. Because persistent programming languages in general and the Napier88 system in particular are still new to the GIS community, the basic concepts and the main features of these languages will be described in the next chapter.

CHAPTER 3 : NAPIER88 AND ITS USE AS THE IGIS DEVELOPMENT TOOL

3.1 Introduction

During the 1960s and 1970s, programming languages and database management systems were developed quite separately from one another. However, in order to develop application software, an interface between the programming language and the database has to be created or established. The creation of such an interface not only consumes considerable resources, but it also results in a number of problems. Two main problems are *the semantic gap*, which is the difference between the data model used in an application program and its representation in the database, and *the impedance mismatch*, which is the inconsistency of data types existing between the program language domain and the database domain [Atkinson and Buneman, 1987; Khoshafian and Abnous, 1990; Kim, 1990; Cattel, 1991]. These problems become matters of major concern when dealing with complex applications such as CAD, CAE and CASE, let alone GIS. During the 1980s, the need to produce an integrated system combining both programming facilities and database management became widely recognised, and since then many attempts have been made to construct programming languages which are completely integrated with a database management system. These languages are termed *database programming languages* (DBPLs). Atkinson and Buneman [1987] have given a detailed overview of the development in DBPLs and have identified the requirements that such a development has to fulfil in terms of three criteria: *data type completeness, persistence*, and *expressive power.* So far, during the 1990s, the main development of DBPLs has tended also to incorporate object-oriented concepts in the system design, and this has resulted in the emergence of *object-oriented database programming languages* (OODBPLs) or *object-oriented database management systems* (OODBMSs). On the other hand, another important development in DBPLs has been that of *persistent programming languages* (PPLs) which adopt a uniform approach to persistence in pursuit of a fully integrated database programming environment.

Both OODBPLs and PPLs are able to meet the three criteria of a DBPL set out above and to alleviate most of the problems of the semantic gap and the impedance mismatch. However, there are dissimilarities between OODBPLs and PPLs. These major differences can be summarised as follows [Atkinson and Buneman, 1987; Dearle *et al.*, 1989; Cattell, 1990; Atkinson, 1992a; Batty, 1992; Cooper, 1993; Halokunnas, 1994]: -

- An OODBPL is generally formed by extending an existing object-oriented programming language, usually C++, through the provision of database capabilities, whereas a PPL is usually developed as a completely new system and is independent of any existing programming language.

- An OODBPL is based on the object-oriented data model and places its emphasis on the data management of objects, whereas a PPL centres on a consistent treatment of the data used both in programming and database management so that it works well with any kind of data model.

- In an OODBPL, procedures are treated as properties of data. By contrast, in a PPL, the program and the data have equal rights - procedures are independent and operate on data, but they can also perform data manipulation for object-oriented management. Moreover, procedures are first class values in a PPL and this promises more flexibility for the behavioural aspects of an application.

- An OODBPL may need to deal explicitly with the movement of data between programs and a database, but this activation is not necessary in a PPL.

- In a PPL, the persistence of data is independent of data types, whereas in an OODBPL, it may be dependent on certain data types.

- An OODBPL normally provides a structured query language (SQL), object management toolkits, and interfaces to other programming languages, whereas a PPL often lacks these features.

- PPLs tend not to support inheritance since no satisfactory match has been found between this property and the features listed above (*i.e.* SQL, toolkits, *etc.*) in terms of the checking of types.

Perhaps the most important difference, however, is that OODBPLs have been developed into commercial products. Therefore, as discussed in Section 2.8.2, much of the current research into GISs has concentrated on the OODBPL approach. However, a PPL can provide a more flexible environment than an OODBPL. In principle, this elegant feature should provide an ideal environment for the integration of geographical data. Therefore, research exploring the capabilities of PPLs for the development of a GIS, particularly an IGIS, should be encouraged. Although PPLs are still experimental and are not yet available as commercial products, the firmly integrated programming/database feature of PPLs has drawn the author's attention to carry out an exploration into the suitability of the persistent programming language Napier88 as an IGIS development tool.

In this chapter, firstly the essence of PPLs is described with a discussion of their implications for geographical data handling. This is followed by an overview of the Napier88 system, with an emphasis on the language aspects, including the basic design principles, its main characteristics, the type system and the persistent store environment. Thereafter, the most important facilities of Napier88 that are relevant to the design of IGISs and their databases are described.

3.2 Persistent Programming Languages

Persistent programming languages are defined as those languages which allow any of their values to have lives of any duration [Atkinson and Morrison, 1990]. The concept of persistent language systems was introduced by Atkinson [1978] as a means of simplifying the task of programming. The initial motivation for building such systems arose from the difficulties of storing and retrieving data structures in CAD/CAM research [Atkinson, 1978]. The first data type complete PPL, PS-algol, was successfully implemented in 1980 after some difficulties were experienced during attempts to integrate persistence with the well known Algol 68 and Pascal languages. Since then, considerable research has been devoted to the investigation of the concept of persistence and its implementation. Consequently, a number of other persistent programming systems have been developed including Galileo, Amber, Poly, Napier88, *etc.* [Atkinson and Morrison, 1990; Morrison *et al.*, 1993a]. Currently, work to enhance existing functions and to develop new features (or both) is continuing. Several persistent systems including Napier88 already have quite a lot of functions available for practical applications.

Persistent programming systems can be regarded as the appropriate underlying technology for the construction and maintenance of large, long-lived, object-based application systems. Morrison *et al.* [1993a] summarised the advantages of using persistent programming systems as follows: -

- *improving programming productivity as a result of simpler semantics;*
- *removing ad hoc arrangements for data translation and long term data storage; and*
- *providing protection mechanisms over the whole environment.*

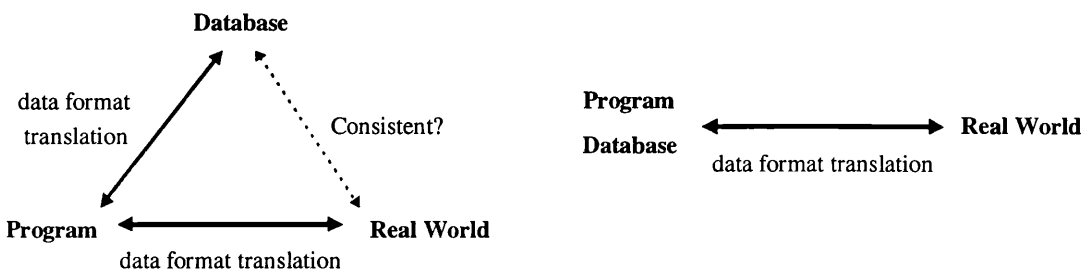
All these benefits come from the underlying principle of orthogonal persistence by which a persistent programming system is supported. The concept of orthogonal persistence; the principles used for the provision of persistence; and the persistent type system form the essential elements of PPLs. Therefore, each of them is described in turn in the following subsections.

3.2.1 The Concept of Orthogonal Persistence

The persistence of data is the length of time for which data exists and is usable. In traditional computer systems, the persistence of data can be described by the following six categories [Atkinson *et al.* 1983; Brown, 1989]:

1) data that only exists within the evaluation of an expression,
2) data that is local to the activation of a procedure,
3) data that is global to a program or outlives the procedure that created it,
4) data that exists after the execution of a program,
5) data that exists between versions of a program, and
6) data that outlives the program that created it.

There is an explicit distinction between *short-term data* (categories 1 to 3) and *long-term data* (categories 4 to 6) in terms of data persistence. Short-term (transient) data is only valid inside a program, and is lost once a program terminates. Long-term (persistent) data is stored outside the program context. The persistence of short-term data is usually supported by a programming language, whereas the persistence of long-term data is supported by a DBMS or a file system. In such systems, the data format translation must be included in any operation, in the course of which, data needs to be transferred between the programming and database domains. Not only will many resources have to be consumed on data format translation, but it is also very difficult for programmers to keep three mappings consistent between the three domains: the real world, the program representation and the database representation. In particular, the persistence of the modelled data from the real world into a database needs to pass through two steps: (i) the modelled data first maps to appropriate data types provided by a program; and then (ii) these data types in the program domain again map to the corresponding data types used in a database. Thus, a programmer has to ensure that the persistent data in the database domain is always kept logically consistent with the real world domain. Fig. 3.1(a) illustrates the concept of mappings between different domains in a traditional programming language.



(a) Traditional programming languages *(b) Persistent programming languages*

Figure 3.1 The concept of mappings between different domains in (a) traditional and (b) persistent programming languages

The required consistency becomes almost unachievable since the actual data and programs complicate the mappings. As a result, data represented in the three domains are very difficult to maintain in a harmonious state in the long run.

A persistent programming language eliminates the distinction between short-term and long-term data persistence by integrating a programming language and the database into a single system. In a persistent programming system, persistence is an orthogonal property of data. That is, the manner in which data is manipulated is independent of its persistence. The same mechanisms operate on both short-term and long-term data, avoiding the traditional need for separate systems to control access to data of differing degrees of longevity [Atkinson, 1992a; Kirby 1992]. Hence, a persistent programming system only requires a single mapping from the modelled system of the real world to programs since there is no data translation required between the program and the database domains [Atkinson and Buneman, 1987]. Fig. 3.1(b) depicts the concept of mappings between different domains in a persistent programming language.

3.2.2 The Principles for the Provision of Persistence

According to the concept outlined above, a persistent programming system is devised and constructed with the integration of a programming language and database facilities to provide a consistent treatment of the data used in both the programs and the database. There are two essential principles which guide the provision of persistence in the design and architecture of persistent programming languages [Atkinson *et al.*, 1983; Atkinson and Buneman 1987; Atkinson and Morrison, 1990; Atkinson, 1992a]:

The Principle of Persistence Independence

The persistence of a data object is independent of how the program manipulates that data object, and conversely, a fragment of program is expressed independently of the persistence of the data it manipulates.

The Principle of Persistent Data Type Orthogonality

In line with the principle of data type completeness, all data objects, whatever their type, should be allowed the full range of persistence.

Based on these principles, the use of all data in a persistent programming system is independent of its persistent properties, including where it is kept, how long it is kept, and in what form it is kept. In other words, a persistent programming system removes the need to explicitly distinguish between the use of short-term and long-term data. Moreover, a persistent programming system supports data type completeness. This means that any data

type with rich data structures can be constructed, managed and stored in an integrated programming/database environment.

3.2.3 The Persistent Type System

In persistent language systems, type systems are languages for describing the data types which programs in the language can manipulate and provide the mechanisms whereby sentences in those languages can be interpreted. Types are denoted by expressions in the type algebra provided by the type system. Variables have types which denote the subset of values they may hold. Each value has a type [Atkinson, 1992a]. Some system types are built with a regular data structure *e.g.* an array, a list and a relational table. A new data type can be constructed by using predefined type constructors. Hence, a user is able to create new data types with appropriate data structures for a specific application quite easily. Conceptually, the set of data types in a PPL program is roughly equivalent to a schema in the context of DBMSs. Similarly, a type system in a PPL represents a data model in a DBMS. The vocabulary equivalencies between PPLs and DBMSs are shown in Table 3.1.

Persistent Programming Languages	**Database Management Systems**
Type System	Data Model
Data Type	Schema Element
Variable	Database
Value	Instantaneous DB Extent

Table 3.1: The vocabulary equivalencies between PPLs and DBMSs [Atkinson, 1992a]

There are two separate but interacting aspects present in type systems: these are the *control of complexity* and the *protection of data.* The *control of complexity* is the ability to structure data in a regular form, whereas the *protection of data* is the ability to protect data from accidental or deliberate misuse [Morrison *et al.*, 1990]. The complexity of a type system is determined by the number of its defining rules. These type rules should be able to be applied consistently throughout the design of the underlying system. On the other hand, it is necessary to have data protection and recovery mechanisms to prevent data misuse and loss from hardware and software failure. Generally speaking, the greater the enforcement on the protection aspects of types, the less flexibility there will be in the modelling capabilities of application systems. For example, most language systems employ totally static checking for types in order to become more secure in type correctness and more efficient in program execution. These are known as *strongly typed systems.* However, such language systems cannot dynamically accommodate any change in types. As a result, the cost of totally static checking for types is that any alteration to type involves the total re-compilation of the whole system. On the contrary, some language systems perform totally

dynamic checking for types to provide flexibility in controlling complexity at the run-time. These are known as *weakly-typed systems*. The drawback of totally dynamic checking for types is that it results the system being intrinsically less safe and less efficient [Morrison *et al.*, 1987]. Therefore, the design of type systems needs to have a balance between flexibility and safety.

3.2.4 Implications for Geographical Data Handling

Geographical data handling deals with complex real world objects which usually comprise or are represented by a mixture of spatial, attribute, and temporal components. Geographical objects (features) first map to a conceptual data model through some data generalisation and abstraction process. This mapping is known as *conceptual modelling* and involves identifying the object's pertinent characteristics, and constructing object relationships, *etc.* The design of an appropriate conceptual data model depends on different geographical data representations, and the intended or required applications. This conceptual organisation is then translated into programs and a database representation which is often called *logical modelling*. This is concerned with the organisation of geographical data in programs and databases, and in particular with data structures. However, data persistence in the course of logical modelling is quite different depending on whether between a traditional database management system or a persistent programming system is being considered.

For example, Fig. 3.2 illustrates the modelling processes and the differences in data persistence for rivers between a traditional system and a persistent system. The locations of rivers in the real world are first captured and stored in digital form and are then conceptualised as a spaghetti model using the entity-relationship approach. In this instance, a river is represented as an entity which is composed of (*i.e.* has a relationship with) polylines (*i.e.* entities); a polyline consists of a series of points; and a point has three spatial components - its X, Y, Z coordinates. It should be noted that the same data model may be implemented in the form of different data structures. For instance, the spaghetti data model representing the rivers may be implemented as a *List*, as an *Array*, and so on. In the example shown in Fig. 3.2, both the traditional and the persistent systems use *List* structures in their program representation. However, when the programs store data in the databases, the data retains the same *List* structure in a persistent system, while it may have to be translated into other data structures in a traditional DBMS. In this example, it can be seen that the format translation required in a traditional system results in the problem of impedance mismatch. In practice, the semantic gap problem may also appear quite readily since a geographical database normally consists of many types of features and they may be represented by several data models for different applications. By contrast, in a persistent

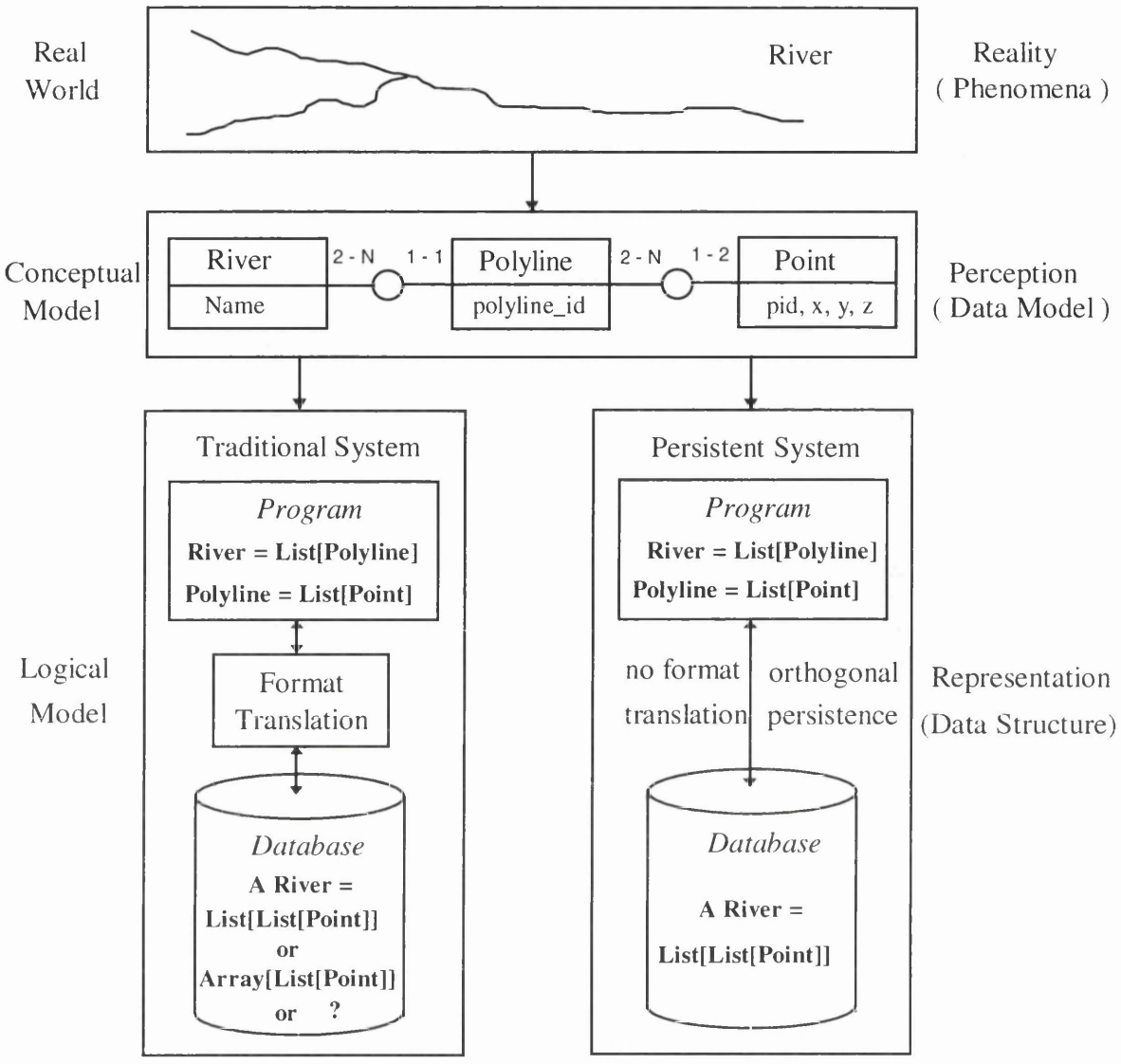


Fig. 3.2 The differences of data persistence between a traditional and a persistent system

system not only is data translation for persistence unnecessary, but furthermore any data can be seen as having the same representation of a data format or data structure which can be used in both programs and databases.

In summary, a persistent programming system removes the inconsistencies of data representation in the three domains - the real world, programs, and databases - through the mechanism of orthogonal persistence. So it can provide the features of persistence, type completeness and expressive power in a single integrated programming/database environment. This unique feature is especially useful for the organisation and maintenance of the integrated database demanded by an IGIS.

3.3 *Napier88 Overview*

This section gives a short summary of Napier88. A complete functional description of Napier88 can be found in Morrison *et al.* [1989; 1993b], Connor *et al.* [1991] and Kirby *et al.* [1994], while the implementation of bulk type aspects is given in Atkinson *et al.* [1993a]. It should be noted that Napier88 is still a research language. Thus all the manuals are rather terse in explaining language features and in describing system functions. In particular, Napier88 lacks a proper user's manual that would contain detailed information on each function and illustration of this by examples. Therefore, any average application programmer deciding to adopt persistent system technology should not underestimate the effort required to understand fully the tools described here. At present an alternative source for learning the basic concepts and features of Napier88 can be found in Atkinson [1992b].

Napier88 and its predecessor PS-algol are persistent programming languages developed at the University of St. Andrews in collaboration with the University of Glasgow. In addition, the University of Adelaide in Australia has also participated actively in the system development. The design of the Napier system began in 1985 [Atkinson and Morrison, 1990], and it was named after the famous mathematician who invented the principle of logarithms [Morrison *et al.*, 1989]. As the name suggests, Napier88 (Release 1.0) was introduced in 1988. It was originally designed as a testbed for experiments concerned with type systems, programming environments, concurrency, bulk data, object stores and persistence. As a consequence of its status as a research language, the Napier88 system is in a state of continual evolution. Recently it has been upgraded with a number of new features in Release 2.0 [Morrison *et al.*, 1993b] as well as bulk type libraries that can be used in both Releases [Atkinson *et al.*, 1993a].

3.3.1 *Language Design Principles*

The Napier88 system follows in the algol tradition as did its predecessors S-algol and PS-algol. These programming languages were designed based on three principles originally outlined by Strachey [Strachey, 1967]. These can be stated as follows [Atkinson *et al.*, 1984; Dearle *et al.*, 1989; Morrison *et al.* , 1989; 1993b]: -

The Principle of Data Type Completeness

All the data must have the same "civil rights" and the rules for using the data types must be complete, with no gaps.

The Principle of Abstraction

The process of extracting the general structure to allow the non-essential details to be ignored.

The Principle of Correspondence

The rules for introducing and using names should be the same everywhere in a program.

This language design makes for the Napier system having a small number of defining rules allowing no exceptions and leads to a less complex yet very powerful PPL. *The principle of data type completeness* means that Napier88 is a very rich type system and ensures that all data types may be used in any combination in the language. All data types have the same rights to be declared, to be assigned to and to be assigned, to have equality defined over them, and to persist. *The principle of abstraction* is invoked by identifying the semantically meaningful syntactic categories and providing abstractions over them. This mechanism allows the control of complexity with a high degree of abstraction. For example, procedures are regarded as abstractions over expressions and statements; polymorphism as an abstraction over type; abstract data types as abstractions over declarations; and vectors and structures are regarded as abstractions over all data types in the store. The abstraction mechanism is particularly important for software reuse because the code for an abstraction is only written once, while that written for a particular specialisation may be used many times [Morrison *et al.*, 1987]. *The principle of correspondence* makes Napier88 programs easy to read, understand and remember using a consistent naming scheme for declaring names in program blocks, in the parameters of procedures, in the fields of records, and so on. Based on these three principles, in addition to the provision of orthogonal persistence, the Napier88 system is able to provide powerful features for the construction and maintenance of large and long-lived application systems.

3.3.2 Language Characteristics

The Napier88 system consists of the programming language and its persistent environment located in a persistent store. Initially, the persistent store, also called the stable store, is populated by standard libraries and the system uses objects within the persistent store to support itself. Napier88 then allows the extension of the persistent store using the new values created by user programs. In the persistent store, type systems are used to represent the data models of both data and programs. In effect, a program fragment is treated as just another data value. The model of persistence in Napier88 is that the system automatically determines the persistence of objects by its reachability from a root object. In other words, when a program terminates, all its data objects may be destroyed except those that the type program has arranged to be reachable from the root object. The persistent store is also stable, that is, it is transformed from one consistent state to the next. Therefore, the persistence of a data object from a program into a persistent store is performed by the user by ensuring that the data object is reachable from the root object of the store. However, if the data object is fetched from the store, then when a program terminates, a stabilisation

operation is automatically performed and so the data object still persists [Morrison *et al.*, 1993b].

The general characteristics of the Napier88 persistent programming language can be summarised as follows [Dearle *et al.*, 1989; Atkinson *et al.*, 1992b; Morrison *et al.*, 1993b]:

-

* **Block structured**: The visibility of an identifier is determined by its lexical scope. This means that the scope of an identifier starts immediately after the declaration and continues up to the next unmatched **end**.
* **Procedural language**: The order of evaluation is strictly from left to right and top to bottom except where the flow of control is altered by one of the language clauses. Parentheses in expressions can be used to override the precedence of operators.
* **Strongly typed**: The system is mostly statically checked for type correctness except for two types (**env** and **any**) which must be dynamically checked.
* **Initialising declarations**: Type and identifier declarations are allowed to occur anywhere in sequences of statements.
* **Type inference**: The types of declared identifiers are inferred from the initialising expression, but the types of procedure parameters and results are not inferred and these must be explicitly stated by the programmer.
* **Type completeness**: There are no restrictions on constructing types.
* **Orthogonal persistence**: Models of data are independent of longevity. The lifetimes of data are determined by the duration of their reachability from a distinguished root, *PS()*.
* **A type secure stable store**: The system does not allow the types of objects to be arbitrarily changed, *i.e.* it does not feature *type coercion*. The system also has a mechanism for making locations constant in the store. Type checking is always performed for the type security of the store over a spectrum of times under user control.
* **Reflection**: The compiler is callable at run-time and this can be used to construct programs which extend themselves at run-time to deal with novel data types as they are encountered [Cooper and Kirby, 1994].
* **Automated support of persistence**: Data is moved automatically from the persistent stable store to the active space as it is needed and is returned automatically if it has been updated at the end of the program or whenever a stabilisation operation is applied.
* **Higher-order procedures**: Procedures are data objects that can be passed as arguments to other procedures; be returned as the results of procedures; be components in data structures; and be assigned to variables.
* **Parametric polymorphism** allows the sharing of code among data types that have a common structure, *i.e.* generic type procedures may be quantified by any number of types rather than a different one having to be written for each type.
* **Abstract (existential) data types**: The data object can be manipulated without anyone being able to discover its implementation or representation. The information hiding

feature of abstract data typing means that objects have a "public" interface, but the representation and the implementation of these interfaces are "private" (*i.e.* hidden form the user).

* **Graphical data types** are provided for both line drawings and raster images.
* **Type equivalence** is based on structural equivalence rules, *i.e.* any aliases, recursion variables, and operator applications are fully factored out before equivalence is accessed.
* **Localised dynamic binding**: A collection of bindings which denotes an environment (type **env**) can be added to and removed from another environment. Values from environments can be used in programs through a dynamically bound and dynamically type checked mechanism.
* **Dynamic binding and deferred type checking**: The type **any** denotes any type and provides a holder for any value, including those for values not yet defined. This mechanism permits hidden types, and provides the capacity for handling future types and incremental type checking.
* **Incremental loading mechanisms**: Persistent procedures constitute all the Napier88 libraries and are the basis for separate compilation. The incremental loading mechanisms perform the linking and loading of these libraries and separate compilation units. At the same time, the linking and loading mechanisms check that all the types match correctly.

It should be noted that several of these characteristics are regarded as novel or unique features in terms of programming language design, notably the provision of orthogonal persistence, automated support of persistence, graphic data types, high-order procedures, and incremental loading mechanisms. Such features will certainly have some impacts on IGIS development, so they will be described further in the next section. All the above characteristics apply to both Releases 1.0 and 2.0. Essentially, Napier88 Release 2.0 has only two changes to the language: (i) a dynamic abstract witness model for abstract types and (ii) the availability of type operators, but the persistent store environment (which will be described in Section 3.3.4) has also been significantly enriched and reorganised [Morrison *et al.*, 1993b].

The Napier88 system is designed as a layered architecture consisting of a compiler, the Persistent Abstract Machine (PAM) and the persistent storage architecture. All the structural layers are virtual allowing implementation on any platform [Morrison *et al.*, 1993b]. The present system is implemented by a compiler written in Napier88, and an interpreter of the PAM written in C. The interpreter runs under Unix operating systems on top of a stable store [Atkinson, 1992b]. At present, Napier88 only runs on two kinds of platform: the Sun SPARC and DEC Alpha graphical workstations. However, porting to other platforms has been planned [Kirby *et al.*, 1994]. Detailed information about the Napier88 system and its relevant publications can currently be obtained through the Internet

from the World Wide Web (WWW) server (http://www-fide.dcs.st-andrews.ac.uk) of the Persistent Programming Research Group at the University of St. Andrews.

3.3.3 The Napier88 Type System

As has been described above (See Table 3.1), a type system in persistent programming languages is equivalent to a data model. The Napier88 type system is based on the notion of types as a set structure imposed over the value space [Morrison *et al.*, 1993b]. The Napier88 language provides a set of type rules, including predefined base types and type constructors. Because the type constructors obey the *Principle of Data Type Completeness*, which has been described in Section 3.3.1, therefore an infinite set of data types associated with any complexity of structures and any degree of abstraction can be constructed. This mechanism ensures that the Napier88 type system employs only a small number (12) of type rules, yet is very powerful in controlling the modelling aspects of application systems. The Napier88 system consists of three kinds of system types: base (or elementary) types, compound (or constructed) types, and bulk types. These system types are the fundamental elements for building application systems. The base types are built-in and cannot be further decomposed. The base type can be classified into either the elementary store types or the scalar types according to whether or not they can behave as stores. The compound types are constructed by supplying types as parameters to type constructors. The bulk types are provided for the description of regular data structures through type construction defining instances that are arbitrarily large collections of elements [Atkinson *et al.*, 1991]. These system types are summarised in Table 3.2 [Atkinson *et al.*, 1992b].

Type Categories		**System Types**
Base Types	Scalar Types	integer (**int**), real (**real**), boolean (**bool**), string (**string**), picture (**pic**), pixel (**pixel**), null (**null**).
	Elementary Store Types	file (**file**), environment (**env**), any (**any**).
Compound Types		vector (**vector**), structure (**structure**), image (**image**), procedures (**proc**), variant (**variant**), ADT (**abstype**), polymorphic procedures.
Bulk Types		Lists (**List**), Maps(**Map**), Sets, Strings, Vectors.

Table 3.2 : System types provided by Napier88

On the other hand, Napier88 adopts a mixture of static and dynamic type checking mechanisms. The **env** and **any** types are dynamically checked, but all other type checking is static. This arrangement allows the user to perform late type checking in order to accommodate unanticipated type changes during run-time. The Napier88 type system employs so-called eager type checking where types will be checked as early as possible in the life cycle [Morrison *et al.*, 1990]. In other words, static data types are checked for their correctness during the compilation time except that the checks on dynamic types are delayed since they are required to be flexible for type changes during the run-time. The sensible design of the type checking mechanism ensures that the Napier88 system has the facilities of type safety and type flexibility, as well as efficiency in program execution.

3.3.4 Persistent Store Environment

In Napier88, all system objects, user programs and data are kept in a persistent store. The persistent store may contain unlimited numbers of procedures and default variables available for user applications. All the information in the persistent store is organised in a hierarchical (tree-like) structure. Each node in the tree represents an environment. The concept of an environment is similar to that of a block in block-structured programming languages or a directory in Unix. The environment holds the bindings of the currently visible identifiers, where a binding is a set of quadruples describing the attributes of identifiers. Each quadruple comprises an identifier, a type, a value and a variable/constant location indicator. An environment can be bound into other environments or removed from its bound environment [Atkinson, 1992b]. The root of the tree represents the whole persistent environment, *i.e.* the persistent store. The root environment, which is yielded by calling the predefined procedure *PS( )*, gives access to various libraries of useful procedures and other values. The leaves of the tree represent persistent data which may be in the form of any data type which is supported by the system or has been defined by users. The environment structure of the persistent store is conceptualised as shown in Fig. 3.3.

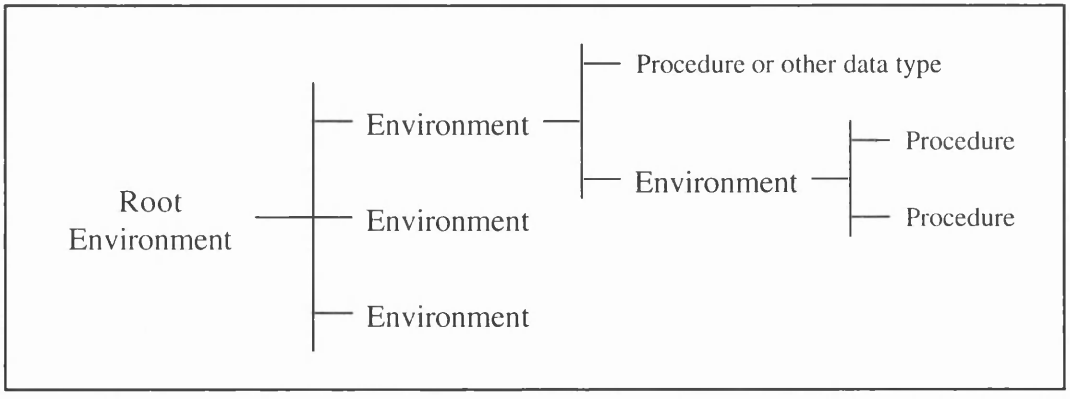


Figure 3.3 Environment structure of the persistent store

Basically, the persistent store is a single large file located within the operating system. The contents of the persistent store are invisible to users and are not accessible by other systems. However, Napier88 provides a store browser for users to view the store contents. Any data object in the persistent store can be accessed from Napier88 programs by making the environment holding that data object reachable from the root environment. Similarly, any data object in Napier88 programs will automatically persist in the persistent store after the programs terminate provided that the data object has been bound to an environment as well as one which is reachable from the root environment.

The creation of an initial store can be done by copying an existing store which has been populated with standard libraries or by setting up an empty store and then running a number of Napier88 programs for the installation of appropriate libraries. The initial population of objects in the persistent store forms an important part of the programmer's facilities because all of these objects will be heavily used by Napier88 application programs. The persistent store is intended to provide system programmers, application developers, and end-users with the means to construct reusable objects including data and software programs. In principle, the contents of the persistent store will remain reusable and extensible with the evolution of application software, even if the application programs which originally created them will no longer be used. Eventually, the persistent store will contain large bodies of long-lived data and program modules, yet must still have the capability to support unanticipated applications which will arise in the future. Therefore, the persistent store created by the Napier88 Release 1.0 system could be used directly in Release 2.0 as well as any future releases without any change or reconstruction.

However, there are some major changes of the persistent store environment in Napier88 Release 2.0. These include concurrent execution and data access; reflective programming for system evolution; a new organisation of the initial stable store; and considerable enhancement of the utility procedures available in the standard library [Morrison *et al.*, 1993b; Kirby *et al.*, 1994]. As a result, the persistent stores used in the two releases (1.0 and 2.0) are incompatible, mainly because the standard library environment is structured differently between Release 1.0 and 2.0. The incompatibility between the two releases is probably a cost that simply has to be paid for to ensure a better persistent system. Nevertheless, it will be regarded as a major drawback with regard to the use of Napier88 in this particular IGIS research. Another major difference is that the compilation of Napier88 programs in a batch mode is quite different between both releases. Release 2.0 uses a compilation module which is written in Napier88 and has been integrated into the persistent store, whereas Release 1.0 is based on an independent compilation system which is written in PS-algol [Brown, 1989; Kirby *et al.*, 1994]. Although there are several other differences between the two releases, the standard library contains the following primary facilities that generally speaking apply to both releases [Connor *et al.*, 1991; Kirby *et al.*, 1994]:

- A set of procedures supports basic programming activities including arithmetical operations; I/O control and operations; the control of graphical devices; vector graphic display; raster image display and operations; mouse and keyboard event control, *etc.*

- An integrated programming environment supports the interactive development of Napier88 programs. It allows the user to compose and execute programs and to examine their effects on the persistent store.

- A window management system (WIN) provides graphic user interface (GUI) facilities for user interface programming. It allows the creation and manipulation of user interfaces employing windows, menus, icons, dialogue boxes, a mouse and keyboard event handling.

- A store browser can display a representation of any Napier88 value in either a graphical or text mode.

- A set of procedures which forms the Napier88 compiler for compiling Napier88 programs in an interactive mode. This facility supports hyper-programming which allows the Napier88 programs to contain embedded direct references to values, locations and types in the persistent store.

In addition to the standard library, a family of bulk type libraries developed at the University of Glasgow can also be installed in the persistent store. These bulk type libraries provide data types with built-in data structures and support a set of operations which can be applied to them. Using bulk types libraries, data objects with regular data structures can be easily represented by predefined bulk types. That is, any data which may carry a data model can be constructed as a bulk type and denoted as a single object stored in the persistent store. Those bulk type libraries which currently have been implemented or are in the process of being implemented are *Lists, Maps, Sets, Strings, Vectors, Conversions*. Several other bulk type libraries such as *Trees, Graphs, Rings* and *Matrices* will be added in due course [Atkinson *et al.*, 1993a; 1993b].

Fig. 3.4 is a typical example that illustrates an initial persistent store populated with the Napier88 Standard Library Edition 2.2 and the Bulk Libraries Release 1 [Atkinson, *et al.*, 1993a; Kirby *et al.*, 1994] in which the contents display the names of the main environments. Each environment may further contain a set of procedures, values, or environments. For instance, all the procedures in the *Outline* and *Raster* environments, which are used for the manipulation of vector and raster data respectively, are also shown in Fig 3.4.

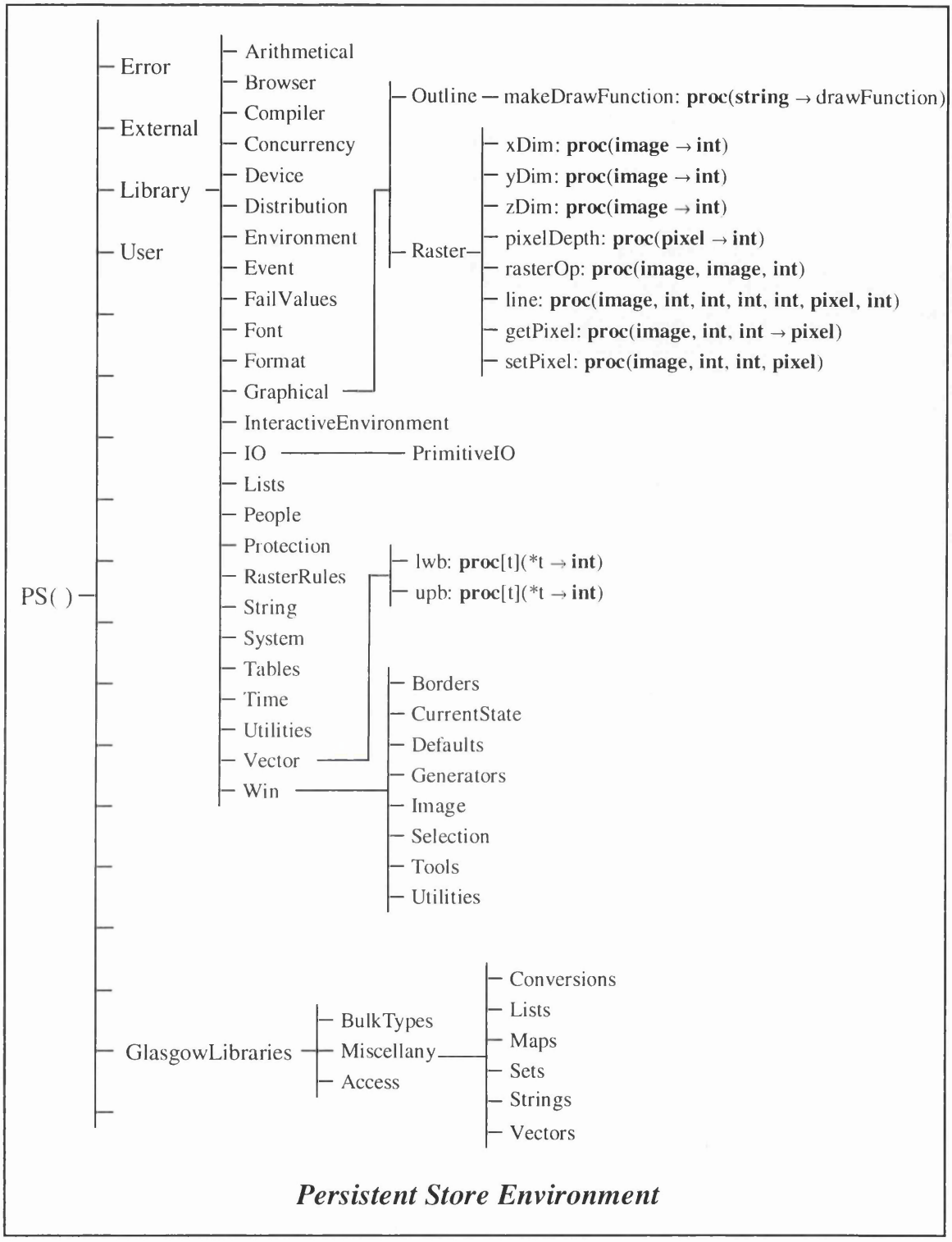


Figure 3.4 A partial overview of the persistent store populated with the Napier88 Standard Library Edition 2.2 and Bulk Libraries Release 1

N.B. 1) The environment *Library* given above in the diagram refers to the Standard Library; the environment *GlasgowLibraries* refers to the Bulk Libraries as discussed in the text.

2) The environment *Vector*, which forms part of *Library*, contains two procedures which determine the high and low bounds of a vector. The environment *Vectors*, which forms part of *GlasgowLibraries*, contains procedures such as sorting a vector, applying a function to each element of a vector, *etc.*

3) The environment *Outline* is in fact dealing with the drawing of vector graphics, as discussed in this thesis.

It should be noted that the *User* environment in the persistent store is reserved for users to store their information. Thus any programs and data that a user wishes to make persistent in the store should be organised and kept in the *User* environment. Hence, in the course of constructing an IGIS database, all the data, including their associated programs, will be incrementally accumulated within this environment.

3.4 Important Facilities for IGIS Development

Having described the general capabilities of the Napier88 system, it can be seen that Napier88 is a database programming language designed originally for the purpose of building persistent application systems in which the construction and maintenance of large, long-lived data objects is the major concern. GIS applications involve the establishment of such systems. A geographical database that contains large volumes of maps, images and attributes has to keep being persistent and consistent over a long period of time. Therefore, Napier88 may be used as a GIS development tool, particularly for an IGIS. Before carrying out this research, some of the important facilities that can be considered to be useful for the development of an IGIS are further summarised as follows: -

- Both the programs and the data are tightly integrated into a single programming/database environment. A persistent store acts as a single database where both the software and the data are stored. This facility lays the foundations of a single working/storage environment for the development of an IGIS.

- The data type completeness and the orthogonal persistence of Napier88 allow the possibility for any kind of geographical features to be constructed as a data type and placed in the persistent store. This facility allows any data type to work within an IGIS without restrictions.

- The bulk type libraries provide programmers with abundant data types for organising geographical data into a variety of regular data structures depending on the specific needs of individual applications. This facility allows specific geographical features, which are particularly complex objects associated with rich semantics, to be stored as compound objects without the need to decompose them into a number of simple components.

- Three fundamental data types (**pixel**, **image** and **pic**) are provided to facilitate graphical capabilities. The **pixel** and **image** types are the essential elements required for handling raster images, while the **pic** type is the basic element available for dealing with vector maps [Morrison *et al.*, 1986]. This feature allows both vector and raster types of

geographical data to be organised and manipulated within the same database environment.

- The abstract data types and the parametric polymorphism provide a powerful abstraction mechanism for adequately describing geographical objects to a high degree of abstraction as well as composing new objects out of existing objects. The abstract mechanism not only allows an implementation of object-oriented data management but also has the advantage of software reuse. This facility enables programmers to create an IGIS which has both object-oriented programs and databases.

- The incremental construction mechanism allows programs and data to be incrementally constructed and enhanced. This facility allows programmers to control the evolution of programs and data in an IGIS.

- The WIN window management system provides utilities for producing a graphical user interface and frees the application programmer from the necessity of writing the source code needed for user interface programming [Cutts *et al.*, 1989]. This facility allows programmers to create a user friendly IGIS using the user interface tool kits.

The utilisation of these facilities in the development of an IGIS forms an important part of this research. The subsections which follow give a brief description, by means of examples, of the syntax of the Napier88 system, emphasising the usage of these facilities in geographical data handling. These examples are based on the persistent store environment of Napier88 Release 1.0. Full descriptions of the language syntax of Napier88 and the user's libraries can be found in "The Napier88 Reference Manual" [Morrison *et al.*, 1989; 1993b], "The Napier88 Standard Library" [Kirby *et al.*, 1994] and "Towards Bulk Types Libraries for Napier88" [Atkinson *et al.*, 1993a].

3.4.1 Running a Napier88 Program

As with any programming languages, a Napier88 program contains the source code which may be created and edited by a text editor. For example, Fig. 3.5 illustrates a Napier88 program which can be used to compute the distance of a straight line joining the two points (x_1, y_1) and (x_2, y_2).

In Fig. 3.5, the words in boldface are reserved in the language. A string of characters which starts with a **!** and terminates by a newline is a comment such as line 1. Line 2 obtains the *Arithmetical* and *IO* environments from the persistent store and binds them to the clauses after the **in**. Line 3 gets a binding, which has the identifier *sqrt* and the type **proc**, from the *Arithmetical* environment and uses it within the scope following the **in**. The procedure *sqrt*

```
! program name: st_line_dis.N                                                    ! 1
use PS() with Arithmetical, IO: env in                                          ! 2
use Arithmetical with sqrt: proc(real → real) in                                ! 3
use IO with readReal:  proc(→ real);                                            ! 4
            writeReal: proc(real);                                              ! 5
            writeString: proc(string) in                                        ! 6
begin                                                                           ! 7
   let distance := 0.                                                           ! 8
   writeString("Input x coordinate of the first point: "); let x1 = readReal( ) ! 9
   writeString("Input y coordinate of the first point: "); let y1 = readReal( ) ! 10
   writeString("Input x coordinate of the second point: "); let x2 = readReal( ) ! 11
   writeString("Input y coordinate of the second point: "); let y2 = readReal( ) ! 12
   distance := sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1))              ! 13
   writeString("The distance = "); writeReal(distance)                          ! 14
end                                                                             ! 15
```

Figure 3.5 A Napier88 program that computes the distance of a straight line

takes one argument **real** and yields an argument of the same type **real**. Similarly, lines 4, 5 and 6 obtain procedures *realReal*, *writeReal* and *writeString* from the *IO* environment. All the bindings are used in the block enclosed by the **begin** (line 7) and the **end** (line 15). Within the block, there is a sequence of clauses and declarations. Line 8 declares an identifier *distance* associating it with a value (0.) by a **let** statement. The assign ":=" operator means that the identifier is a variable which may be updated. The type of the identifier is inferred by the expression to the right of the assign, which in this case is the type **real**. Lines 9 to 12 write a prompt message to the standard output and take a value of real from the standard input. The equal "=" operator means that the identifiers are constant. In line 13, the value of the expression to the right of the assign ":=" is evaluated to update the value of identifier *distance*. Finally, line 14 outputs the result.

The program file *st_line_dis.N* can be compiled with the command line under the Unix operating system.

```
npc st_line_dis.N
```

This will produce a file called *st_line_dis.out* which can be executed with the command line.

```
npr st_line_dis.out
```

It should be noted that, after the execution of the program, all the data is discarded and is therefore not placed in the persistent store because all the identifiers used, *i.e. x1*, *y1*, *x2*, *y2* and *distance*, are not reachable from the root object *PS()*. The way to make data persistent in the persistent store is explained later in Section 3.4.6.

3.4.2 Vector Graphics

The vector graphic facilities allow the user to produce line drawings in an infinite two-dimensional real space. A line drawing is represented by the picture (**pic**) data type in Napier88. In order to draw a vector picture, a Napier88 program needs to include three main parts:

1. Initialisation of a graphical device;
2. Definition of a colour map;
3. Construction and drawing of vector pictures.

1. The initialisation of a graphical device is concerned with the selection of the device with which a picture will be drawn and with the specifications of the initial size and location of the window if an X-terminal is being used. At the time of writing, Napier88 only supports a small number of output devices including X-terminals; Tektronix vector graphic terminals 4010, 4006, 4107; the QMS L800 Laserprinter; the g6320 colour plotter and the cs4800 printer. Among these, X-terminals are the most commonly used display devices. Using X-terminals, an X-window can be obtained by the implementation of the following two statements.

   In the first place, a window file object is created by

   let *wf* = open("WINDOW: XDIM:w, YDIM:h, ZDIM:d, XPOS:x, YPOS:y", m)

   where w, h, d are measured in pixels and are the width, height and depth of the window respectively;
   x, y are measured in pixels and are used to specify the distance of the window from the left or right and top or bottom edges of the screen, respectively; and
   m is the access mode (0 = read only; 1 = write only; 2 = read and write).

   In the second place, the window file *wf* is supplied as a parameter in the procedure *getScreen* as follows:

   let *x_win* = getScreen(*wf*)

   which results in an image, *i.e.* an X-window.

2. The definition of a colour map is to create a colour look-up table for pixels in an X-window. Every X-window opened within the Napier system has an associated colour map. The colour map has 256 entries corresponding to the 256 permutations of eight bits (on/offs). Basically the system is set up for screens of eight bit-planes. Each entry contains a 24 bit number corresponding to 256 blue levels by 256 green levels by 256 red levels. Therefore, a colourmap can be created by choosing any 256 colours from around 16 million. By default, 256 entries are set to the background colour of the X-

window, *i.e.* black (0). A colour map can be defined by repeating the use of the *colourMap* procedure which is used to set one of the entries:

colourMap : **proc**(fd : **file** ; p : **pixel** ; i : **int**)

where fd is the file descriptor (*e.g. wf*) of the X-window whose colour is to be adjusted;
p is the particular pixel whose colour is to be changed in the map; and
i is the new colour. *e.g.* 65,280 = 0 * 256 * 256 + 255 * 256 + 0 would be the strongest possible green (0 Blue 255 Green 0 Red).

A Napier image may be set to contain a certain number of bit-planes. A pixel in the image is defined as the concatenation of the **on** or **off** of the pixel in each bit-plane at the corresponding position. For example,

let *a_pixel* = **off** ++ **on** ++ **off** ++ **off**

means that the pixel has 4 bit-planes and the bit values which are numbered from 0 to 3 are 0, 1, 0, 0. The colour of the pixel may be assigned by mapping the pixel to an integer which represents the combination of the RGB intensities. For example, the assignment of *a_pixel* to green colour can be done by *colourMap*(*wf*, *a_pixel*, 65280).

3. The construction and drawing of vector pictures is regarded as being the manipulation of the 'picture' data types. The simplest picture is a point. For example,

let *point* = [x, y]

represents the point (x, y) in two-dimensional space. There are two binary operators on pictures, namely join '^' and combine '++'. The ^ operator forms a new picture by joining the first picture to the second by a straight line from the last point of the first picture to the first point of the second. The ++ operator also forms a new picture by including all the subpictures of both the operand pictures. For example, a polyline which consists of 3 straight line segments can be represented as follows:

let *polyline* = [x1, y1] ^ [x2, y2] ^ [x3, y3] ^ [x4, y4]

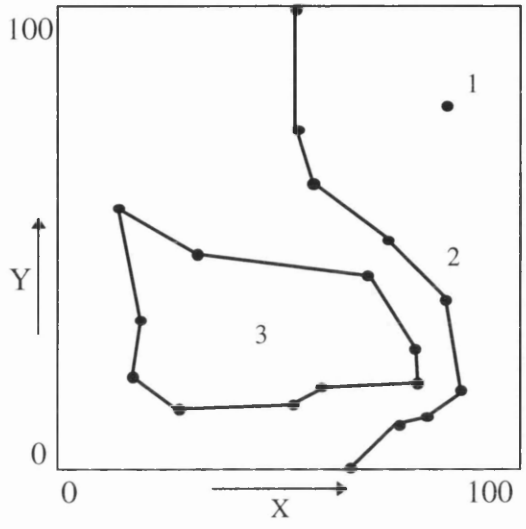
And a new picture may be formed as a "set" of the point and the polyline pictures by the statement

let *point_and_polyline* = *point* ++ *polyline*

The Napier88 system also provides five transformations that can operate on pictures, including **shift**, **scale**, **rotate**, **colour**, and **text**.

Fig. 3.6 is a simple but complete example demonstrating the use of all the vector graphic facilities described above. First of all, a vector data set, which is a table of coordinates, is created by means of a data acquisition method. A Napier88 program is then written to use the data for drawing a vector map in an X-window. It should be noted that this example is rather simplified. In practice, a data set will usually be read into a Napier88 program by

utilising the *IO* facilities rather than be entered directly into the program. In this example, an array, *i.e.* the **vector** type, is used to hold the data.



Feature ID	Coordinates (x, y)
1	(83.7,78.3)
2	(50.9,100.0)(50.9,73.6)(54.5,61.1) (70.6, 49.2)(83.7,36.4)(87.0,16.5) (80.1, 11.6)(73.9, 9.1)(64.2, 0.0)
3	(13.0,56.4)(29.9,45.8)(67.1,41.5) (77.2,25.7)(77.5,18.5)(56.7,17.8) (51.1,13.4)(25.9,12.2)(16.1,20.0) (17.6,32.2)(13.0,56.4)

```
! Program name: vector_graph.N
type drawFunction is variant(imageDraw: proc(image, pic, real, real, real, real);
                             fileDraw: proc(file, pic, real, real, real, real);
                             fail: null)
use PS () with Graphical, Device, IO, System, User: env  in
use Graphical with Outline: env in
use Outline with makeDrawFunction: proc(string → drawFunction) in
use Device with getScreen: proc(file → image);
                  colourMap: proc(file, pixel, int)  in
use IO with PrimitiveIO: env;
              readLine: proc(→ string);
             writeString: proc (string) in
use PrimitiveIO with open: proc(string, int → file);
                       close: proc (file → int) in
use System with abort: proc ( ) in
begin
!  Part1:  initialise a graphical device
   let window_file = open("WINDOW: XDIM:200, YDIM:200, ZDIM:4", 2)
   if window_file = nilfile do { writeString ("Cannot open an X window'n"); abort( ) }
   let screen = getScreen (window_file)
   let draw = makeDrawFunction("image")'imageDraw
!  Part 2:  define a colour map
   let white = on ++ on ++ on ++ on
   let black = off ++ off ++ off ++ off
   let red = on ++ off ++ off ++ off
   let green = off ++ on ++ off ++ off
   let blue = off ++ off ++ on ++ off
   let gray = off ++ off ++ off ++ on
   colourMap(window_file, white, 255 * 256 * 256 + 255 * 256 + 255)
   colourMap(window_file, black, 0)
   colourMap(window_file, red, 255)
   colourMap(window_file, green, 255 * 256)
   colourMap(window_file, blue, 255 * 256 * 256)
```

```
    colourMap(window_file, gray, 223 * 256 * 256 + 223 * 256 + 223)
!   Part 3: construct and draw vector pictures
    let feature := vector 1 to 3 of nilpic
    feature(1) := colour [83.7, 78.3] in blue  ! house
    feature(2) := colour [50.9, 100.0] ^ [50.9, 73.6] ^ [54.5, 61.1] ^ [70.6, 49.2] ^
                         [83.7, 36.4] ^ [87.0, 16.5] ^ [80.1, 11.6] ^ [73.9, 9.1] ^
                         [79.1, 0.0] in red      ! road
    feature(3) := colour [13.0, 56.4] ^ [29.9, 45.8] ^ [67.1, 41.5] ^ [77.2, 25.7] ^
                         [77.5, 18.5] ^ [56.7, 17.8] ^ [51.1, 13.4] ^ [25.9, 12.2] ^
                         [16.1, 20.0] ^ [17.6, 32.2] ^ [13.0, 56.4] in green     ! swamp
    let border = colour [0., 0.] ^ [0., 100.] ^ [100., 100.] ^ [100., 0.] ^ [0., 0.] in black
    let vec_map := border
    for i = 1 to 3 do  vec_map := vec_map ++ feature(i)
    draw(screen, vec_map, -5.0, 105.0, -5.0, 105.0)
    let pause = readLine( )
    let void = close (window_file)
end
```

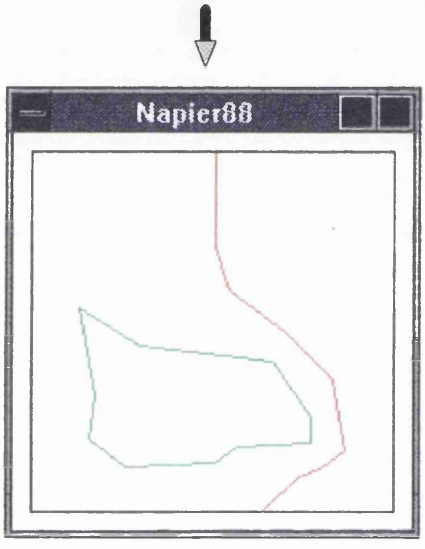


Figure 3.6 An example utilising the vector graphic facilities of Napier88

A **vector** type is very useful for handling geographical data because it provides a method of grouping together objects of the same type. The elements of the vector can be of any type. However all the initial values of the elements must be defined beforehand. There are three forms to allow different ways of providing these values.

1. The first form specifies the lower and upper bound and initialises every element to the same value. For example:

let *feature* := **vector** 1 **to** 3 **of nilpic**

This statement declares a variable *feature* of three elements with initial values equal to a picture with no points (**nilpic**).

2. The second form of vector initialisation supplies the low bound and a list of initial values, for example:

   let *feature* := **vector** @ 1 **of [nilpic, nilpic, nilpic]**

   The length of list determines the upper bound.

3. Finally, the third form uses a procedure to provide the initial values, for example:

   let *feature* := **vector** 1 **to** 3 **using proc**(i: **int** → **pic**); **nilpic**

   This form can also be used to represent multi-dimensional arrays. For instance,

   let *cell* := **vector** 1 **to** m **using proc**(i: **int** → ***int**); **vector** 1 **to** n **of** 0

   produces a two-dimensional array of m by n elements with initial values equal to '0'.

In the program of Fig. 3.6, each feature is constructed as a picture and is also coloured differently. The three features are then combined as a single picture (*i.e. vec_map*) using the "++" operator with the aid of the loop statement (*i.e.*, the **for** clause). Finally, the resultant picture is displayed in an X-window. This is carried out by the *draw* procedure which can be accessed by supplying a parameter "image" in the procedure *makeDrawFunction* and taking a variant *imageDraw* from the parameter returned. The picture is first mapped to a bounding rectangle specified by the four real parameters (xmin, xmax, ymin, xmax), and is then scaled and shifted to fit the X-window. The screen hardcopy shown in Fig 3.5 has been mapped onto an area slightly bigger than the map coverage, so that all the line drawing, including the map border, can be seen.

3.4.3 Raster Graphics

The raster graphic facilities allow the user to create and manipulate images. An image is a rectangular grid of pixels and is represented by the **image** data type. There are two ways of initialising images.

1. The first form specifies the x and y dimensions and initialises every pixel to the same pixel expression. For example,

   let *cell* = **image** 20 **by** 15 **of on ++ on ++ off ++ on**

   create an image *cell* with 20 pixels in the x direction and 15 pixels in the y direction. All pixels have the depth 4 with initial value **on ++ on ++ off ++ on**. The origin of the image is in the lower left corner, which has the address 0, 0.

2. The second form of image initialisation supplies the x and y dimensions and uses an existing image as a background pattern to create an image. For example,

   let *ras_img* := **image** 800 **by** 600 **using** *cell*

will create an image *ras_img* of size 800 x 600 pixels and will then copy the image *cell* onto it as many times as is necessary to fill it in both directions, starting at the address 0, 0.

Aliases to parts of images can be set up by using the *limit* operation, for example:

let *sub_img* = **limit** *ras_img* **to** 300 **by** 200 **at** 400, 300

will set *sub_img* to be that part of *ras_img* which starts at 400, 300 and has size 300 by 200 pixels. This operation does not make a new copy of that part of *ras_img* but merely copies the pointer to it. The *limit* operation is particularly important for performing raster operations, *i.e.* **ror, rand, xor, copy, nand, nor, not** and **xnor**. For example,

copy limit *ras_img* **to** 400 **by** 300 **at** 200, 150 **onto** screen

will write the defined section of the *ras_img* which is the central part and transfers a quarter size of the image onto the screen.

Fig. 3.7 is an example which illustrates the use of the raster graphic facilities. As was the case with vector graphics (see first paragraph of Section 3.4.2), a Napier88 program also requires three main parts to utilise the raster graphic facilities. The initialisation of a graphical device and the definition of a colour map are the same as those described in Section 3.4.2 and Fig. 3.6. This example (in Fig. 3.7) uses the raster data shown in Fig. 2.2 to create a raster image. Essentially it is the raster equivalent of the vector image shown in Fig. 3.6. The size of the image is 12 x 12 cells. Each cell (or source cell) is defined as being equivalent to 15 x 15 screen pixels (target pixel). The background colour of the raster image and the colour of each feature are also defined. This example employs a two-dimensional array to manipulate the raster data which is also put directly into the program After the raster image has been constructed, the resultant image is projected onto the central part of the X-window as shown in Fig. 3.7.

Once again, it should be noted that the raster data used by a Napier88 program generally comes from raster files. Thus in practice, a raster graphic program should provide the capabilities to import raster files and construct them as images. In this instance, a blank image is first created depending on the image dimensions as determined from the file header (or file descriptor) of an image. The image data is then read into the program and converted to pixels. Each pixel is set to its correct position in the image using the procedure *setPixel* provided in the standard library.

```
! Program name: raster_graph.N
! Part 1 and Part 2 are the same as those in the program vector_graph.N
! Part 3: construct and display raster images
  let m = 12; let n = 12    ! cell dimension
  let size = 15             ! cell_size = 15 pixels
```

```
let bg_cell = image size by size of gray
let house_cell = image size by size of blue
let road_cell = image size by size of red
let swamp_cell = image size by size of green
let ras_img := image size * m by size * n using bg_cell
let cell := vector 1 to m using proc(i: int → *int); vector 1 to n of 0
! create a raster image
cell(1, 7) := 2
cell(2, 7) := 2
cell(3, 7) := 2; cell(3, 11) := 1
cell(4, 7) := 2
cell(5, 2) := 3; cell(5, 8) := 2
cell(6, 2) := 3; cell(6, 3) := 3; cell(6, 9) := 2
for j = 2 to 8 do cell(7, j) := 3; cell(7, 10) := 2
for j = 2 to 9 do cell(8, j) := 3; cell(8, 11) := 2
for j = 2 to 9 do cell(9, j) := 3; cell(9, 11) := 2
for j = 2 to 6 do cell(10, j) := 3; cell(10, 11) := 2
cell(11, 9) := 2; cell(11, 10) := 2
cell(12, 8) := 2
! display a raster image
for i = 1 to m do
   for j = 1 to n do
      case cell(i, j) of
      0 : { }
      1 : copy house_cell onto limit ras_img at (j - 1) * size, (n - i) * size
      2 : copy road_cell onto limit ras_img at (j - 1) * size, (n - i) * size
      3 : copy swamp_cell onto limit ras_img at (j - 1) * size, (n - i) * size
default : { }
copy image 200 by 200 of white onto screen  ! set white background for the X-window
copy ras_img onto limit screen at 10, 10
```

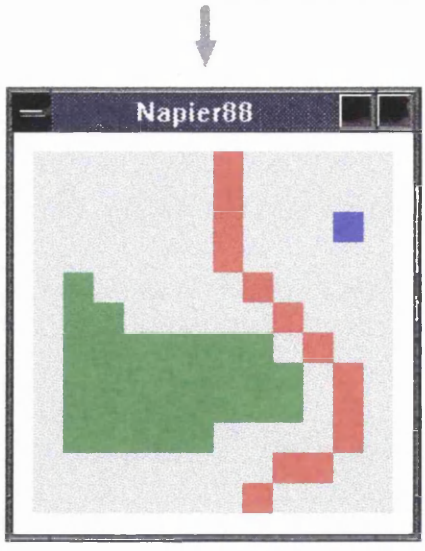


Figure 3.7 An example utilising the raster graphic facilities of Napier88. The map depicted is basically that previously shown in its vector form in Fig. 3.6

3.4.4 Bulk Type Libraries

The bulk type libraries support two essential activities. Firstly, they allow regularities in structure to be described. Secondly, they support powerful and succinct notations for the execution of computing with such regular structures [Atkinson *et al.*, 1993b]. Typically, a bulk type describes a value of arbitrary size (*i.e.* a bulk value) by repeating an element of some other type as many times as is necessary. The bulk type libraries are designed to support several families of bulk types. A family is a collection of bulk types that have a similar set of operations with a similar algebra that relates these operations to one another. At the time of writing, only a family which has set and sequence-like properties is available; other families such as graph-like bulk types and totally unordered types are still under development [Atkinson *et al.*, 1993a]. The set and sequence-like family consists of a number of members including *Lists*, *Maps*, *Sets*, *Strings*, *Vectors* and *Conversions*. Both *Lists* and *Maps* have been used intensively in this research project, so they will be discussed separately and in more detail in the following two subsections. As for the other bulk types, the *Sets* library is used to deal with ordered sets using a B-tree representation; the *Strings* library is provided for the handling of a sequence of characters of an arbitrary length; the *Vectors* library extends the facilities for the manipulation of the standard vector type; and the *Conversions* library is provided for conversions between the other five bulk types.

3.4.4.1 The Lists Library

The *Lists* library contains a collection of procedures for the construction and manipulation of a sequence of homogeneous elements which are in the form of the *list* data structure. The *list* type is essential for the vector representation of linear features such as roads, rivers, coastlines, *etc.* For example, a road may be represented as a list of points, where each point has a structure of x and y coordinates. In fact, point and polygonal features can also be represented in the *list* form. A point feature may be represented as a *list* type of a single element, whereas a polygon feature can be regarded as a list of line components which close back on the starting point. Thus all vector-based geographical features can use a single consistent type *list* for their data representation. The bulk type *List* used in the *Lists* library is defined by

```
rec type List[T] is variant(full: Cell[T]; empty: null)
&    Cell[Q] is structure(hd: Q; tl: List[Q])
```

This is a parametric recursive type which can be of any type but which requires that all the elements in a list should be of the same type. The identifier *T* enclosed in square brackets signifies that it is a parametric type. The parametric type, which generally represents different types, may be specialised for use, such as *List*[**int**], *List*[**string**], *List*[*List*[**real**]],

etc. This mechanism of the parametric type is called parametric polymorphism or polymorphism for short [Morrison *et al.*, 1987]. Polymorphic procedures which use parametric types are extremely useful because the same Napier code can be re-used on data with the same structure but containing different types [Atkinson, 1992b]. The *Lists* library provides about 150 *list* processing functions which are sufficient for any kind of operations being carried out on data of the *List* type. For example, in order to make a list of coordinate pairs, an empty list can be created as follows:

```
type XY is structure(x, y: real)
let xy_list := l_make[XY]( )
```

The first statement declares a structure type *XY*, with two fields *x* and *y* of type **real**. Note that the **structure** type is a compound type constructor which can also be used to group different types together. The second statement uses the procedure *l_make* to make a new (empty) list *x_list*. The *xy_list* may then be appended one element at a time, either at the head or the tail of the list by the procedure *l_prepend* or *l_append*. For example,

```
for i = 1 to 9 do
begin
    let x = readReal( )
    let y = readReal( )
    let xy = XY(x, y)
    xy_list := l_prepend[XY](xy, xy_list)
end
```

The above code fragment reads the coordinate pairs repeatedly 9 times from the standard input and constructs the list *xy_list* for the line feature 2 given in Fig. 3.6. After a list has been built, plenty of operations may be applied to the list. For example, a new location of the feature 2 may be produced by shifting the amount of sx in x-direction and sy in y-direction. This can easily be implemented utilising the following statements.

```
let shift_location = proc(pos: XY → XY)
begin
    let new_pos = XY(pos(x) + sx, pos(y) + sy)
    new_pos
end
let new_xy_list := l_map[XY, XY](xy_list, shift_location)
```

The procedure *l_map* applies the function *shift_location* to every element of the list *xy_list* and results in the new list *new_xy_list*. In this case, the procedure *shift_location* is passed as an argument to the procedure *l_map*. Also the procedure *makeDrawFunction* used in the program *vector_graph.N* (see Fig. 3.6) can return a procedure for the drawing of the feature. This characteristic of regarding procedures as data objects in Napier88 is called high-order procedures.

It is possible to organise a map as a list of geographical features. This arrangement may be suitable for map production in which data is usually organised sequentially and is operated on in a batch mode. However, it is inappropriate for querying geographical features due to the needs for a fast response time and for the processing to be carried out in an interactive mode. Because the only access to a list is from a pointer to the first element (head) of the list, so the traversal of a list cannot be avoided in the course of accessing an element. Hence, the list type may be well suited for the representation of individual features, but the performance is often poor when a large number of elements are linked together into a long list. In order to aggregate many geographical features into a bulky yet manageable unit, the *Maps* library can be used to achieve this purpose.

3.4.4.2 The Maps Library

This particular Library has nothing to do with maps in the cartographic or GIS sense or use of the word. In the specific context of Napier88, the term *Maps* (more properly finite mappings) is used to describe a generalisation of sets, relations, and sparse arrays. A *map* is the stored and updateable representation of a partial function, and may be considered as a set of pairs of tuples, denoting the domain and range respectively. A *map* will trivially represent sets, arrays, sparse arrays, index structures, and relations [Atkinson *et al.*, 1991; 1993b]. The *Maps* library provides a representation of the mappings required between any two Napier88 types. The full implementation of the *Maps* library with all of its features has not yet completed. However, a particular form of the *map* type has been implemented and has been in use for several years. This particular form has a domain and a range, each of one type. The domain must be a set such that no two elements may have the same value for their domain. The user provides the equality and ordering tests for the domain. The *Maps* library contains a collection of procedures for the construction and manipulation of the data of the *map* type. The type of a *map* for values of type *A* to values of type *Z* is represented as *Map*[A, Z]. Each pair stored in the *map* contains one value from the domain of type *A* and the corresponding value from the range of the type *Z* [Atkinson *et al.*, 1992b; 1993a]. For example, a *map* type may be created for storing the geometry of geographical features as follows:

```
let eq = proc(a, b: int → bool); {if a = b then true else false}
let lt = proc(a, b: int → bool); {if a < b then true else false}
let geometry := m_empty[int, List[XY]](eq, lt)
```

The first two statements provide the equality and ordering tests for the domain of type **int**. In this instance, the values in the domain are arranged in an ascending order. The third statement uses the procedure *m_empty* to prepare an empty *map* (*geometry*) for storing elements. Each element stored in the *geometry* object represents a mapping from an integer

value to a list of coordinate pairs *XY*. For example, each feature in Fig. 3.6 can be constructed as an element combining a feature identifier (*id*) and a list of coordinates (*xy_list*) in pairs. Thereafter, each element may be put into the *geometry* object in turn using the statement

m_isu_insert[**int**, List[XY]](geometry, id, xy_list)

The procedure *m_isu_insert* inserts an element with domain **int** and range *List*[*XY*] into the *map* (*geometry*). As a result, the object *geometry* has a bulk value which comprises three tuples, each corresponding to a feature. The procedure includes an "*_isu_*" in the identifier which means that this operation will update any instance of the *map* in situ, without changing the identifier of the *map*. In other words, this procedure only updates the content of an old object and does not result in the creation of a new object in the manner that the operation of the procedure *l_map* does in the example of utilising the *List* library. In fact, the naming convention, *i.e.* the use of "*_isu_*", will apply to all the procedures provided by the bulk type libraries. Furthermore, each procedure which belongs to a specific library has been designed to be associated with a prefix, such as '*l*' represents '*List*', '*m*' stands for '*Map*', *etc.* These naming conventions allow programmers to have a good chance of guessing the meaning or name of a procedure. In practice, therefore, this convention is very convenient when planning or executing the utilisation of the bulk type libraries.

An element in a *map* may be easily accessed by the procedure *m_find*. For example, the statement

let swamp_coords = m_find[**int**, List[XY]](geometry, 3)

will return a list of coordinates representing the feature swamp. Thus the *Maps* library is very useful for the modelling and indexing of geographical data - matters which will be discussed in more detail in Chapters 5 and 7 respectively. In particular, the elements of a *map* can be efficiently inserted and accessed because the *Maps* library has implemented an adaptive data structure, which is based on both the binary search and the B-tree, in order to tune itself automatically to the various number of elements present in a *map* [Atkinson *et al.*, 1993b].

3.4.5 Abstract Data Types

Abstract data types may be used where the data object displays some abstract behaviour which is independent of its representation type [Morrison *et al.*, 1993b]. An abstract data type can be manipulated without any need to discover its implementation or representation. For example, the procedures for drawing three kinds of entity (point, polyline and polygon) may be implemented differently, yet they all exhibit the same basic behaviour - that of drawing an entity.

In order to draw a point, the procedure may be implemented as follows:

```
type XY is structure(x, y: real)
type Extent is structure(xmin, xmax, ymin, ymax: real)
let draw_point := proc(pt: XY; pt_col: pixel; window: image; extent: Extent)
begin
    let point = colour[pt(x), pt(y)] in pt_col
    draw(window, point, extent(xmin), extent(xmax), extent(ymin), extent(ymax))
end
```

This procedure makes a picture *point* from a pair of coordinates *pt*, colours the picture in *pt_col*, and displays it within a drawing *extent* on a *window*. Similarly, the procedure for drawing a polyline may be implemented as follows:

```
let draw_polyline := proc(pl: List[XY]; pl_col: pixel; window: image; extent: Extent)
begin
    let polyline := [l_first[XY](pl)(x), l_first[XY](pl)(y)]
    pl := tl[XY](pl)
    while pl isnt empty do
    begin
        let xy = hd[XY](pl)
        polyline := polyline ^ [xy(x), xy(y)]
        pl := tl[XY](pl)
    end
    polyline := colour polyline in pl_col
    draw(window, polyline, extent(xmin), extent(xmax), extent(ymin), extent(ymax))
end
```

The differences between these two procedures are:

1. the identifier and the type of drawing entity (*pt*: *XY* vs. *pl*: *List*[*XY*]); and
2. the construction of a picture (*point* vs. *polyline*).

Because the implementation part (*i.e.* the expressions between begin and end) of the procedures is invisible to users, so the only difference is the data type (*i.e.* *XY* and *List*[*XY*]) used in the interface part.

The use of these two procedures for drawing geographical features is not very convenient, because the data type of a feature has to be determined before applying one of these procedures. For example, the drawing of the three features shown in Fig. 3.6 requires the uses of the *draw_point* procedure for feature 1 and the *draw_polyline* procedure for features 2 and 3. In order to have a general procedure for drawing features independent of their data types, an abstract data type can be used to achieve this aim. An abstract data type *Feature* for such objects can be defined by

```
type Feature is abstype[coords](id: int;
                                window: image;
                                extent: Extent;
```

```
                    id_col: proc(int → pixel);
                    location: proc(int → coords);
                    display: proc(coords, pixel, image, Extent))
```

The type *coords*, known as the witness type, is used to define an abstract data type over types such as *XY*, *List*[*XY*], *etc*. The abstract data type interface is declared between the round brackets. In the above case, the *Feature* has six fields: *id*, *window*, *extent*, *id_col*, *location* and *display* provided for the interface. The implementation of the interface depends on the requirements of the particular representation types involved. For example, the field *display* can be implemented as *draw_point* or *draw_polyline* depending on whether the data type is *XY* or *List*[*XY*], and the procedures required for the fields *id_col* and *location* may be further implemented as follows:

```
let id_col := proc(id: int → pixel)
begin
    let col = case id of
                1 : blue
                2 : red
                3 : green
            default : white
    col
end
```

```
let get_xy := proc(id: int → XY); l_first[XY](m_find[int, List[XY]](geometry, id))
let get_xy_list := proc(id: int → List[XY]); m_find[int, List[XY]](geometry, id)
```

where *geometry* is a bulk object of type *map* (see Section 3.4.4.2). Thereafter, abstract data objects of type *Feature* may be created. For example,

```
let house = Feature[XY](1, screen, extent, id_col, get_xy, draw_point)
let road = Feature[List[XY]](2, screen, extent, id_col, get_xy_list, draw_polyline)
```

will create objects *house* and *road* which have the same abstract data type *Feature*. However, their representations are different; in this particular case, the *house* object uses the point type *XY* and the *road* object uses the polyline type *List*[*XY*]. Once an object of type *Feature* has been created, one can no longer tell the specific representation used for *coords*. Based on the abstract data type *Feature*, a procedure for drawing any type of feature may be implemented and used as follows:

```
let draw_feature = proc(feature: Feature)
use feature as F in
begin
    let id = F(id)
    let window = F(window)
    let extent = F(extent)
    let id_col = F(id_col)
```

```
    let location = F(location)
    let display = F(display)
    display(location(id), id_col(id), window, extent)
end
```

```
! uses of the procedure
draw_feature(house)
draw_feature(road)
```

The **use** clause is a scoping and renaming device. The abstract data type *feature* is renamed as *F* in the clause following the **in**. By giving the object a constant name *F*, it can be assured statically that the interface procedures will only be applied to the objects of the correct representation [Morrison *et al.*, 1987]. The procedure *draw_feature* provides a public interface available for drawing all kinds of features. The important point about this procedure is that it will operate on objects of the abstract data type *Feature* irrespective of their implementation. For example, the change of the procedure *id_col* to rearrange colours for features or a change in the representation of a *house* object from its representation as a point to that of a polygon when displaying it at a larger scale only needs a change in the internal representation of abstract data objects. These changes do not result in a change to the procedure *draw_feature*.

Thus, abstract data types provide a mechanism whereby a clear separation is made between the interface and the implementation of the data type. The information-hiding feature of abstract data types is essential for the development of an object-oriented software and database for an IGIS.

3.4.6 Making Data Persistent and Reusing Persistent Data

All the data objects created in the previous programs can be placed into the persistent store, *i.e.* they will exhibit orthogonal persistence. The environment *User* of the persistent store is reserved for users to organise and manage their persistent data. Initially, the environment is empty. Both programs (*i.e.* data objects of **proc** type) and data of any type may be made persistent in the environment *User*. The environment structure in the environment *User* may be designed and constructed by the users. A new environment is created by using the standard procedure *environment* and it may be added to the environment *User* by the **in** ... **let** ... notation for declarations. For example, the environment *User* prepared for the storage of procedures and data may be set up by the following program.

```
use PS( ) with IO, User: env; environment: proc(→ env) in
use IO with writeString: proc(string) in
if User contains Programs or User contains Data then
   writeString("User already contains Programs or Data, no action taken.'n")
else
```

```
begin
    in User let Programs = environment( )
    in User let Data = environment( )
end
```

In order not to destroy existing valuable information inadvertently, the operator **contains** is used to test whether the name already exists in the specified environment. This program creates two new environments - *Programs* and *Data* - in the environment *User*. Afterwards, procedures and data may be added to these two environments *Programs* and *Data* respectively also by using the **in**... **let** ... notation declarations. For example, adding the following statements into the program *vector_graph.N* one line before the last **end** will make the *vector_map* persistent in the environment *Data*.

```
use User with Data: env in
in Data let vec_map := vec_map
```

Similarly, the following statements may be added to a program which contains the procedure *draw_point*. This arrangement will make this procedure persistent within the environment *Programs*.

```
use User with Programs: env in
in Programs let draw_point := draw_point
```

However, procedures are usually developed independently and will be shared by many programs. The development of a procedure has to accommodate any future changes, and these changes must not result in the re-compilation of the program which uses this procedure. This situation may be realised by using the incremental construction methodology supported by Napier88's incremental loading mechanisms. The philosophy behind this methodology is that of employing the L-value binding which libraries of the procedure values hold as procedures variables. Thus the amendments to the procedures should be achieved by assignment to these variables. This methodology is implemented by setting up the initial procedure variables as stubs. These are variables containing standard dummy values, and these variables will be given useful values by assignment from separate programs [Atkinson, 1992b]. For example, the following program sets up stubs for the procedures *draw_point* and *id_col*.

```
type XY is structure(x, y: real)
type Extent is structure(xmin, xmax, ymin, ymax: real)
use PS( ) with User, GlasgowLibraries: env; environment: proc(→ env) in
use User with Programs: env in
use GlasgowLibraries with Miscellany: env in
use Miscellany with uninitialised: proc[T](string → T); uninitialised_void: proc(string) in
begin
    in Programs let draw_point := proc(pt: XY; pt_col: pixel; window: image; extent: Extent)
```

```
                                    uninitialised_void("draw_point")
    in Programs let id_col := proc(id: int → pixel); uninitialised[pixel]("id_col")
end
```

The procedure *uninitialised* or *uninitialised_void* is used to provide a dummy value for each "set-up" procedure depending on whether it will return a value or not. It should be noted that a set of type declarations may also be pre-compiled and saved for reuse. For example, the type declarations such as *XY, Extent, Feature, etc.* used in the previous examples may be collected and saved as a file. This file is then processed by the command `nps` which compiles and saves the type declarations. Once the storage of the type declarations has been made, a program may be compiled against these available types. The following program is then used to initialise the procedure *draw_point* without needing to specify the type declarations in the program.

```
use PS( ) with Graphical, User: env in
use Graphical with Outline: env in
use Outline with makeDrawFunction: proc(string → drawFunction) in
use User with Programs: env in
use Programs with draw_point: proc(XY, pixel, image, Extent) in
begin
    let draw = makeDrawFunction("image")'imageDraw
    draw_point := proc(pt: XY; pt_col: pixel; window: image; extent: Extent)
    begin
      let point = colour[pt(x), pt(y)] in pt_col
      draw(window, point, extent(xmin), extent(xmax), extent(ymin), extent(ymax))
    end
end
```

After binding some geographical data and the procedures used in the previous examples, the environment *User* will have a structure which is shown in Fig. 3.8.

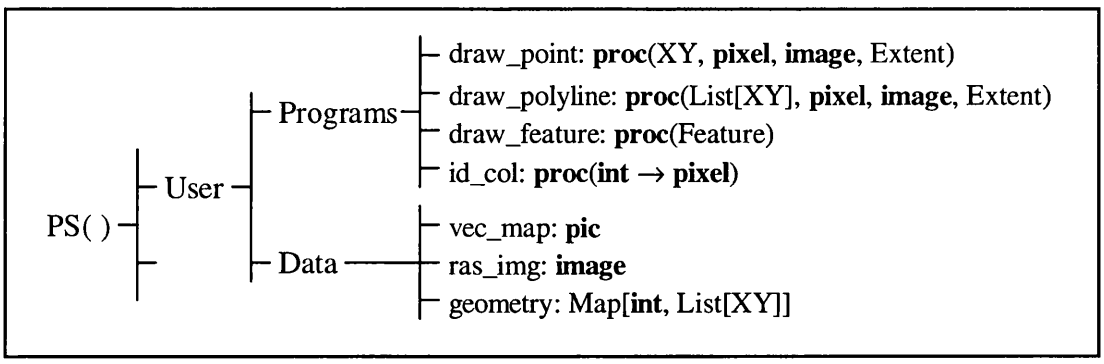


Figure 3.8 The environment *User* after the binding of procedures and data

The way in which programs and data from the environment *User* can be retrieved for reuse is exactly the same as that used with procedures from the standard library. For example, the

procedures *draw_polyline* and *id_col* and the data *geometry* may be retrieved from the persistent store to draw the feature 3 in a blue colour by using the following statements.

```
! reuse.N
use User with Programs, Data: env in
use Programs with draw_polyline: proc(List[XY], pixel, image, Extent);
                    id_col: proc(int → pixel) in
use Data with geometry: Map[int, List[XY]] in
! identifiers screen and extent declared as before
draw_polyline(m_find[int, List[XY]](geometry, 3), id(1), screen, extent)
```

This program need not be recompiled if the internal representation of the procedures has been changed. For example, if the colours in the procedure *id_col* have been rearranged, then the only requirement is to edit, compile and run this procedure. The program *reuse.out* can be used to give a different result without being recompiled.

It should be noted that unwanted procedures and data may be removed from their bound environments. For example, the statement

drop geometry **from** Data

will remove the binding (or object) *geometry* from the environment *Data*. The effect is that the binding *geometry* is no longer reachable from the environment *Data*. The dropped binding may still remain in the persistent store if other bindings to the value in the dropped binding are still valid [Morrison *et al*, 1993]. The space allocated for unused objects may be eliminated by performing a garbage collection using the command `nprgc`, followed by running the store compaction command `nprcompact`.

3.5 Summary

For geographical data handling, the support of persistent objects is of the greatest importance to a GIS. Most programming languages (whether conventional or object-oriented) only allow certain types of objects to outlive the execution of a program. Once the program terminates its execution, all objects except those of certain persistent types become inaccessible. Such programming languages cannot provide a seamless and elegant support for the persistence of geographical data, due to the already mentioned problems of the semantic gap and the impedance mismatch. Things become still worse when the program (and the programmer) is concerned with the integration of various types of geographical data, as in an IGIS. Therefore, there is a need to treat persistence as an orthogonal property of data. That is to say, persistence should be independent of data type and the way in which the data is manipulated. There have been many attempts to add persistence to a programming language. Now persistent programming languages are being developed as the technology that will provide an answer to such a requirement.

Napier88 is probably the most prominent of the persistent programming languages. The Napier88 system allows any object to persist in a persistent store irrespective of the representation of the object. There is no difference in the usage, referencing or access of objects that are stored in the memory or in the persistent store. The support of orthogonal persistence for data objects by Napier88 forms a good basis for the provision of an integrated programming/database environment. Therefore, the provision of orthogonal persistence is regarded as the most important facility for the development of an IGIS. Apart from this, Napier88 has several language features which are novel, including automated support of persistence, graphic data types, high-order procedures and incremental loading mechanisms. Napier88 also provides a collection of standard libraries and bulk libraries to facilitate system development. Some of the facilities provided as standard are in fact essential for the development of an IGIS. They include vector graphics, raster graphics, bulk type libraries, abstract data types, WIN, *etc.* These facilities seem to form the basis for the creation of an IGIS with a full degree of integration. Thus the suitability of Napier88 as an IGIS development tool will be examined in detail in the Chapters that follow.

CHAPTER 4 : THE IGIS SYSTEM ARCHITECTURE

4.1 Introduction

As has been described in Chapter 3, the fundamental concepts and the design principles of the persistent programming language Napier88 are quite different to those of conventional programming languages. In particular, Napier88 treats persistence as an orthogonal property of data to provide a firmly integrated programming/database environment. In principle, such an environment is an ideal foundation for building a geographical database with a full degree of integration. However, because the persistent environment is entirely supported by the Napier88 language itself, no interface is provided for users to make use of alien resources, *e.g.* embedding Unix commands or other languages' library facilities into a Napier88 program; providing linkages to commercially available DBMSs, *etc.* Therefore, in order to take advantage of the unique feature of the persistent environment, the design and development of an IGIS needs to start from scratch and depend solely on the resources provided by the Napier88 system. In other words, the **complete** method (see Section 1.2.2) is currently the only approach which may be used to carry out an IGIS development using Napier88.

The development of an IGIS with full functionality is a relatively complex undertaking and requires many man-years of effort. In general, a GIS comprises four important components (hardware, software, database, people) and provides five major functions (input, manipulation, management, query and analysis, output) for dealing with spatially referenced data in order to supply information in support of decision making and other activities [Aronoff, 1989; McRae, 1989; Star and Estes, 1990; Maguire and Dangermond, 1994]. All these components and functions interrelate and diversify the subsystems forming an IGIS. Therefore, careful design is vital for the success of a GIS besides having an efficient implementation. In practice, a GIS design may be subdivided into two major stages: its institutional (or external) design and its technical (or internal) design. The institutional design deals with the relationships that exist between the GIS and the world outside the system. The design phases of this category mainly comprise a user requirement analysis and a user resource analysis. On the other hand, the technical design is concerned with the functions, data models and data structures of a GIS. The design phases of this stage comprise the functional design and the database design [McRae, 1989; Marble, 1994]. These two stages of the GIS design are closely related and need to be conducted in harmony in order to develop a successful GIS.

The institutional design involves a wide range of issues, including data availability, data licences and costs, equipment availability and costs, data accuracy standards, data sharing

and security, data exchange standards, operational procedures, staff education and training, and so on [Lauer, 1991]. Many of these issues are administrative, managerial, or even political in nature. Consequently, it is rather difficult to carry out an institutional design which is fully comprehensive for an IGIS because of their complexity. Therefore, the institutional design is often ignored in practice. As a result, quite a number of examples have been reported in the technical press showing that the failure of a GIS was due to a lack of an appropriate institutional design. However, in spite of its importance, a discussion of the institutional design lies outside the main context of this research topic, so it will not be further discussed in this chapter.

On the other hand, the technical design deals purely with the system functionality, data models and data structures aspects implemented in an IGIS. Before carrying out the technical design, the features which are required in an IGIS have to be determined. It has been pointed out in Chapter 2 that the integration of heterogeneous data from diverse sources into a single system is one of the primary features needed in a GIS. There may be other demands to include advanced technical features in an IGIS. These may include handling the temporal dimensions of data as well as its spatial dimensions. Other requirements might be to provide version management within a large multi-user IGIS for long transactions; a seamless geometric and topologic database; and an object-oriented front-end processor for system customisation. Still other requirements or possibilities would be the provision of a virtual database allowing access to external databases; distributed geographic databases over a world-wide network for large-area or global projects; multi-media presentations of geographical information; the encoding of spatial knowledge into a database for spatial reasoning, and so on [Newell and Theriault, 1989; Laurini and Thompson, 1992; Laurini, 1994; Maguire and Dangermond, 1994; Newell, 1994]. However, many of these features are still in the stage of research and development. In addition, they involve a variety of technologies including database management, networking, multi-media, object-oriented programming, *etc.* Therefore, the inclusion of all of these advanced features in the IGIS design may end up with the system either being too complicated to implement or too big to handle. Since a fully integrated database is the backbone of all these advanced features, the IGIS system architecture presented in this thesis is centred on the issue of the integration of various types of geographical data. Accordingly, the functional design and the database design are focused on this particular point.

Since this is the first attempt to develop an IGIS using Napier88, there were no relevant GIS procedures written in Napier88 available for the system development - although in Abdallah's [1990] research, quite a large number of PS-algol procedures had been developed for his prototype GIS. However, none of these procedures could be reused in the Napier88 system because the two language systems are incompatible. Furthermore,

Abdallah's prototype GIS was only concerned with the handling of vector data and was based primarily on a spaghetti data model. It was clear therefore that the system architecture of Abdallah's prototype GIS would not be adequate for the development of an IGIS. Therefore, the system architecture of the new IGIS based on Napier88 has required a completely new design. Thus the remainder of this chapter is concerned first of all with the design of the IGIS with its associated data, products and other systems based on the persistent programming/store environment of Napier88. Thus, the design criteria have been drawn from a discussion of the design considerations required for the IGIS development. This is followed by a further discussion of the two designs which form the essential elements of the IGIS, namely, the functional design and the database design. Finally, the overall system architecture of the IGIS is presented.

4.2 The Persistent IGIS and its Surroundings

The design of an IGIS based on the persistent programming language Napier88 needs to identify the individual components of an IGIS and its surroundings. The IGIS is built within an integrated programming/database environment, *i.e.* Napier88's persistent programming/store environment or persistent environment for short. This persistent environment provides a seamless condition for running programs as well as storing persistent data. For convenience, such an IGIS based on the Napier88 system will be termed a persistent IGIS. This persistent IGIS with its surrounding is conceptualised in Fig. 4.1. It should be noted that this Figure depicts the logical data flow between different components rather than the physical system configuration.

The persistent IGIS consists of application programs and a persistent store. The application programs, which are stored as files using the Unix file system (not shown in Fig. 4.1), comprise a suite of application modules. Each application program is designed for a specific application. The persistent store contains a specific collection of data objects for IGIS applications, in addition to the Standard Library and the Bulk Libraries supported by the Napier88 system (as explained in Chapter 3). These data objects, which normally will be stored in the *User* environment, may be classified into two categories: a software library and an integrated database. The software library is an aggregation of basic GIS modules, procedures, default constants and variables provided for use in the application programs, whereas the integrated database is composed of three kinds of data: vector, raster and attribute. When running or executing the persistent IGIS, an application program is firstly loaded into the main memory of the computer by the Napier88 system. The application program then automatically accesses the data objects, which are reachable from the root object *PS()*, from the persistent store and copies them into the application program memory. Conversely, when the program terminates, the data objects used in the program

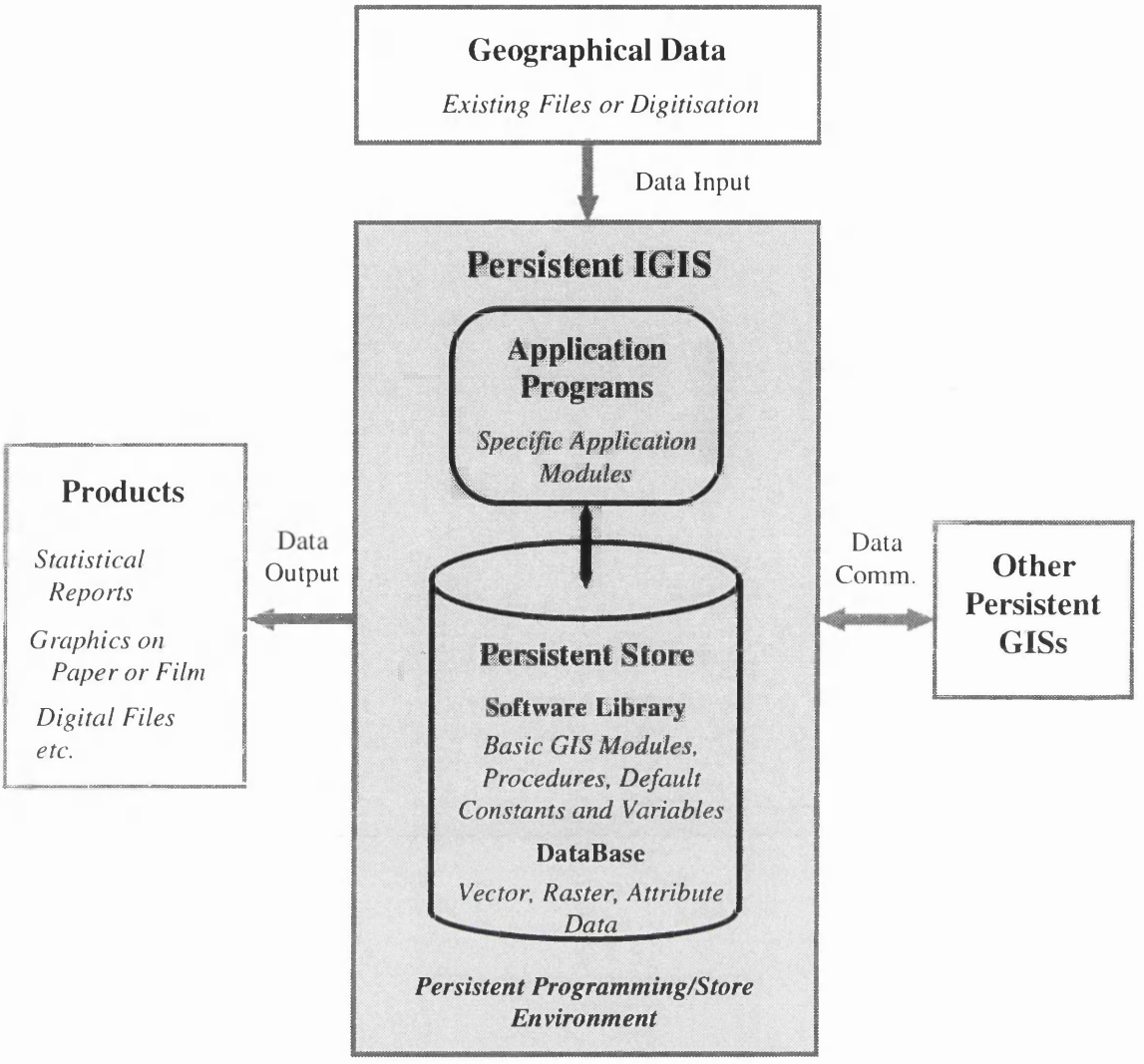


Figure 4.1 The Persistent IGIS with its surroundings

will be placed in the database if they are reachable from the *PS()*. It will be seen that the application program and the database are tightly integrated in the persistent environment while the Napier88 system is running.

On the other hand, the persistent IGIS needs to communicate and interact with the outside world for the purposes of data input, data output and data communication. The data input operations are required for the import of existing digital files and to support the data acquisition being carried out using various digitising devices. The data output operation is used to plot graphics, to print reports or to export digital files. As for the data communications aspect, this has to be provided for the sharing of data with other persistent GISs.

4.3 Design Considerations

From the system design of a persistent IGIS given above, it is apparent that the system development mainly involves the construction of the relevant software library and of the database which will be held within the persistent store in order to supply data objects for application programs. In other words, the construction of the geographical data in the database and the provision of the associated software library are the main concerns. As has been mentioned before, the application programs and the persistent store are closely integrated within the system environment of Napier88. Any data will have the same representation, including its format and its structure, both in the programs and in the persistent store. Because the data types stored in the persistent store will be used directly in the programs, so the software design is quite dependent on the database design, and *vice versa*. As a result, care in the design of the organisation of geographical data in the persistent store is a major factor in ensuring the successful development of a persistent IGIS. Thus the design of an integrated database in the persistent store is the most critical issue for the development of a persistent IGIS.

Consideration of the technical requirements for an integrated geographical database and the associated software library determines the key design criteria for a persistent IGIS. In this section, only those requirements that are more or less specific to the design of a persistent IGIS will be discussed. These basic requirements include consideration of

(i) the various forms of the vector, raster and attribute types of geographical data required for database integration;
(ii) the various functions of geo-processing systems required for software integration;
(iii) the data models and data structures needed for both the software and the database; and
(iv) the superimposition and concurrent processing of vector data and raster data in a persistent IGIS.

A detailed discussion of each of these matters will be given in the subsections that follow.

4.3.1 Various Forms of Geographical Data Types Needed for Database Integration

Before the design of an integrated database can begin, the anticipated capability of the IGIS should be determined. The functions of an IGIS may be confined to the applications of geographical query and analysis. Alternatively they may be extended to include the capabilities of relevant digital mapping and image processing systems should these be very important to the data input aspects of the IGIS. Generally speaking, GISs provide very limited capabilities for digital mapping or image processing operations. For example, most

existing vector GISs support a facility for the digitisation of existing maps. However, often they do not have a complete set of map editing functions such as those which will be found in digital mapping systems. In order to acquire some of the capabilities of digital mapping or image processing, the general practice is for a digital mapping system to be used as a front-end processor of a vector GIS or an integrated GIS. Similarly an image processing system may be used as a front-end processor of a raster GIS or an integrated GIS. In this instance, the output data formats of these front-end processors often need to be translated into the input data formats of the GIS. In fact, the data conversion and the repetition of some functions can be avoided if the functions of digital mapping and/or image processing are integrated into an IGIS. However, different functions of an IGIS may be involved at different stages in the geographical data handling. This will result in the complexities of data types in different forms having to be designed and accommodated in the persistent store.

Fig. 4.2 illustrates an overview of the different data types and the various forms of each data type that may be involved in different systems relevant to geographical data handling. Having regard to this geographical data handling, each type of geographical data may be generally classified as falling into one or other of three forms: either raw, processed or derived data. Each form may contain one or several formats or data structures, each specifically designed for a particular stage in the data processing. The raw form, which is often the simplest possible form, may be used with the data which has been newly created by digital acquisition devices. The processed form can be used when the prerequisite manipulations such as correcting errors, changing map projections, executing coordinate transformations, structuring the data, building topology, *etc.* has been carried out on the data. As for the derived form, it may be necessary to employ this when the data is generated from the processed form for specific applications. In general, digital mapping systems and image processing systems mainly cover the data processing operations carried out between the raw form and the processed form, whereas a vector GIS, a raster GIS and an integrated GIS are concerned primarily with the data processing implemented between the processed form and the derived form.

In each geo-processing system, an interim form - which normally represents the internal format used in the system - may be required for the efficient processing and storage of the data. For example, in a digital mapping system, the raw form may be used when the data is newly created. Then the raw data is edited, pre-processed and restructured into an internal format which represents the interim form. Finally, the interim data may be converted into the processed form representing a data format which may be used in other vector-based systems or for exporting digital files. Similarly, a vector GIS may import digital files from digital mapping systems which are already in a processed form and then transform them into the internal format of the GIS. Furthermore, the interim data may be extracted from or

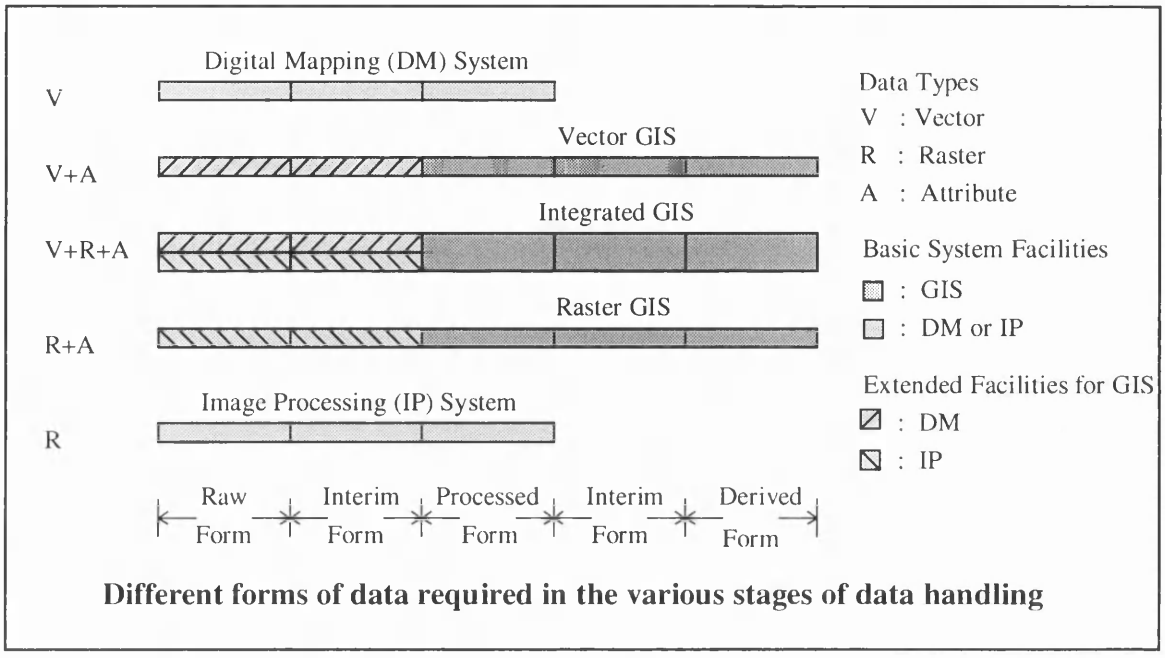


Figure 4.2 Geographical data handling may cover different forms of the vector, raster and attribute data types used in different geo-processing systems

reformatted into the derived form for other applications. In terms of an IGIS, it may not only accommodate the three data types (vector, raster, attribute), but it may also cover the processed, the interim and the derived forms required for each type of data processing. However, if an IGIS extends from basic GIS functions to the image processing functions associated with digital photogrammetric or remotely sensed data, then the raster data component needs to include the raw and the interim forms of data which derive from image processing. The same holds true for the vector data component should an IGIS be extended to include digital mapping capabilities.

It should be noted that the concept depicted in Fig. 4.2 has been simplified and only illustrates the major forms involved in a geo-processing system. In practice, such a system may have to cover all forms of one or several of the data types described above, *e.g.* a vector GIS usually includes a data format in the raw form needed for map digitisation, in addition to the processed form, the interim form and the derived form required for most operations.

As has been discussed in Section 2.5 and has also been demonstrated by the representative commercial IGIS products outlined in Section 2.6, the current trend in IGIS development is to include the functions of digital mapping systems as well as those of image processing systems. Therefore, the design of data types in an integrated database has to ensure that an IGIS is able to accommodate all the foreseeable requirements in these respects. Thus the

database of a persistent IGIS will contain the three main data types together with the different forms of each data type required for a whole range of geographical data handling.

4.3.2 Various Functions of Geo-processing Systems for Software Integration

Apart from the different forms of each data type which need to be included in an integrated database, it is also necessary to consider the associated software modules which will utilise the integrated database. Because the data flow and the linkages between the various software modules affect the organisation of geographical data in the database, so the requirement of ensuring that the various functions of different geo-processing systems can be integrated within the software has also to be considered. IGIS software can often comprise or be decomposed into dozens of modules. Each module is available for a specific application. For example, at the first level of software decomposition, an IGIS may be broken up into six main modules: the core module, the digital mapping module, the image processing module, the vector GIS module, the raster GIS module and the attribute handling module. These modules may be further subdivided into many small modules such as the vector data import module, the vector data export module, the raster data import module, the raster data export module, *etc.* The core module, which comprises a number of functional modules which provide basic facilities, may be used to link up and communicate with all other modules. All of these modules may employ different forms of one or several data types for their operations. Fig 4.3 is a typical example which illustrates the data flow and the linkages which could exist between different software modules in the situation described above.

It should be noted that the attribute handling module shown in this example also contains an import/export function, a processing function, *etc.*, but these have not been further decomposed into small modules. Different types of data (vector, raster and attribute) may be imported into or exported from the IGIS through an appropriate import/export module (or a function in the case of attribute data) in addition to those to be created within the IGIS using data collection functions which are not shown in Fig. 4.3.

In terms of vector data processing, raw vector data may be created by using the digital mapping module. The raw vector data is then processed and transformed into the interim vector data which is structured into the form required for most digital mapping operations. The interim vector data may be further converted into the processed vector data which is then restructured and prepared for general vector-based applications. The processed vector data may join the attribute data, which may be imported from attribute files via the attribute handling module, to form the interim vector/attribute data required for most vector GIS applications using the vector GIS module. Furthermore, the interim vector/attribute data

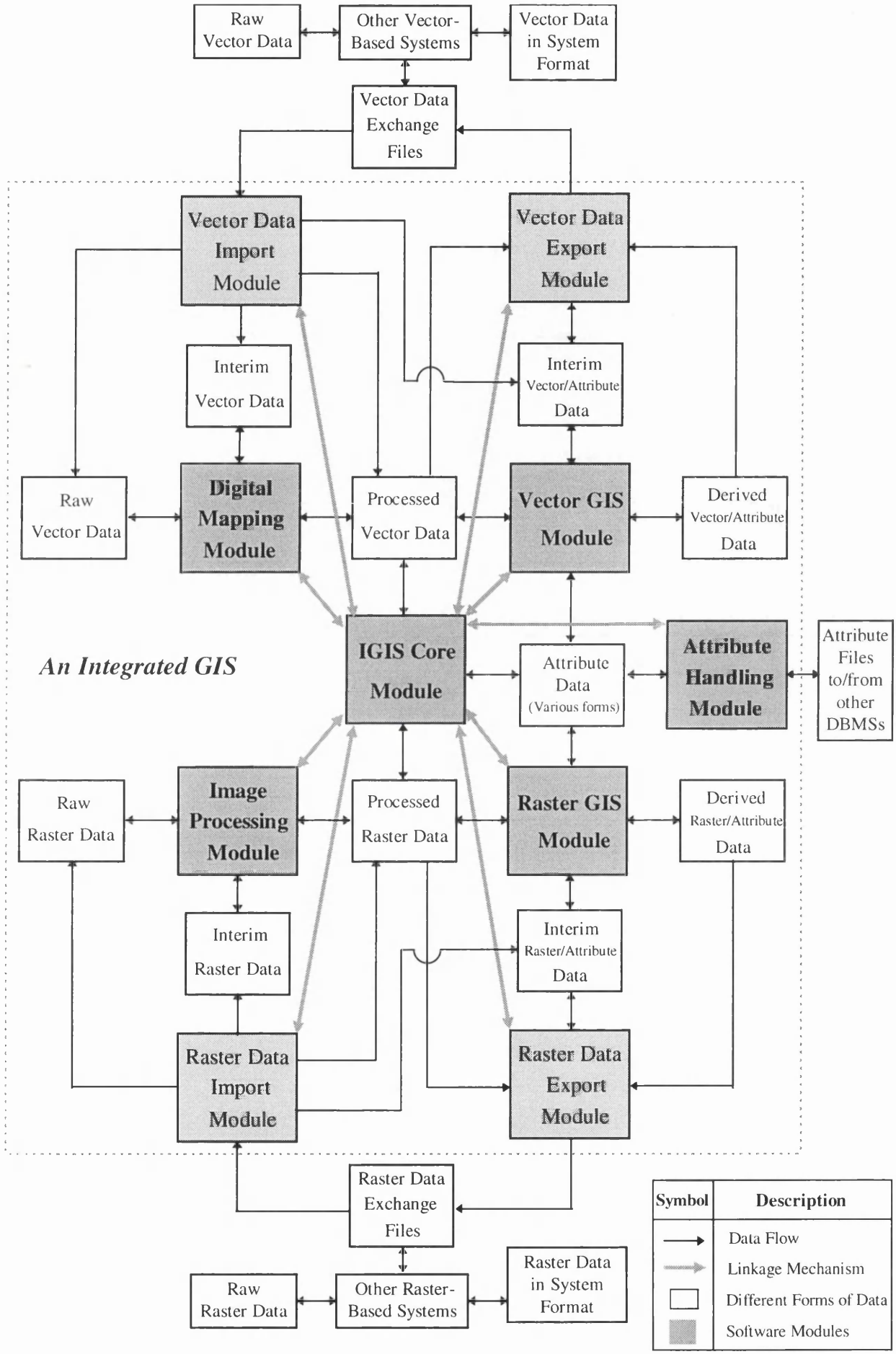


Figure 4.3 Data flow and linkages between the main modules in an IGIS

may have to be extracted and restructured in order to run specific applications within the vector GIS module or for transfer to other vector-based systems using the vector data export module. On the other hand, vector data can be imported from existing files in appropriate exchange formats by using the vector data import module. Depending on the status of the data held in an existing file and the capability of an exchange file format, the vector data may be imported to the IGIS in the form of the raw vector data, the interim vector data, the processed vector data or the interim vector/attribute data. For example, if a map has been digitised using a CAD software package, then the digitised data may be stored as a file in DXF format. If the file has been edited, then the file may be converted into the form of the interim vector data required specifically for the applications of the digital mapping module. Alternatively it can be produced in the form of the processed vector data generally needed for various uses in vector-based software modules. Otherwise it can simply be supplied in the form of the raw vector data which can be further edited by the digital mapping module. However, if a digital map has been created by a digital mapping system, often it will have been stored as a file in a geographical data exchange format, *e.g.* DLG, DIME, TIGER, NTF or SDTS. This kind of exchange file format is able to carry additional information such as data structures, attributes, *etc.* along with the actual vector data. Therefore, such a file may be converted into the form of the interim vector data or the interim vector/attribute data which is used solely in the digital mapping module or the vector GIS module respectively. In addition, it also may be converted directly into the form of the processed vector data for general applications of vector data in the IGIS.

The same concept of the data flow and associated linkages utilised in the vector-based software modules can also be applied to the raster data processing. For example, if a raw image has been acquired from the scanning of an aerial photograph or recorded from a remote sensing device and stored as a file employing a format such as TIFF, GIF, SPOT, Landsat TM, *etc.*, then the image file may be imported into the IGIS and converted into the form of the raw raster data. The raw raster data can then be accessed by the image processing module of the IGIS and restructured into the form of the interim raster data needed for efficient image processing operations. After the image data has been processed, it can be further converted into the form of the processed raster data used for general raster operations. However, if the image file has already been handled by an image processing software package before its import into the IGIS, *e.g.* an orthoimage file produced by the ERDAS system, then the file may be converted directly into the form of the processed raster data. Furthermore, the processed raster data may join the attribute data from other sources to form the interim raster/attribute data required for most raster GIS applications or it may be converted into the form of the derived raster/attribute data used for particular applications.

The vector-based modules and the raster-based modules are all linked together by the IGIS core module to provide either uni-format or dual-format applications. For example, the superimposition of vector maps and raster images may be carried out by simply accessing the processed vector data and the processed raster data using the IGIS core module. A further specific application may be carried out by an appropriate software module such as using the vector GIS module to perform a network analysis.

In summary, the anticipated functions of software modules in an IGIS have to be considered in the design stage since the overall capability of an IGIS depends not only on the support of an integrated database but also the data flow and the linkages between the software modules in the IGIS.

4.3.3 Data Models and Data Structures

Starting from the real world, the data modelling process is usually divided into four different levels:

(i) the external model,
(ii) the conceptual model,
(iii) the logical model and
(iv) the internal model.

The external model is arranged so that potential users can define their own subset of the real world relevant to specific applications. The conceptual model is a synthesis of all the external models which is realised in the form of the data model. The logical model is the implementation of the conceptual model in an information system which takes the form of a data structure. The internal model is concerned with the representation of the data within the storage media, *e.g.* in the form of its file structures. The relationship between the various data modelling levels is illustrated in Fig. 4.4 [Maguire and Dangermond, 1991; Laurini and Thompson, 1992].

Since the conceptual model and the logical model are in many ways the critical parts of geographical data modelling, they have been widely discussed in GIS literature [Aronoff, 1989; Star and Estes, 1990; Egenhofer and Herring, 1991; Maguire and Dangermond, 1991; Peuquet, 1991; Cromley; 1992; Laurini and Thompson, 1991]. It should be noted that, while these two modelling levels are often termed as data models and data structures in the GIS community, the usage of this terminology may be quite different from the conventions used within the computing community.

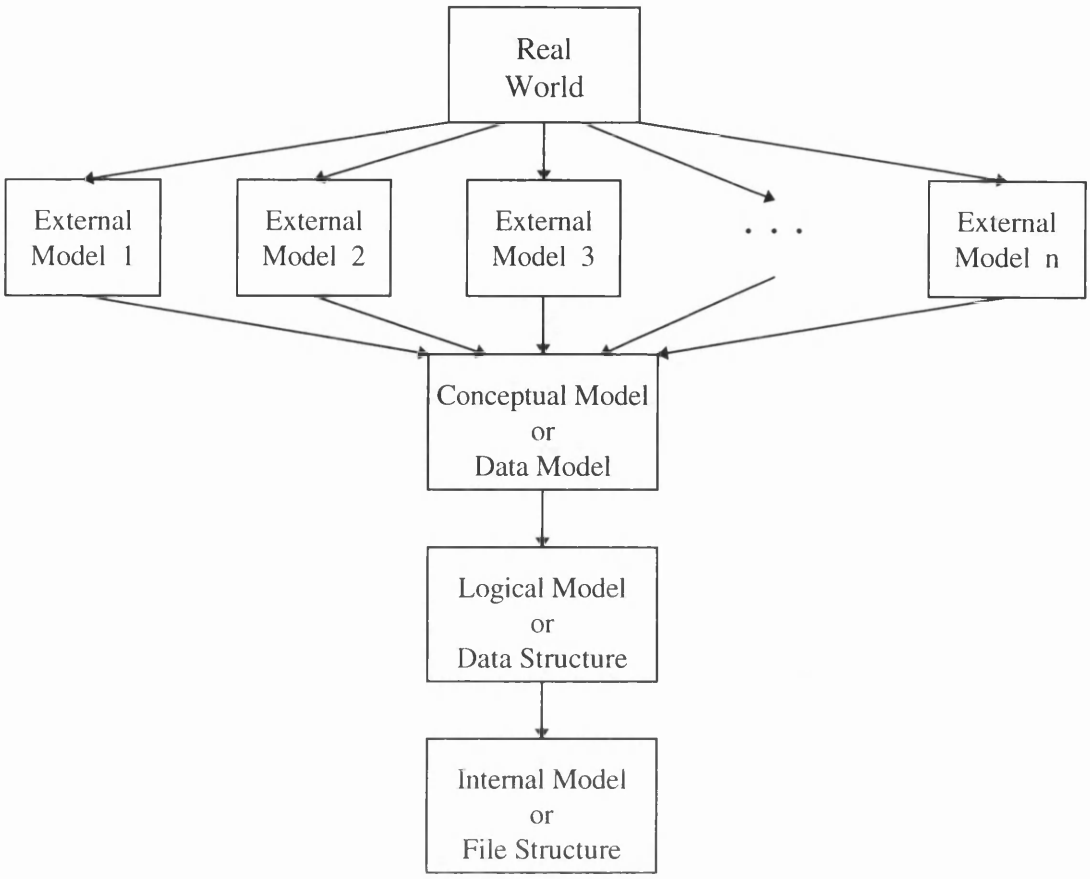


Figure 4.4 Data modelling levels

The goal of data modelling in a GIS is to consistently, robustly and efficiently organise geographical data in a computer system so that it can provide fast user queries and effective operations. The selection of a particular data model may considerably influence the processing of geographical data. Various data models have been developed to represent the three basic types of geographical data: vector, raster and attribute. Table 4.1 shows three representative data models for each of the data types that are commonly used in GISs [Peuquet 1991; Egenhofer and Herring 1991].

Data Types	Representative Data Models
Vector	Spaghetti, Topological, Hierarchical
Raster	Regular, Irregular, Hierarchical
Attribute	Hierarchical, Network, Relational

Table 4.1 The representative data models used in GISs

Each representative data model may have a number of variants depending on the complications of the data model required in or generated by a particular application. For example, the topological model may be further classified into the link and node, the partial

topological and the full topological models [BSI 1992]. In addition, the implementation of a data model may use various data structures. For example, the hierarchical raster model may be implemented as a quadtree or a R-tree or a derivative of these structures. In general, vector data commonly uses the spaghetti, the link and node, and the full topological models; raster data often employs the grid cell and the linear quadtree models; while the attribute data usually uses the relational model.

These data models differ in their powers and capabilities so that they provide various possibilities and different suitabilities within applications. In the past, it has been impossible to accommodate all these data models in a single GIS for use in any potential application. In the traditional approaches, some compromises between the data models and their potential applications are usually made in order to determine or define a suitable data model. Thus a commercial GIS usually dedicates one type of data model to suit general applications, *e.g.* the topological model is the most commonly-used data model in vector GISs. However, as has been discussed in Chapter 3, the design of a data type in a persistent system is similar to the design of a schema in a DBMS. Thus the provision of a data model in a persistent IGIS can simply be carried out by designing a suitable type system. In addition, there is no limit to the number of type systems that can be used in a persistent system. Therefore it is possible to provide users with a few or many (or even all possible) data models coexisting in a persistent IGIS.

For example, each main software module shown in Fig. 4.3 can accommodate a different data model. The digital mapping module may use the spaghetti model; the vector GIS often uses the topological model; the image processing module may use the regular raster model; the raster GIS may employ the hierarchical raster model; and the attribute handling module uses the relational model. Furthermore, each data model may be implemented with several data structures for the representation of different forms of data types. The implementation of the various data structures may be carried out by designing appropriate data types in order to promote the best system performance. Thus each form of data type described in Section 4.3.1 can be associated with a specific data structure in the data. For instance, the commonly used raster data structures are arrays (a matrix of integers or bytes), bitmaps (a matrix of pixels or bits), run length encoding, quadtrees and pyramids [Laurini and Thompson, 1992]. They can be optimally implemented for each form of the raster data. Thus the raw raster data may use an array or a run length encoded (RLE) structure; the interim raster data will often use a bitmap or an RLE structure; the processed raster data may use either a bitmap, an RLE or a linear quadtree (LQT) structure; the interim raster/attribute data may employ a LQT along with relation tables; and the derived raster data will normally use a bitmap structure.

The selection of a particular data model to represent a specific kind of geographical data depends on the requirements of a specific application. The data models used in GIS are many and varied. However, the optimal data modelling of geographical data for applications is another complex issue in GIS research and is not the main concern of this particular research effort. Instead, this research concentrates on how to implement these varied data models in a persistent store, *i.e.* taking the representation of geographical data with multiple data models and the implementation of various data structures into consideration in the overall design of a persistent IGIS.

4.3.4 Superimposition and Concurrent Processing of Vector Data and Raster Data

The various considerations discussed in Sections 4.3.1 and 4.3.2 - comprising the database integration and the software integration respectively - are intended to reach the storage level of integration. Using the approach - that of multiple data modelling - discussed in Section 4.3.3, it is possible to store all the required data in a form suitable for use at several levels of detail. Thus both the necessary multi-scale map data and multi-resolution image data and the appropriate data models and data structures associated with them can all be integrated within a single database. However, in order to have a full degree of integration, the design considerations for achieving both the display level and the process level of integration should not be neglected.

The superimposition of vector data and raster data requires the provision of capabilities to access both types of data from large volumes of these data and to superimpose them on a window-type display. This requirement involves utilising the capabilities of both vector graphics and raster graphics; the use of spatial indexing mechanisms; the display of multi-scale map data or multi-resolution image data; the overlay of geo-referenced vector and raster data in a single display; the provision of a graphical user interface; and the allocation of bit-planes for a graphical display device suitable for both vector and raster data. On other hand, the concurrent processing of vector and raster data needs the simultaneous use of multiple data structures and requires the software to support a wide range of spatial queries and analytical operations. Satisfying these requirements mainly involves the successful implementation of multiple data modelling (see Section 4.3.3) as well as a spatial indexing and interrelation mechanism for accessing both vector and raster data.

When considering the design of an IGIS, the spatial indexing and interrelation of geographical data are major concerns. The role of spatial indexing is to deal with the need to obtain geographical data based on location. Many factors affect the data retrieval performance of a spatial indexing technique. These include the organisation of the spatial data; the characteristics of index keys; the design of the data structures used for building indices, and so on. As for the interrelation of geographical data, three types of links need to

be established: the vector-to-attribute, raster-to-attribute and vector-to-raster links. Each linkage requires a bi-directional connection. The mechanisms for linking spatial (vector or raster) data with attribute data are very well developed so this is easier to implement than the linking of vector and raster data. Various spatial keys are possible for the linkage between vector and raster data, such as feature identifiers, coordinate values, grid cells, ZIP code, *etc.* However, very little research effort has been expended on the subject of linking vector and raster data. As a result, there is no well established method currently available for this purpose. Therefore it is necessary to adopt a spatial key and to develop a scheme of linkages between the vector and raster data.

4.4 Design Criteria

Based on the various design considerations discussed in the previous section (4.3), the key criteria for the design of a persistent IGIS may be summarised as follows: -

- The provision of an integrated geographical database and an integrated GIS software library.
  - Three types of geographical data (vector, raster and attribute) as well as different forms of each data type must be stored in a single database within the persistent store.
  - All modules, procedures, default variables and constants, *etc.* relevant to the use of the database must also be organised as a software library and stored in the same persistent store.

- The coexistence of multiple geographical data models which are capable of being implemented with various data structures.
  - Different kinds of data models representing different types of geographical data will be coexistent within the same database.
  - Each data model can be implemented with the several types of data structure required for different applications.

- The capability of superimposition and interrelation of vector maps and raster images.
  - Multi-scale map data and multi-resolution image data will be spatially indexed and linked in the same database.
  - The persistent IGIS must be able to display vector maps, raster images or a superimposition of both.

It should be noted that this outline is centred specifically around the integration of vector map data and raster image data into a persistent store and it is by no means exhaustive for the development of a fully-fledged IGIS. Instead, the list of the design criteria given above

has been chosen specifically in order to attempt the constructing of a truly integrated geographical database so as to provide a framework for the later development of a fully integrated GIS.

4.5 Functional Design

The functionality of a GIS is closely related to the needs of its actual or potential applications. Because of the nature of its highly diverse applications, it is impossible to develop a scheme of GIS functionality which is completely comprehensive. However, the functional design of a persistent IGIS may include the most important and widely used generic functions. Further specific functions may be individually extended and combined to produce the relevant software modules. A functional classification scheme has been designed for the persistent IGIS based on the work presented by Aronoff [1989], Maguire and Dangermond [1991] and Hartnall [1993a]. Within the software component, six main modules can be identified or categorised for the major types of functions that need to be implemented in the persistent IGIS. Each of these main modules is further divided into several sub-modules. Each sub-module represents a collection of functions grouped together for a definitive application. Fig. 4.5 shows a hierarchical classification of the major types of functions designed for the persistent IGIS. Each main module is briefly described as follows:

1. **Capture & Manipulate**
   This module comprises the digital mapping and the image processing sub-modules available for the data input aspects of the persistent IGIS. They mainly contain the functions needed for data acquisition and basic data processing and manipulation. The data acquisition functions involve collecting geographical data from a variety of devices; whereas the data manipulation functions deal with the removal of errors and inconsistencies and the pre-processing of digitised and scanned data in order to provide clean and appropriate data sets for the other modules.

2. **Transfer & Transform**
   The transfer functions deal with the import/export of geographical data to/from other systems using exchange file formats and the need for direct communication with other persistent GISs. The transform functions perform the required scaling, rotation and translation between two data sets using different coordinate systems or projections in order to provide geo-referenced data so that all data sets can be handled in a common coordinate system.

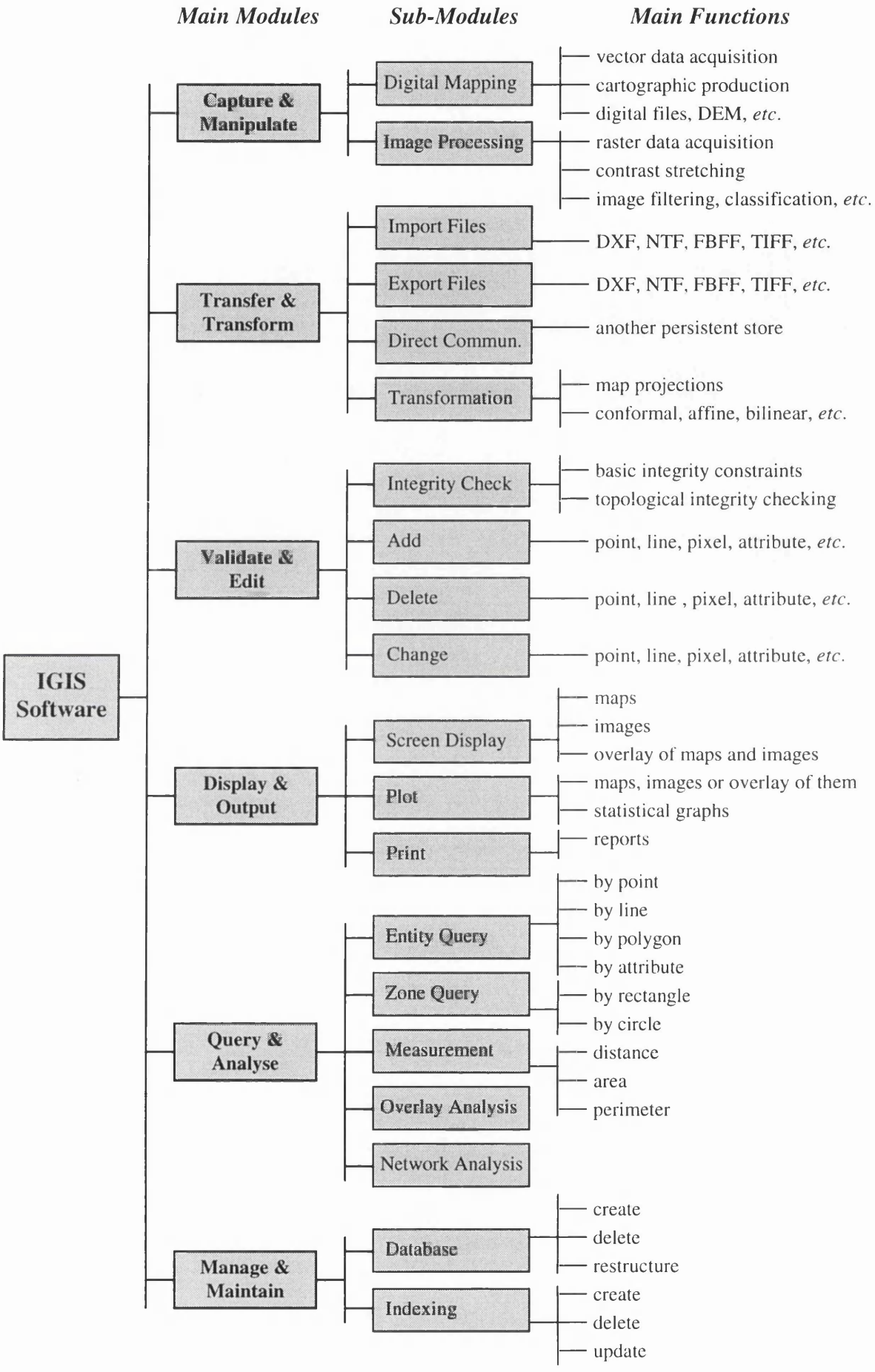


Figure 4.5 Functional design of the software component of the persistent IGIS

3. **Validate & Edit**

   The validate functions are concerned with applying integrity constraints to the data in order to ascertain that data consistencies such as range checks for attributes (basic integrity constraints) or topological checks for geometric primitives (spatial data checking) have been met. The edit functions allow the user to add, delete or modify geographical data.

4. **Display & Output**

   The display functions provide the capability for the presentation of the results in many forms, including maps, images, overlays of maps and images, graphs, statistical reports, tables and so on. The output functions comprise the generation of maps, images, graphs, reports, *etc.* in hard copy form or as digital files.

5. **Query & Analyse**

   The query functions deal with the retrieval of attribute data about spatial features or *vice versa* such as querying the attribute information of geographical features by pointing to entities or by specifying a search region. The analyse functions provide the ability to analyse geographical patterns and relationships through the operations of spatial searching and overlay such as network analysis, overlay analysis, *etc.*

6. **Manage & Maintain**

   The manage and maintain functions are concerned with the management and maintenance of the databases, the spatial indexing of the databases, and the software library. Other operations, such as changing the data structure to suit a specific application, updating a spatial indexing key to react the changes of a database, *etc.* will also be included in this module.

As has been discussed in Section 4.2, the basic GIS modules, procedures, default variables and constants relevant to IGIS functions can be organised as a software library within the persistent store. All the items in the software library may be classified into three categories - *General*, *Graphical* and *GIS* - so that each of them can be stored in a separate environment in the persistent store. The *General* library contains those items available for general programming such as converting a string of digits into an integer; the *Graphical* library consists of procedures relevant to graphical manipulation such as drawing a point, drawing a line string, zooming a map, *etc.*; and the *GIS* environment comprises all the other items except those contained in the *General* and the *Graphical* libraries. The environment for the storage of the software component in the persistent store is illustrated in Fig. 4.6.

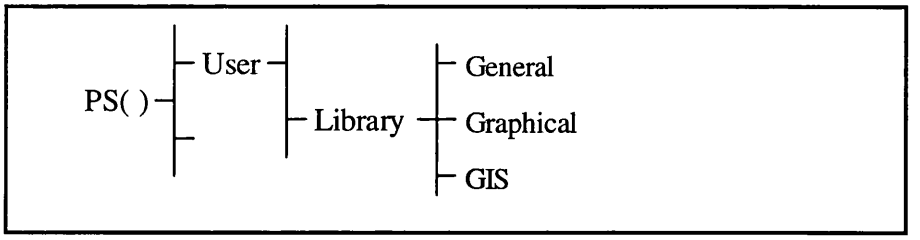


Figure 4.6 The software *Library* environment of a persistent IGIS

4.6 Database Design

The design of the database component required for a persistent IGIS involves two parts:

(1) the set-up of the database environment needed to hold geographical data in the persistent store; and
(2) the creation of type systems to represent a variety of geographical objects.

The first part of the database design is concerned with the organisation of store environments for the various forms of the three basic data types. Four forms (raw, interim, processed and derived) of each data type, which have been discussed in Sections 4.3.1 and 4.3.2, may be considered for inclusion in a persistent IGIS. Each form of data can be grouped together in an environment, *i.e.* three basic data types of the same form are integrated into a database. Therefore, four databases, entitled respectively *Raw*, *Interim*, *Processed* and *Derived*, may be created in the *Database* environment. The *Raw* database can be used to store the data which requires further editing and pre-processing; the *Interim* database may hold the data in the form suitable for digital mapping or image processing; the *Processed* database, which is the core database, is used to store the data in the form available for most GIS applications; and the *Derived* database can be used to store the data which has been restructured for particular applications. Apart from these databases, an *Index* database may be used to store data such as the mbr (minimum bounding rectangle) tables, index tables, *etc.* required for the indexing of the above four databases. The environment for holding geographical databases in the persistent store is illustrated in Fig. 4.7.

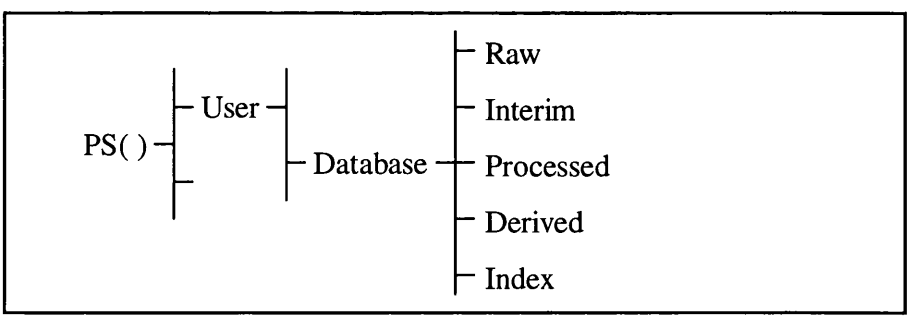


Figure 4.7 The *Database* environment of a persistent IGIS

The second part of the database design deals with the declaration of data types (schema) forming a number of type systems (or data model) which may be used selectively in the persistent IGIS. The design of a type system involves the selection of a data model and the implementation of the selected data model with appropriate data structures. As has been discussed in Section 4.3.3, multiple data models are required for the representation of different kinds of geographical data and multiple data structures are also needed to implement the various forms of a particular kind of data. Unlike conventional IGISs, which often employ a uniform data model for each kind of geographical data, the persistent IGIS is intended to provide several data models for various applications. In other words, the representation of vector map data at different scales may employ an appropriate data model to tune the IGIS to a specific application. The same applies to the representation of multi-resolution raster image data. For example, vector map data at a small scale often requires the link and node model for the applications of network analysis or the fully topological model for the applications of overlay analysis; whereas, at a large scale, the spaghetti model may be employed for map production. Therefore, as in the functional design, the database design can contain the most important and widely used generic data models and it may also be extended for certain specific data models. However, because of the complexities in the design of the type systems required for the many varied types of geographical data, it is inappropriate to discuss this in this section. Instead, the detailed design of data types for various data models will be discussed when dealing with geographical data modelling and organisation in Chapter 5.

4.7 The System Architecture

Based on the functional design (Section 4.5) and the database design (Section 4.6), the architecture of a persistent IGIS may be developed. The overall system architecture of the persistent IGIS developed by the present author is illustrated in Fig. 4.8. The software component and the persistent store, which closely interrelate in the persistent programming/database environment of Napier88, form the core of the persistent IGIS. The software component consists of the six modules provided for basic geographical data handling which were discussed in Section 4.5. Most reusable procedures employed in these modules are organised and stored in the software library of the persistent store. On the other hand, the database component comprises the five databases required for the integration of the different forms of geographical data as well as the index data. The software modules required for specific applications *i.e.* the applications modules, may be developed utilising the underlying core of the persistent IGIS. Furthermore, the persistent IGIS communicates with its surroundings through exchange file formats or by direct data transfer using the transfer and transform module.

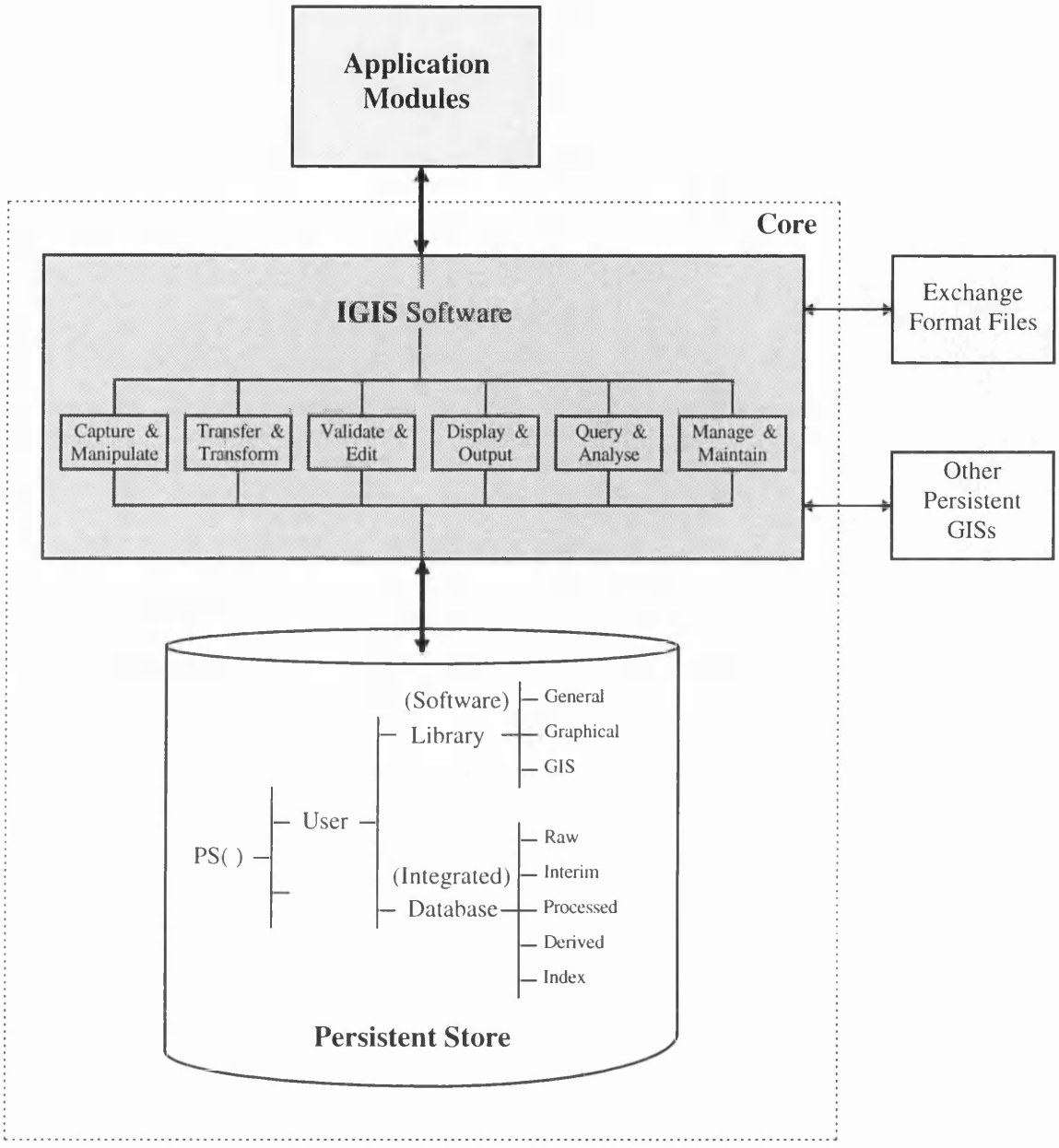


Figure 4.8 The overall system architecture of the Persistent IGIS

In summary, the design of the system architecture has aimed at the development of an IGIS having a full degree of integration. The resulting persistent IGIS will place an emphasis on the support of the integrated software needed for digital mapping, image processing, and vector and raster GIS handling, as well as the provision of an integrated geographical database for the storage of various forms of vector, raster and attribute data.

4.8 Discussion

The system architecture of the persistent IGIS has taken advantage of the unique persistent environment supported by Napier88. As has been discussed particularly in Section 4.4, three criteria are critical for the design of the persistent IGIS:

(i) an integrated geographical database and an integrated software library;
(ii) multiple geographical data modelling and data structures; and
(iii) superimposition and concurrent processing of vector and raster data.

Therefore, this research is oriented towards satisfying and providing these key design features. The essential tasks have been identified and organised into three parts:

(1) geographical data modelling and organisation;
(2) vector and raster interrelation; and
(3) spatial indexing and queries.

Each of these have been carried out and implemented; the results will be presented consecutively in the three chapters (5, 6, 7) that follow.

Another aspect of this research is the implementation of a prototype IGIS to work with large volumes of real geographical data in order to evaluate the capabilities and the performance of the persistent IGIS. Thus the strengths and weaknesses of an IGIS based on the persistent programming language Napier88 may be identified and evaluated from the results. This development and the results derived from it will be described in Chapter 8.

CHAPTER 5 : GEOGRAPHICAL DATA MODELLING AND ORGANISATION

5.1 Introduction

The discussion conducted in the previous chapter revealed that a persistent IGIS requires a number of commonly used data models to accommodate different forms of geographical data. In addition, the functional aspects of a persistent IGIS covering digital mapping, image processing, vector GIS and raster GIS operations must also be catered for. This Chapter describes the organisation of the geographical data held in the persistent store.

Because of its inherent simplicity and flexibility, the data model which has been most widely used for the handling of attribute data in GIS applications has been the relational model. Its organisation of the attribute data into a series of tables is simple to understand and is easily modified. Furthermore, the manipulation of the attribute information of a geographical feature most commonly involves a single data record held in a specific relational table. Therefore, in terms of geographical data handling, attribute information is an inherently simpler type of data to store and to manipulate than spatial data. Thus normally a spatial database management system stores and manipulates the spatial data. The descriptive information is accessed using keys to point to the corresponding attribute data. As such, the various forms of attribute data used in the different stages of geographical data handling may only employ a single data model - the relational model.

By contrast, spatial data is more complex. The manipulation of spatial data involves concepts such as proximity, connectivity, containment, overlay, *etc.* The representation of spatial data requires multiple records to store the location of a single geographical feature and its relationships to other features. Quite apart from this, spatial data usually requires a specific data model for a particular stage or process of geographical data handling - as has already been described in Section 4.3.3. Therefore, spatial data is not easily accommodated by the tabular database environment of a standard relational DBMS. As a result, it is essential that various spatial data models for the handling of different forms of geographical vector and raster data are provided during the development of a persistent IGIS.

This chapter first gives a brief introduction to both the conceptual modelling and the logical modelling involved in this development. Then some commonly used spatial data models which have been adopted for the representation of vector and raster geographical data in the persistent IGIS are discussed. Based on these data models, a set of data types which represent various geographical data can then be declared to prepare for the building of an integrated geographical database. This is followed by the setting up of the database

environment in the persistent store for the construction of the integrated geographical database. Finally, the organisation of multiple-scale vector map data and multi-resolution raster image data in the database will be discussed.

5.2 Conceptual and Logical Data Modelling

The provision of a data model for a persistent IGIS based on Napier88 is to allow the design of a type system for the data model. A type system can be created by declaring a set of data types which represent the entities of a data model and their relationships. In fact, the design of a type system basically involves two major steps in the data modelling process (see Fig. 4.4), namely conceptual modelling and logical modelling. These are further discussed in the following two subsections respectively.

5.2.1 Conceptual Data Modelling

Conceptual modelling often makes use of a formal approach known as the entity-relationship approach, The entity-relationship formalisation contains the following components [Laurini and Thompson, 1992; AGI, 1994]:

* Entity - A real world object or digital phenomenon.
* Entity class - A specified group of entities.
* Relationship - The association between entities or entity classes.
* Attribute - A trait, quality or property that is a characteristic of an entity or a relationship.
* Cardinality - The degree of relationship expressed by four numbers defining the minimum and maximum number of entities occurring in a relationship, in both the forward and the reverse directions.
* Integrity constraints - A predicate which must be matched in order to ensure the integrity of the model.

The conventions used for depicting entity-relationship by diagrams are varied. Fig. 5.1 illustrates the entity-relationship diagram employed in this thesis. An entity or entity class is represented by a rectangle; A relationship is depicted by a circle linking two or more entities or entity classes. Each entity or entity class has an entity name. Attributes are shown as lists within the rectangle boxes. Each attribute consists of an identifier and a data type separated by a ":" mark. [Laurini and Thompson, 1992].

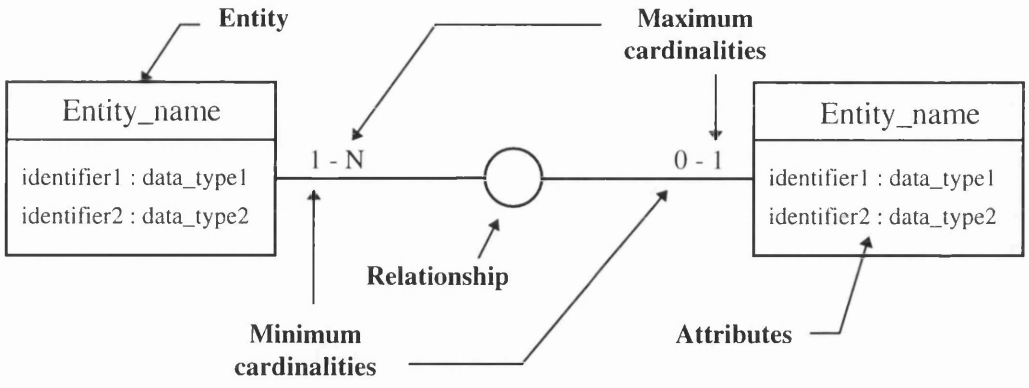


Figure 5.1 Nomenclature for entity-relationship diagrams (Adapted from Laurini and Thompson, 1992)

For example, the conceptual modelling of a polygon map which consists of polygons and points may be carried out as follows. Each polygon may be made up of a number of points (minimum 3 but unspecified maximum (denoted by N)), whereas each point may appear as an isolated point (minimum 0) or may be formed as a component of a particular polygon (maximum 1). The entity-relationship diagram between the polygon and the point entities can be depicted as Fig. 5.2.

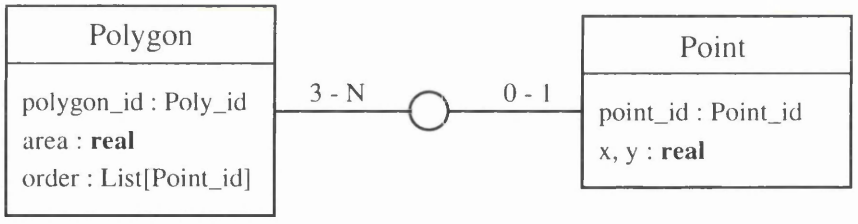


Figure 5.2 A conceptual model for polygon maps illustrated by an entity-relationship diagram

5.2.2 Logical Data Modelling

The second important step is to map conceptual data models to internal data structures. The logical modelling deals with the transformation of the entity-relationship into the data types associated with appropriate data structures. For instance, the entity-relationship diagram shown in Fig. 5.2 may be transformed into a type system utilising Napier88 as follows: -

```
type Poly_id is string
type Point_id is int
type Polygon is structure(area: real; order: List[Point_id])
type Polygon_table is Map[Poly_id, Polygon]
type Point is structure(x, y: real)
type Point_table is Map[Point_id, Point]
```

The first two lines declare identifiers *Poly_id* and *Point_id* with the types **int** and **string** respectively. The third line declares a data type *Polygon* which has a structure with two fields - *area* and *order* - of type **real** and *List*[*Point_id*] respectively. The field identifier *order* has a list of data values (point identifiers) indicating the sequence of points in the polygon in order to draw the shape correctly. The fourth line declares a data type *Polygon_table* which represents a mapping from a polygon identifier of type *Poly_id* to a polygon entity of type *Polygon*. Similarly, the last two lines declare the data types required for the point entity and the point table.

5.3 Spatial Data Models and the Design of Type Systems

The spatial component of geographical data can be represented either by a vector model or by a raster model. As has been discussed in Section 4.3.3, many data models have been developed to represent spatial data and each data model may have a number of variants. Five of these data models have been adopted in the design of the persistent IGIS. They include the spaghetti, the link and node and the polygon-based data models for use with vector data as well as the grid cell and the linear quadtree data models for handling raster data.

The naming conventions for data types relevant to data models have been devised as follows: -

DM_ represents a prefix for all types particularly used in a specific data model.

The prefixes used for the five data models are:

SP = Spaghetti
LN = Link and Node
PB = Polygon-based
GC = Grid Cell
LQT = Linear Quadtree

DM_e represents an entity in a specific data model, where e is the entity name. For example, SP_point denotes a point entity in the spaghetti data model;

DM_k_e represents a mapping table that will hold instances of the entity e. In this case, k is a short word or abbreviation for the entity identity which indicates the key of this table. For example, *LN_pid_point* represents the point table in the link and node data model and *pid* donates that the point identifier *point_id* is the key of the table;

E_id represents the data type of the entity identity e_id. E is a word that capitalises the first letter of the word e. For example, the data type of the point identifier *point_id* is represented as *Point_id.*

In this section, the design of a type system for each data model is described in turn in each of the five subsections (5.3.1 - 5.3.5) that follows. Each subsection explains the concept of

the data model and describes how a variant of the data model may be implemented as a type system using Napier88.

5.3.1 The Spaghetti Data Model

The spaghetti data model is the simplest vector data model for the representation of geographical data. Although the spaghetti data model is not well suited to the representation of geographical data used for GIS analyses, it is an important data model for the digital mapping aspects of an IGIS. Therefore, the spaghetti data model constitutes a particular component of the multiple data models required by the persistent IGIS.

5.3.1.1 The Concept

In this model, a geographical data set is regarded as a collection of coordinate strings grouped together with no inherent structure. A point is encoded as a single XY coordinate pair; a line as a string of XY coordinate pairs; and a polygon as a closed loop of XY coordinate pairs. Each entity (point or line or polygon) becomes a single logical record placed sequentially in the digital file containing the acquired data. The data file of XY coordinates is actually the form in which the vector data are stored in the computer.

The spaghetti data model is very simple and is easy to understand. It is an efficient data model for digital map production because spatial relationships which are extraneous to the plotting process are not stored. However, the spaghetti data model is rather inefficient for most types of spatial analyses since any spatial relationships must be derived by computation. Furthermore, the coordinate list describing the common boundary between adjacent polygons must be recorded twice, once for each polygon. This will result in the potential inconsistency of common boundaries between two digitising operations and the consequent increase in the storage requirement [Peuquet, 1984; Aronoff, 1989; Peuquet, 1991].

5.3.1.2 The Type System

Three entities can be identified in the spaghetti data model described above, *i.e.* point, line and polygon entities. In practice, the polygon entity may be treated as a special case of the line entity where the fist and last points have the same coordinate values. Apart from this, the text entity which is used to represent cartographic annotations may also have to be considered in the production of digital maps. Therefore, two compulsory entities (point and line) and one optional entity (text) are required in the design of a type system. The entity-relationship diagram of this particular variant of the spaghetti data model is illustrated in Fig. 5.3.

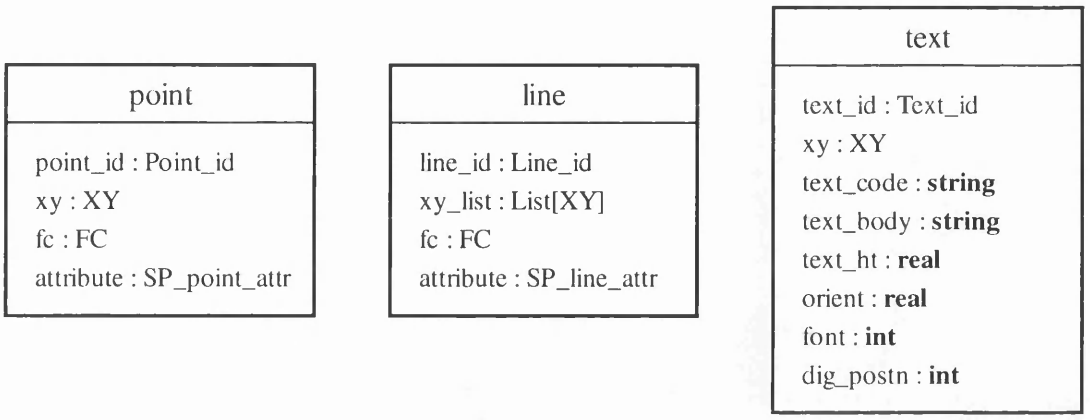


Figure 5.3 The entity-relationship diagram of a spaghetti data model

Because the spaghetti data model does not contain topological data, so the three entities are quite independent of one another without any relationship existing between them. It should be noted that several data types employed in the attribute lists are system types, including **int**, **real**, **string**, *List* which are supported by the Napier88 system, whereas the other data types are user-definable and may be specified to accommodate the needs of applications.

In Fig. 5.3, each entity type contains a particular attribute (entity identity) - point_id, line_id and text_id respectively - which denotes a unique value for each entity. The entity identity can be specified as a key used in a mapping table which is represented as a data type *Map*. The type system for the spaghetti data model depicted in Fig 5.3 may be declared as follows:

```
type SP_point is structure(xy: XY; fc: FC; attribute: SP_point_attr)
type SP_pid_point is Map[Point_id, SP_point]
type SP_line is structure(xy_list: List[XY]; fc: FC; attribute: SP_line_attr)
type SP_lid_line is Map[Line_id, SP_line]
type SP_text is structure(xy: XY; text_code, text_body: string; text_ht, orient: real; font,
                          dig_postn: int)
type SP_tid_text is Map[Text_id, SP_text]
type SP_tid_txt is variant(sp_tid_text: SP_tid_text; none: null)
type SP_DM is structure(point: SP_pid_point; line: SP_lid_line; txt: SP_tid_txt; fcd: FCD)
```

where *xy* or *xy_list* is the position of an entity;

fc is a feature code;

attribute is the descriptive information about the point and line entities; *i.e.* they are represented by the reserved types *SP_point_attr* and *SP_line_attr* which may be defined by the user;

text_code is used to categorise text;

text_body is a string of characters;

text_ht is the height of the text;

orient is used to define the orientation of the text string;

font is the numerical identifier for the font used for the display;

dig_postn identifies the position of a digitised point with reference to the bounding rectangle containing the text. For example, Fig. 5.4 illustrates that the *dig_postn* may have a value in the range 0 to 8;

point, *line* and *txt* where each represents a table for the corresponding entity;

fcd is a table containing feature codes and their descriptions associated with the data set;

FCD is a data type *Map* which maps an element of a feature code in the domain (*FC*) to its corresponding feature description in the range (*FD*), *i.e.* **type** FCD **is** Map[FC, FD].

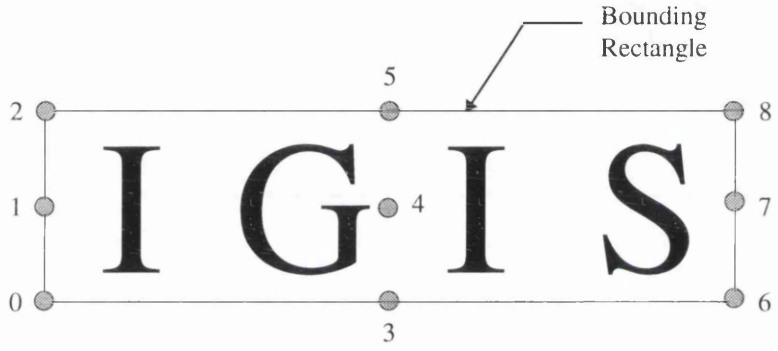


Figure 5.4 Digitised position indicators

It should noted that because the text entity is optional in this data model, so it may or may not be present in the database. This is handled by the **variant** type *SP_tid_txt* which has two selectors *sp_tid_text* and *none* containing respectively a table and no table for text data.

5.3.2 The Link and Node Data Model

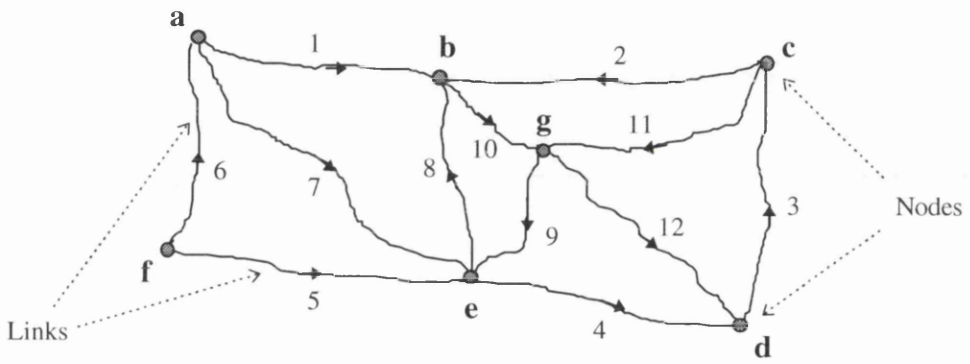
The link and node data model, also known as the Arc-Node data model, is one of the most popular methods for encoding spatial relationships among entities [Peuquet, 1991]. This data model explicitly records adjacency information between links and nodes which is essential for the implementation of network analysis, *e.g.* querying the shortest path in terms of the distance between two cities. The link and node data model is generally regarded as a particular form of the topological data model. There are several variants available for the representation of the link and node data model. The complexity of a link and node data model depends on what kind of spatial relationships exist and how they can be represented explicitly in the data model. For example, a link and node data model may

only encode basic spatial relationships - the link and the node topology - or it may additionally include the polygon topology.

The data model presented in this section is aimed specifically for use in network analysis. Therefore, both the link and the node topology are required in this particular form of the link and node data model.

5.3.2.1 The Concept

In this data model, a link is defined as a line string without any logical intermediate intersection, whereas a node is the start or end of a link and may be shared by several links. Spatial relationships between links and nodes are stored with cross-referencing. Fig. 5.5 illustrates the concept of the link and node data model.



Link Topology

Link_id	From_node	To_node
1	a	b
2	c	b
3	d	c
4	e	d
5	f	e
6	f	a
7	a	e
8	e	b
9	g	e
10	b	g
11	c	g
12	g	d

Node Topology

Node_id	Link_list
a	1, -6, 7
b	-1, -2, -8, 10
c	2, -3, 11
d	3, -4, -12
e	4, -5, -7, 8, -9
f	5, 6
g	9, -10, -11, 12

Figure 5.5 The concept of the link and node data model

In the link table, a link record comprises at least three basic elements: a link name (link_id) that identifies the link, and two node identifiers (from_node and to_node) for "from" and

"to" the endpoints of the link. In this way, both the connectivity and the direction of the link are explicitly recorded. On the other hand, a node record in the node table may contain a node name (node_id) and a list of link identifiers (link_list) that share the node. A number of additional attributes (not shown in Fig. 5.5) may be included in the link and node tables. Alternatively they may be linked to them by pointers.

Since the link and node data model explicitly retains elementary spatial relationships between links and nodes, it is very efficient for network analysis. Apart from this, this data model often stores the geometry of links and nodes in a separate geometry table. Therefore, the geometry of data can be recorded without redundancy, *i.e.* each line segment is recorded only once [Aronoff, 1989; Laurini and Thompson, 1992]. However, with this organisation, it is inefficient for polygon-based applications since the polygon topology is not stored explicitly in the database and has to be built during run time.

It should be noted that the node topology may be derived directly from the link topology, and *vice versa*. The storage of both link and node tables will result in some redundancy in the database. However, because both types of topology are used intensively during network analysis, it may be more efficient to store them in the database rather than to store only one of them and construct the other when it is needed.

5.3.2.2 The Type System

From the above discussion, it is clear that three compulsory entities (link, node and geometry) are required for the link and node data model. Apart from these, other optional entities can be added to accommodate particular applications. Thus, several variants of the link and node model may be formed. For example, the polygon entity may be added to the data model for handling polygon-based applications. A variant presented here is to include the point, line, text and attribute entities in the link and node data model. This is intended to allow the persistent IGIS to deal with those geographical features such as buildings, spot heights, contours, *etc.* which do not comprise a network and which supply additional information to that provided by the links and nodes. The entity-relationship diagram of this particular form of the link and node data model is illustrated in Fig. 5.6. Based on this diagram, the type system may be described as follows: -

```
type LN_point is structure(geom_id: Geom_id; attr_id: Attr_id)
type LN_pid_point is Map[Point_id, LN_point]
type LN_line is structure(geom_id: Geom_id; attr_id: Attr_id)
type LN_lid_line is Map[Line_id, LN_line]
```

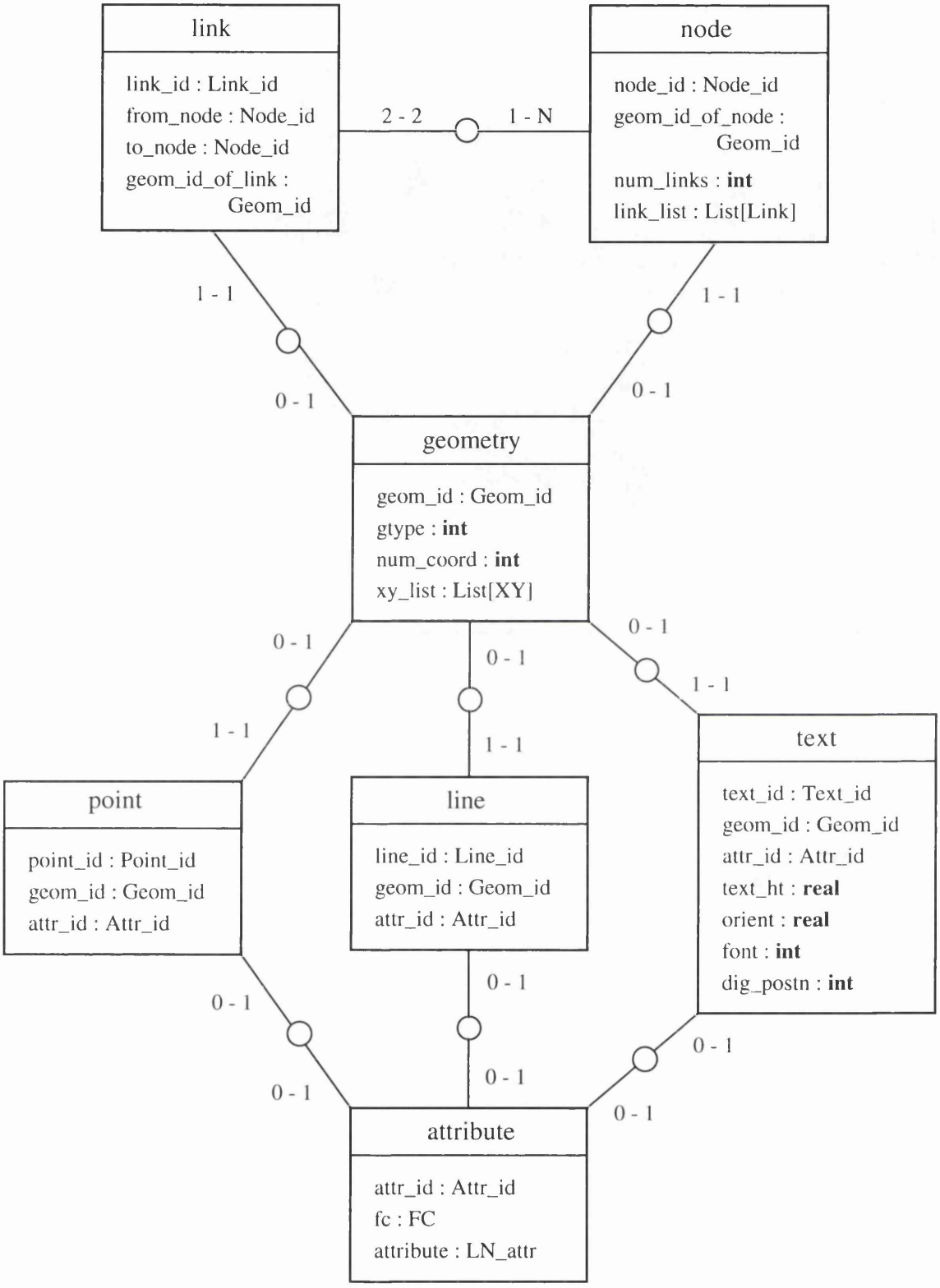


Figure 5.6 The entity-relationship diagram of a link and node data model

```
type LN_text is structure(geom_id: Geom_id; attr_id: Attr_id; text_ht, orient: real;
                          font, dig_postn: int)
type LN_tid_text is Map[Text_id, LN_text]
type LN_tid_txt is variant(ln_tid_text: LN_tid_text;  none: null)
type LN_geometry is structure(gtype, num_coord: int; xy_list: List[XY])
type LN_gid_geometry is Map[Geom_id, LN_geometry]
type LN_attribute is structure(fc: FC; attribute: LN_attr)
type LN_aid_attribute is Map[Attr_id, LN_attribute]
type LN_link is structure(from_node, to_node: Node_id; geom_id_of_link: Geom_id)
type LN_kid_link is Map[Link_id, LN_link]
type Link is structure(direction: int; geom_id_of_link: Geom_id; orient: real; level: int)
type LN_node is structure(geom_id_of_node: Geom_id; num_links: int; link_list: List[Link])
type LN_nid_node is Map[Node_id, LN_node]
type LN_DM is structure(point: LN_pid_point; line: LN_lid_line;
                        geometry: LN_gid_geometry; attribute: LN_aid_attribute;
                        link: LN_kid_link; node: LN_nid_node; txt: LN_tid_txt;
                        fcd: FCD)
```

where *geom_id* is a cross-reference to the geometry table;

attr_id is a cross-reference to the attribute table;

text_ht is the height of text;

orient (in *LN_text*) is used to define the orientation of the text string;

font is the numerical identifier of the font used for the display;

dig_postn identifies the position of a digitised point with reference to the bounding rectangle containing the text (see Fig. 5.4);

ln_tid_text is a text table;

gtype is geometry type. For example, 1 = point, 2 = line, 3 = arc, 4 = circle, *etc.*;

num_coord is the number of coordinate pairs;

xy_list is the position of an entity;

fc is a feature code;

attribute is the descriptive information about entities - it is represented by the reserved types *LN_attr* which may be defined by the user;

from_node and *to_node* are the node identifiers for "from" and "to" the endpoints of a link respectively;

geom_id_of_link is the identity of a record in the geometry table containing the coordinates of the link;

direction indicates whether the link starts at this node or ends at it. For example, 1 = link starts at node, 2 = link ends at node;

orient (in *Link*) is the azimuth of the first or last segment of the link at this particular node. For example, if the direction is "1", then the azimuth is of the first segment, and if "2" it is of the last segment;

level shows the relative levels of the links at the node. For example, if the links represent linear features which actually intersect on the ground, then they have the same value, otherwise they each have a different value;

geom_id_of_node is the identity of a record in the geometry table containing the coordinates of the node;

num_links is the number of links at the node; and

link_list is a list of link identifiers sharing the node.

5.3.3 The Polygon-based Data Model

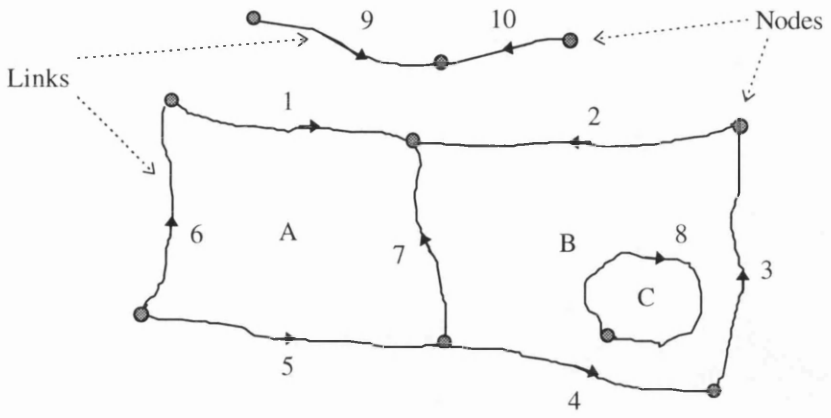
The polygon-based data model is also a widely used method of defining spatial relationships in a GIS. This data model defines the spatial relationships between polygons and chains. The polygon-based data model can also be regarded as a form of the topological data model. As the name suggests, this data model is very useful for polygon-based applications, *e.g.* the overlay analysis of land use data and land capability data. Thus the particular form of the polygon-based data model discussed in this section is oriented specifically towards those applications involving overlay analysis.

It should be noted that the data model presented here is entirely different from the so-called "whole polygon data model" in which each polygon is encoded in the database as an independent entity similar to the polygon entity described in Fig. 5.2.

5.3.3.1 The Concept

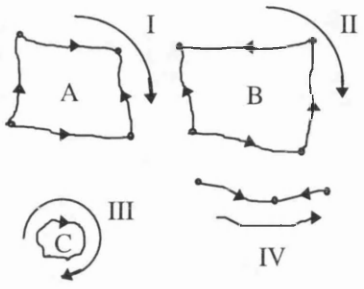
In this data model, the basic logical entity is the chain. Therefore, it is sometimes called a chain-based data model. A chain is defined as a collection of directed links. As in the link and node data model, a link is defined as a line string without any intermediate intersection. Thus a chain may represent a polygon or a complex line (polyline). The node topology is seldom used in those applications which are primarily polygon-based, so it has not been included in the data model. The concept of this polygon-based data model is illustrated in Fig. 5.7.

Both the polygon and the chain topology are the essential components of the data model. In the chain table, a chain record comprises at least two basic elements: a chain name (chain_id) that identifies the chain and a list of link identifiers (link_list) which comprise the chain. In the polygon table, each polygon record has, at the very least, a polygon name



Chain Topology

Chain_id	Link_list (geom_id_of_link)
I	1, -7, -5, 6
II	-2, -3, -4, 7
III	8
IV	9, -10

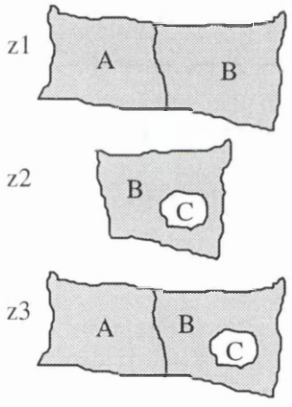


Polygon Topology

Poly_id	Chain_id
A	I
B	II
C	III

Complex Polygon Topology

Cpoly_id	Poly_id_list
z1	A, B
z2	B, -C
z3	A, B, -C



Link Topology

Link_id	Left_poly	Right_poly
1	0	A
2	B	0
3	B	0
4	B	0
5	A	0
6	0	A
7	A	B
8	B	C

Figure 5.7 The concept of the polygon-based data model

(poly_id) that identifies the polygon and provides a cross-reference to a record in the chain table. Based on these two tables, other spatial relationships can be constructed [BSI, 1992].

In the example shown in Fig. 5.7, the link topology which defines the relationship between polygons and links may be derived from the chain and the polygon topology. The complex polygon topology defines a complex polygon which shall be represented by one or more polygons. "Complex polygon" is used here since it is the term defined in the NTF specification (BS7567) as follows [BSI, 1992]: - *This feature shall contain one or more polygons that do not necessarily form a contiguous extent. The constituent polygons may overlap. Component polygons may be flagged to show where they are to be added or subtracted to create the final area.*

From the above discussion, it can be seen that the chain entities and the polygon entities may be regarded as the fundamental elements required for the construction of complex objects. With the addition of the point entity, the polygon-based (or chain-based) data model may be used to deal with very complex geographical features. For example, a park object can be formed by a combination of several polygons (*i.e.* a list of chains), polylines (also a list of chains) and points. Thus it may be used as an underlying data model for the object-oriented data management.

5.3.3.2 The Type System

In this data model, five entities (polygon, chain, link, geometry and attribute) are compulsory. Other entities may be added to the data model. In fact, two entities have been included: the cpolygon (complex polygon) and the collection. The word "collection" is used here since it is the term defined in the NTF specification (BS7567) as follows [BSI, 1992]: -

A collection shall contain one or more features of any type within the same section. Any specific collection shall not refer to itself.

The cpolygon may contain one or more polygons. The constituent polygons may be independent or overlapped. The "collection" may contain one or more polygons or complex polygons. The entity-relationship diagram of this particular form of the polygon-based data model is illustrated in Fig. 5.8. As has been noted earlier, the data model presented here is aimed specifically at polygon-based applications; thus the "collection" does not include entities other than the polygon and the cpolygon. However, this data model may easily be extended to embody one or more features of any type in the "collection". The type system which corresponds to the diagram shown in Fig. 5.8 may be described as follows: -

```
type PB_geometry is structure(gtype, num_coord: int; xy_list: List[XY]; attr_id: Attr_id)
type PB_gid_geometry is Map[Geom_id, PB_geometry]
type PB_aid_attribute is Map[Attr_id, PB_attribute]
type PB_polygon is structure(chain_id: Chain_id; geom_id: Geom_id; attr_id: Attr_id)
type PB_polyid_polygon is Map[Poly_id, PB_polygon]
type PB_link is structure(direction: int; geom_id_of_link: Geom_id)
type PB_chain is structure(num_parts: int; link_list: List[PB_link])
type PB_cid_chain is Map[Chain_id, PB_chain]
type PB_polyid_sign is structure(poly_id: Poly_id; sign: string)
type PB_cpolygon is structure(num_parts: int; polyid_sign_list: List[PB_polyid_sign];
                              geom_id: Geom_id; attr_id: Attr_id)
type PB_cpolyid_cpolygon is Map[Cpoly_id, PB_cpolygon]
```

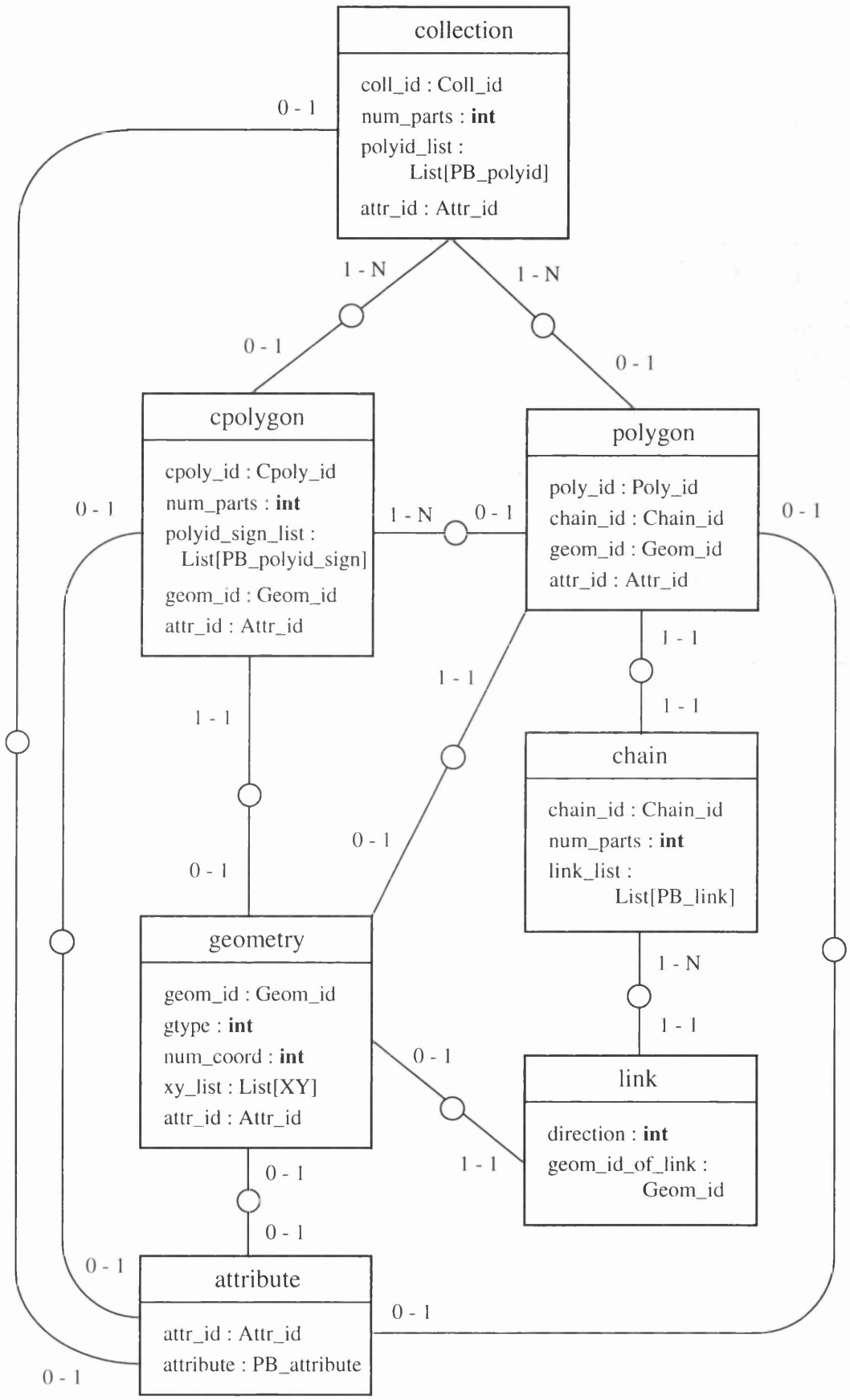


Figure 5.8 The entity-relationship diagram of the polygon-based data model

```
type PB_polyid is structure(poly_id: Poly_id; poly_type: int)
type PB_collection is structure(num_parts: int; polyid_list: List[PB_polyid]; attr_id: Attr_id)
type PB_collid_collection is Map[Coll_id, PB_collection]
type PB_DM is structure(collection: PB_collid_collection;
                        cpolygon: PB_cpolyid_cpolygon;
                        polygon: PB_polyid_polygon;
                        chain: PB_cid_chain; geometry: PB_gid_geometry;
                        attribute: PB_aid_attribute; fcd: FCD)
```

where *gtype* is geometry type. For example, 1 = point, 2 = line, 3 = arc, 4 = circle, *etc.*;

num_coord is the number of coordinate pairs;

xy_list is the position of an entity;

attr_id is a cross-reference to the attribute table;

chain_id is a cross-reference to the chain table;

geom_id (in *PB_polygon*) which indicates the location of *poly_id* is a cross-reference to the geometry table;

geom_id (in *PB_cpolygon*) which indicates the location of *cpoly_id* is a cross-reference to the geometry table;

direction controls the order of coordinates in the geometry table for the link. For example, 1 = the direction of the link is the same as the order of the coordinates, 2 = the direction of the link is the reverse of the order of the coordinates;

geom_id_of_link is the identity of a record in the geometry table containing the coordinates of the link;

num_parts (in *PB_chain*) is the number of links in a chain;

num_parts (in *PB_cpolygon*) is the number of polygons in a complex polygon;

num_parts (in *PB_collection*) is the number of polygons and complex polygons in a "collection";

link_list is a list of the link identifiers required for the chain;

poly_id is the polygon identity;

sign indicates whether the polygon referred to by *poly_id* shall be added or subtracted to create a complex polygon;

polyid_sign_list is a list of (*poly_id*, *sign*) that creates a complex polygon;

poly_type indicates whether the type of polygon is a simple or complex polygon. For example, 1 = simple polygon, 2 = complex polygon;

polyid_list is a list of polygon identifiers that create a collection of features.

5.3.4 The Grid Cell Data Model

The grid cell is the simplest form of the raster data model. In this data model, the space is subdivided into regular cells - which may be square or rectangular. The square cellular decomposition is the most commonly used method for dealing with geographical raster data. The square cell can be directly represented as a matrix or array which is supported by most programming languages. Furthermore, it can also be interfaced easily to the hardware devices commonly used for spatial data input and output [Peuquet, 1984; Aronoff, 1989]. Thus the grid cell data model may be used in those applications involving the use of image data which has been derived from remote sensing or scanning maps; in the organisation of map libraries, *etc.* However, this section will place an emphasis on the image processing aspect of an IGIS.

5.3.4.1 The Concept

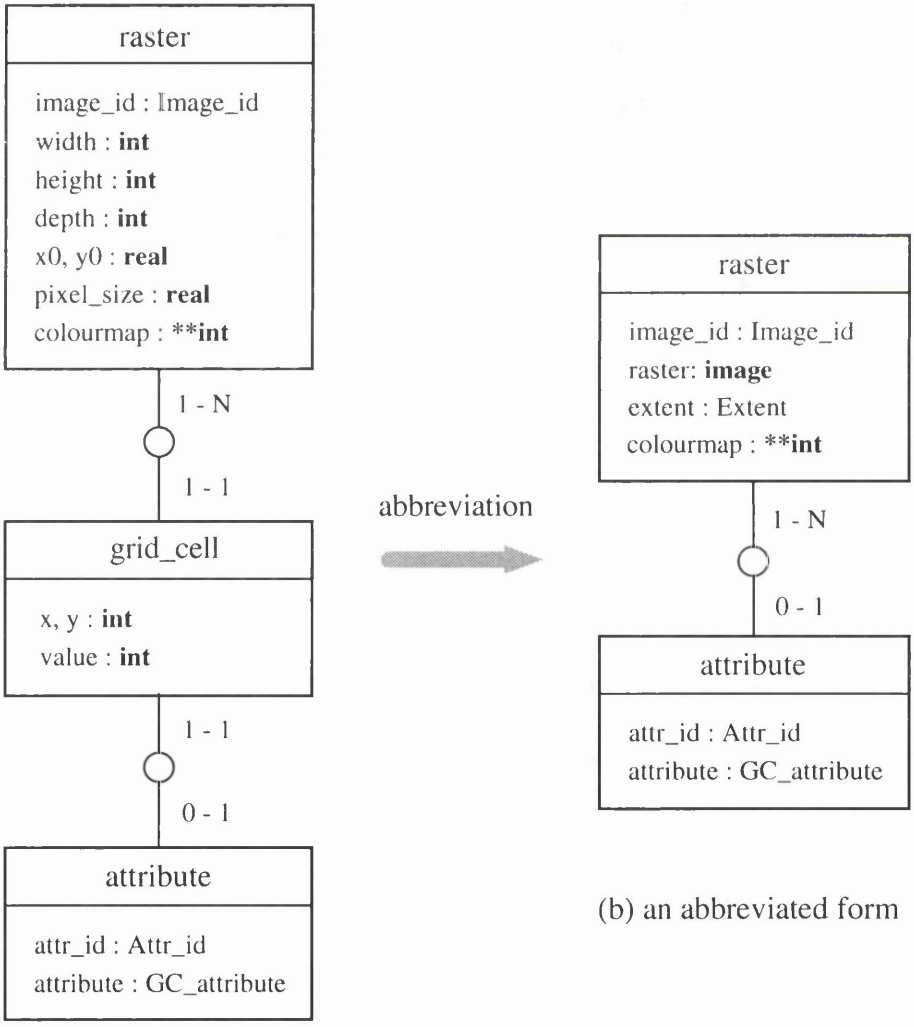
In this data model, the basic unit is the grid cell. Each cell represents an area of the land surface. The location of each cell or pixel is defined by its row and column numbers. The value assigned to the cell indicates the attribute that it represents. Geographical features can be represented by grid cells. A point feature is denoted by a single cell, a line feature by several cells with the same value forming a linear grouping, and an area feature by an aggregation of contiguous cells, all having the same value. The conversion of geographical data into grid cells mainly involves the existence of an *a priori* fixed grid cell size, supplemented by methods of determining which attribute lies in a specific cell, the level of description required for the attributes and the registration of a raster image into a terrain or geographical reference system [Laurini and Thompson, 1992]. Since the attribute of each cell is stored as a unique value, the total number of values to be stored is the product of the number of columns times the number of rows. In addition, a colourmap which contains a colour look-up table for displaying pixels (see Subsection 3.4.2) is often associated with a raster image. Thus the image size (width by height by depth), the pixel size, the coordinates of the origin (frequently the upper left corner of the image), and an attribute look-up table are usually regarded as being the basic elements forming the grid cell data model.

The grid cell data model can easily be accommodated to array data structures. Therefore, the overlay operations used with raster images of different themes may be carried out efficiently by comparing the attribute values for identical grid cells. However, this data model tends to take up more storage space than the corresponding vector data models in order to precisely represent geographical features. Thus several raster data models such as run-length encoding, quadtree, *etc.* have been developed to reduce the storage space required with raster image data. Apart from this, another drawback is that the topological relationships between geographical features are not explicitly represented in the data model.

Nevertheless, because of its simplicity, the grid cell is still the most widely used data model for handling raster data.

5.3.4.2 The Type System

Three entities can be identified in the grid cell data model, *i.e. raster*, *grid_cell* and *attribute*. The *raster* entity contains a matrix of grid cells (*i.e.* pixels) identified by row and column identifiers *width* and *height* together with the relevant auxiliary information (*i.e. depth*, *x0*, *y0*, *pixel_size* and *colourmap*). The *grid_cell* entity consists of the location (*x*, *y*) and the attribute code (*value*) of a pixel. The *attribute* entity provides the descriptive information for each attribute code. A general form of the entity-relationship diagram of the grid cell data model is illustrated in Fig. 5.9(a).



(a) a general form

(b) an abbreviated form

Figure 5.9 The entity-relationship diagram of the grid cell data model

As has been discussed in Chapter 3, Napier88 supports both **image** and **pixel** data types and also provides raster facilities for handling images. Thus the *grid_cell* entity can be replaced by the **pixel** type and can be embedded in the **image** type. Also, the identifiers *width*, *height* and *depth* can be removed from the *raster* entity because they are already included implicitly in the **image** type. Napier88 supports the procedures *xDim*, *yDim* and *zDim* in the *Raster* environment of the Standard Library as depicted in Fig. 3.4 for users to determine the overall image dimension. Furthermore, the origin of the image and the pixel size may be defined in a data type *Extent* which indicates the ground coverage of an image. Therefore, the entity-relationship diagram of the grid cell data model may be abbreviated as shown in Fig. 5.9(b).

The type system of the grid cell data model may be declared as follows: -

```
type Extent is structure(x_min, y_min, x_range, y_range: real)
type GC_aid_attribute is Map[Attr_id, GC_attribute]
type GC_DM is structure(raster: image; extent: Extent;  colourmap: **int;
                        attribute : GC_aid_attribute)
```

where *x_min* and *y_min* indicate the ground coordinates of the origin for an image;

x_range and *y_range* denote the dimensions of a whole image in both the x- and y-directions respectively within the ground reference system;

extent indicates the ground coverage of the image via the definition of the *x_min* and *y_min* and the corresponding *x_range* and *y_range*;

GC_attribute is a data type for the representation of the descriptive information about individual pixels;

raster contains the image data;

colourmap provides a colour look-up table for the display of the image. The data type ****int** denotes a matrix of integers holding the values of the RGB intensities for each representative colour. *i.e.* the dimension of the matrix is 3 x n, where n is the number of colours or grey scales; and

attribute represents the attribute table of the image.

In order to represent the same image as an appropriate data type which can be used in different stages of image processing, the following two data types have been declared for handling the raw form and the interim form of an image respectively.

```
type Rawimage is structure(data: *int; width, height, depth: int; colourmap: **int)
type Interim_image is structure(raster: image; colourmap: **int)
```

Because both the raw and the interim forms are intended for use during the image processing aspects of the persistent IGIS, so the attribute table and the auxiliary information for geo-referencing have been excluded in these two data types. It should be noted that the *Rawimage* employs an one-dimensional vector of type **int** to hold the image data in row order. This has been provided for some image processing operations which manipulate pixels in type **int** more efficiently than in type **pixel**.

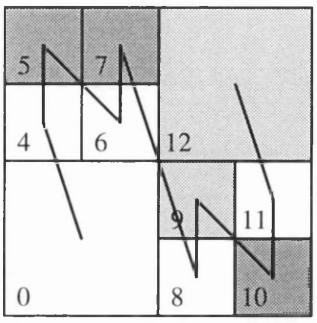
5.3.5 The Linear Quadtree Data Model

The quadtree is a commonly used data model for handling raster data. It is formed by recursively subdividing each non-homogeneous quadrant until all quadrants are homogeneous with respect to a selected property, or until a predetermined cut-off depth is reached [AGI, 1994]. The quadtree data model is particularly useful when the raster data are relatively homogeneous and do not require frequent updating. As such, the quadtree data model can provide the properties of variable spatial resolution and more efficient storage of data. It is also regarded as an efficient indexing scheme for spatial data. As a result, in addition to representing the raster data, the quadtree data model is often used in applications such as the building of the pyramid model required to provide a multiple level representation of image data, together with spatial indexing and searching of the geographical data [Aronoff, 1989; Star and Estes, 1990; Peuquet, 1991; Laurini and Thompson 1992]. Several variants of the quadtree have been developed such as the region quadtree, point quadtree, PM quadtree, *etc.* A detailed overview of these various quadtree models can be founded in Samet [1989].

In the persistent IGIS, a particular form of the quadtree - the linear quadtree - has been adopted to deal with the raster data. The linear quadtree data model employs a locational reference scheme that represents the relative location of the two-dimensional kind by a one-dimensional system. The linear quadtree data model is very widely used in current geo-processing systems because of its efficiency in terms of performance and its simplicity in terms of implementation, as explained below.

5.3.5.1 The Concept

Various locational references schemes are possible for addressing a quadtree. Peano ordering (or N ordering) is probably the most widely known because it is particularly convenient for computer implementation. The principle of the Peano ordering is to use a one-dimensional path, which consistently orders the four quadrants at each level in a SW, NW, SE, NE sequence, to thread each quadrant only once through the two-dimensional space of the data set [Laurini and Thompson, 1992]. Fig. 5.10(a) illustrates the scheme of Peano ordering for quadtree quadrants.



(a) Peano ordering of quadrants

Peano Key	Side Length Code	Attribute Code
5	1	a
7	1	a
9	1	b
10	1	a
12	2	b

(b) Quadtree table for Peano ordering

Figure 5.10 A linear quadtree ordered by Peano key

The numerical coding for identifying each quadrant using integers is called the Peano key. Conventionally, only those quadrant blocks which contain information are included in the appropriate table, as shown in Fig. 5.10(b). Thus each non-white quadrant (nos. 5, 7, 9, 10 and 12 in Fig. 5.10(a)) is included as a record consisting of three items: its Peano key, side length and attribute code. The side length code often represents the number of the smallest quadrant size present on the side of the quadrant block.

The Peano ordering of quadrants has several advantages. Principally, it employs single dimension addressing instead of using row and column identifiers to identify quadrants, so it requires less storage space. In addition, it can facilitate the retrieval of data from storage devices because usually neighbouring quadrants in quadtree space are stored close together. Furthermore, the Peano key may be used as a spatial key because the transformation between the Peano key and the coordinate pair (x, y) can be easily be performed by bit interleaving. This will be described in Chapter 7 when dealing with spatial indexing. As with other data models, there are various pros and cons to be considered when deciding to adopt or employ the linear quadtree data model. One of the major disadvantages is the time taken to create and modify a linear quadtree, especially for complex areas [Aronoff, 1989; Laurini and Thompson, 1992]. However, the linear quadtree data model is generally regarded as a good choice for handling geographical data which require few changes, *e.g.* the storage of processed images used for backdrops or the spatial indexing of base maps.

5.3.5.2 The Type System

In terms of handling raster data, three basic entities can be identified in the linear quadtree data model. They are the *raster*, the *quadrant* and the *attribute* entities. The *raster* entity is used to represent a raster image before decomposing it into quadrants or after reconstructing it from its quadrant components. Because the representation of a raster image can be handled by the grid cell data model (see Subsection 5.3.4), thus the *raster*

entity may be excluded from the linear quadtree data model. The *quadrant* entity consists of the Peano key (*peano_key*), the side length (*side_length*) and the attribute code (*attr_id*) of a quadrant. The *attribute* entity contains the descriptive information for each attribute code. The entity-relationship diagram of the linear quadtree data model is illustrated in Fig. 5.11.

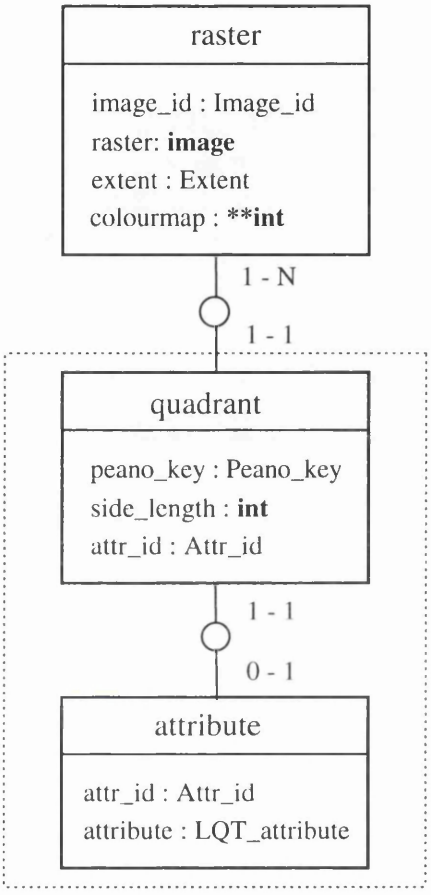


Figure 5.11 The entity-relationship diagram of the linear quadtree data model

The dotted-line block in Fig. 5.11 consists of two entities essential for building the linear quadtree data model. The corresponding type system may be declared as follows: -

```
type LQT_quadrant is structure(side_length: int; attr_id: Attr_id)
type LQT_peano_quadrant is Map[Peano_key, LQT_quadrant]
type LQT_aid_attribute is Map[Attr_id, LQT_attribute]
type LQT_DM is structure(quadrant: LQT_peano_quadrant; attribute: LQT_aid_attribute;
                         extent: Extent; colourmap: **int; pixel_size: real; depth: int)
```

where *side_length* is the number of pixels on the side of the quadrant block;

attr_id is a cross-reference to the attribute table;

quadrant and *attribute* each represent a table for the corresponding entity;

LQT_attribute is a data type for the representation of the descriptive information about quadrants;

extent indicates the ground coverage of the image;

colourmap provides a colour look-up table for the display of the image;

pixel_size is the size of the pixel in terms of its ground length; and

depth is the depth of the image in terms of the number of bit planes used for its implementation.

5.4 Creating An Integrated Geographical Database

As has been noted in Section 4.6, four forms of geographical data have been considered in the database design of the persistent IGIS. The main component in the persistent IGIS database environment is the *Processed* database which has been designed to store processed vector map data and raster image data in various data models. In the *Processed* database, basemaps and baseimages which provide fundamental spatial information for geographical data handling constitute the major part of the data. In terms of constructing a basemap database, each basemap may employ any one of the three vector data models discussed in Section 5.3; thus it can be represented as a bulk value of the type *SP_DM* or *LN_DM* or *PB_DM*. In addition, all basemaps may be grouped together into an aggregated data value, *i.e.* regarding it as a basemap library. This can be carried out by declaring a type system for dealing with all basemaps as follows: -

```
type Basemap_DM is variant(spaghetti: SP_DM;
                           link_node: LN_DM;
                           polygon_based: PB_DM)
type Basemap is structure(data_model: Basemap_DM; attribute: Basemap_attr)
type Base_Maps is Map[Map_id, Basemap]
```

where *spaghetti, link_node* and *polygon_based* identifiers mean that a map data set may use the spaghetti or the link and node or the polygon-based data model;

data_model represents the data model of a specific basemap;

attribute may contain general descriptive information about a particular basemap such as its map scale, map projection, geodetic reference system, geodetic datum, data accuracy, the date of its creation, the source of the data, the data acquisition method, *etc.*

Similarly, each processed baseimage may use either the grid cell or the linear quadtree data model and can be represented as a bulk value of the type *GC_DM* or *LQT_DM*. Also, all

processed images can be aggregated into a baseimage library by the declaration of a type system as follows: -

```
type Baseimage_DM is variant(grid_cell: GC_DM;
                             linear_quadtree: LQT_DM)
type Baseimage is structure(data_model: Baseimage_DM; attribute: Baseimage_attr)
type Base_Images is Map[Image_id, Baseimage]
```

where *grid_cell* and *linear_quadtree* identifiers will indicate whether an image data set will use the grid cell or the linear quadtree data model;

data_model represents the data model of a baseimage;

attribute may contain general descriptive information about a baseimage such as its image resolution, the date of its creation, the source of data, the data acquisition method, *etc.*

Having created the above type systems, two variables *base_maps* and *base_images* of type *Base_Maps* and *Base_Images* may be created for the storage of basemaps and baseimages respectively as follows:

```
! program name: mk_processed_db.N
!                making a database environment for the storage of basemaps and baseimages
type Map[A, Z] is MapRep1to1[A,Z]
use PS() with IO, User, GlasgowLibraries: env; environment: proc(→ env) in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps: env in
use Maps with m_empty: proc[A, Z](proc(A, A → bool), proc(A, A → bool) → Map[A,Z]) in
use User with Library, Database: env in
use Library with General: env in
use General with eq_str, lt_str: proc(string, string → bool) in
use Database with Processed: env in
use IO with writeString: proc(string) in
begin
   ! the base_maps component
   if Processed contains base_maps then
      writeString("The Processed environment already contains base_maps, no action taken.'n")
   else
   begin
      in Processed let base_maps := m_empty[Map_id, Basemap](eq_str, lt_str)
   end
   ! the base_image component
   if Processed contains base_images then
     writeString(" The Processed environment already contains base_images, no action taken.'n")
   else
   begin
       in Processed let base_images := m_empty[Image_id, Baseimage](eq_str, lt_str)
   end
```

end

This program checks whether the *Processed* environment already contains the required components: - variables *base_maps* and *base_images*. If neither of them exists, then an empty database is created for it and is bound to the *Processed* environment. Thereafter, the *base_maps* and the *base_images* can be used to store all the processed forms of geographical data.

The databases for the other forms (raw, interim and derived) of geographical data can be treated in a similar way. For example, the *Raw* database may hold a group of *raw_maps* collected in the spaghetti data model as well as a group of *raw_images* in the grid cell data model. The *Interim* database may comprise a collection of map and image data (*interim_maps* and *interim_images*) restructured in the different forms suitable for digital mapping and image processing. Finally the *Derived* database will contain the specific data sets generated from the *Processed* database for particular applications. Examples might be that all *road_maps* would be organised in the link and node data model for network analysis, that all *landuse_images* are built with the linear quadtree data model for overlay analysis and so on. Fig. 5.12 illustrates the multiple modelling of geographical data integrated within the *Database* environment in the persistent store.

The *Database* environment contains four databases - *Raw*, *Interim*, *Processed* and *Derived*. Each database is used to store a particular form of geographical data which may comprise different vector and raster data types embedded within various data models. Thus the overall *Database* environment represents an integrated geographical database in the persistent IGIS.

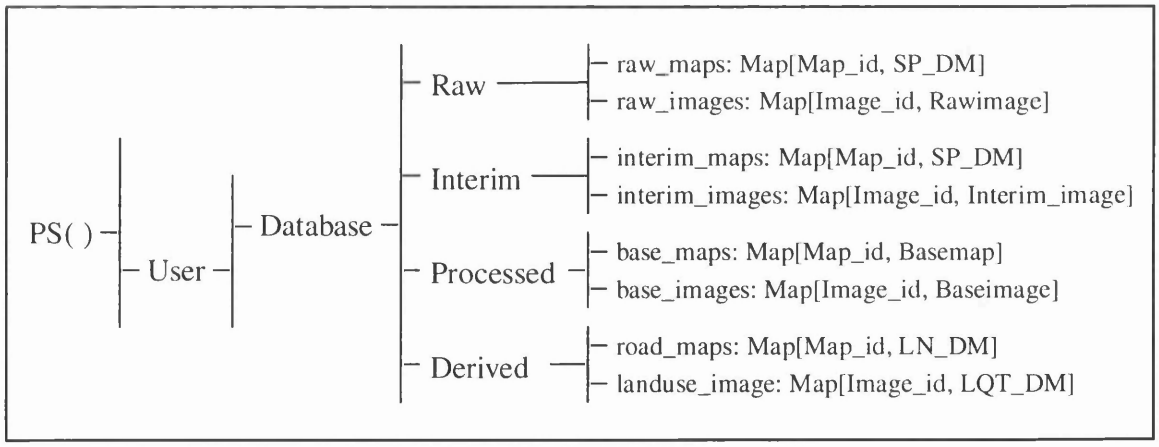


Figure 5.12 An example of the multiple modelling of geographical data integrated in the *Database* environment within the persistent store

5.5 Constructing An Integrated Geographical Database

In the previous section, the creation of new and empty databases in the persistent store for the construction of various forms of geographical data has been discussed. This section is concerned with the construction and entry of geographical data into these databases. As has been noted earlier, the *Processed* database is the key component of an integrated geographical database in the persistent IGIS. In addition, the acquisition of available geographical data from mapping agencies is a common practice. Very often, these data have been processed and stored using an exchange file format. Therefore, this section concentrates on how to build the *Processed* database component of the integrated geographical database. The same procedure may apply to other databases for the construction of different forms of data.

The construction of the *Processed* database requires the development of an import module which comprises a set of data transformation programs in the persistent IGIS. Each program is used to read a particular exchange file format such as NTF, TIFF, FBFF (Flat Binary File Format), *etc.* and to convert the data into persistent objects represented with an appropriate data model, *i.e.* one of the type systems discussed in Section 5.3. The persistent objects are then saved in the persistent store. Figure 5.13 illustrates the process of importing exchange file formats into the *Processed* database of the persistent store.

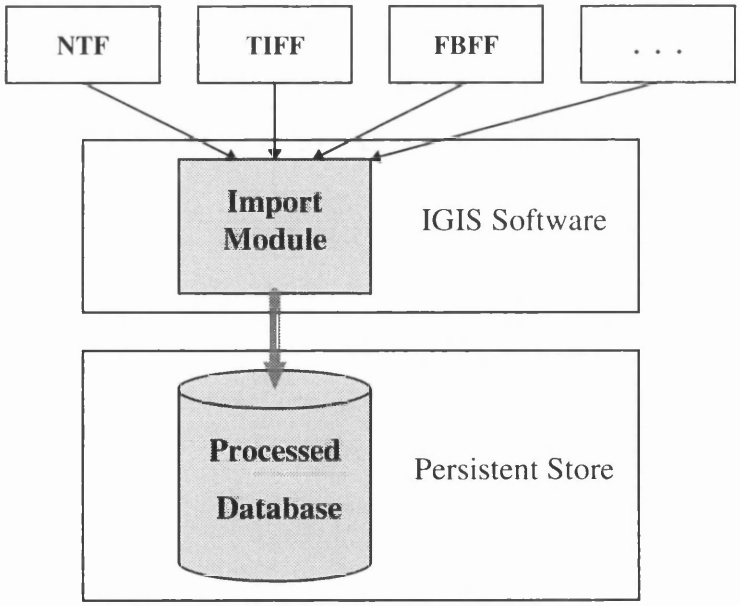


Figure 5.13 The process of importing exhange files into the persistent store

In order to carry out experiments with an integrated geographical database, a set of sample data covering the Port Talbot area in South Wales has been used. This data set consists of three vector map files and two raster image files. The vector map files are mainly used for

the construction of different data models in a database, whereas the raster image files are aimed at handling different forms (raw, interim and processed) of geographical data. The vector files are stored in the same NTF v 2.0 format, but use different data models (the spaghetti, the link and node and the polygon-based models respectively); whereas both raster files employ the same data model (the grid cell), but are stored in different file formats (FBFF and TIFF respectively). Table 5.1 gives a summary of the characteristics of the different components of the test data set. It should be noted that this data set will also be used for the experiments described in Chapter 6 and 7.

File Name	SS8087	270190	2144	PTBAND1	SS88SW
Data Type	Vector	Vector	Vector	Raster	Raster
File Format	NTF v 2.0 level 2	NTF v 2.0 level 3	NTF v 2.0 level 3	FBFF 8-bit	TIFF 8-bit palette colour uncompressed
Data Model	Spaghetti	Link and Node	Polygon-based	Grid Cell	Grid Cell
Coverage (km)	1 x 1	5 x 5	25 x 25	20 x 20	10 x 10
Coordinate / Pixel Resolution (m)	0.10	1	0.1	25	5
Map Series	Landline	OSCAR	Boundary-line	–	–
Data Source	Surveyed at scale of 1 : 2,500.	Derived from maps at source scales of 1 : 1,250, 1 : 2,500 and 1 : 10,000.	Derived from the definitive 1 : 10,000 boundary records.	Extracted from a Landsat TM scene 204/24, 22nd July 1984. 800 x 800 pixels	Scanned from maps at scale of 1 : 50,000. 2,000 x 2,000 pixels
File Size (bytes)	133,400	210,271	390,665	640,000	4,005,880
Supplier	Ordnance Survey	Ordnance Survey	Ordnance Survey	NRSC	MR-Data Graphics

Table 5.1 The description of the test data set

Based on the database environment presented in Fig. 5.12, the *Processed* component may contain a collection of vector map data in the variable *base_maps* as well as a collection of raster image data in the variable *base_images*. The organisation of the vector map data and the raster image data into these two variables has been carried out for the test data set and will be described respectively in the two subsections that follow.

5.5.1 Organising Vector Map Data

As has been shown in Fig. 5.13, a data translation program needs to be developed for the conversion of the NTF data into the Napier88 data types. This subsection first gives a brief description of the NTF exchange format. This is followed by the process of constructing persistent data objects for the three vector map files.

5.5.1.1 An Overview of NTF v 2.0

The National Transfer Format (NTF) v 2.0 is the standard format [BS7567] used in the UK for the exchange of digital map data. It is also being used for all of digital map products created by the Ordnance Survey [Ordnance Survey, 1993a]. NTF supports a family of five data models (simple spaghetti, complex spaghetti, link and node, partial topology and full topology) as well as user-defined data models. Thus it may be used for a wide variety of applications and for transfers of data of varying degrees of complexity. In order to provide the features of efficiency and easy of use, five levels of complexity have been defined in NTF to accommodate various needs for data transfer. The characteristics of each level can be summarised as follows [BSI, 1992]: -

Level 1 is used for simpler types of vector data. Points, lines and texts are separate entities, and each may be given one feature code and one attribute value;

Level 2 is also used for simpler types of vector data. However, it allows the addition of many attributes to the point, line and text entities. Text may be linked to a feature as an attribute;

Level 3 supports a variety of data models that may include network data, polygons, semantic relationships and complex features;

Level 4 allows the transfer of data using either a full topological model or a link and node data model; and

Level 5 is a user definable format and is intended mainly for use with highly specialised data sets that do not fit easily into levels 3 or 4.

A NTF transfer set comprises a volume header and one or more databases which are further subdivided into sections. The volume header provides information about the overall transfer set, such as the date of creating the transfer set; a unique reference number for the transfer set; the NTF level, and so on. Each database commences with a database header record followed by any or none of the records. The database header record gives brief information about a database, such as the database name, the data dictionary, the feature classification scheme, the data quality, the type of the data model, *etc.* Each section contains a section header record and the section data (data records). The section header record, which may be followed by the section quality record and/or the data quality record, contains information

which is essential for interpreting and processing some of the field in the data records. The section data contains the data about features, topology and geometry, and their relationships. The overall NTF file structure is illustrated in Fig. 5.14.

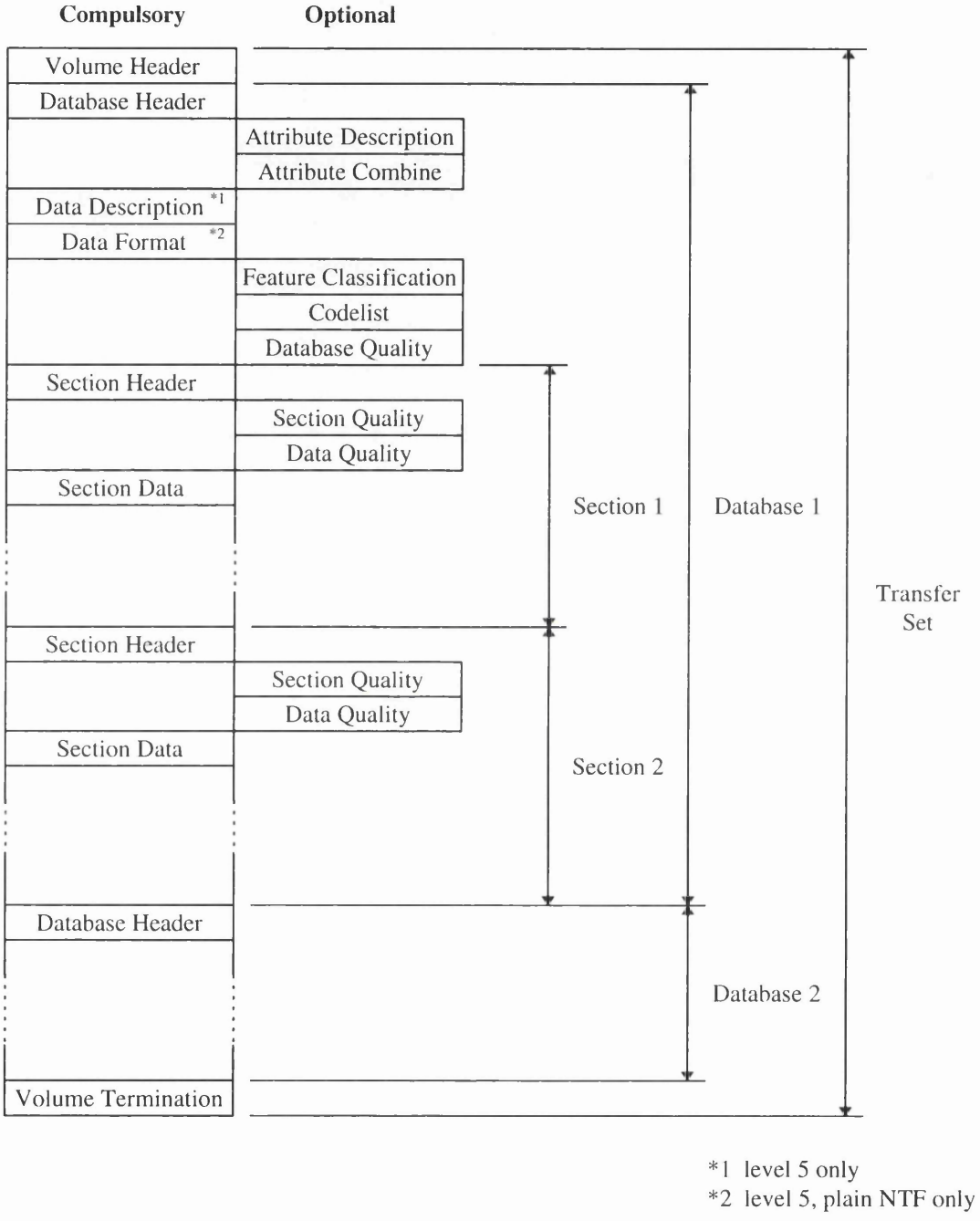


Figure 5.14 The NTF file structure [BSI, 1992]

5.5.1.2 Constructing Vector Map Data

Before constructing persistent objects, data types for handling attribute should be defined. The three files SS8087, 270190 and 2144 included in Table 5.1 use the spaghetti, the link

and node and the polygon-based data models respectively. The attribute data types required specifically for dealing with NTF files may be declared as follows: -

The spaghetti data model

```
type Landline_point_attr is structure(orient: real; symbol_code: int)
type Contour_point_attr is real      ! height
type SP_point_attr is variant(landline: Landline_point_attr; contour: Contour_point_attr)
type Contour_line_attr is real       ! height
type SP_line_attr is variant(landline: null; contour: Contour_line_attr)
```

where *orient* is used to define the orientation of the point symbol;

symbol_code is the code of the point symbol;

landline represents digital maps at scales of 1:1,250, 1:2,500 and 1:10,000;

contour represents digital contour maps at the scale of 1:50,000.

The link and node data model

```
type LN_attr_ssm is structure(RB, RU: bool; OR: real; PN, NU: string)
type LN_attr_oscar is structure(SY: int; LL: real; SC, PN, RN, FW: string)
type LN_attr is variant(small_scale_map: LN_attr_ssm; oscar: LN_attr_oscar)
```

where *RB* denotes the representative point bounded by a linear feature;

RU denotes the representative point not bounded by a linear feature;

OR is used to describe the orientation of a point feature from Grid East, anti-clockwise. If absent, the feature is not deemed to have an orientation;

NU represents numbered features;

SY represents the date of digitisation - yymmdd;

LL represents the length of the chainage link;

SC represents the source scale; possible values are : "A" = 1:1,250 or 1:2,500, "B" = 1:10,000, "C"= 1:50,000, "D" = 1:100,000+;

PN represents any name, ended by a "\";

RN represents any valid road number. This must start with M, A or B - and may be ended with (T) or (M);

FW represents the form of the road. "D" = Dual-carriageway, "R" = Roundabout;

small_scale_map represents digital maps at scales of 1:625,000 and 1:250,000;

oscar represents **O**rdnance **S**urvey **C**entre **A**lignment of **R**oads (OSCAR). It is a digital road network derived from Ordnance Survey maps at scales of 1:1,250, 1:2,500 and 1:10,000.

The polygon-based data model

```
type PB_attribute is structure(AI, LK, PI, HW, LV: int; HA: real; fc: FC;
                               NM, OP, SD, CT: string)
```

where *AI* is the identifier of an administrative area;
LK is a link identifier;
PI is a polygon identifier;
HW is a mean high water flag;
LV is the level of an administrative area type;
HA is the area (in hectares) of polygons in a tile;
fc is a feature code;
NM represents any name;
OP represents the *opcs-code* for city, district and ward only;
SD is the superseded date;
CT is the change type.

Based on the NTF file structure shown in Fig. 5.14 and the type systems designed in Subsections 5.3.1 ~ 5.3.3, a Napier88 program has been developed for the import of an NTF file into the *Processed* database - this will be described later in Chapter 8. In general, data conversion from one format to another is a straightforward task. However, the import of data objects into databases involves more complicated structuring of data than the conventional file format conversion. Fig. 5.15 is a flowchart illustrating the primary steps required in this data import process. The dotted-line blocks (1 and 2) represent the two major parts needed in the data import program. They can be outlined as follows: -

1. The initialisation of the entity tables, and
2. The construction of data objects into entity tables which form a *basemap* in the *base_maps* table.

The initialisation of entity tables concerns the creation of entity tables with data types conformable to a data model. That is, it involves the construction of a set of new and empty tables needed to hold the data objects for each entity. The respective entity tables needed for the handling of the three data models may be declared as follows: -

Spaghetti data model (SS8087)

```
let sp_pid_point := m_empty[Point_id, SP_point](eq_int, lt_int)
let sp_lid_line := m_empty[Line_id, SP_line](eq_int, lt_int)
```

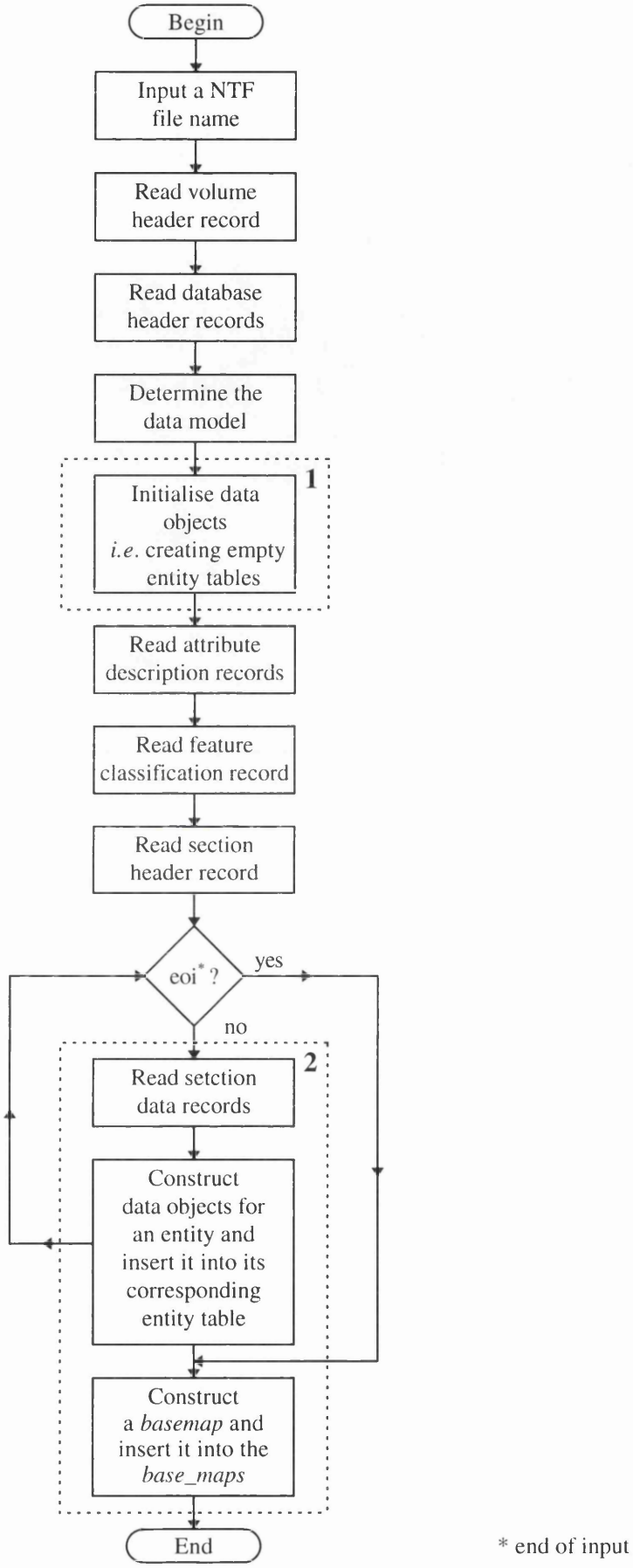


Figure 5.15 The flowchart shows the primary steps of reading a NTF file and converting it into a *basemap* which is placed into the *base_maps* in the *Processed* database.

```
let sp_tid_text := m_empty[Text_id, SP_text](eq_int, lt_int)
let fcd := m_empty[FC, FD](eq_str, lt_str)
```

Link and node data model (270190)

```
let ln_pid_point := m_empty[Point_id, LN_point](eq_int, lt_int)
let ln_lid_line := m_empty[Line_id, LN_line](eq_int, lt_int)
let ln_gid_geometry := m_empty[Geom_id, LN_geometry](eq_int, lt_int)
let ln_aid_attribute := m_empty[Attr_id, LN_attribute](eq_int, lt_int)
let ln_kid_link := m_empty[Link_id, LN_link](eq_int, lt_int)
let ln_nid_node := m_empty[Node_id, LN_node](eq_int, lt_int)
let ln_tid_text := m_empty[Text_id, LN_text](eq_int, lt_int)
let fcd := m_empty[FC, FD](eq_str, lt_str)
```

Polygon-based data model (2144)

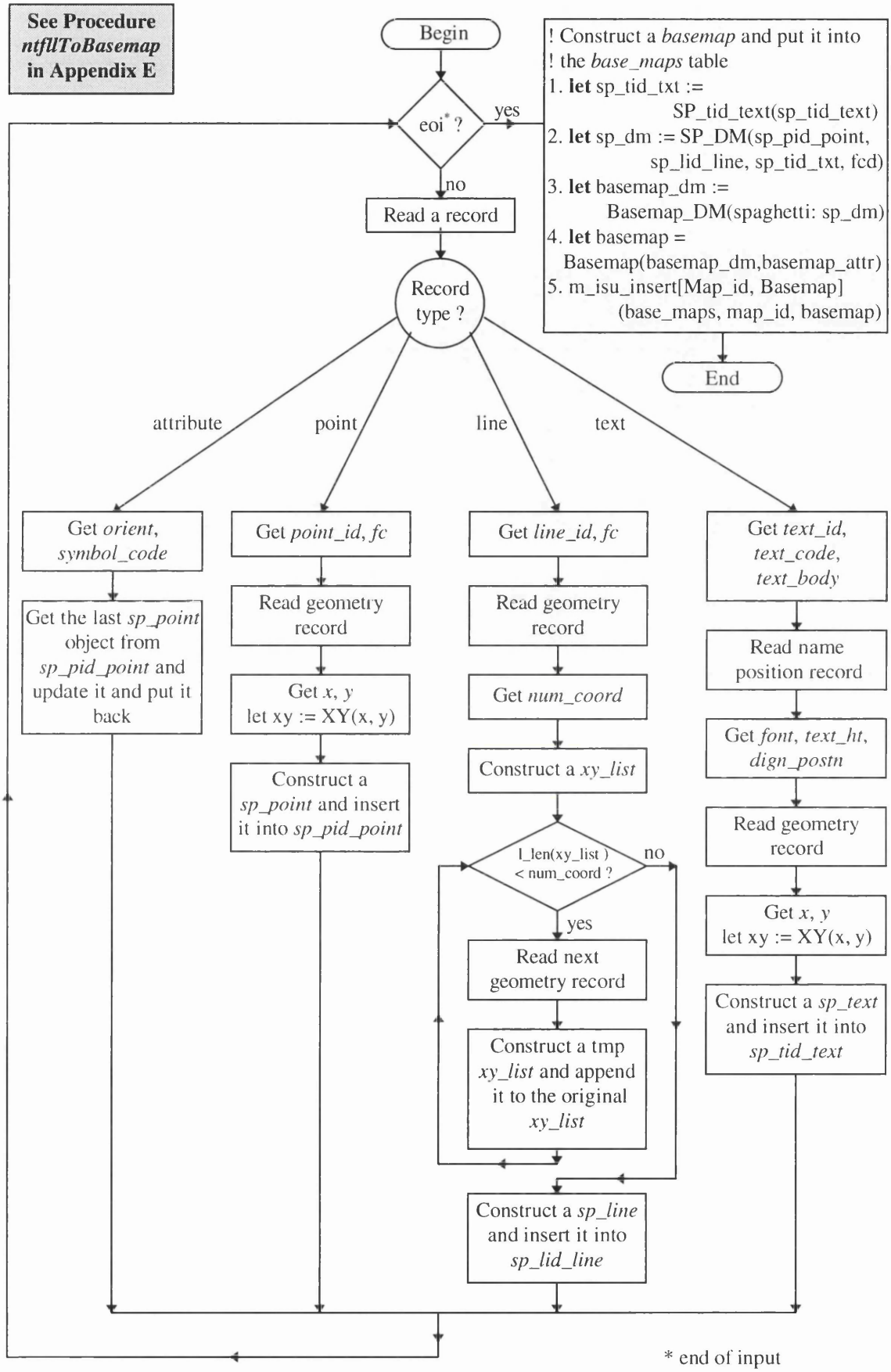
```
let pb_gid_geometry := m_empty[Geom_id, PB_geometry](eq_int, lt_int)
let pb_aid_attribute := m_empty[Attr_id, PB_attribute](eq_int, lt_int)
let pb_polyid_polygon := m_empty[Poly_id, PB_polygon](eq_int, lt_int)
let pb_cid_chain := m_empty[Chain_id, PB_chain](eq_int, lt_int)
let pb_cpolyid_cpolygon := m_empty[Cpoly_id, PB_cpolygon](eq_int, lt_int)
let pb_collid_collection := m_empty[Coll_id, PB_collection](eq_int, lt_int)
let fcd := m_empty[FC, FD](eq_str, lt_str)
```

where *m_empty* is a procedure provided by the *Maps* library for creating an empty *Map* (See Section 3.4.4.2),

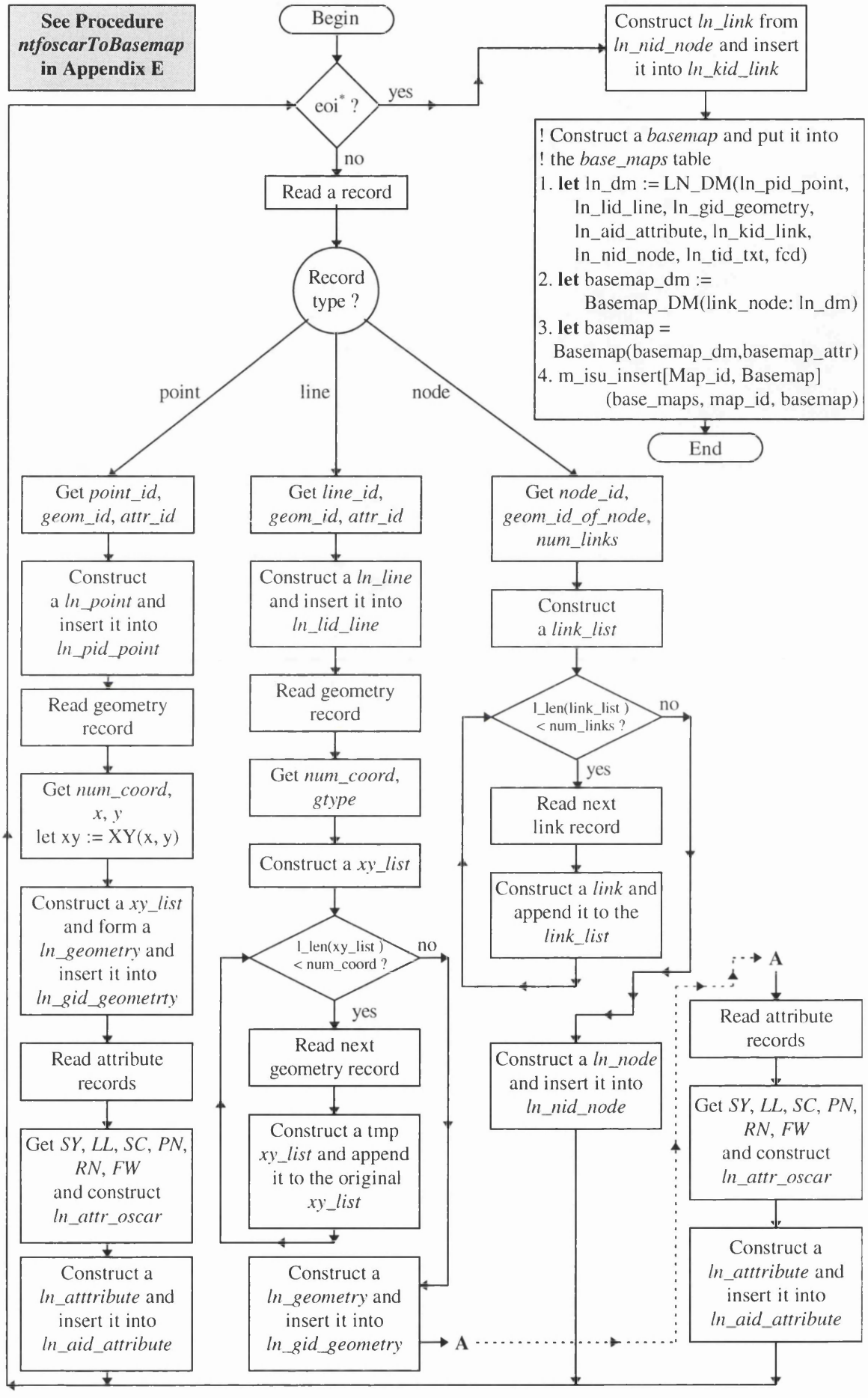
eq_int and *lt_int* are procedures which provide the equality and ordering tests for the domain type of **int**, and

eq_str and *lt_str* are procedures which provide the equality and ordering tests for the domain type of **string**.

The construction of data objects into entity tables deals with the reading of section data records and their conversion into the predefined entities. The constructed entities are then inserted into their entity tables. Finally, all the entity tables are aggregated and formed into a basemap which is inserted into the *base_maps* table. Fig. 5.16 gives an expanded flowchart of the dotted-line block 2 shown in Fig. 5.15 for each of the data sets used, *i.e.* the Landline, the OSCAR and the Boundary-line data.

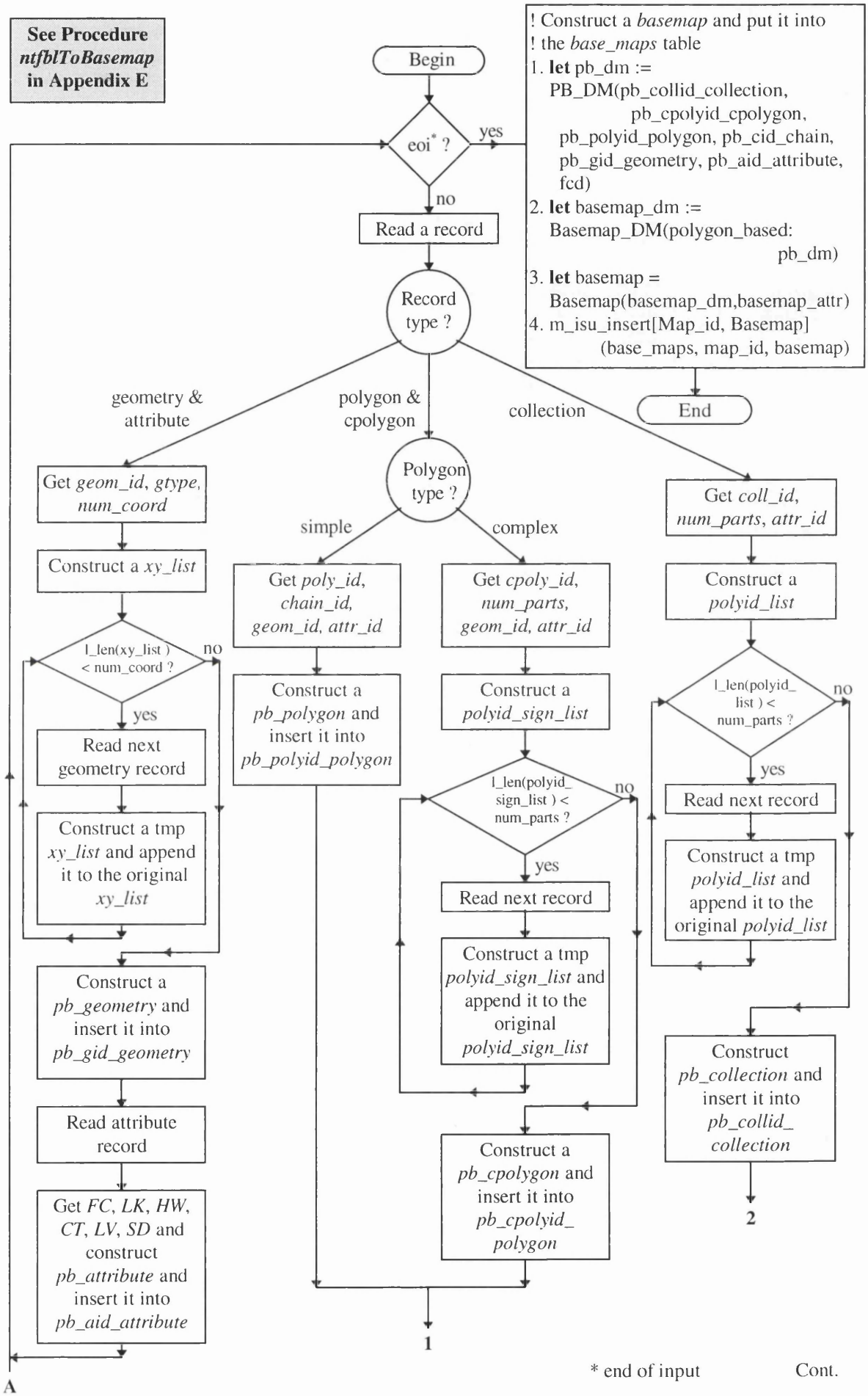


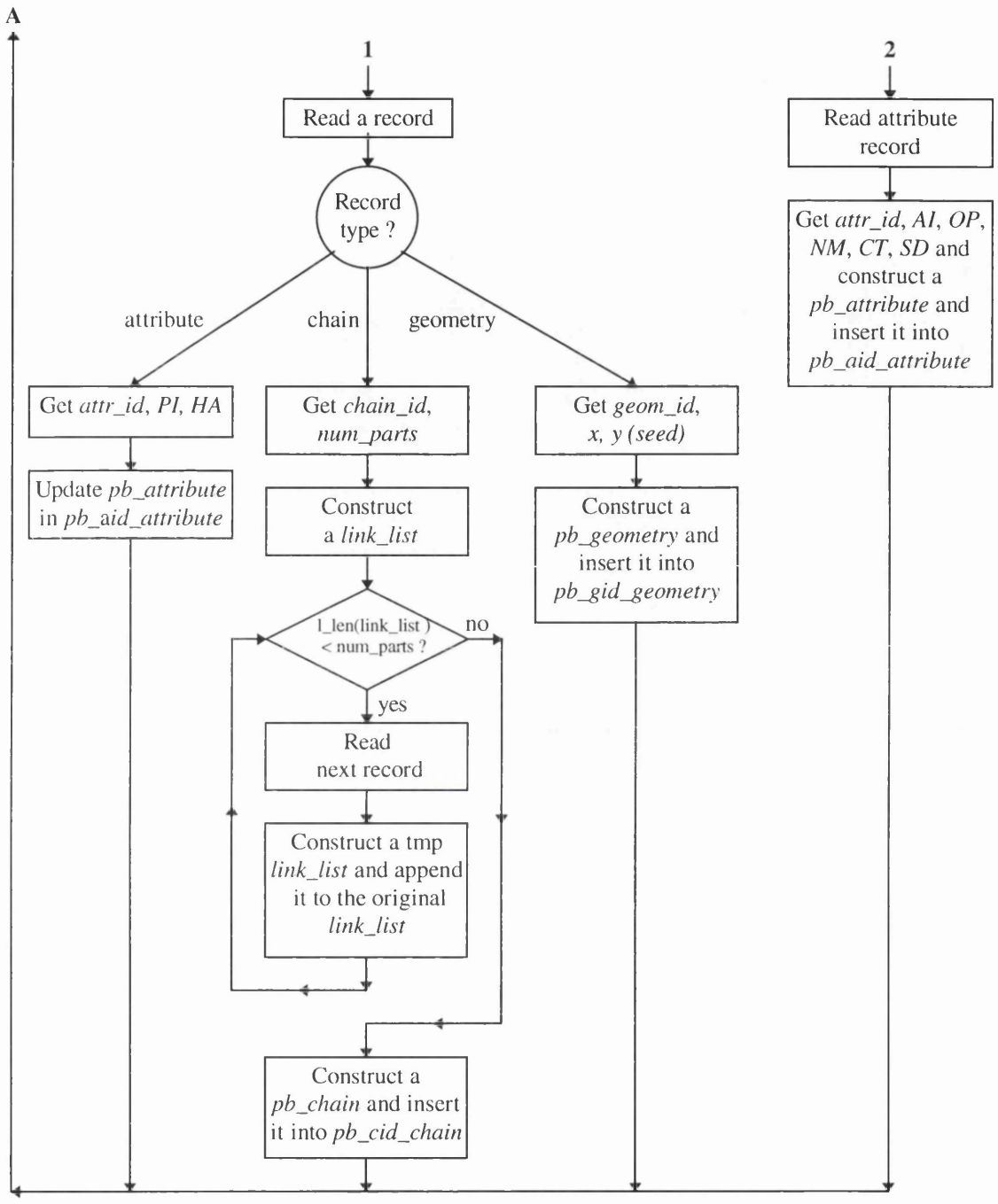
(a) Flow diagram for the "Landline" map data



(b) Flow diagram for the "OSCAR" map data

* end of input


See Procedure ntfblToBasemap in Appendix E
Begin
eoi* ?
yes
no
Read a record
Record type ?
! Construct a basemap and put it into
! the base_maps table
1. let pb_dm :=
PB_DM(pb_collid_collection,
pb_cpolyid_cpolygon,
pb_polyid_polygon, pb_cid_chain,
pb_gid_geometry, pb_aid_attribute,
fcd)
2. let basemap_dm :=
Basemap_DM(polygon_based:
pb_dm)
3. let basemap =
Basemap(basemap_dm,basemap_attr)
4. m_isu_insert[Map_id, Basemap]
(base_maps, map_id, basemap)
End
geometry & attribute
polygon & cpolygon
collection
Get geom_id, gtype, num_coord
Construct a xy_list
l_len(xy_list) < num_coord ?
no
yes
Read next geometry record
Construct a tmp xy_list and append it to the original xy_list
Construct a pb_geometry and insert it into pb_gid_geometry
Read attribute record
Get FC, LK, HW, CT, LV, SD and construct pb_attribute and insert it into pb_aid_attribute
A
Polygon type ?
simple
complex
Get poly_id, chain_id, geom_id, attr_id
Construct a pb_polygon and insert it into pb_polyid_polygon
Get cpoly_id, num_parts, geom_id, attr_id
Construct a polyid_sign_list
l_len(polyid_sign_list) < num_parts ?
no
yes
Read next record
Construct a tmp polyid_sign_list and append it to the original polyid_sign_list
Construct a pb_cpolygon and insert it into pb_cpolyid_polygon
1
Get coll_id, num_parts, attr_id
Construct a polyid_list
l_len(polyid_list) < num_parts ?
no
yes
Read next record
Construct a tmp polyid_list and append it to the original polyid_list
Construct pb_collection and insert it into pb_collid_collection
2
* end of input
Cont.



(c) Flow diagram for the "Boundary_line" map data

Figure 5.16 An expanded flowchart of the dotted-line block 2 shown in Fig. 5.15

Based on the flowcharts presented in Fig. 5.15 and Fig. 5.16, three independent Napier88 procedures - *ntfllToBasemap*, *ntfoscarToBasemap* and *ntfblToBasemap* (See Volume II Appendix E) - may be developed for the import of data derived from Ordnance Survey map series - *i.e.* the Landline, OSCAR and Boundary-line respectively. In fact, Ordnance Survey also provides data derived from other map series such as the 1:625,000, 1:250,000 and

1:50,000 scale series in NTF v 2.0 format. A data conversion procedure for each of them can also be developed in a similar way. Therefore, all these procedures can be combined together into a NTF v 2.0 import module in order to handle OS data of varying degrees of complexity. The actual implementation of a NTF v 2.0 Import module is given in Chapter 8.

Using the above three procedures, vector map files SS8087, 270190 and 2144 can be imported into the *base_maps* table in the *Processed* database. The map file name which is unique may be directly used as the key, *i.e. map_id*, for the *base_maps* table. After importing the test map data into the persistent store, the *base_maps* table contains three entries. Each entry has a map identifier and a set of map data associated with a specific data model. The map data in the *base_maps* table can be loaded into Napier88 programs directly and used in various applications. Therefore, it can be seen that vector map data with different data models has been integrated into the *base_maps* table in the *Processed* database.

5.5.2 Organising Raster Image Data

As has been done for vector map data, so a data translation program is also required for the conversion of each image file format into Napier88 data types. There are dozens of image exchange file formats - industrial or *de facto* standard - available for transferring raster image data between different systems. Unlike map data exchange files which may be in the form of various different and complex data models, image data exchange files often employ a simple data model - the grid cell (see Section 5.3.4). However, raster images in an uncompressed form often require a large data volume. Therefore, many image exchange file formats have been designed with several data compression techniques employed internally to implement a reduction in the overall data volume. An alternative way of implementing data compression is to use general data compression programs such as PKZIP, ZOO, *etc.* to achieve the required storage reduction. However, a compressed image file has to be decompressed back to its original form before any application program can be applied to the image data.

In Table 5.1, the two raster image files are PTBAND1 and SS88SW. The PTBAND1 is in FBFF (flat binary file format), whereas the SS88SW is in TIFF. FBFF, which is the simplest image format, organises a two-dimensional array of image data into an one-dimensional array. FBFF stores image data from left to right and from top to down, starting at the upper left corner of the image. FBFF can be very easily imported into a system. However, the image dimension (width x height x depth) has to be specified when reading a FBFF image, because FBFF does not contain a header for holding this essential information.

By contrast, TIFF was designed specifically to promote the interchange of digital image data. The design of TIFF is so rich and extensible that it is widely used for scanner output and related applications. Since TIFF is one of the most commonly used file formats for the exchange of raster image data, therefore the next subsection first gives a brief introduction to TIFF. Thereafter, the construction of raster image data into the *base_images* table of the *Processed* databases from a FBFF or TIFF file will be discussed.

5.5.2.1 An Overview of TIFF v 5.0

The Tagged Image File Format (TIFF) v 5.0 has been jointly developed by the Aldus and Microsoft Corporations in collaboration with leading scanner vendors and other interested parties. TIFF was designed primarily to reduce the proliferation of proprietary scanned image formats by scanner vendors and desktop publishing software developers [Aldus & Microsoft, 1988]. However, because TIFF is a powerful and flexible format, so it has been widely used in many other applications, *e.g.* in image processing operations, raster map production, *etc.*

In TIFF, four classes have been defined as follows: -

* Class B for bilevel (1-bit) images,
* Class G for grayscale images,
* Class P for palette colour images, and
* Class R for RGB full colour images.

A TIFF file begins with an 8-byte image file header that points to one or more image file directories. These image file directories contain information about the image as well as pointers to the actual image data. The image file header contains the byte order (bytes 0 -1), the TIFF version number (bytes 2-3) and the offset (in bytes) of the first image file directory (bytes 4-7). The image file directory (IFD) may be found at any location in the file after the header and must be followed by the image data that it describes. An IFD consists of a 2-byte count of the number of entries (*i.e.* the number of fields), followed by a sequence of 12-byte field entries, and followed by a 4-byte offset of the next image file directory (or 0 if none follow). Each 12-byte IFD entry contains a tag (bytes 0 -1), a field type (bytes 2-3), the length of the field (bytes 4-7) and the value offset (bytes 8-11) [Aldus & Microsoft, 1988]. The overall TIFF file structure is illustrated in Fig. 5.17.

The information content of an IFD can be identified by its tag. For example, IFDs with tag = 256, 257 and 258 represent the width, height and depth of an image respectively. In TIFF, an image may be divided into a number of strips; each strip may be of any length and be at

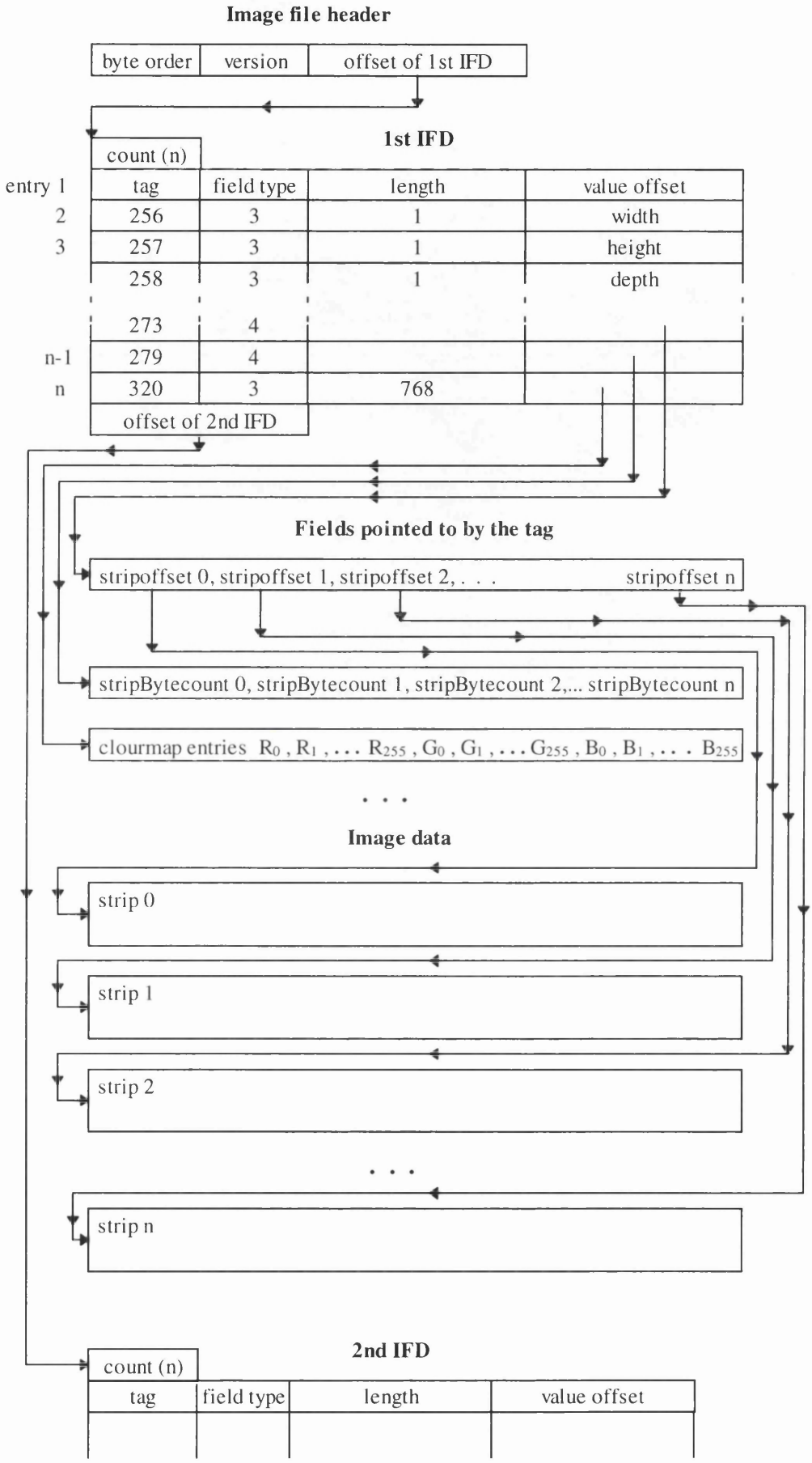


Figure 5.17 The TIFF file structure

any location in the file. Each strip may be located by the byte offset of that strip and the number of bytes in that strip. Strip byte offsets and strip byte counts of all strips can be obtained from IFDs with tag = 273 and 279 respectively. In addition, for palette colour images, an IFD with tag = 320 gives the location of a colourmap in the file. The field type is a code which represents the data type used in an IFD. The code list is as follows: 1 = an 8-bit unsigned integer; 2 = an 8-bit ASCII byte; 3 = a 16-bit unsigned integer; 4 = a 32-bit unsigned integer; 5 = a fraction represented by two 32-bit unsigned integers. The first represents the numerator of a fraction, the second the denominator. The length is specified in terms of data types, not the total number of bytes. The value offset contains either an actual value or an address value pointing to a specific location anywhere in the file.

Since TIFF uses pointers (byte offset) quite liberally, a TIFF file can be read easily from a random access storage device. Apart from this feature, TIFF also provides several data compression techniques including Packitbits compression (for bilevel scanned and paint type files), CCITT Group 1 D facsimile compression (for bilevel data) and LZW (Lempel-Ziv & Welch) compression (for raster images)

5.5.2.2 Constructing Raster Image Data

As has been noted earlier, raster image files often employ a grid cell data model. The conversion of raster image data into Napier88 data types is much simpler than that for vector map data. Depending on the file structure of an image exchange file format, a raster image file may or may not contain basic image information and a colourmap. For example, a FBFF image does not contain information about the image dimension and a colourmap, so it is necessary to provide these essential data manually. By contrast, a TIFF image carries these essential data and other auxiliary data along with the image data; thus they can be read automatically from a TIFF image. Fig. 5.18 illustrates the primary steps required for constructing and inserting various forms of image data into different databases from a FBFF or a TIFF image file.

Depending on the status of an input image, the image data may be converted into the *raw_images*, the *interim_images* or the *base_images* tables in different databases. In general, if the image data requires further image processing operations such as geometric correction, image filtering, contrast stretch, image classification, reduction of an image depth, *etc.*, then the image file should be converted into a *rawimage* because the representation of image data in an integer array would give a better performance during these image processing operations.

See Procedures *fbffToRaw*, *tiffToRaw* and *tiffToInterim* in Appendix E

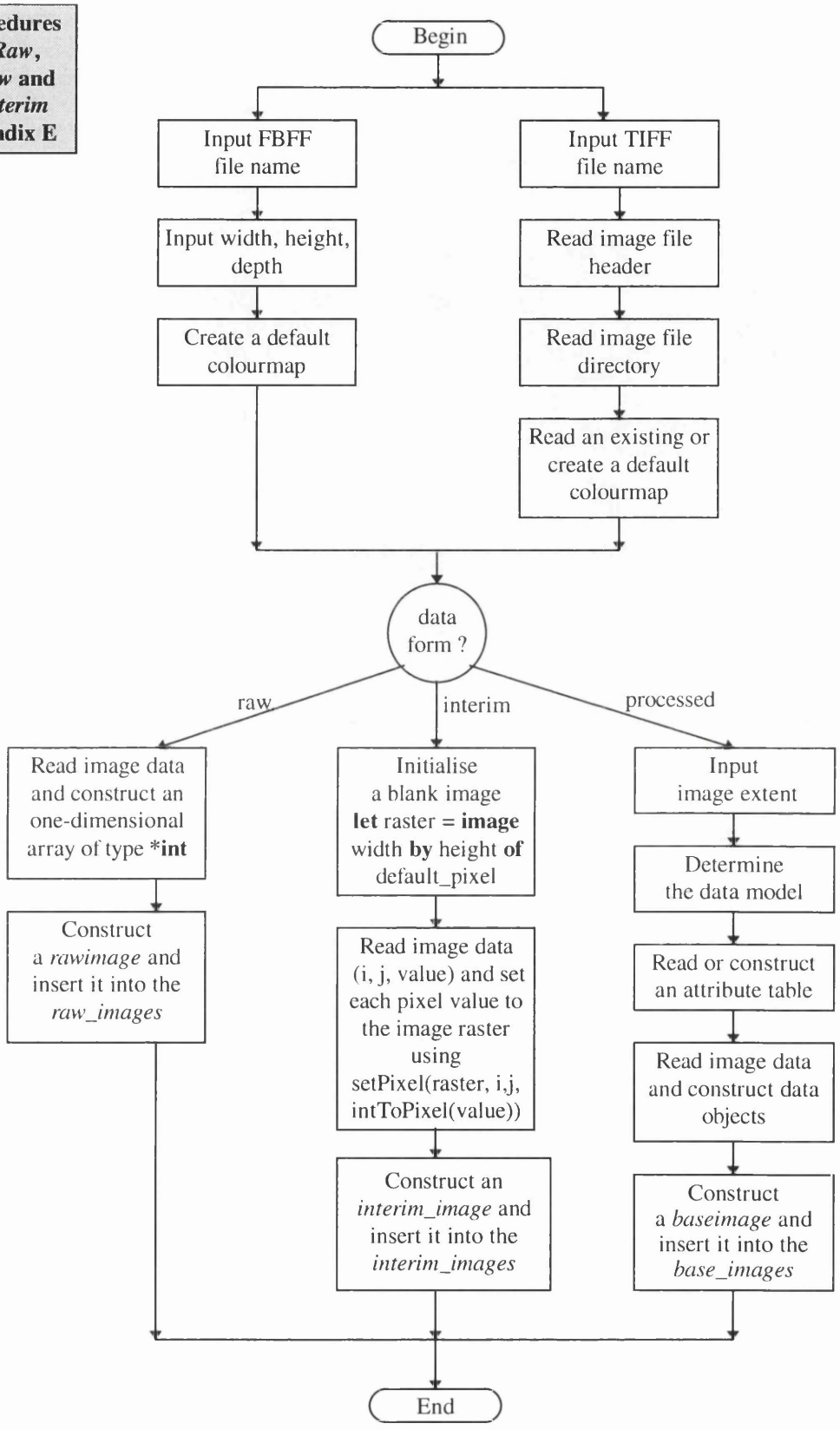


Figure 5.18 The flowchart shows the primary steps required for constructing and inserting various forms of image data into different databases from an FBFF or a TIFF image file

However, if the image data does not need any further image processing operation, but does require some image management operations, *e.g.* to extract an image, to trim an image, to change a colourmap, to overlay two images and so on, then the conversion of the data into an *interim_image* would be more convenient for achieving this purpose.

As for the construction of a *baseimage*, this is a prerequisite for GIS applications. In order to register an image on a map, the ground coverage of an image has to be determined and entered. In addition, an attribute table associated with the image needs to be created. After selecting a data model, the image data can be read and constructed into data objects of predefined Napier88 data types such as the type systems declared in Subsections 5.3.4 and 5.3.5. Finally, a *baseimage* may be constructed and inserted it into the *base_images* table in the *Processed* database.

According to the flowchart presented in Fig. 5.18, six import procedures (2 file formats x 3 data forms) may be developed. Before carrying out the import of two test image files into image databases, the status of each of the two files was first examined. The PTBAND1 file is a band of an extracted Landsat TM scene and has been geo-corrected to fit the OSGB National Grid. However, a contrast stretch has not yet been performed on the image PTBAND1. On the other hand, the SS88SW file is a quarter tile of raster data scanned from the Ordnance Survey's 1:50,000 scale map SS88. Any scale or rotation distortion in the captured image has been removed. It is clear that these two image files require no further image processing operations except that a contrast stretch may have to be performed on the image PTBAND1. However, although both images are in 8-bit form, in order to superimpose both the vector maps and the raster images on an 8-bit graphical display (as will be discussed later in Chapter 6), the depth of each of these two images needs to be reduced. Therefore, two import procedures *fbffToRaw* and *tiffToRaw* (See Volume II, Appendix E) have been developed to convert and transfer both image files PTBAND1 and SS88SW4 into the *raw_images* table.

After importing both images into the *raw_images* table in the *Raw* database, several procedures have been developed to deal with image manipulation and management operations. The linear contrast stretch procedure is first used to perform this operation on the *rawimage* PTBAND1. The image depth reduction procedure is then carried out to reduce the image depth of both the resultant *rawimage* PTBAND1 and the *rawimage* SS88SW from 8-bits to 4-bits. Thereafter, both *rawimages* are converted into *interim_images* by the *rawToInterim* format conversion procedure. The *interim_images* may be previewed and trimmed by the image view and trim procedure in order to extract specific parts of images that are of interest to the user. The extracted images can be further associated with the extent of the coverage of the image data and may be converted and transferred into the *base_images* table using the *interimToBaseimage* conversion

procedure. The overall flowchart showing the import of these two test images to construct the *base_images* is given in Fig. 5.19.

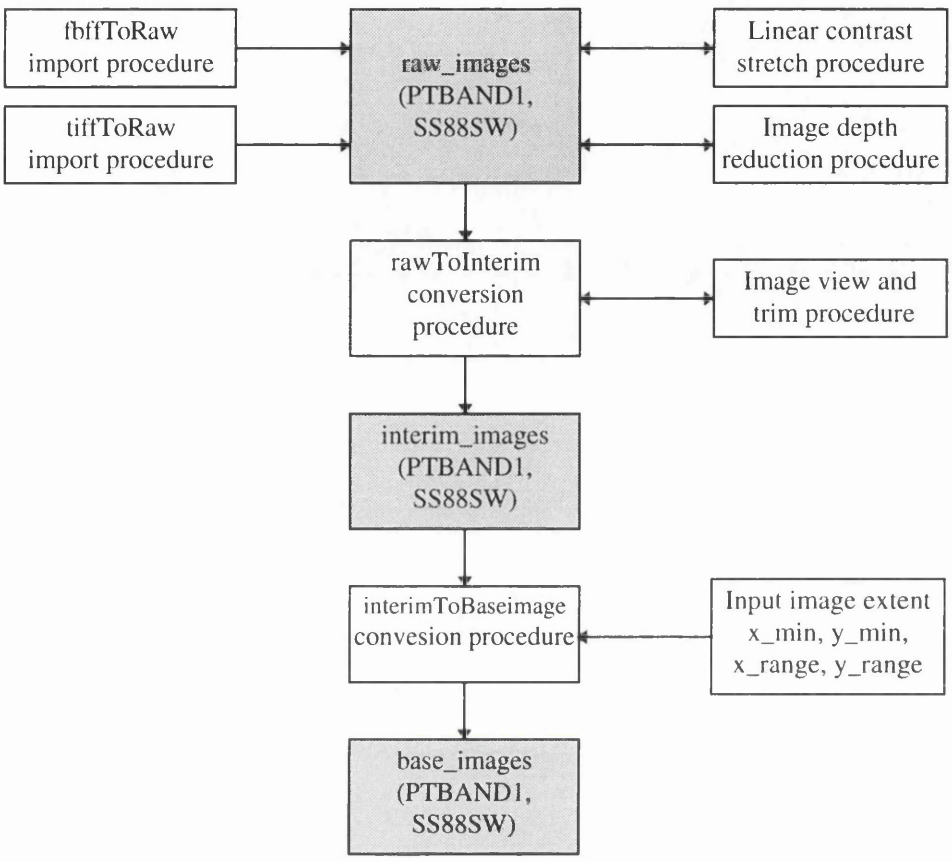


Figure 5.19 The flowchart showing the import of two test images to construct the *base_images* table

Having constructed the *base_images* table in the *Processed* database, the table contains two entries. Each entry has an image identifier, *i.e. imag_id*, and a *baseimage* associated with a data model. In this particular example, both entries employ the same grid cell data model. However, the *base_images* have been transferred from the *Raw* database to the *Processed* database through the *Interim* database.

5.6 Summary

In this chapter, the characteristics of three vector data models and two raster data models commonly used for the representation of geographical data have been discussed. A type system has also been designed for each data model. These type systems form a framework for the creation of a persistent geographical database which is essential for the building of an IGIS.

A set of test data has been selected to construct an integrated geographical database. Vector map data have been organised into the *base_maps* table with different data models, whereas raster image data have been processed in the *Raw* and the *Interim* databases before being entered into the *base_images* table. Both *base_maps* and *base_images* tables are integrated in the *Processed* database. The integrated database now contains the test data which will be used further for the experiments carried out and described in the next two chapters.

CHAPTER 6 : SUPERIMPOSITION AND INTERRELATION OF VECTOR MAPS AND RASTER IMAGES

6.1 Introduction

In the previous chapter, the organisation of both vector map data and raster image data for use in an integrated geographical database has been discussed. The required features of multiple data modelling, and the provision and manipulation of multiple scale maps and multiple resolution images have all been realised in the persistent database environment supported by Napier88. In other words, through the provision of those features, the persistent IGIS has achieved the storage level of integration. However, the provision of both the display and the process levels of integration is as important as the support of the storage integration and indeed such a provision is necessary in order to reach the full degree of integration that is required in an integrated GIS.

The process level of integration is concerned with the concurrent processing of vector and raster data within a single system. In order to provide this feature, a persistent IGIS has to embody a facility allowing the selective retrieval of geographical data. This will result in a requirement that all the geographical data contained in the database will have to be spatially indexed in advance. Since spatial indexing is quite a complex issue, thus the indexing and search of geographical data for data manipulation will be treated specially in Chapter 7. Therefore, this chapter concentrates on an exploration into the possibilities of providing the display level of integration required for a persistent IGIS.

This chapter first deals with the possibilities of displaying vector maps or raster images separately and the procedures required to implement these operations. Next the arrangement of the colours needed for the display of vector maps and raster images is discussed. This is followed by an account of the procedures involved in the superimposition of vector maps and raster images. Finally, the combination and interrelation of vector maps and raster images through the use of a spatial key is also discussed in some detail.

6.2 The Separate Display of Vector Maps or Raster Images

The display of vector maps or raster images on a terminal or a computer monitor is a basic facility which is essential for most GIS operations or applications. Graphical display of maps or images allows the user to interact with and analyse the data which has been queried and to preview the resultant data before the final output is generated either in digital or in hard copy form. Thus the provision of a facility for the separate display of maps or images is a prerequisite for the development of a persistent IGIS. Furthermore, the superimposition

of vector maps and raster images is also based on the support of this facility. Therefore, this section discusses the development of the display functions needed for the viewing of the maps or images required for the persistent IGIS.

In order to display maps or images on the screen, a number of operations have to be carried out. Figure 6.1 gives the flowcharts showing the major steps required for the display of a basemap and a baseimage. These operations may be categorised into two parts:

1. The retrieval of a basemap or baseimage from the *Processed* database, and
2. The display of a basemap or baseimage on the screen.

These are discussed further in the following two subsections.

6.2.1 The Retrieval of a Basemap or Baseimage from the Processed Database

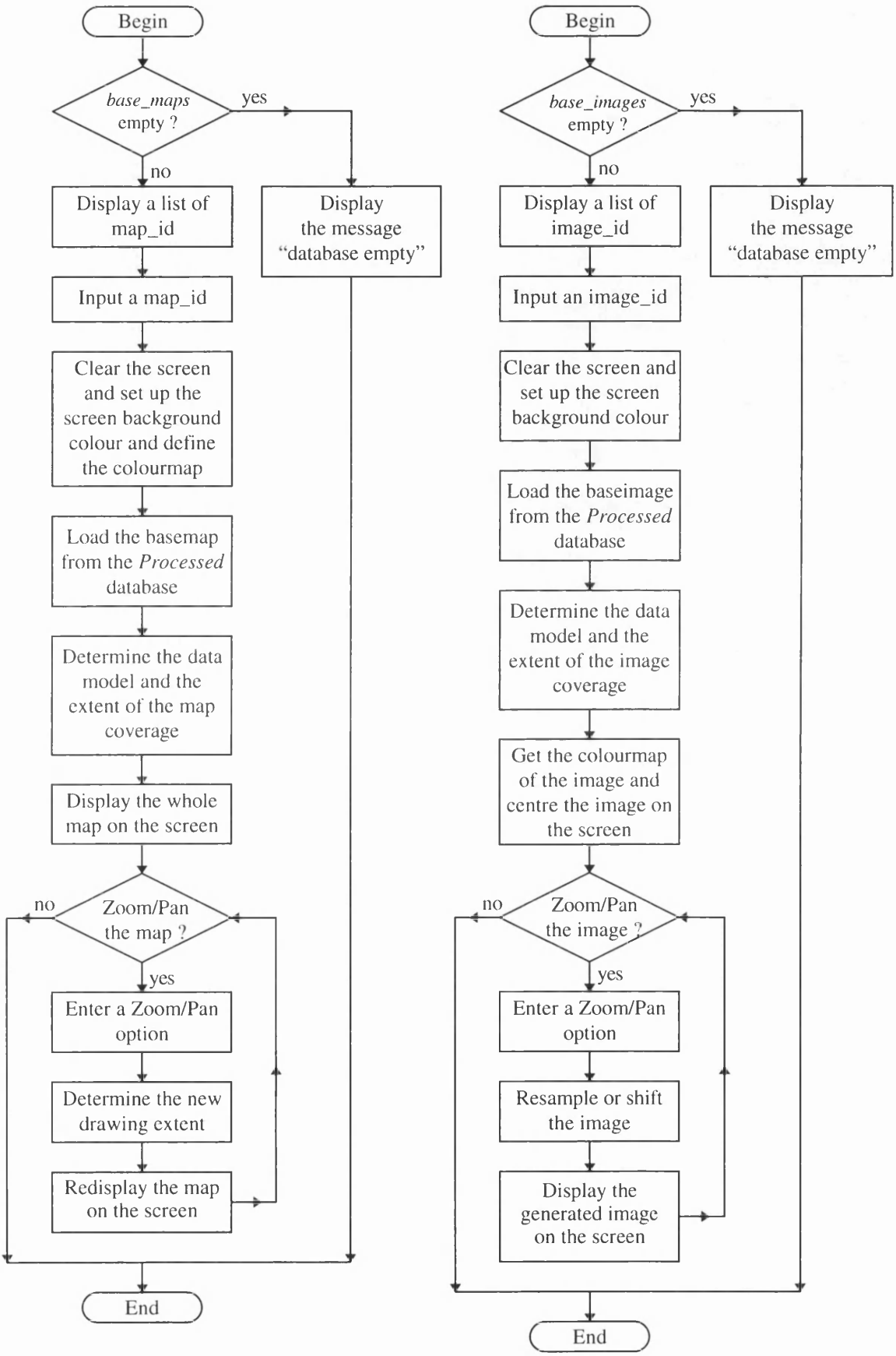
The retrieval of a basemap or baseimage first requires a search of the *Processed* database to be carried out to find out whether the required map or image is available as well as the actual actions of retrieving the data if and when it is found. Since basemaps and baseimages are stored respectively in the *base_maps* and the *base_images* tables of the data type *Map*, the database search and retrieval operations can be performed fairly easily using the procedures *m_isEmpty*, *m_apply*, *m_contains*, *m_find*, *m_copy*, *etc.* provided by the *Maps* Library. For example, the retrieval of a basemap can be carried out by the following operations. First of all, the status of the *base_maps* table may be checked by the statement

```
m_isEmpty[Map_id, Basemap](base_maps)
```

This will result in a Boolean value (**true** or **false**) which indicates whether the *base_maps* table is empty or not. If the *base_maps* table is not empty, a list of the *map_id* entries contained in the table may be obtained. This can be carried out by first creating a procedure, which prints the *map_id* of an entry in the table, and then applying this procedure to every element in the table using the procedure *m_apply*. This operation may be implemented as follows: -

```
let prtMapId = proc(map_id: Map_id; basemap: Basemap)
    { writeString(map_id ++ " "); }
m_apply[Map_id, Basemap](base_maps, prtMapId)
```

The procedure *prtMapId* is used to print a *map_id*. The procedure *m_apply* which applies the procedure *prtMapId* to every element of the *base_maps* table will produce a list of all



(a) The display of a basemap (b) The display of a baseimage

Figure 6.1 The flowcharts for the display of a basemap and a baseimage

the *map_id* entries. Furthermore, the procedures *m_contains* and *m_find* can be used to check that the *base_maps* table contains the specific *map_id* that is required and then to access the basemap data for this particular *map_id*. This operation can be implemented as follows: -

```
if m_contains[Map_id, Basemap](base_maps, map_id) do
    { let basemap = m_find[Map_id, Basemap](base_maps, map_id) }
```

The first statement checks whether the *base_maps* table contains the specified *map_id*. If it does, then the second statement is followed to retrieve the basemap corresponding to this designated *map_id*.

Similarly, a corresponding set of operations to those described above can be applied to search for and retrieve a baseimage from the *base_images* table in the *Processed* database.

After a particular basemap or baseimage has been found in the database, its data model can then be determined. Also, all the entity tables associated with the basemap or baseimage (See Subsections 5.5.1 and 5.5.2) can be copied into the main memory using the procedure *m_copy* provided for this operation in GIS applications.

6.2.2 The Display of a Basemap or Baseimage on the Display Screen

Having retrieved a basemap or baseimage from the database, the whole map or image can then be displayed on the display screen using a set of drawing/display procedures, including *drawPoint*, *drawLineString*, *drawText*, *drawRectangle*, *viewImage* and so on (See Appendix D). These procedures have been developed employing the fundamental vector graphics and raster graphics facilities provided and supported by Napier88 - which have already been discussed in Subsections 3.4.2 and 3.4.3. However, for the user of the system to be able to view or manipulate any part of the map or image, further graphics facilities such as "zoom", "pan", "rotate and "clip" functions need to be developed. In the present implementation of the persistent IGIS, a "zoom" facility for the close-up viewing of a map and a "pan" facility used to roam across the image have been developed. These two basic facilities are essential for the subsequent operations - such as the superimposition of vector maps and raster images and the querying of spatial entities. The development of the graphics capabilities required for zooming a map and panning an image are described in following sections.

6.2.2.1 Viewing and Zooming a Basemap

Before the zoom operation is carried out on a basemap, the whole of the basemap is usually displayed and viewed in a display window on the screen in order to provide an overview of the map content and coverage. This operation requires the fitting of a map into a display window. Depending on the coverage of a map and the aspect ratio (*i.e.* height / width) of a window, a map may fit a window vertically or horizontally or both. This situation can be represented by the two cases shown in Fig. 6.2. The first case shows the situation where a map needs to fit a window vertically, whereas the second case shows the situation where the map has to fit the display window horizontally. The exact fitting of a map into a window - both horizontally and vertically - can be regarded as a special circumstance (*i.e.* $w_m * h_w / w_w = h_m$) of either the first of these two cases (which is treated in Fig. 6.2) or the second case.

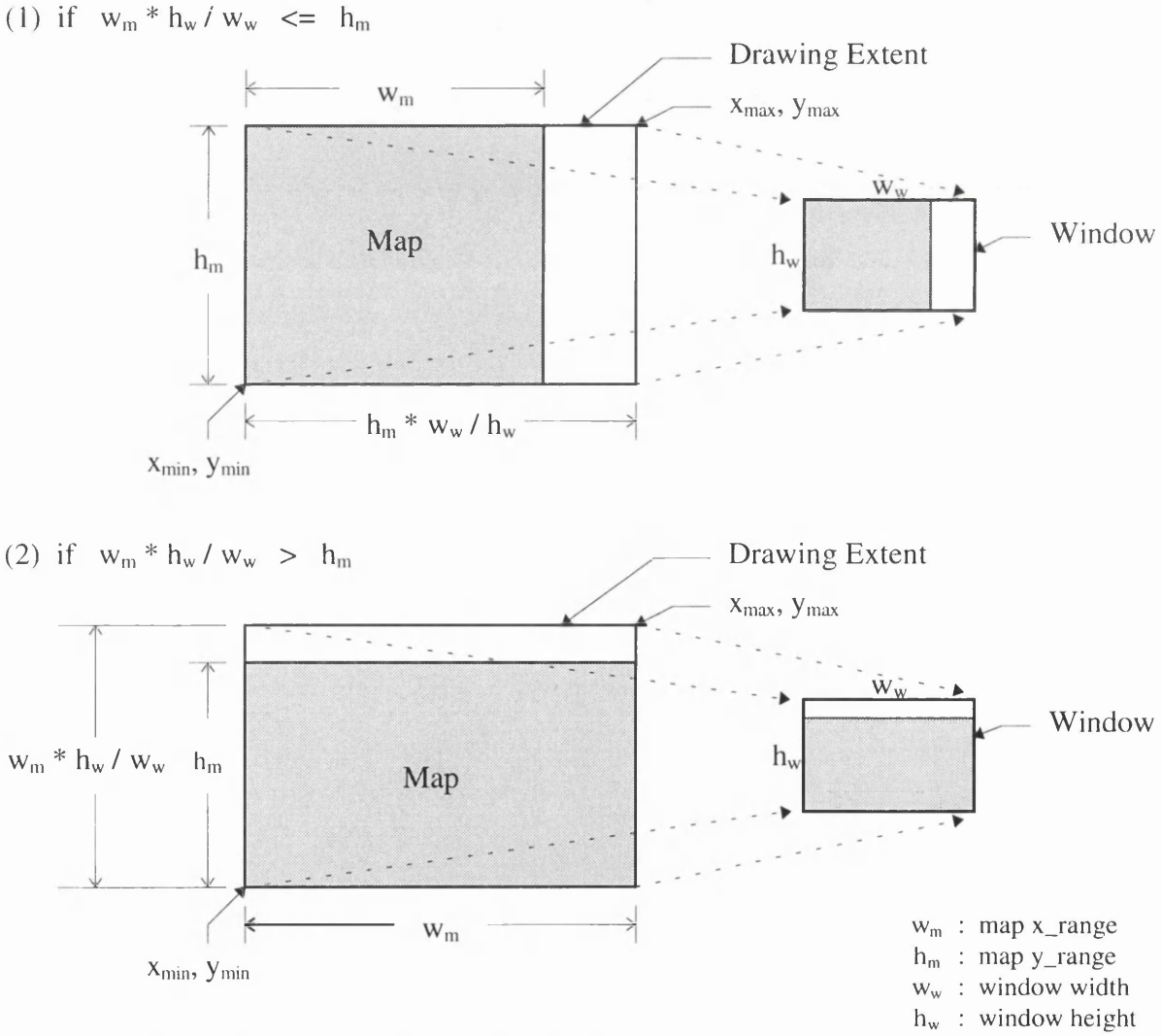


Figure 6.2 The fitting of a map into a display window

The algorithm for handling this fitting operation may be described as follows: -

```
if map_xrange * window_height / window_width <= map_yrange
then
    drawing_xrange = map_yrange * window_width / window_height
    drawing_yrange = map_yrange
else
    drawing_xrange = map_xrange
    drawing_yrange = map_xrange * window_height / window_width
```

Based on the above algorithm, the drawing extent (x_{min}, y_{min}, x_{max}, y_{max}) required for the display of the whole map can be determined, where x_{min} and y_{min} are the ground coordinates of the south-west corner or the origin of the map and $x_{max} = x_{min}$ + drawing_xrange, $y_{max} = y_{min}$ + drawing_yrange. Thereafter, the drawing extent can be used by the drawing procedures to display the whole map in a window. Conceptually, the drawing extent can be viewed as the ground coverage of a "virtual window". Because the virtual window and the display window have the same aspect ratio (*i.e.* drawing_yrange / drawing_xrange = window_height / window_width), it ensures that the displayed map retains the same scale in both the x- and y- directions.

Zooming a map is the operation needed either to magnify the map displayed on the display window to see more detail or to shrink it to view more of the map with less detail. The zooming operation can be achieved in any one of several different ways. It may be carried out in a very simple way that magnifies or shrinks the current display a stated number of times or in a more complex way that permits the user to pan around a box representing the viewing window encompassing the entire generated portion of the map and to enlarge or shrink it in a dynamic manner. Each method of zooming has its pros and cons. Thus the provision of several different options for a zoom command has become a common practice in current CAD and GIS packages. Therefore, a number of zooming options for viewing a map have been implemented in the persistent IGIS. The primary operations involved in each of these zooming options are described in Table 6.1.

In fact, the zooming operation is the determination of a new drawing extent either from the current drawing extent or from the extent of the map coverage. The new drawing extent is supplied to the graphics procedures for the generation of a new display. The map is clipped to the area of the bounding rectangle defined by the drawing extent. The entities within the rectangle are then "mapped" onto the display window. The methods used for the determination of drawing extent for the "Zoom Number (N)", the "Zoom Centre" and the "Zoom Window" options are illustrated in Fig. 6.3.

Based on the concept described above, a procedure *getZoomExtent* has been developed to carry out the zoom operations needed for the viewing of maps (See Appendix D). Having developed the drawing procedures required to display a basemap as well as the zoom

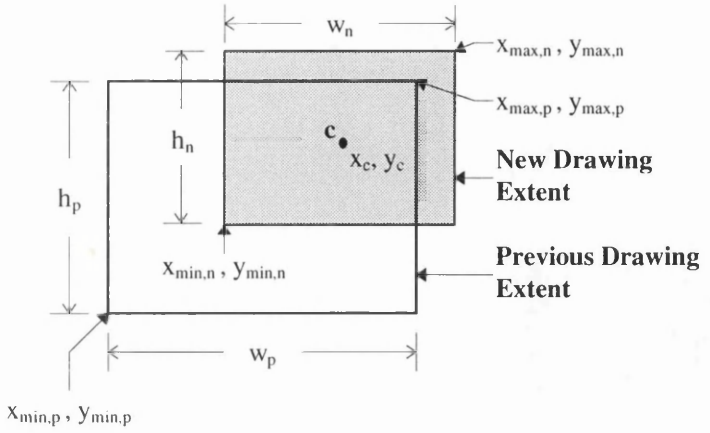
Option	Operation
Zoom All	This places the entire map in a display window. This operation is the same as that needed to display the whole map in a window which has been described above.
Zoom Number (N)	This enters a magnification/reduction number relative to the Zoom All display, *i.e.* Zoom All = Zoom 1.
Zoom Centre	This specifies the centre point of the area of the map to be displayed using a cursor controlled by a mouse and enters a magnification/reduction number relative to the scale of the currently displayed map.
Zoom Window	This designates the rectangular area of the map to be drawn as large as possible within the display window by specifying two diagonal corners using a mouse.

Table 6.1 The primary operations of the different zooming options

procedures to give a close-up view of any part of this map, any basemap may be drawn in a display window and any part of the displayed map can be enlarged or reduced. Figure 6.4 shows the display of the whole of OS map SS8087 in an X-window which has a size of 800 x 600 pixels. The entire window, including the title, frame, client area of the window (*i.e.* the area inside the window), *etc.*, has been captured from the screen and then rotated anti-clockwise 90 degrees using an image processing utility program. Afterwards, the zoom procedure has been used to enlarge a window area (the dashed-line block in Fig. 6.4) using the "Zoom Window" option. The display of the designated window area on completion of the zoom operation has also been captured, rotated and then shown in Fig. 6.5.

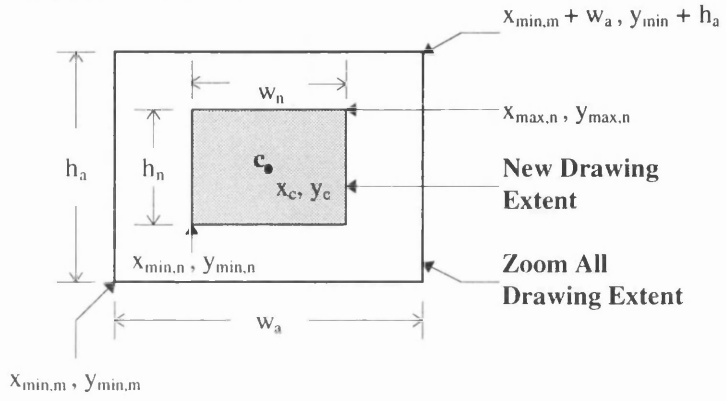
In both Fig. 6.4 and Fig. 6.5, the background colour of the display windows has been set to a light grey. Three foreground colours (red, green and black) have been used to represent the map. Text annotations are represented in black, building outlines in red and all other features in green. It should be noted that the title area (displayed at the top of the window) has been displayed in black, so that the default title "Napier88" (also in black) is invisible. It is known that the colour of this title area is affected by the setting of the colourmap specified in drawing programs. However, the Napier88 reference manual does not give a procedure describing how to change or control the colour of the title area. Thus the colour of the title area displayed in an X-window must be regarded as being somewhat "unpredictable". This can be demonstrated by comparing the various windows captured from the screen which have been included in this thesis; an example of this unpredictability is the contrast between the title areas contained in Fig. 6.4 and Fig. 6.8 illustrated later in this chapter.

Zoom Centre



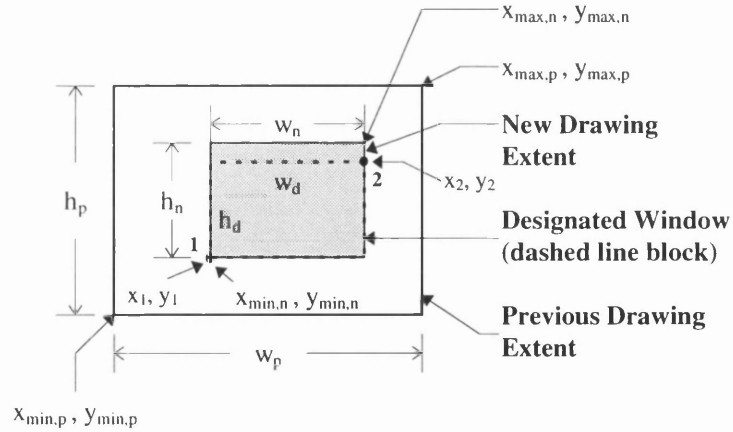
$w_p = x_{max,p} - x_{min,p}$
$h_p = y_{max,p} - y_{min,p}$
$w_n = w_p / f$
$h_n = h_p / f$
$x_{min,n} = x_c - w_n / 2$
$y_{min,n} = y_c - h_n / 2$
$x_{max,n} = x_c + w_n / 2$
$y_{max,n} = y_c + h_n / 2$

Zoom Number (N)



if $w_m * h_w / w_w <= h_m$
then
$w_a = h_m * w_w / h_w$
$h_a = h_m$
else
$w_a = w_m$
$h_a = w_m * h_w / w_w$
$x_c = x_{min,m} + w_a / 2$
$y_c = y_{min,m} + h_a / 2$
$w_n = w_a / f$
$h_n = h_a / f$
$x_{min,n} = x_c - w_n / 2$
$y_{min,n} = y_c - h_n / 2$
$x_{max,n} = x_c + w_n / 2$
$y_{max,n} = y_c + h_n / 2$

Zoom Window



$w_d = x_2 - x_1$
$h_d = y_2 - y_1$
if $w_d * h_w / w_w <= w_d$
then
$w_n = h_d * w_w / h_w$
$h_n = h_d$
else
$w_n = w_d$
$h_n = w_d * h_w / w_w$
$x_{min,n} = x_1$
$y_{min,n} = y_1$
$x_{max,n} = x_{min,n} + w_n$
$y_{max,n} = y_{min,n} + h_n$

w : the width of a bounding rectangle	w_w,h_w : the width and height of the display window
h : the height of a bounding rectangle	w_p, h_p : the width and height of the previous drawing extent
f : the magnification/reduction factor	w_n, h_n : the width and height of a new drawing extent
p : previous drawing extent	w_m, h_m : the width and height of the map
n : new drawing extent	w_a, h_a : the width and height determined from the "Zoom All" case
a : zoom all extent	w_d, h_d : the width and height of a designated window area (all distances are given in ground length)
m : map extent	
d : a designated window	
c : centre of the area to be displayed	

Figure 6.3 The determination of the drawing extent required for various zooming options

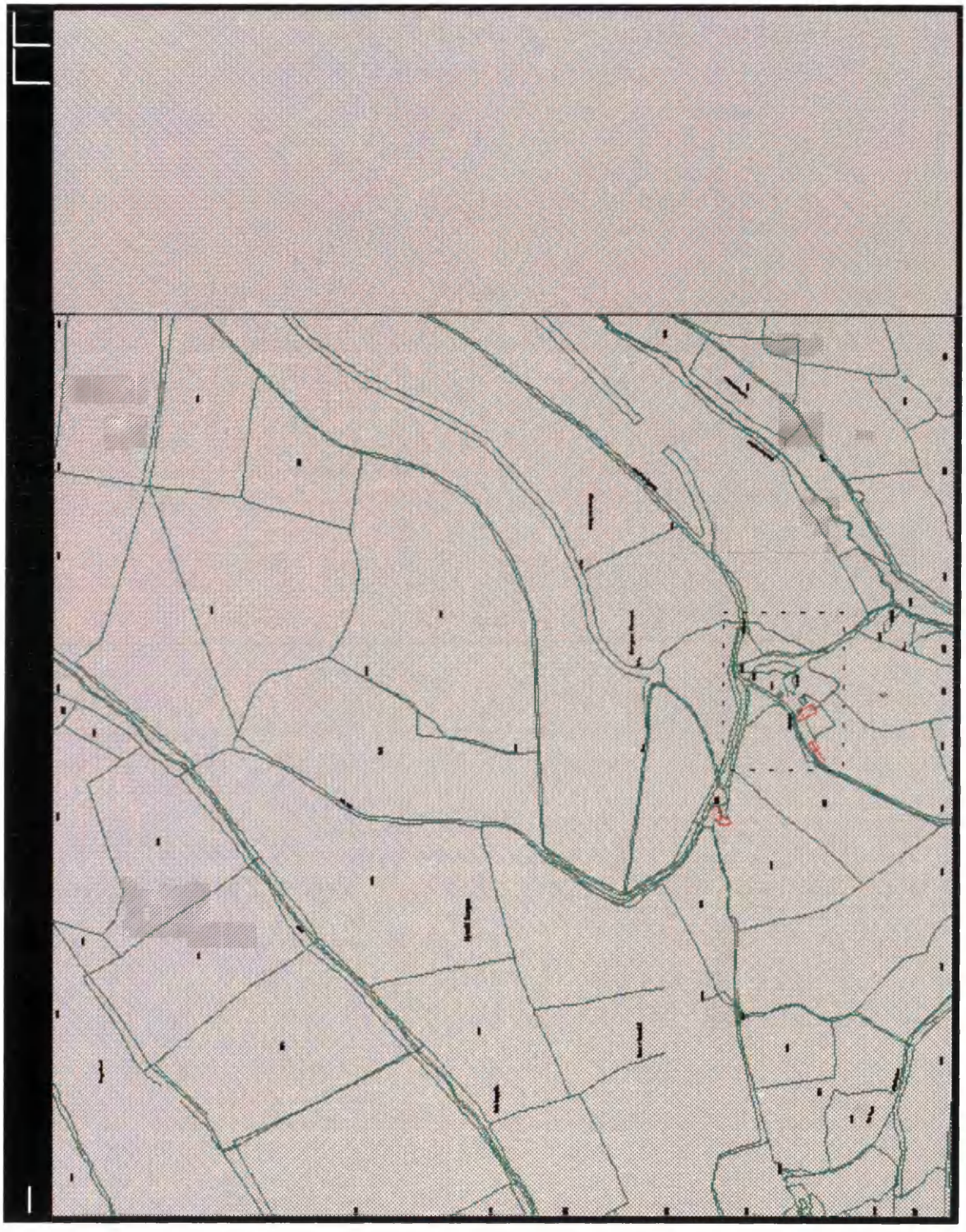


Figure 6.4 The display of the whole map SS8087

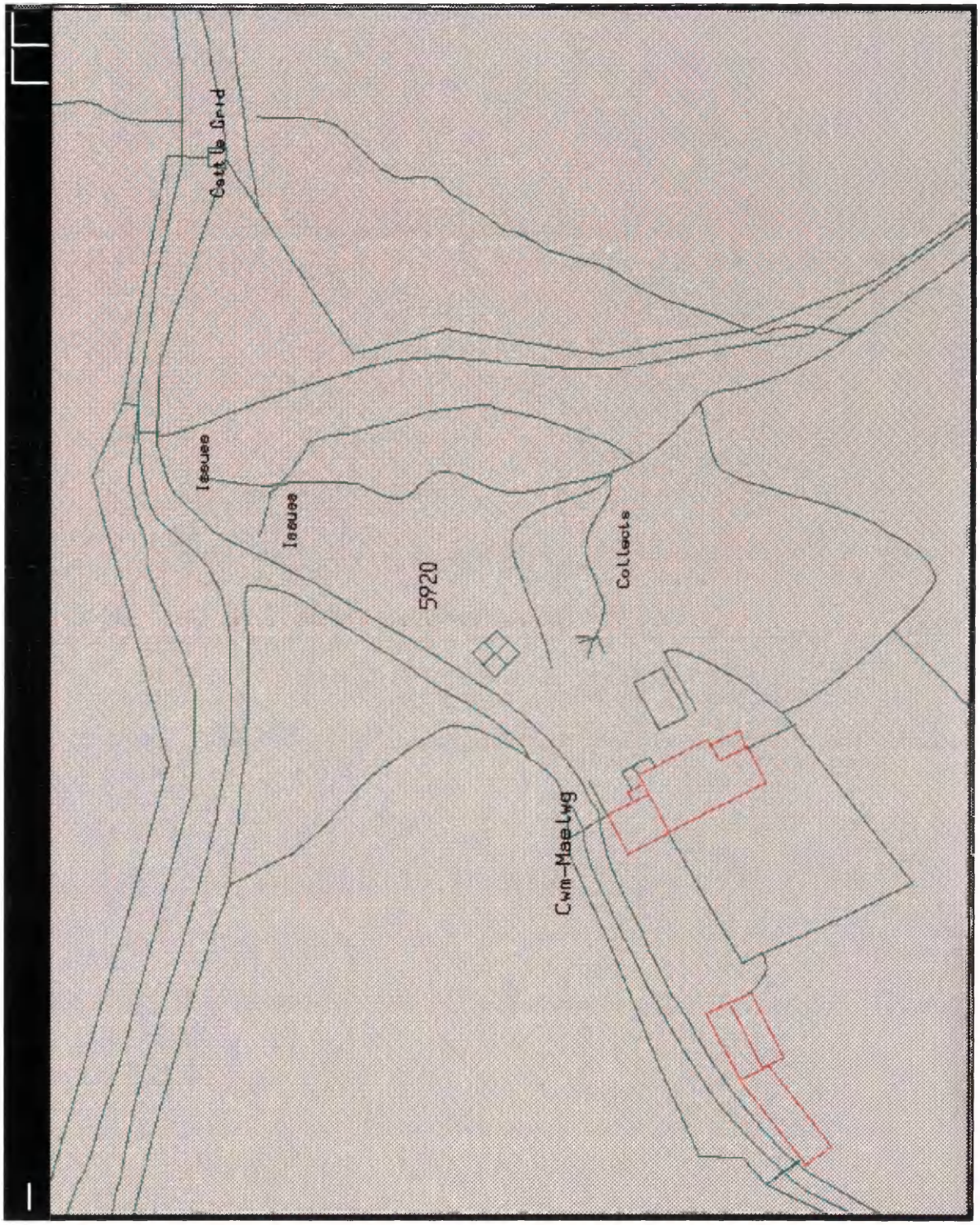


Figure 6.5 The zoom-in display of the dashed-line block area in Fig. 6.4

6.2.2.2 Viewing and Panning a Baseimage

As has been done when zooming a basemap, so a zoom function can also be developed for the viewing of a baseimage. This is particularly useful for showing an overview of the coverage of a baseimage. Zooming an image requires a resampling process. Sampling at a higher rate (*i.e.* scaling up) generates more samples, and thus a larger image. On the contrary, sampling at a lower rate (*i.e.* scaling down) generate fewer samples, and thus a smaller image. Zooming an image at the scale of an integer value relative to the actual pixel size does not require an interpolation or a stretching process, so it can be implemented fairly easily. However, zooming an image at the scale of a real value involves reconstructing the original continuous function at the given sample points and then resampling at a different rate [Schumacher, 1992]. This operation will require quite a complex operation. Therefore, a complete function for zooming an image has not yet be implemented in the persistent IGIS. Instead, this section places an emphasis on the development of the viewing function required for most raster operations since these are being carried out on images which are being displayed at their actual pixel size rather than on the images that have been resampled. Thus a pan function, which is essential for viewing an image displayed at the scale of 1:1, (where one image pixel is "mapped" onto one screen pixel), has been developed in the persistent IGIS.

In order to facilitate the panning of an image, it is advantageous to display an image by first registering the centre of the image to the centre of the display window when it is first loaded. This operation can be carried out by the statement

copy limit raster **at** xr, yr **onto limit** window **at** xw, yw

which will display the limited section of the image *raster* starting at x_r, y_r onto the limited section of the display window starting at x_w, y_w. Because the size and the depth of the limited sections are not specified in the above statement, therefore the maximum size and the maximum depth of both the raster image and the display window are taken by default to perform automatic clipping on the edges and depths of the limited sections. There are four possible cases whereby a raster image may overlap with a display window. The starting points for setting the limits of a raster image and a display window - (x_r, y_r) and (x_w, y_w) - required for the above statement are varied in each case. The values of the coordinates (x_r, y_r) and (x_w, y_w) for each case are illustrated in Fig. 6.6.

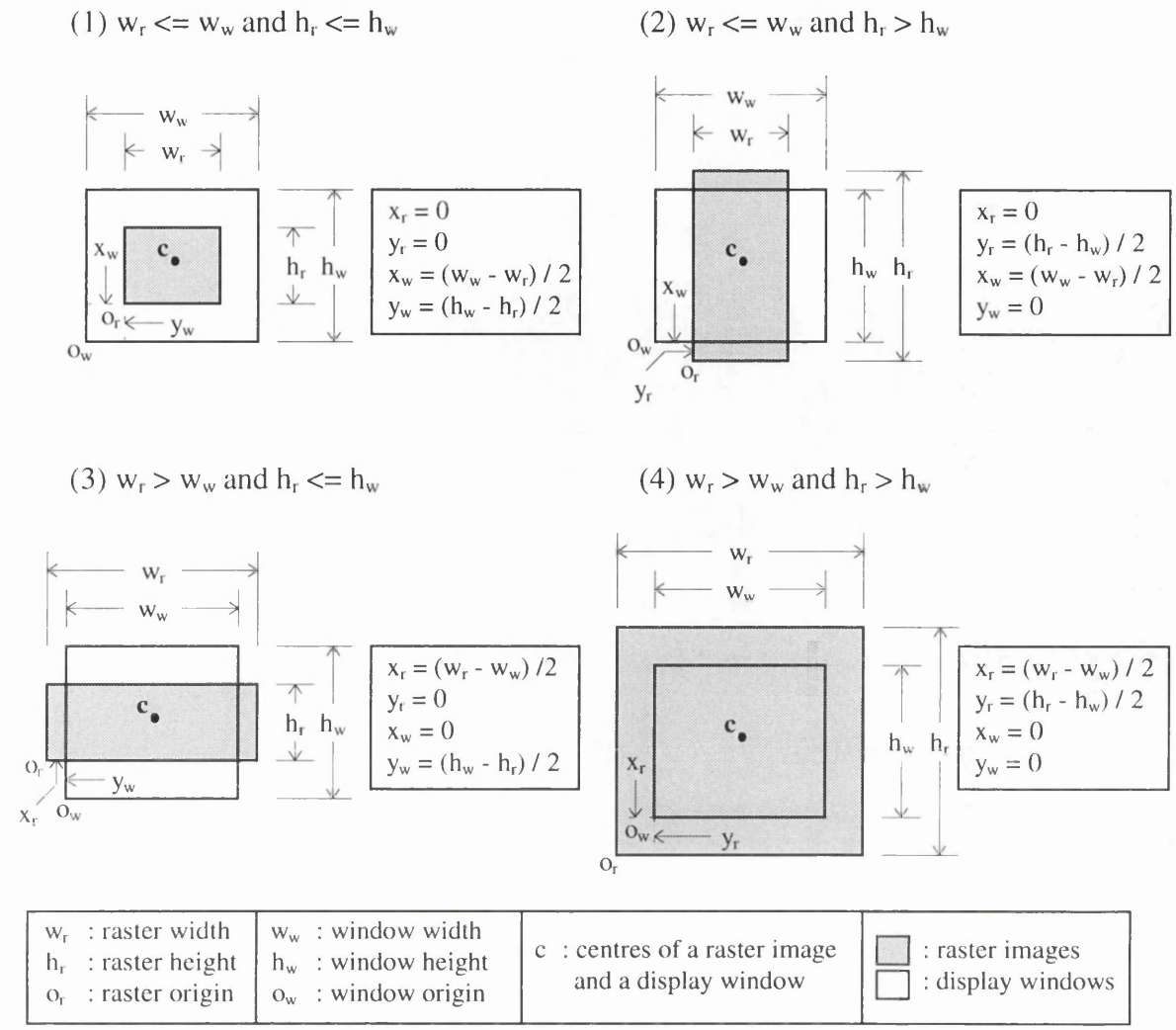
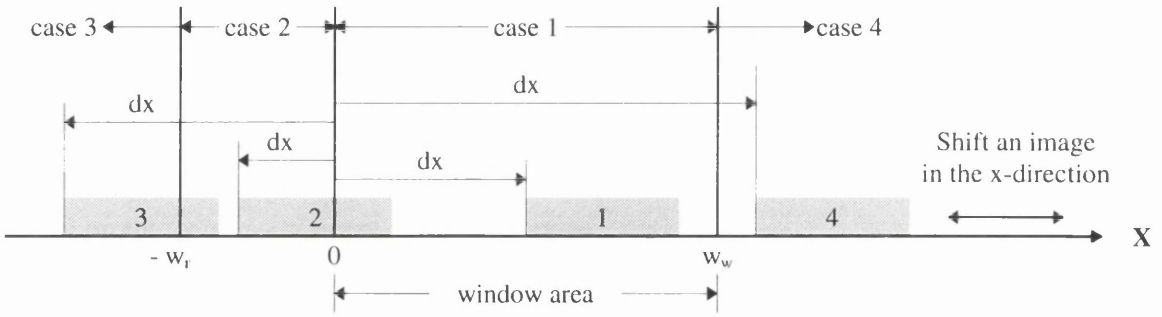


Figure 6.6 The determination of (x_r, y_r) and (x_w, y_w) for the different overlay cases of a raster image and a display window

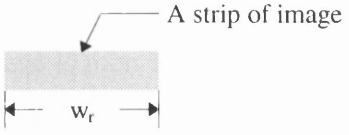
The algorithm for the implementation of each of the four cases illustrated in Fig. 6.6 may be described as follows: -

```
let x_r := 0; y_r := 0
let x_w := 0; y_w := 0
if raster_width <= window_width
then
    x_w := (window_width - raster_width) / 2
else
    x_r := (raster_width - window_width) / 2
if raster_height <= window_height
then
    y_w := (window_height - raster_height) / 2
else
    y_r := (raster_height - window_height) / 2
```

This algorithm can be implemented as an image display procedure that will centre an image onto a window. If an image is bigger than the window, then a panning function will be needed to view that part of an image which is outside the window. The panning function is provided to move an image from side to side, up and down, *etc.* within the window. Several methods may be used to achieve such a panning operation. For example, the movement of an image may be carried out using the keys on a keyboard which control the movement of the screen cursor or employing a mouse to drag the image dynamically across the display screen, *etc.* In the current implementation of the persistent IGIS, a panning function has been developed by drawing a line on the display window in order to indicate the shift of the image. Thus the relative displacements of the image in both the x- and y- directions (dx and dy) can be determined in terms of pixels. Then, the dx and dy values can be used to recompute the (x_r, y_r) and (x_w, y_w) values required for the projection of an image onto a window. Fig. 6.7 illustrates the determination of the new values for x_r and x_w considering the shift of the image only in the x- direction, *i.e.* across the width of the screen.




```
if dx >= 0 and dx < w_w                 ! case 1
then
    x_w = dx ; x_r = 0
else if dx < 0 and dx >= - w_r          ! case 2
then
    x_w = 0 ; x_r = - dx - 1
else if dx < - w_r
then
    x_w = 0 ; x_r = w_r - 1             ! case 3
else
    x_w = w_w - 1 ; x_r = 0             ! case 4
```



w_r : raster width
w_w : window width

Figure 6.7 The determination of x_r and x_w by shifting dx

The statements described in Fig. 6.7 are made on the specific promise that the displacement is referenced to the origin of the display window. In order to centre an image on a window, the displacement should be referenced to the centre of the window. This operation requires a translation movement, which can be carried out by the replacement of dx with $dx + w_w / 2$, in the above statements. The same concept can also be applied to the y-direction. Therefore, the overall algorithm for panning an image can be described as follows: -

```
! shift an image in the x-direction
if dx >= - window_width / 2  and  dx < window_width / 2
then
    x_w = dx + window_width / 2 ; x_r = 0
else if  dx < - window_width / 2  and  dx >= - window_width / 2 - raster_width
then
    x_w = 0 ; x_r = - (dx + window_width / 2) - 1
else if  dx < - window_width / 2 - raster_width
then
    x_w = 0 ; x_r = raster_width - 1
else
    x_w = window_width - 1 ; x_r = 0

! shift an image in the y-direction
if dy >= - window_height / 2  and  dy < window_height / 2
then
    y_w = dy + window_height / 2 ; y_r = 0
else if  dy < - window_height / 2  and  dy >= - window_height / 2 - raster_height
then
    y_w = 0 ; y_r = - (dy + window_height / 2) - 1
else if  dy < - window_height / 2 - raster_height
then
    y_w = 0 ; y_r = raster_height - 1
else
    y_w = window_height - 1 ; y_r = 0
```

Based on the panning concept described above,

(i) the procedure *getDragDxy*, which returns the displacement dx and dy by dragging a "rubber-band" line; and

(ii) the procedure *drawImage*, which can handle the shift of an image,

have been developed (See Appendix D). These two procedures can be used to display and pan a baseimage. Fig. 6.8 shows that the display of the image SS88SW centres on an X-window. The dotted line indicates that the displayed image is then shifted in the north-east direction through the use of the panning function. The displacement is about 400 pixels in both the x- and the y- directions. The resultant display of the image after panning is shown in Fig. 6.9.

6.3 *The Provision and Arrangement of Colours for the Display of Maps and Images*

Colour is a very effective tool for conveying information to the user of a GIS. A colour monitor is commonly used to represent the different categories of geographical features in different colours. The screen resolutions of the colour monitors used in a GIS normally range from 640 x 480 to 1,024 x 1,280 pixels with a capability of displaying 256 colours [Aronoff, 1989]. In general, a range of 256 colours is more than sufficient to display and manipulate vector map data but is quite limiting for the viewing and processing of raster

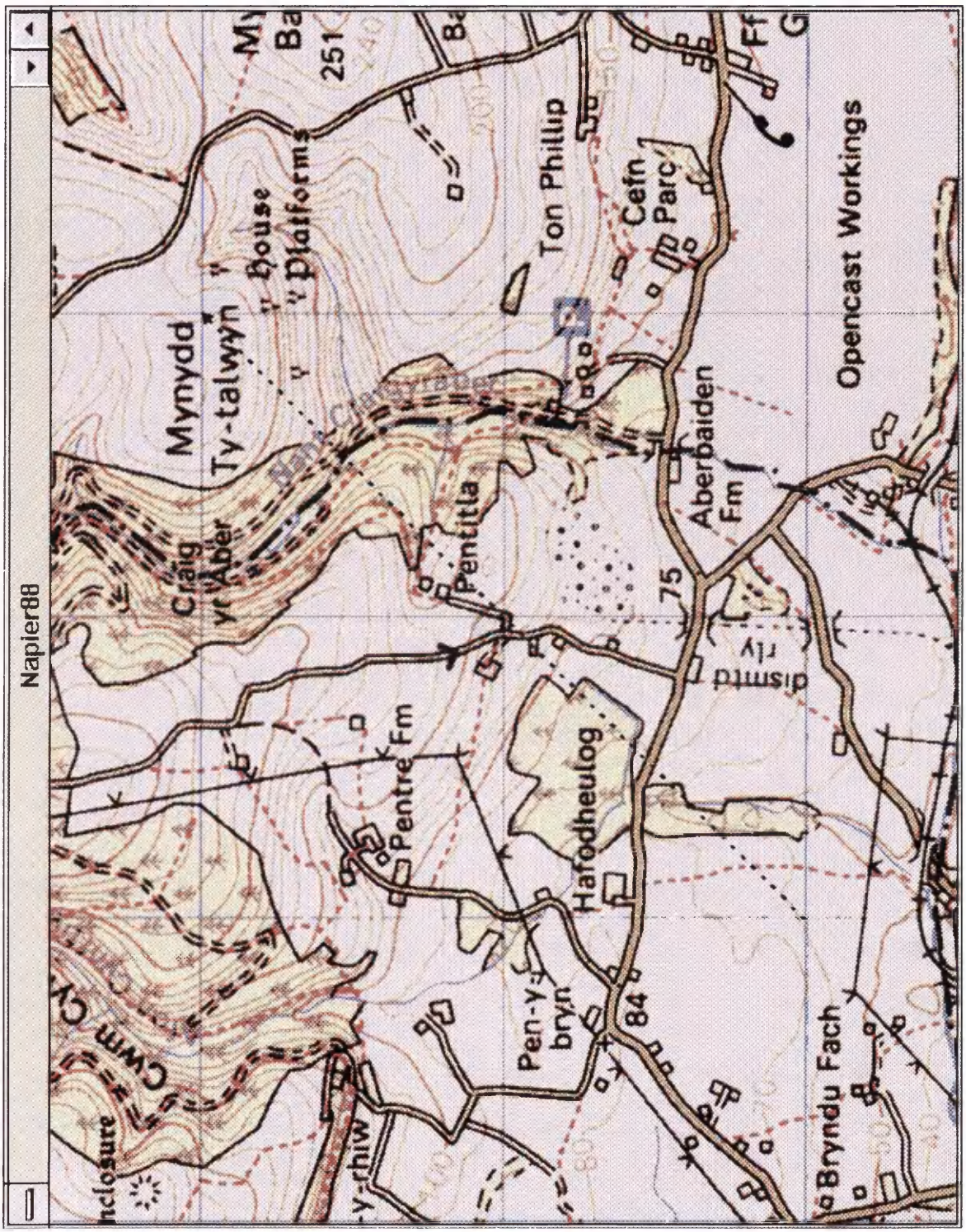


Figure 6.8 The display of the central part of the scanned raster image of OS map SS88SW

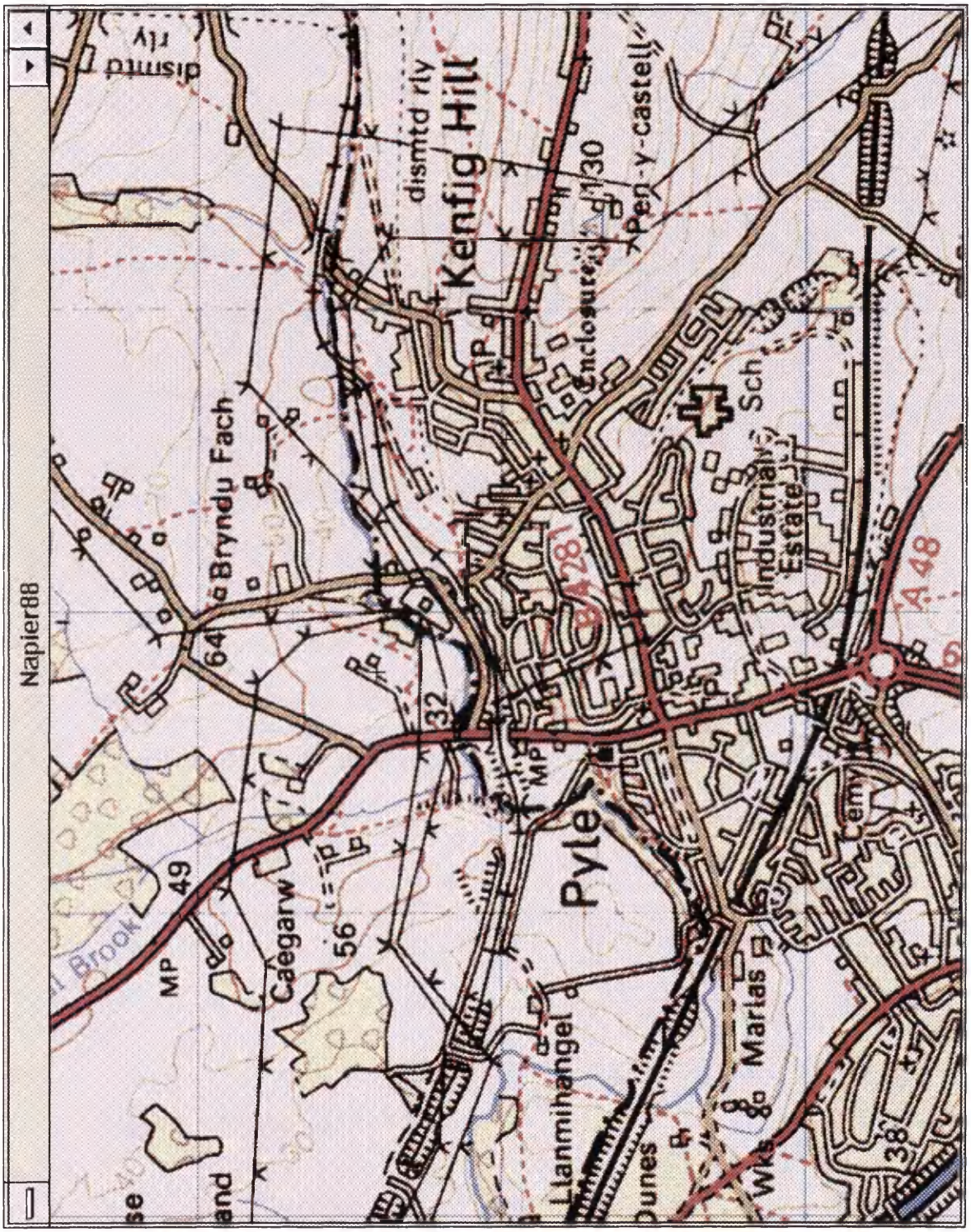


Figure 6.9 The resultant display of the scanned raster image of OS map SS88SW after panning

image data, particularly multiple-band remotely sensed data. In other words, image processing operations will often require a colour monitor providing a higher resolution and more colours than those used in the other aspects - such as the digital mapping, vector GIS and raster GIS operations - of an IGIS. As has been discussed in Chapters 4 and 5, both the system architecture and the database design of the persistent IGIS have included the potential to provide the capabilities required for image processing operations. Hence, the prototype IGIS is able to extend its functions to deal with remotely sensed data, in which the fine graduations of shading and colour present in the image are retained, using a high resolution colour monitor. However, an 8-bit frame buffer (also known as a image plane) for displaying images is normally used in a GIS. Therefore, this section will be oriented towards the use of relatively low-cost colour monitors which have resolutions ranging from 640 x 480 to 1,024 x 768 pixels and a minimum of 256 different colours for the handling and display of the geographical data.

In an 8-bit plane colour monitor, the values that represent the screen image can range from 0 to 255. Each value stored in the frame buffer represents the colour to be displayed at a specific pixel location on the screen, *i.e.* a maximum of 256 different colours can be produced. A colour look-up table, sometimes called a colourmap, is used to convert pixel values into appropriate red, green and blue values to control the colour display tube. Each of the red, green and blue values can accommodate numbers in the range 0 to 255, so the combination of the three values gives a range of 256 x 256 x 256 (*i.e.* 16,777,216) colours. However, because the numbers stored in the frame buffer can have only 256 different values, so the colour look-up table can only contain a maximum of 256 entries. Each entry can select one of 256 red values, 256 green values, and 256 blue values. Therefore it may represent any one of the full range of colours but only 256 such entries are available in total. In the Napier88 system, the procedure *colourMap* is provided for the setting of any entry in the colour look-up table - as has already been described in Section 3.4.2. Thus a colour look-up table can be defined by repeating the use of the *colourMap* procedure which establishes a "mapping" from a particular pixel to a 24-bit number corresponding to an entry in the overall range of 256 blue levels by 256 green levels by 256 red levels.

It should be noted that a colour look-up table can also be defined to display monochrome images with different gray levels. This is particularly important for the display of a digital orthophoto image or the image derived from one channel of a remotely sensed device such as the SPOTS or Landsat TM scanner. The settings of a gray-level colourmap can be carried out by assigning a range of intensities between black and white, *i.e.* a gray level index is defined by assigning the same intensity to the red, green and blue values.

As has been discussed in Chapter 5, a baseimage is already associated with a specific colourmap which is stored in the database. Therefore, the colourmap of a baseimage can be

easily acquired and used also for the image display. By contrast, a basemap does not contain a default colourmap for representing different types of features. The operation required for the colour display of a basemap is quite a different matter. In a basemap, the individual features are already categorised by a feature classification scheme. Each feature carries a code to indicate what the feature is: *e.g.* a building, fence, road, river, *etc.* The number of feature codes used may vary depending on the specification used by a particular mapping organisation for a specific map series, on the application requirements, and so on. For example, Ordnance Survey's "Landline" data employs 30 feature codes and 6 additional text codes [Ordnance Survey, 1994]. In principle, each code used in a map can be assigned a particular colour. However, if too many colours appear on a single map, it becomes difficult to distinguish the individual features. Therefore, in order to display a map with several commonly-used colours, it is necessary to organise the map's feature codes in a hierarchical manner or to reclassify them into several categories by regrouping the relevant features into a number of groups. Thus, besides the definition of a colourmap for the commonly-used colours, the colour settings used for the display of basemaps involve the creation of a feature-colour table by assigning one of the predefined colours to a group of feature codes.

In terms of the separate displays of basemaps or baseimages which have been discussed in the previous section, a quite independent colourmap holding up to 256 entries can be defined for use in each of the graphic display programs. In other words, all the 8 bit planes will be dedicated to display vector maps or raster images when the persistent IGIS is dealing with uni-format data, *i.e.* either vector or raster data. However, in order to simultaneously display both map and image data in a common window, it is vital to allocate the number of colours (or bit planes) used to represent each of them. The determination of the number of colours required for the representation of map or image data is a complex subject and is beyond the scope of this thesis. In general, the required number of colours used in vector maps lies between 8 (= 3 bits) and 16 (= 4 bits), whereas in raster scanned map images it lies between 16 (= 4 bits) and 32 (= 5 bits) while with the raster images acquired from remote sensing, it will often be more [Sproull *et al.*, 1985; Aronoff, 1989; Ordnance Survey, 1993b]. On the other hand, the choice available when allocating the number of bit planes required for the simultaneous handling of both vector and raster data is quite limited - either 3 bit planes for vector data and 5 bit planes for raster data or 4 bit planes for both of them. Considering the fact that raster images are often made available and stored in 1, 4 or 8 bit form and are often used as backdrops for assisting the manipulation of vector data when dealing with dual-format data, so the second alternative has been selected and implemented in the persistent IGIS. It should be point out that the current implementation for the allocation of a fixed number of bit planes each for vector and raster data is regarded as an intermediate solution in the development of the persistent IGIS. Eventually, the persistent IGIS should include a scheme that can dynamically optimise

and allocate a much larger number of bit planes for the provision of the facility to handle dual format data.

The allocation of the bit planes used for the concurrent display and processing of dual-format data may be carried out by assigning bit planes 0 to 3 for dealing with raster data and 4 to 7 for the handling of vector data. Fig. 6.10 illustrates the arrangement of bit planes used to deal with this situation. With this configuration, 16 different colours can be defined for the representation of the vector data using the high-bit part of the frame buffer, whereas 16 different colours or gray levels may be used for raster data using the low-bit part. In order to display vector maps and raster images in a common window, a default colourmap can be defined as follows: -

```
let vec_cndx := vector 0 to 15 of off ++ off ++ off ++ off
let ras_cndx := vector 0 to 15 of off ++ off ++ off ++ off
for i = 0 to 15 do  { vec_cndx (i) := colourToPixel(i, 4) }
for i = 0 to 15 do  { ras_cndx (i) := colourToPixel(i, 4) }
! define colours for vector data
let ct := rgb(16)
for i = 0 to 15 do
   for j = 0 to 15 do
      { colourMap(window_file, ras_cndx(j) ++ vec_cndx(i),
                                          ct(i,3) * 256 * 256 + ct(i,2) * 256 + ct(i,1) }
! define gray scales for raster data
ct := grayLevel(16)
for i = 0 to 15 do
   { colourMap(window_file, ras_cndx(i) ++ off ++ off ++ off ++ off,
                                          ct(i,3) * 256 * 256 + ct(i,2) * 256 + ct(i,1) }
```

The first two statements declare two arrays to hold the colour indices in pixel representation for vector data and for raster data respectively. The next two statements convert the colour indices from integer to pixel (bits) representation. This is followed by the settings of 256 entries in the colour look-up table. The procedure *rgb* (See Appendix D) supplies the red, green and blue values of the 16 predefined commonly-used colours required for the procedure *colourMap*. The 256 entries in the colour look-up table can be divided into 16 groups, *i.e.* the vector map colour index 0 to 15 shown in Fig. 6.10. Each group contains 16 entries which will have the same combination of 0's and 1's for bit planes 4 to 7 but can have any combinations of 0's and 1's for bit planes 0 to 3. At this moment, the colour look-up table will display 16 colours depending only on the control of bit planes 4 to 7. In other words, the content of bit planes 0 to 3 will have no effect on the colour display. However, in order to use a raster image as a backdrop during the handling of vector data, the raster image has also to be displayed. This operation can be carried out by a further change in the colour look-up table. The procedure *grayLevel* (See Appendix D) creates a range of 16 intensities between black and white. The intensities are then used to

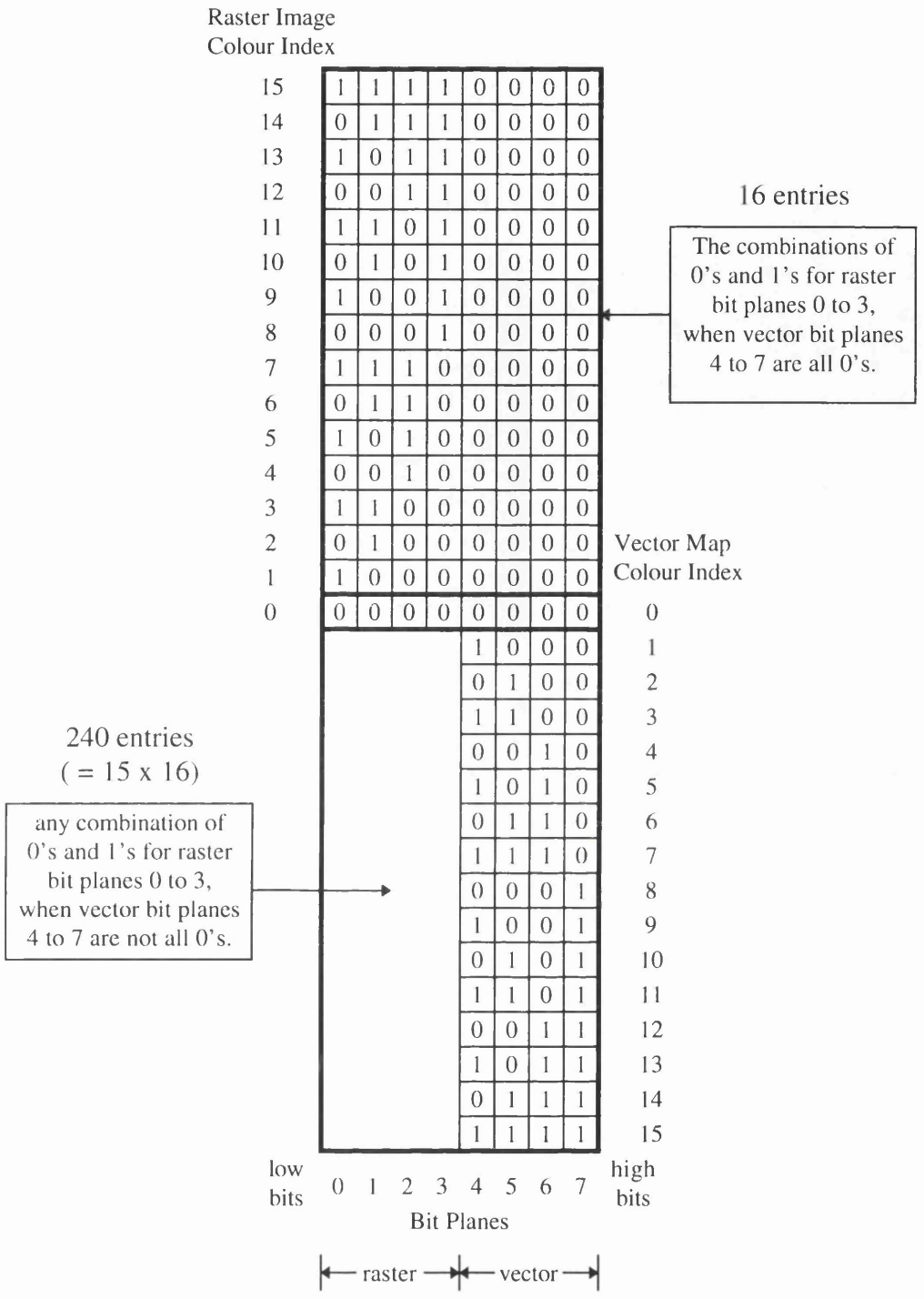


Figure 6.10 The allocation of bit planes for handling both vector and raster data concurrently

change 16 entries of the colour look-up table if the bit planes 4 to 7 are all 0's. This change means that, if there is no vector data (*i.e.* map colour index = 0) to display, then only the raster data will be displayed. Therefore, it may be said that the setting of the colourmap gives the vector map data a higher priority in terms of display than the raster data, so that the vector information is always displayed on top of the raster images.

During the display of both vector maps and raster images, the default 16 colours and the 16 gray scales defined in the colour look-up table will be dynamically changed to meet the application requirements. On the one hand, the RGB intensities for a specified vector map colour index can be redefined to display a particular colour. On the other hand, when a new raster image is loaded, the colourmap of the image (*i.e.* the RGB intensities of the 16 image colour indices) will be used to change the intensities of the corresponding 16 entries in the colour look-up table. That is to say, a maximum of 16 different colours can be produced for the display of each data type (vector or raster) but the actual colours used in the display may be selected from the full range of 16.7 million colours. Thus, using this arrangement of bit planes, the persistent IGIS is able to provide the facilities required for the superimposition and the concurrent processing of both vector and raster data.

6.4 The Superimposition of Maps and Images

Having allocated bit planes as well as having arranged colours for the handling of dual format data, a map and an image can easily be overlaid in a common window. For example, the superimposition of a 16-colour map *A* and a 4-bit image *B* in a window, which corresponds to the drawing extent defined by x_min, y_min, x_max and y_max, may be carried out as follows: -

```
draw(window(4 | 4), A, x_min, y_min, x_max, y_max)
copy B onto window(0 | 4)
```

The first statement draws the map *A* in the defined window area starting at bit plane 4 and using the depth of the 4 available bit planes, namely numbers 4, 5, 6 and 7. The second statement copies the image *B* to the bit planes 0 to 3 of the window. In this overlay operation, the result will be an appropriate display both for the map *A* and for the image *B*. However, the locations on the map will not agree with the corresponding locations on the image, unless both the map and the image are referenced to a common coordinate system.

In order to carry out this correctly registration operation, one of two possible approaches may be used. The first approach calculates the relative position between a map and an image, and then transforms the coordinates of the map into the coordinates that fit the image or *vice versa*. This approach is usually implemented graphically and interactively by choosing well-defined features that can be easily and precisely identified on both the map and the image. After the locations of the corresponding points have been measured on the screen, a coordinate transformation is (usually a linear conformal or an affine transformation) then carried out to geometrically correct and eliminate the differences in position between the map and the image.

The second approach provides the absolute position for both maps and images, *i.e.* all maps and images are referenced to the same ground coordinate system. Because all the test data used in this research have been referenced to the OSGB National Grid coordinate system, therefore the second approach has been adopted and implemented in the persistent IGIS. In terms of the superimposition of a basemap and a baseimage, the extent of the map and the image coverage can be obtained from the database. Every basemap or baseimage stored in the database contains information on the extent of its coverage, *i.e.* the origin (x_min and y_min) and the range (x_range and y_range) - as described already in Chapter 5. Based on this information, the bounding rectangles of a basemap ($x_{min,m}$, $y_{min,m}$, $x_{max,m}$, $y_{max,m}$) and a baseimage ($x_{min,i}$, $y_{min,i}$, $x_{max,i}$, $y_{max,i}$) can be determined. In order to view an overlay of the basemap and the baseimage, a drawing extent ($x_{min,d}$, $y_{min,d}$, $x_{max,d}$, $y_{max,d}$) which corresponds to the window viewing area is defined to select an area of interest to be displayed. Fig. 6.11 illustrates that the concept of overlaying a map and an image which are referenced to the same ground coordinate system. Using this approach, the overlay of a map and an image will be correctly displayed because the map and the image have been geometrically registered in a common coordinate system.

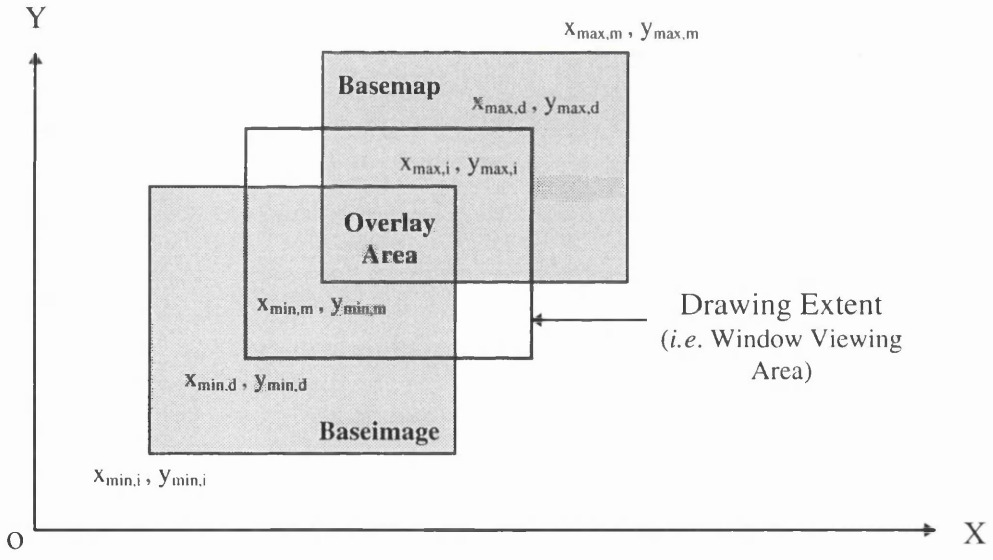


Figure 6.11 The basemap, baseimage and viewing window are all referenced to the same ground coordinate system

Based on the concept discussed above, a display function for the superimposition of basemaps and baseimages has been developed for use in the persistent IGIS. Fig. 6.12 and Fig. 6.13 provide examples of the superimposition of a basemap on a baseimage. The first example (Fig. 6.12) shows the basemap "OSCAR" 270190 superimposed on the Landsat TM Band 1 image of Port Talbot (PTBAND1). The image displayed in the window uses 16 gray scales and covers an area of 20 km x 15 km on the ground (using 800 x 600 pixels and a pixel size of 25 m), whereas the map only uses 2 colours (green and yellow) out of the 16

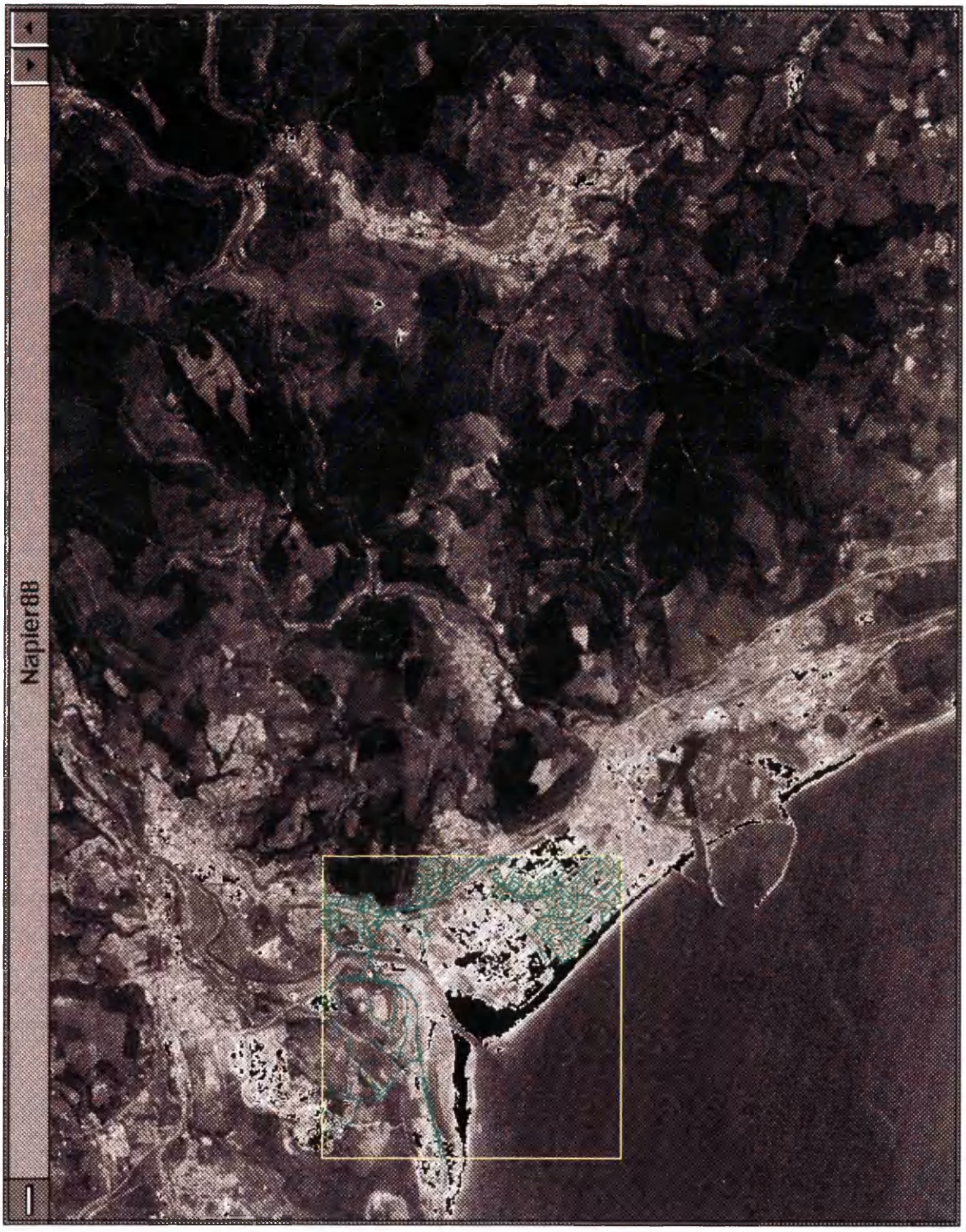


Figure 6.12 The superimposition of the OS map 270190 and the Landsat TM image PTBAND1

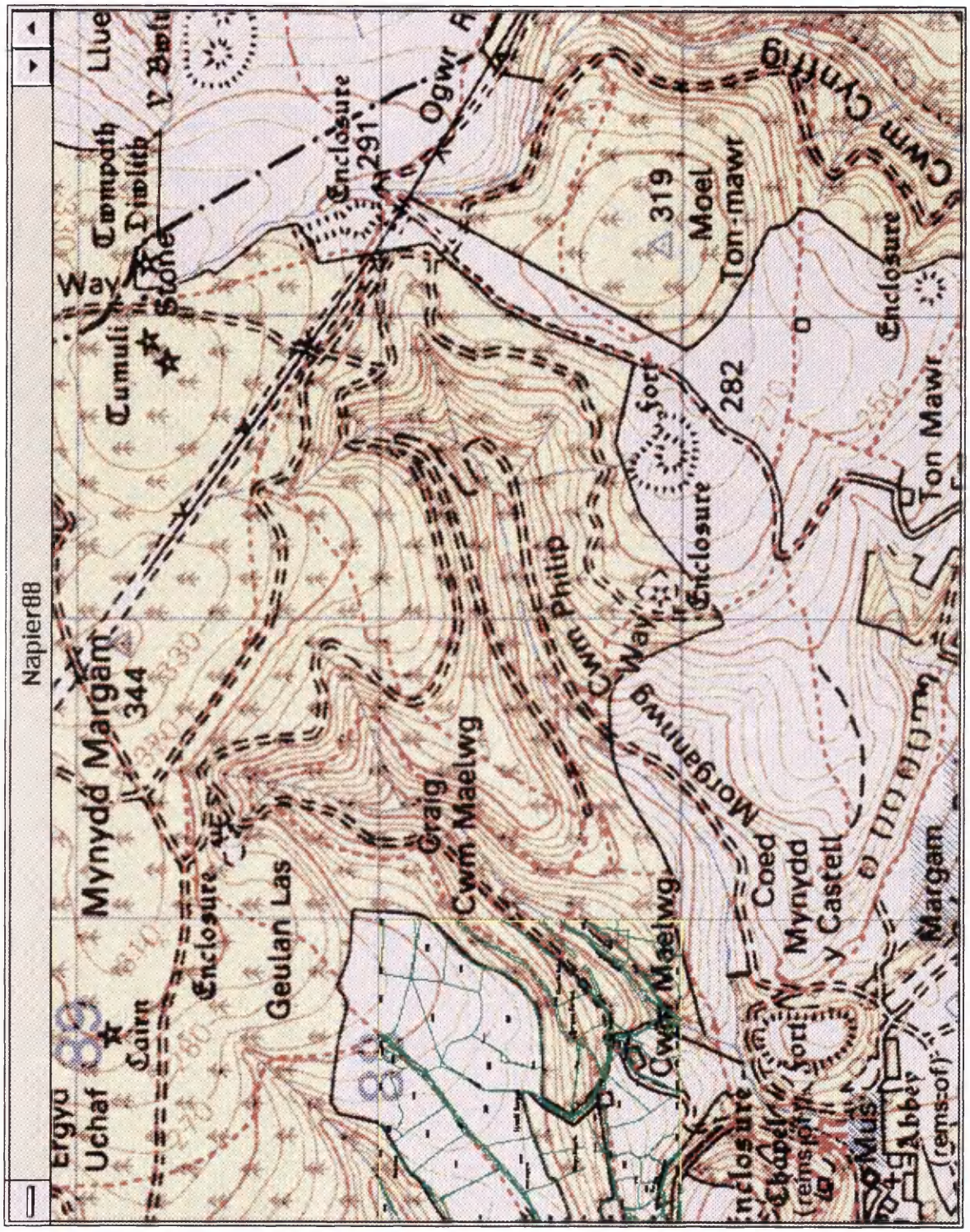


Figure 6.13 The superimposition of the OS map SS8087 and the OS scanned map image SS88SW

predefined colours and covers an area of 5 km x 5 km. The second example (Fig 6.13) shows the basemap "Landline" SS8087 superimposed on the scanned "Colour Raster" image SS88SW. The image displayed in the window uses 16 colours and covers an area of 4 km x 3 km on the ground (utilising the same 800 x 600 pixel displays, but with a pixel size of 5 m), whereas the map only uses 4 (green, black, red and yellow) out of 16 predefined colours and covers an area of 1 km x 1 km on the ground.

6.5 The Interrelation of Maps and Images

In the previous sections (6.2 and 6.4), the querying of a basemap or a baseimage is carried out by designating an identifier, *i.e. map_id* or *image_id* in the graphic display program. In spite of the fact that both the maps and the images are referenced to the same ground coordinate system, the relative positions of a map and an image have still to be derived from the values of the extent of their coverages contained in the database. The identifiers *map_id* and *image_id* do not include nor do they imply spatial relationships such as containment, overlap, *etc.* between the maps and the images. In order to spatially query basemaps and/or baseimages by specifying a location or an area, an exhaustive search through the *Processed* database is necessary to find out which maps and/or images cover the queried location or area. To provide the facility to spatially query vector maps and raster images, a spatial indexing method based on the Peano ordering of a uniform grid cell system has been developed for the persistent IGIS. The Peano ordering - the principle of which has already been discussed in Subsection 5.3.5 in the context of addressing a linear quadtree - is used here to index the identifiers of the relevant maps and images using the spatial key *Peano*. In other words, the maps and images are interrelated in a spatial index table.

The basic principle of this approach is that multi-scale maps and multi-resolution images are referenced to a common grid cell system. The grid cells covered by a map or an image are indexed by the spatial key *Peano*. The spatial key *Peano* comprises the *peano_key* and the *side_length* of a grid cell. A *peano_key* can be computed uniquely from a coordinate pair (x, y) and *vice versa* - this principle will be discussed in more detail later in Chapter 7. Two procedures *xyToPK* and *pkToXY* (see Appendix C) have been developed for this conversion. Thus the *peano_key* of a grid cell can be determined from its origin, usually located at the south-west corner of the grid cell. Because a map series is generally tiled in order to cover a particular area, the tile size can be used as the grid cell size in the spatial indexing system so as to simplify the process of indexing maps without needing to subdivide a map into grid cells. However, normally the coverage of individual images such as aerial photographs, orthophotos or remotely sensed images will not fit the area covered by a specific map, so the process of subdividing an image into grid cells can hardly be prevented. Therefore, using map tiles as grid cells in a spatial indexing system, a map series and the related images can be spatially indexed using peano keys of the map tiles. The *peano_key* of

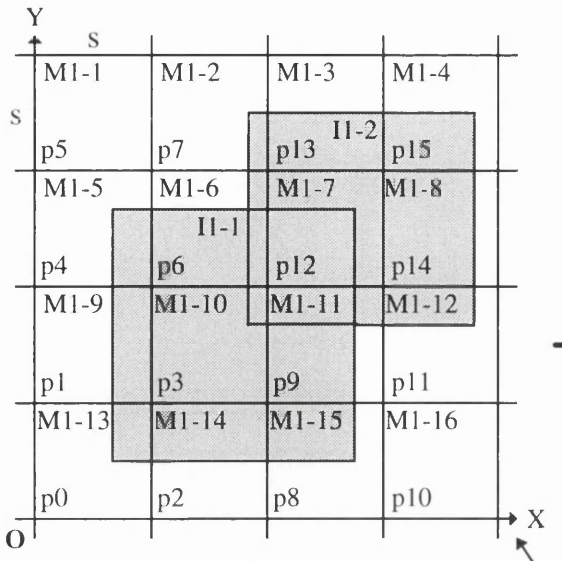
a particular map tile is determined from the coordinates of the south-west corner of a tile or map. In other words, the peano key of a map tile is used to index the map identifier of the map tile and the image identifiers of those images which overlay the map tile. The relationship between *peano_key* and *map_id* is a one-to-one mapping, whereas the relationship between *peano_key* and *image_id* is a one-to-many mapping. Fig. 6.14(a) illustrates the concept of indexing a map series and the related images.

The *peano_key* in the index table related to Fig. 6.14(a) can uniquely identify a grid cell (map tile). However, with this arrangement, a separate index table is required for each map series. Fig. 6.14(b) illustrates another map series which covers the same area as that shown in Fig. 6.14(a) and uses a common coordinate system; however the size of the map tile is doubled. A separate index table which contains 4 entries for this map series will also be generated. As such, quite a number of index tables would have to be created in order to handle multi-scale maps and multi-resolution images. As a result, the querying and the maintenance of these index tables will become tedious and error-prone. A solution to this problem is to arrange that all the index tables are combined into a single index table. For example, the two index tables created for the different map series shown in Fig. 6.14 may be merged together. However, the *peano-key* can then no longer identify a unique grid cell because two maps in different map series may share a common map origin, *i.e.* the *peano_key* of these two maps may have the same values. For example, the map M1-5 in Fig. 6.14(a) and the map M2-1 in Fig. 6.14(b) have the same *peano_key*. In order to create a composite table for all map series, another identifier - *side_length* - is used to join the *peano_key* and form a spatial key *Peano*. Thus the *side_length* is used to distinguish different map series. Using the spatial key *Peano*, each entry in a composite index table such as that shown at the bottom of Fig. 6.14 can be identified uniquely.

Based on the Peano key spatial indexing, the following data types can be declared to represent the indices of multi-scale maps and multi-resolution images respectively.

```
type Peano is structure(peano_key: int ; side_length: real)
type Map_Index is Map[Peano, Map_id]
type Image_Index is Map[Peano, List[Image_id]]
```

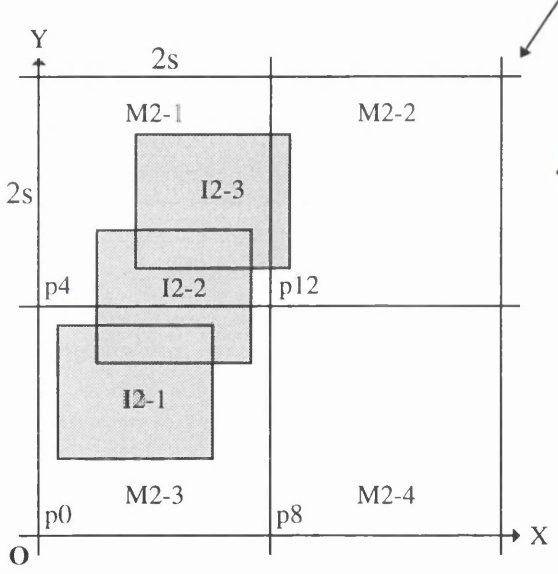
Making use of the spatial indexing methods described above, a map identifier and a list of image identifiers can be determined by pointing with a mouse-controlled index mark or cursor to a position on the display and specifying the value of the *side_length*. Hence, the appropriate map and the corresponding image data can easily be found in the persistent store. The required data is then loaded into a program together with the appropriate data



(a) Indexing a map series and the related images

Peano_key	Map_id	Image_id
p0	M1-13	I1-1
p1	M1-9	I1-1
p2	M1-14	I1-1
p3	M1-10	I1-1, I1-2
p4	M1-5	I1-1
p5	M1-1	nil
p6	M1-6	I1-1, I1-2
p7	M1-2	I1-2
p8	M1-15	I1-1
p9	M1-11	I1-1, I1-2
p10	M1-16	nil
p11	M1-12	I1-2
p12	M1-7	I1-1, I1-2
p13	M1-3	I1-2
p14	M1-8	I1-2
p15	M1-4	I1-2

Map Tiles



(b) Indexing another map series and its related images

Peano_key	Map_id	Image_id
p0	M2-3	I2-1, I2-2
p4	M2-1	I2-2, I2-3
p8	M2-4	nil
p12	M2-2	I2-3

The Composite Key *Peano*

Peano_key	side_length	Map_id	Image_id
p0	s	M1-13	I1-1
p0	2s	M2-3	I2-1, I2-2
p1	s	M1-9	I1-1
p2	s	M1-14	I1-1
p3	s	M1-10	I1-1, I1-2
p4	s	M1-5	I1-1
p4	2s	M2-1	I2-2, I2-3
p5	s	M1-1	nil
p6	s	M1-6	I1-1, I1-2
p7	s	M1-2	I1-2
p8	s	M1-15	I1-1
p8	2s	M2-4	nil
p9	s	M1-11	I1-1, I1-2
p10	s	M1-16	nil
p11	s	M1-12	I1-2
p12	s	M1-7	I1-1, I1-2
p12	2s	M2-2	I2-3
p13	s	M1-3	I1-2
p14	s	M1-8	I1-2
p15	s	M1-4	I1-2

M1 : map series 1
M2 : map series 2
I1 : image series 1
I2 : image series 2
p : peano key
s : side length of grid cells

Figure 6.14 Spatial indexing for the interrelation of both maps and images

structures necessary for the display and also provided for the analytical operations being carried out in GIS applications.

6.6 Summary

This chapter has described the development of an essential facility - the graphical display of vector maps and/or raster images - required for the persistent IGIS. First of all, the separate display of vector maps or raster images has been discussed. Two fundamental functions "zoom" and "pan" have been developed for the viewing or manipulation of basemaps and baseimages respectively. These basic display functions combined with the design and implementation of bit planes for the handling of dual format data are then used to develop the overlay capability of vector maps and raster images. The basemaps and the baseimages held in the *Processed* database described in the previous chapter have been used to test these graphics display procedures developed for the persistent IGIS. These tests showed that the graphics primitives provided by the Napier88 system can be used to develop a display/processing program utilising dual format data for the viewing and manipulation of vector maps and/or raster images which is an integral feature of an integrated GIS. However, the Napier88 system does not provide users with an extensive set of graphic procedures in its Standard Library. As a result, some basic graphics procedures have had to be prepared additionally to allow the development of a suitable display facility for the persistent IGIS.

In order to provide and facilitate access to multi-scale maps and multi-resolution images, a spatial indexing method based on Peano ordering has also been developed to provide for the interrelation of these maps and images. The result has been the provision of a spatial key *Peano* which is used to correlate the map and image identifiers corresponding to a particular grid cell. This spatial indexing method allows the user to display both maps and images covering an area by simply specifying a geographical location.

In this chapter, the facilities for the superimposition and interrelation of vector maps and raster images have been developed for the persistent IGIS. With the provision of these facilities, the persistent IGIS is able to achieve the display level of integration. However, in order to reach the process level of integration, the facilities for the indexing and search of geographical data need to be further developed. These will be discussed in the next chapter.

CHAPTER 7 : SPATIAL INDEXING AND QUERIES

7.1 Introduction

An important issue in the design of an IGIS is spatial indexing carried out in the context of a database. The role of spatial indexing is to deal with the need to access and retrieve geographical data based on its location. Many factors affect the data retrieval performance of a spatial indexing technique. These include the organisation of the data storage; the characteristics of the index keys; the design of the data structures used for the building of the indices; the method of physically encoding the data; and so on. Because spatial indexing techniques are not so well developed as alphanumerical indexing techniques, the selection, design and implementation of an appropriate spatial indexing technique is perhaps the most difficult problem requiring to be solved in the development of a GIS. In fact, quite a number of spatial indexing methods have been proposed to deal with the problem of spatial data retrieval. They include the KD-tree (K-Dimensional tree), quadtree, R-tree (Rectangle/range tree), space-filling curves (two- or three-dimensional ordering), grid file, field-tree, cell tree and BSP-tree (Binary Space Partitioning-tree), *etc.* [Samet, 1984; Langran, 1992; Laurini and Thompson, 1992; van Oosterom, 1993]. Furthermore, each method may generate several variants. For example, the KD2-tree, KDB-tree and KD2B-tree are all modified from the basic KD tree, while the R^+-tree and R^*-tree are derivatives of the R-tree [van Oosterom, 1993; Theodoridis and Sellis, 1993]. Each method and its variants have advantages and disadvantages; none emerges as the best. Therefore the selection of an appropriate method for spatial indexing is still a rather complex issue in the design of a GIS.

In spite of this difficulty, several spatial indexing methods have been adopted in commercial GIS software packages. These methods include quadtrees, R-trees, space-filling curves, and their variants. A quadtree is based on the recursive decomposition of space into four quadrants. A quadrant is further subdivided into four subquadrants if the number of data objects in a quadrant exceeds a predetermined maximum capacity. An R-tree is based on an extension of the B-tree for multi-dimensional objects. The descriptions (*i.e.* object-id and mbr (**m**inimum **b**ounding **r**ectangle)) of data objects are stored in leaf nodes and intermediate nodes are built up by grouping rectangles at the lower level. The minimum bounding rectangles of data objects can overlap one other or they can be completely disjoint. A space filling curve is based on the ordering of those grid cells, which cover data objects, by simplifying two dimensional addressing into single dimensional addressing. Peano (or **N**) ordering (which has already been discussed in Subsection 5.3.5) and Hilbert (or Π) ordering are the space-filling techniques which have been most widely-used in spatial

data handling systems [Aronoff, 1989; Laurini and Thompson, 1992; Theodoridis and Sellis, 1993].

As noted above, each spatial indexing method has its strengths and weaknesses. For example, it is relatively easy to generate the data to any level of detail using quadtrees, but the unbalance of the tree structure may exercise a significant influence on the performance of the system when dealing with large datasets. By contrast, R-trees have a height-balanced tree structure so that the access time to any object is nearly constant. However, the parameters associated with the construction of an efficient tree structure, such as the coverage of a node, the overlay between nodes, *etc.*, are difficult to optimise. As for the space-filling curves, they provide some efficiencies in searching operations, but will imply a large number of indices [Samet, 1989; Peloux *et al.*, 1993; van Oosterom, 1993; Theodoridis and Sellis, 1993].

In terms of indexing geographical data, both quadtrees and R-trees are well-suited to deal with points and polygons, but they do not handle lines well, being particularly less efficient for long lines. Theoretically, R-trees will give a better performance than quadtrees when handling polygons because the construction of an R-tree - which is based on grouping the mbrs of polygons - requires only a small number of polygons to be fragmented to store in more than one tree node. Space-filling curves are very efficient for the execution of exact match queries for points, but they are less efficient for other types of geometric queries such a range query. Of these three alternatives, no single approach appears the best. However, it is generally recognised that, in implementing a spatial indexing method, it is advantageous to have some form of hierarchical organisation. In this respect, quadtrees are very well-understood and their algorithms relevant to GIS development have been extensively reported in GIS literature [Ibbs and Stevens; 1988; Gahegan, 1989; Mark *et al.* 1989; Verts, 1989; Shaffer, 1990; Stuart, 1990]. Hence, the use of quadtrees seems to be a good choice to satisfy the spatial indexing requirements of the prototype IGIS.

Linear quadtrees are probably the most well-known variant of quadtrees. A linear quadtree utilises a space-filling curve to order quadrants within a pointerless representation. A linear quadtree may be regarded as a combination of the quadtree and the space-filling curve indexing techniques. Thus it has advantages of hierarchical spatial indexing as well as efficient single-dimensional addressing. The main drawbacks of linear quadtrees is that the spatial keys (*e.g.* Peano keys, Hilbert keys, *etc.*) are sensitive to the orientation and to the position of the Cartesian space origin. However, if all geographical data are converted to the same ground coordinate system in a prior operation, then the occurrence of this problem can be prevented. Both the Peano and Hilbert orderings are robust in structure and efficient in performance. However, the creation of Hilbert keys is more complex than that of Peano keys [Laurini and Thompson, 1992]. Therefore, a spatial indexing method based on the use

of a linear quadtree in conjunction with Peano addressing has been adopted to develop a facility for the indexing and the search of geographical data in the persistent IGIS.

The remaining parts of this chapter first describe the conversions between a Peano key and an xy-coordinate pair. Next the spatial indexing for geographical data by each entity type, *i.e.* for point, line and polygon entities, will be discussed separately. This is followed by a further discussion about the construction of a composite entity index table. Finally, different types of spatial queries, including query by entity types, query by regions, *etc.*, will also be discussed.

7.2 The Conversions between a Peano Key and an xy-coordinate Pair

A linear quadtree ordered by Peano key can be used not only as a spatial indexing method which will be discussed in this chapter, but it can also be used as a data model for handling raster image data. This latter application has already been discussed in Chapter 5. It involves two major operations: -

(i) Quartering - the splitting of a two-dimensional object space into a tree structure of quadrants; and
(ii) Indexing - the one-dimensional addressing of all the quadrants at leaf nodes by Peano ordering.

The quartering operation recursively subdivides the object space into equal-sized quadrants, subquadrants, and so on, until a predetermined condition is reached. The procedures required for quartering each type of entity, namely point, line and polygon entities, are varied. These will be discussed further in the next section. As for the indexing operation, this deals with the construction of an index table for quadrants using Peano ordering. The Peano ordering is a numbering scheme that uses a particular sequence (*i.e.* four quadrants at each level are regularly ordered in an "N-shape") to systematically order all the quadrants at leaf nodes into a one-dimensional path - the detail of the concept has already been given in subsection 5.3.5. Using this approach, each quadrant, which is normally identified by the two coordinates x and y of a reference point (usually located at the south-west corner or at the centre of the quadrant), can be replaced by a unique number called a peano key. The relationship between a peano key and a pair of coordinates is a one-to-one mapping. Thus the peano key of a quadrant can be obtained from the coordinates of the reference point in the quadrant, and *vice versa*. It should be noted again that the point located at the south-west corner of a quadrant has been used as the reference point throughout this thesis.

The conversion of a pair of x, y coordinate values into the corresponding peano key (p_k) can be carried out using the following steps: -

(1) Convert the x and y decimal values into their corresponding binary values;
(2) Combine the digits of these two binary values together to form a set of binary digits using the manipulation of bit interleaving;
(3) Convert the generated set of binary digits of the peano key into its corresponding decimal value (p_k).

A peano key value can also be converted into the pair of x, y coordinates defining a point using a series of steps similar to those described above but in the reverse order. These steps can be described as follows: -

(1) Convert the decimal value of a peano key (p_k) into its corresponding binary value;
(2) Decompose the digits of the binary value into two separate sets of binary digits using the manipulation of bit interleaving;
(3) Convert the two sets of binary digits into their corresponding decimal values, *i.e.* the x and y coordinates, respectively.

Both conversions work well with integer values and real values. The difference is that integer values only need one operation but real values require two operations - one deals with the conversion of those digits lying before the decimal point while the other handles those occurring after the decimal point. Figures 7.1(a) and (b) illustrate the conversions between a peano key and a pair of x, y -coordinates for integer and real values respectively.

Using a 4-byte integer value, the positive number of a peano key will range from 0 to 2^{30} (= 2^{32-2}). In this respect, 1 bit has already been used for the sign. Furthermore the number of binary digits used for a peano key should be an even number in order to decompose it equally into two sets of binary values, in which case, the values of x and y coordinates will range from 0 to 32,768 (= 2^{15}). Therefore, if the side length of the smallest quadrant (called the minimum side length) in a linear quadtree is s (m) on the ground, then the area covered by the quadtree will be 32,768s by 32,768s (m^2). Depending on the size of the minimum side length, the extent of this coverage may or may not be sufficient for GIS applications. The minimum side length of a linear quadtree is determined by the complexity of the geographical data and the maximum number of entries (called the threshold) that can be allowed to be contained in a quadrant. In order to search efficiently for a specific data object from the list of data objects occurring within a quadrant, the threshold value usually has to be kept reasonably small. Thus a small value of the minimum side length will result from the quartering of geographical data for areas with a high density of data. As a result, the whole national grid system of a country may not be able to be accommodated within the coverage extent of a linear quadtree using integer peano keys.

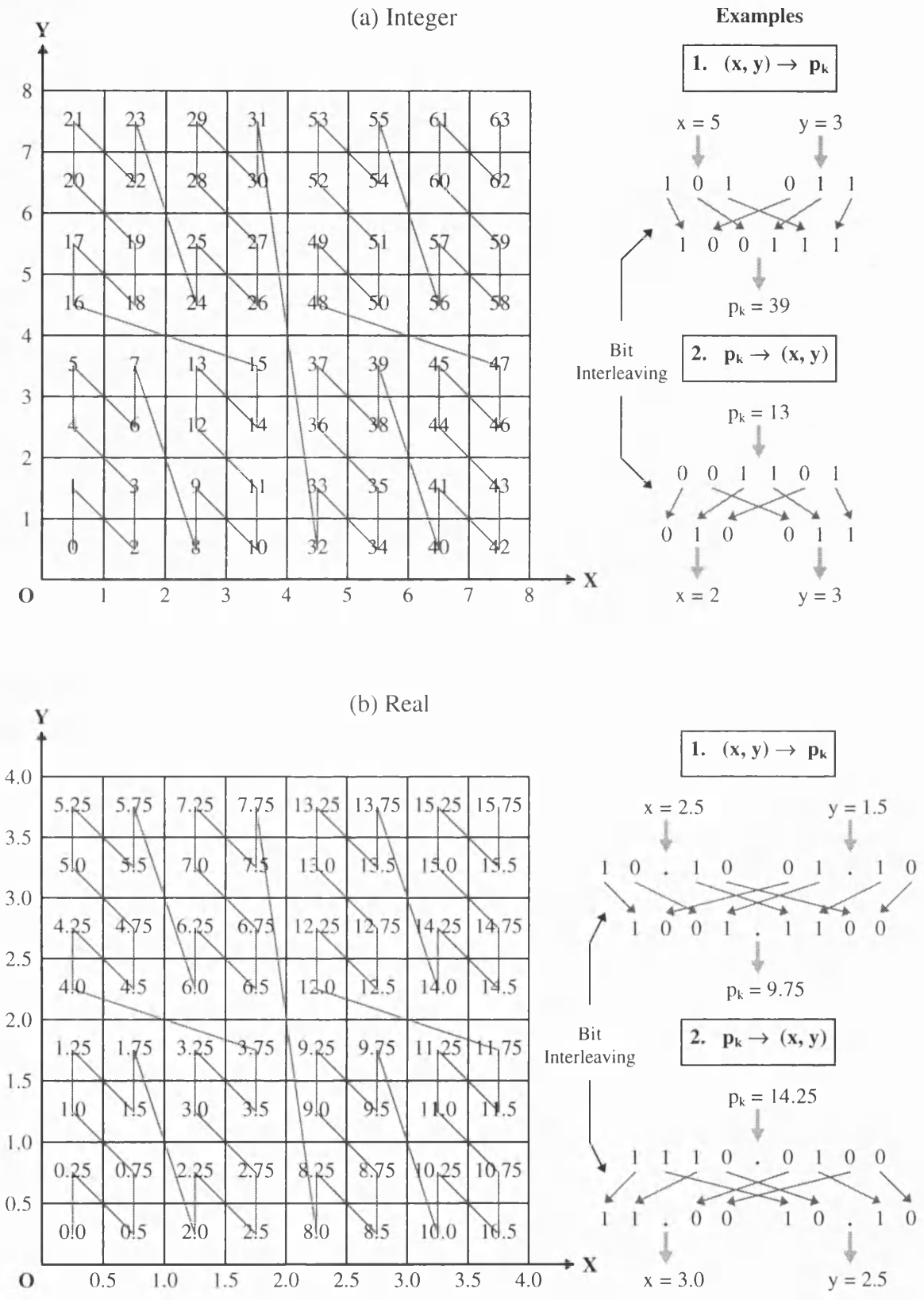


Figure 7.1 Conversions between a peano key and an xy-coordinate pair

The solution to the above problem is to express the peano key as a real value. However, in terms of GIS applications, it is quite meaningless to do further quartering if the minimum side length is smaller than 1 m. In other words, the digits after the decimal points can be truncated. Thus using an 8-byte real value, the positive value of a peano can range from 0. to $2.^{62}$ (*i.e.* maximum 19 digits), and the values of x and y coordinates will range from 0. to $2.^{31}$ (2,147,483,648). The range of these values appears to be sufficient for any GIS application so far envisaged.

Each representation of the peano key, *i.e.* using either an integer or a real value, has its pros and cons. Therefore, two sets of procedures (See Appendix C) have been developed to meet various application requirements. The procedures *pkToXY* and *xyToPK* deal with integer values, whereas the procedures *pkrToXY* and *xyToPKR* handle real values.

7.3 Spatial Indexing of Geographical Data

As has already been discussed in Section 2.2, the vector and raster formats are the two fundamental representations of geographical data. The geometric characteristics of both formats are entirely different. The vector format represents geographical features through the use of three basic entity types - point, line and polygon - whereas the raster format depicts all of them as a matrix of cells or pixels. The raster format has good spatial properties which are inherent in the direct addressing of the pixels. Hence, the pixel values of geographical features can easily be obtained in order to display their attributes. For example, after a baseimage has been displayed on the screen, a mouse-controlled cursor can be used to point at a specific pixel of a geographical feature to obtain the row and column numbers of this pixel relative to the image origin. The procedure *getPixel* provided by the Napier88 Standard Library can be used to determine the pixel value. Thereafter, the pixel value is assigned to an attribute code from which the attribute of the geographical feature may be acquired. Thus it can be seen that raster geographical data is inherently indexed by a matrix or an array structure. However, the raster format is rather inefficient in terms of being able to display and manipulate geographical entities individually. For example, it is very easy to display all the pixels which represent a specific theme among geographical features, but it will require much effort to extract a specific feature or entity for further manipulation.

On the other hand, the vector format represents geographical entities in a natural way, *i.e.* the geographical data is basically feature-based. Comparing it with the use of the raster format, the selective retrieval of geographical vector data can be achieved easily if geographical entities have been spatially indexed using an appropriate method. As a result, the querying of geographical features based on the vector format plays a major role in GIS

operations. Therefore, this section concentrates on the provision of a facility for the spatial indexing of vector geographical data within the persistent IGIS.

7.3.1 General Aspects of Indexing Vector Map Data

In terms of vector geographical data handling, for the persistent IGIS, the three basic entity types (point, line and polygon) in a map dataset have been aggregated into a data object, *e.g.* a *basemap*, associated with an appropriate data model. The vector map data needs to be retrieved from the *Processed* database in a similar manner to that employed for the display of a basemap (See Subsection 6.2.1). Fig. 7.2 illustrates the primary steps required for the spatial indexing of vector map data. Because the geometric characteristics of the three entity types are quite different, so the spatial indexing of each data type needs to be handled individually. In order to facilitate the spatial indexing operation, first of all, the mbr tables for both the line and polygon entities are constructed. The details of this operation will be described in the subsection below (7.3.2). In addition, a temporary coordinate table which correlates an entity identifier with its geometric location is constructed for each entity type. The use of these coordinate tables is to simplify the task of retrieving coordinates for indexing geographical entities. In other words, the retrieval of coordinates can be obtained from a unified form of coordinate table independent of the various geometry tables used with the different data models. Afterwards, three empty tables are created for the construction of the index data required for each of the three entity types. This is followed by the critical part of spatial indexing - the quartering and indexing of the vector map data for each entity type. The dashed-line block shown in Fig. 7.2 represents this core part of the spatial indexing. The quartering operation will vary depending on the entity types, whereas the indexing operation remains the same for each entity type. The quartering operation will be further discussed for each entity type respectively in the subsections 7.3.3 - 7.3.5. As for the indexing operation, it can be carried out using procedures provided by the Napier88 *Maps* Library. The indexing process required in the spatial indexing operation is illustrated in Fig. 7.3.

Initially, the index table can be set up to contain only a single entry which represents a quadrangle covering the area needed for spatial indexing. The initial side length may be determined from the extent of coverage of the quadrangle. For example, in terms of the spatial indexing required for a basemap, the initial index table will contain an entry which consists of a location key and a list of entity identifiers. The location key comprises two elements - a peano key and a side length. As has been discussed in Section 6.5, any quadrant will be uniquely identified using the composite key of these two elements. The composite key *Peano* using integer values for dealing with map and image indices has

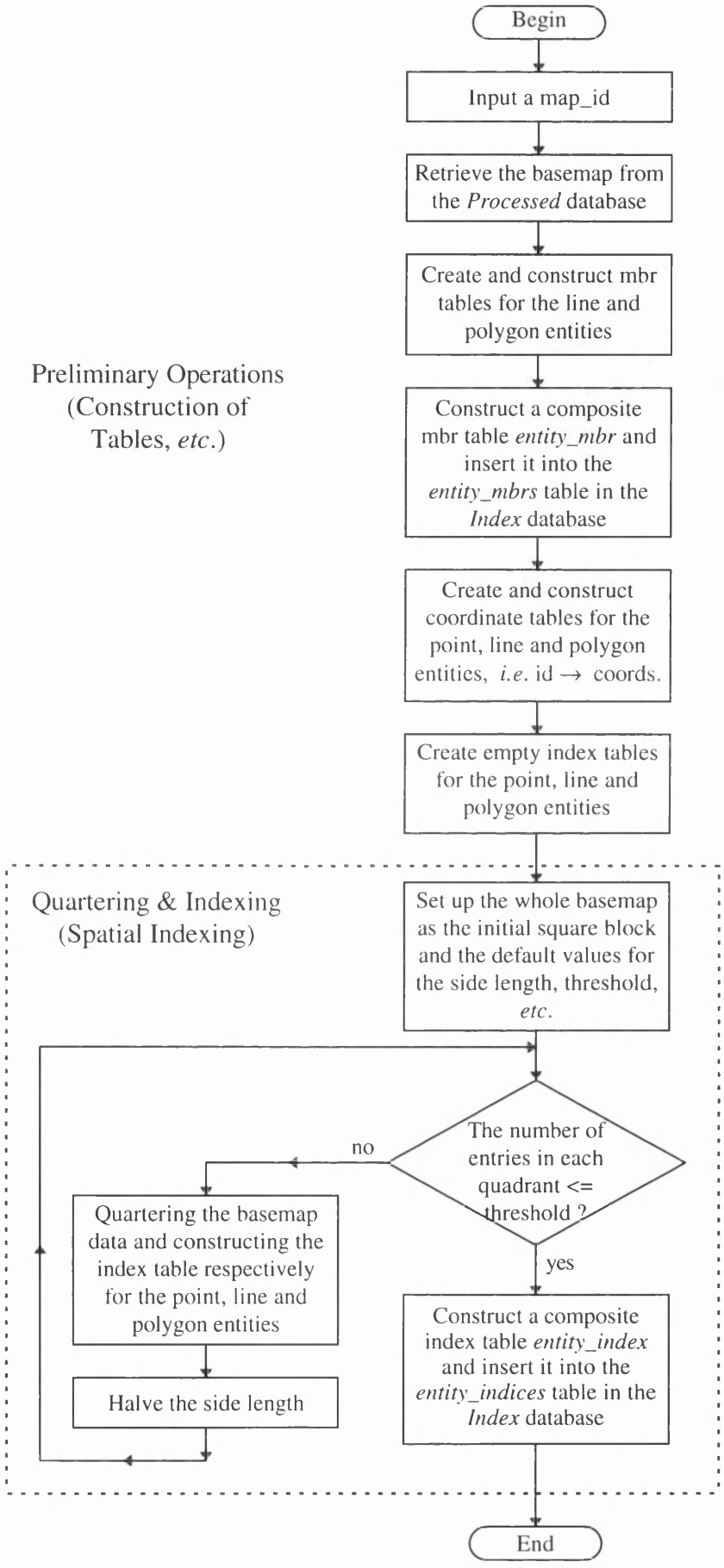


Figure 7.2 The primary steps required for the spatial indexing of vector map data

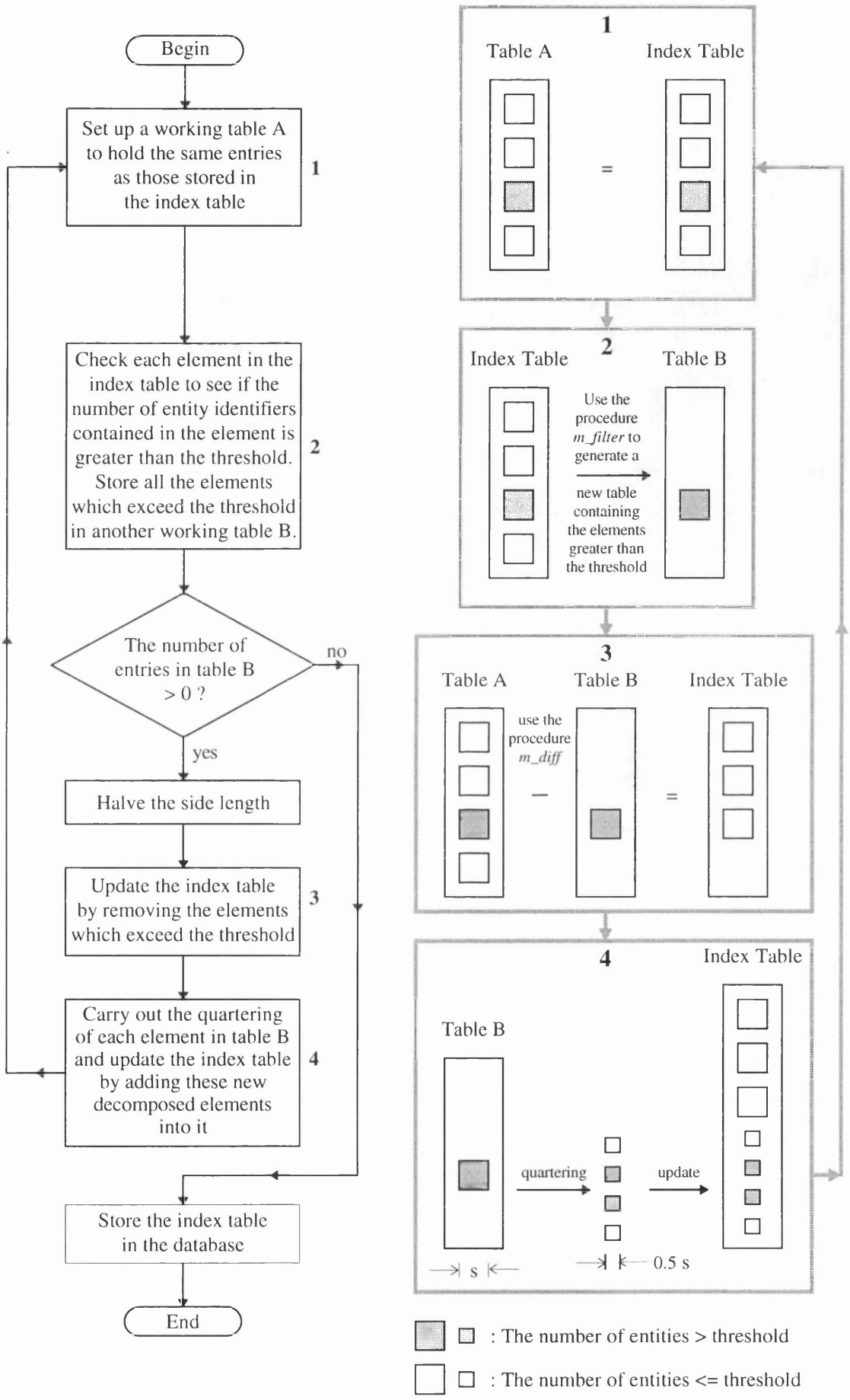


Figure 7.3 The indexing process in the spatial indexing operation for vector map data

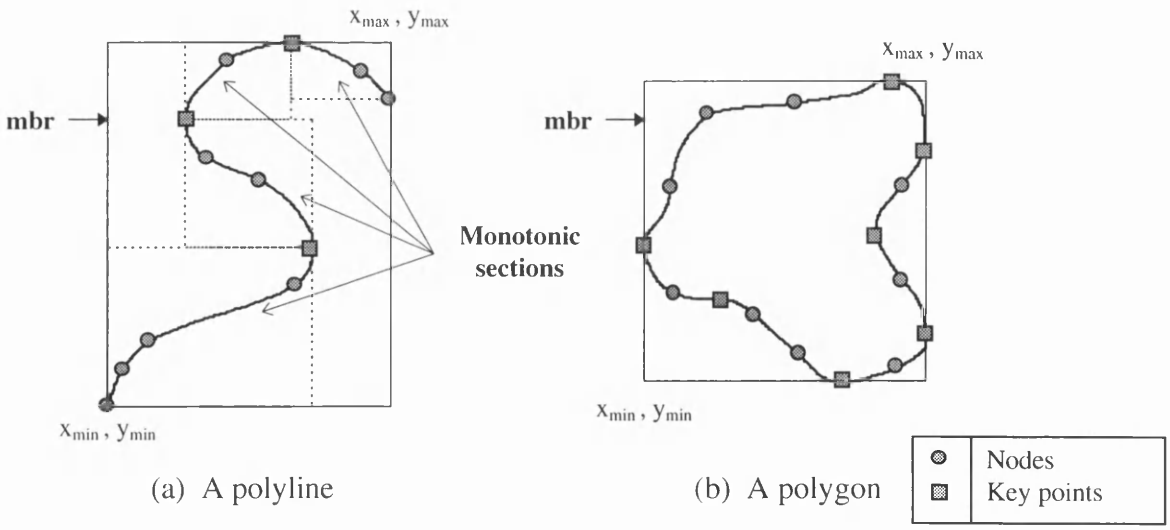
already been defined. In order to extend the range of potential applications, a composite key *Peanor* using real values has also been defined for the use in the entity index tables. The *Peanor* key as well as these entity tables are declared as follows: -

```
type Peanor is structure(peano_key, side_length: real)
type Point_index is Map[Peanor, List[Point_id]]
type Line_index is Map[Peanor, List[Line_id]]
type Polygon_index is Map[Peanor, List[Poly_id]]
```

Using these data types, it can be seen that, after performing the operation of spatial indexing, each entity table will contain the essential information about all the quadrants in a basemap, *i.e.* a peano key, a side length and a list of entity identifiers for each quadrant. This quadrant information can then be used in the search of geographical data.

7.3.2 The Construction of MBR Tables for Line and Polygon Entities

The minimum bounding rectangles of line and polygon entities will provide some limiting conditions for queried entities when the program requires access to and retrieval of spatial data. The mbr of a polyline or polygon can be determined from the minimum and the maximum coordinates of the set of points defined by its vertices. Fig. 7.4 illustrates this concept. In geographical data handling, a polyline or polygon comprises a string of lines. These are represented as a list of coordinate pairs, *i.e. List[XY]*, in the persistent IGIS. Therefore, the mbr of a polyline or polygon can be determined by comparing the coordinate values of each element in the *xy_list*. The procedure *getLineStrMBR* (See Appendix D) has been developed for this purpose.



(a) A polyline (b) A polygon

Figure 7.4 Minimum bounding rectangles for a polyline and a polygon

Because the mbrs created for the spatial indexing of geographical entities will also be used in the operations of querying these entities, so it is advantageous to store them in the database rather than recompute them every time they are needed. In order to store the mbrs for a basemap, the following data types have been declared for their use in the persistent IGIS.

```
type Line_mbr is Map[Line_id, MBR]
type Polygon_mbr is Map[Poly_id, MBR]
type Line_key_pts is Map[Line_id, List[XY]]
type Entity_mbr is structure(line: Line_mbr;
                             polygon: Polygon_mbr;
                             key_pts: Line_key_pts)
type Entity_MBRs is Map[Map_id, Entity_mbr]
```

Thus a mbr table can be constructed for each of the line and polygon entities that are contained in a basemap dataset, namely the *line_mbr* and the *poly_mbr* tables of the data types *Line_mbr* and *Polygon_mbr* respectively. Furthermore these two mbr tables may be combined together into a single table *entity_mbr* of the data type *Entity_mbr*. Afterwards, the *entity_mbr* is inserted into the *entity_mbrs* table of the data type *Entity_MBRs* in the *Index* database.

As has been mentioned before, the (linear) quadtree method is less efficient for the spatial indexing of lines. Generally speaking, the average coverage of the mbrs of the lines in a data set is usually larger than that of the polygons. As a result, the quartering carried out for line data often requires more processing time than that required for polygon data. In order to facilitate the spatial indexing for line data, a particular method has been employed in the operation of creating mbrs for lines. This is carried out by breaking a non-monotonic (poly)line into the monotonic components of that line. The characteristic of a monotonic line is that the x or y coordinates of the vertices will increase or decrease continuously, *i.e.* monotonically, in either the x- or the y- direction, starting from one end point and proceeding to the other end point of the line. Fig. 7.4(a) shows a polyline that has been decomposed into four monotonic sections. The key points where a line is split are constructed as a *list*. A table of the data type *Line_key_pts* which contains a list of the key points for each line in a basemap is then constructed to join the *line_mbr* and the *poly_mbr* tables in the combined table *entity_mbr*. All these tables containing the limiting conditions of line and polygon entities are stored in the *Index* database. Thus they can be used in the operations of spatial indexing and querying.

It should be noted that the "monotonic line" technique can also apply to deal with polygons, particularly very large polygons. However, the trade-off involved in using this technique needs to be justified. Although the technique can improve the efficiency of spatial indexing and queries, it will also increase the processing time and the storage space required for the creation of the *entity_mbr* table. Considering the fact that, in general, the average coverage of polygon mbrs is quite substantially smaller than that of the line mbrs, therefore the "monotonic line" technique has not been implemented for polygons in the persistent IGIS.

7.3.3 Spatial Indexing of Points

In this subsection, the quartering operation of spatial indexing for points will be described. The quartering of a square block for points is quite a simple process. Initially, the point index table contains only a single entry, *i.e.* the square block covering the whole area which has to be spatially indexed. The extent of the coverage of the square - (x_{min} , y_{min}) and s - can be determined from its spatial key *Peanor*. Also, the number of points within this square can be counted from its corresponding list of point identifiers. If the number of points exceeds the threshold of points, then this square is subdivided into four equal-sized quadrants (which are again squares). Because the extent of each quadrant may be easily determined, therefore the spatial key *Peanor* of each quadrant can be created. The result is that all the points indexed by the *Peanor* key of the quartering block at a tree level will be replaced by the *Peanor* keys of the four quadrants at the subtree level. Fig. 7.5 illustrates the concept of indexing points. The points 1 and 2 indexed by the same *Peanor* key of the initial square block will be replaced by the *Peanor* keys of quadrants 1 and 3 respectively.

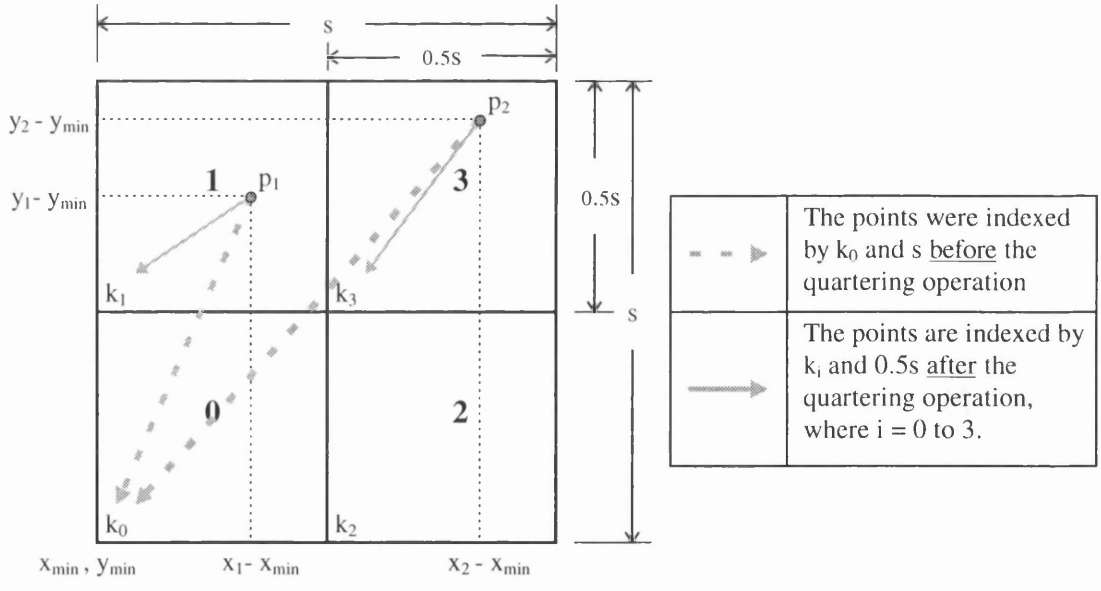


Figure 7.5 The concept of indexing points in the quartering operation

The basic principle of quartering points is to assign each point contained in a quartering block into one of the four quadrants based on its location. In fact, this process creates a new list (or child-list) of point identifiers for each quadrant from the initial list (or parent-list) of point identifiers relevant to the square being quartered. The assignment of a point identifier from the parent-list to one of its child-lists can be carried out as follows: -

(1) Retrieve the coordinates (x, y) of a point from the point coordinate table using the point identifier.

(2) Determine the quadrant in which the point is located using the following formulae: -

$$m = \text{int}[(x - x_{min}) * 2 / s]$$

$$n = \text{int}[(y - y_{min}) * 2 / s]$$

$$q = 2 * m + n$$

where x_{min} , y_{min} are the coordinates of the SW corner of the square being quartered,

s is the side length of the square being quartered,

m, n are the row and the column numbers of a quadrant; these values range from 0 to 1,

q is the quadrant number by *Peano* ordering; the value ranges from 0 to 3,

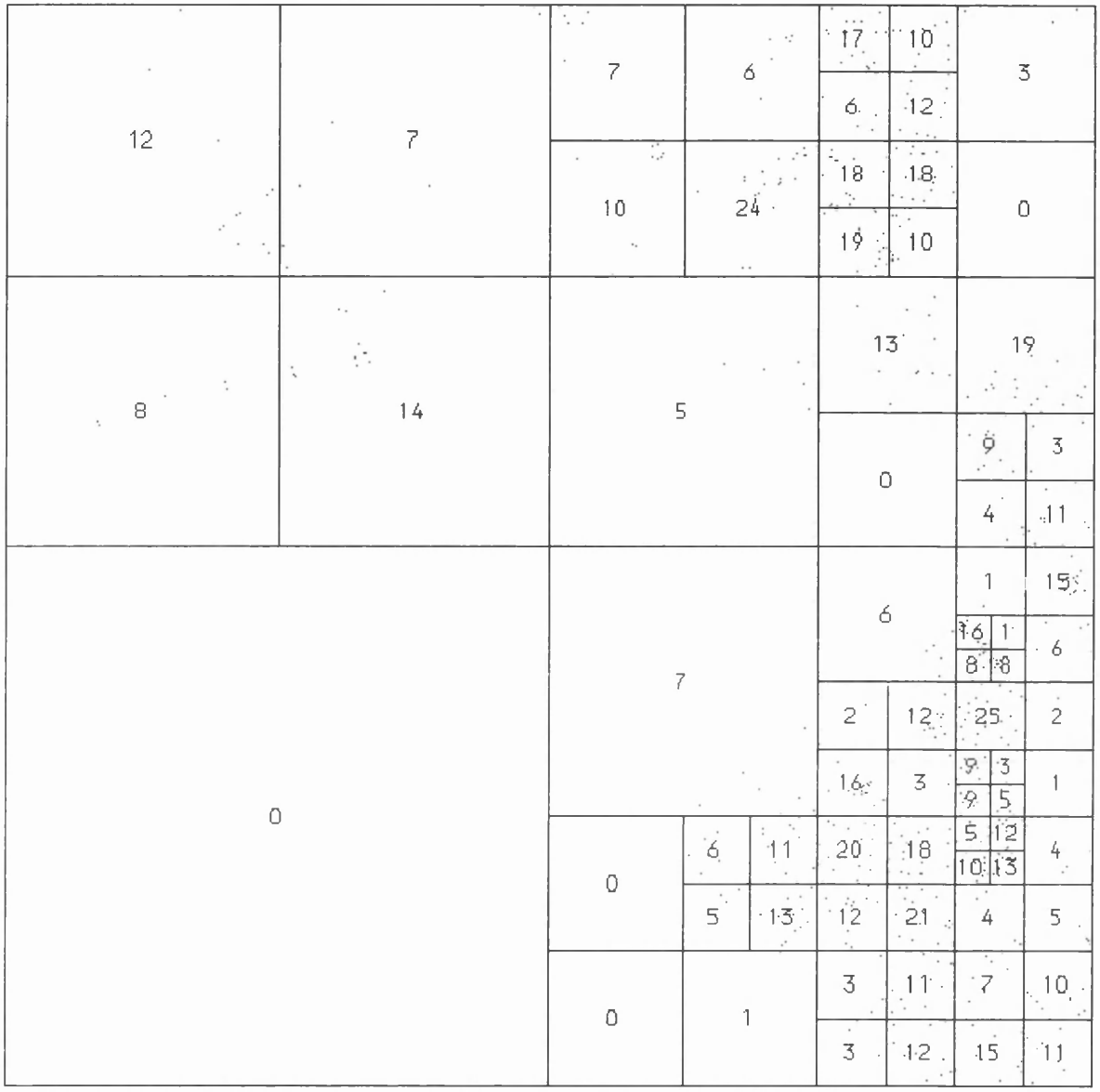
int[ ] represents the integer part of a real value.

(3) Based on the quadrant number determined for the point, append the point identifier to its corresponding list.

After the four child-lists have been constructed, they are inserted into the point index table *point_index* based on their corresponding *Peanor* keys. Also the parent list is removed from the point index table. The same quartering operation will be repeated at the subtree levels until all the entries in the point index table have reached the predetermined condition, *i.e.* the number of points in each quadrant must not be greater than the threshold.

Based on the concept described above, the procedure *lqtNdxPoint* (See Appendix E) has been developed for the indexing of points. Fig. 7.6 shows the linear quadtree diagram after indexing the points contained in the OS map 270190. The numbers in the diagram indicate the number of points being indexed in each quadrant. The accompanying table shows the side length, the total number of quadrants and the number of quadrants which are greater than the threshold at each level of the linear quadtree. In this instance, the quartering operation of the point data has been carried out to tree level 5 so that the number of points in each quadrant is not greater than the threshold.

The threshold value (25) employed in this example is quite arbitrary. The smaller the threshold value that is used, the longer the processing time and the larger the storage space



Level	Side Length (m)	Total No. of Quadrants	Greater than Threshold
0	5000.000	1	1
1	2500.000	4	3
2	1250.000	13	6
3	625.000	31	11
4	312.500	64	3
5	156.250	73	0

The number of points = 700; Threshold = 25;
The mininum side length = 156.25 m; The number of quadrants = 73

Figure 7.6 The linear quadtree diagram after indexing the points contained in the OS map 270190

that is required for spatial indexing. On the other hand, a larger threshold value may result in a slower response time when carrying out a search of the geographical data. Therefore, an optimal threshold is required to give an IGIS a better performance. However, a number of factors will affect the choice of an optimal threshold, including the available computing power, the type of entity to be indexed, the complexities of data models and data structures, the algorithm used for spatial queries, and so on. Therefore, it is necessary to tune the threshold for the persistent IGIS before it can be put into practical applications. The adjustment of the threshold for each entity type will be discussed further in Chapter 8.

7.3.4 Spatial Indexing of Polygons

In much the same way as was done with the indexing of points, so the extent of the coverage of a square block and a list of the polygon identifiers needed within the square can be acquired in advance. The quartering operation can be carried out by checking each polygon against the four quadrants to see whether the polygon overlaps some or all of them. If a polygon overlaps a quadrant, then the polygon identifier is indexed by the *Peanor* key of that quadrant. The check for the overlap between the boundaries of a polygon and those of a quadrant can be divided into two steps: -

(1) Retrieve the mbr of a polygon from the *poly_mbr* table (See Subsection 7.3.2) using the polygon identifier and determine whether the mbr of the polygon overlaps the quadrant or not.
   - if they do not overlap, then the process is stopped.
   - if they overlap and the mbr is completely within the quadrant, then the polygon identifier is indexed by this quadrant and the process is stopped; otherwise the program proceeds to step 2.

(2) A further test is carried out to find out whether the boundary (*i.e.* the line strings and line segments) of the polygon actually overlaps the quadrant or not.
   - if the mbr of a line string intersects the quadrant, then the containment test is performed for the line segments (- the algorithm is described in next subsection -) to find out whether the polygon boundary actually intersects the quadrant. If so, then the polygon identifier is indexed by this quadrant.

These two steps involve a process called the "boxing test" (or the min-max test). Figure 7.7 illustrates the algorithm of the boxing test. The first step requires only a single test between the mbr of a polygon and a quadrant, but the second step may involve many tests between the mbrs of the line strings of a polygon and a quadrant.

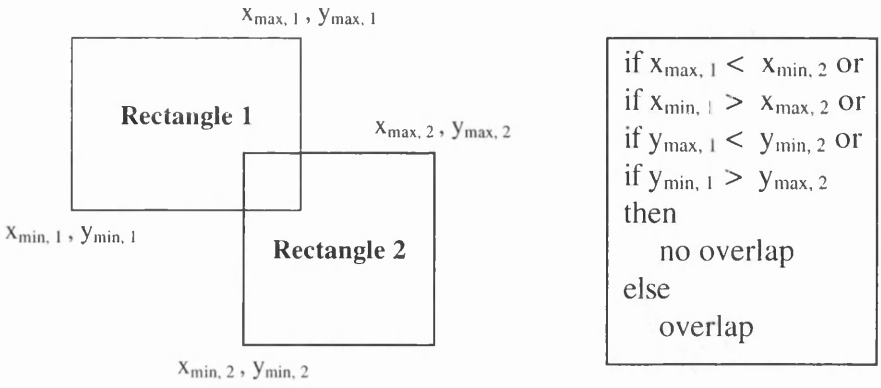


Figure 7.7 The overlap test of the two rectangles

Figure 7.8 illustrates the concept of indexing polygons utilising the quartering operation described above. Polygon A lies completely within a quadrant, but this is not the case with polygons B and C. The mbr of polygon B overlaps four quadrants, but its actual boundary only overlaps three of these quadrants (0, 1 and 2). The mbr of polygon C overlaps quadrant 3, but its actual boundary doesn't. Both polygons B and C require the boxing test to be carried out for their polygon boundaries. The boxing test for the mbrs of the line strings is sufficient to conclude that polygon C does not overlap quadrant 3. However, it still cannot eliminate the possibility that polygon B may be indexed by quadrant 3 since the mbr of one of its line strings overlaps that quadrant. Therefore, it is necessary to further perform the containment test on the line segments of this particular line string (This will be described in more detail later in Section 7.3.5). As a result, polygon A is indexed by the *Peanor* key of quadrant 2, while polygon B is indexed by the *Peanor* keys of three quadrants, 0, 1 and 2.

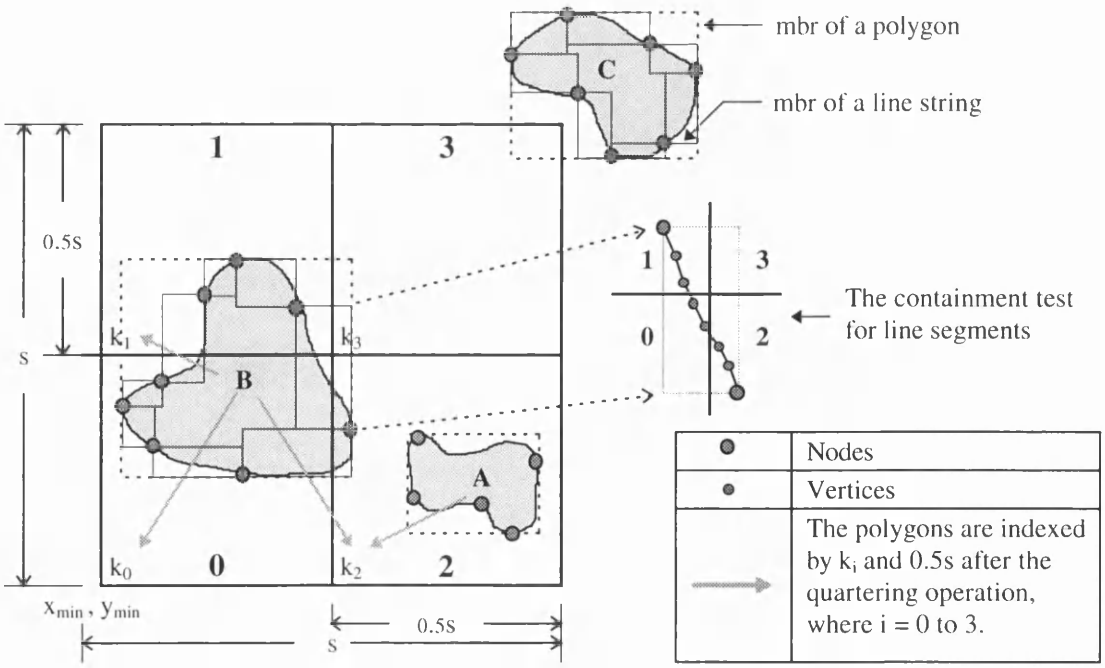


Figure 7.8 The concept of indexing polygons in the quartering operation

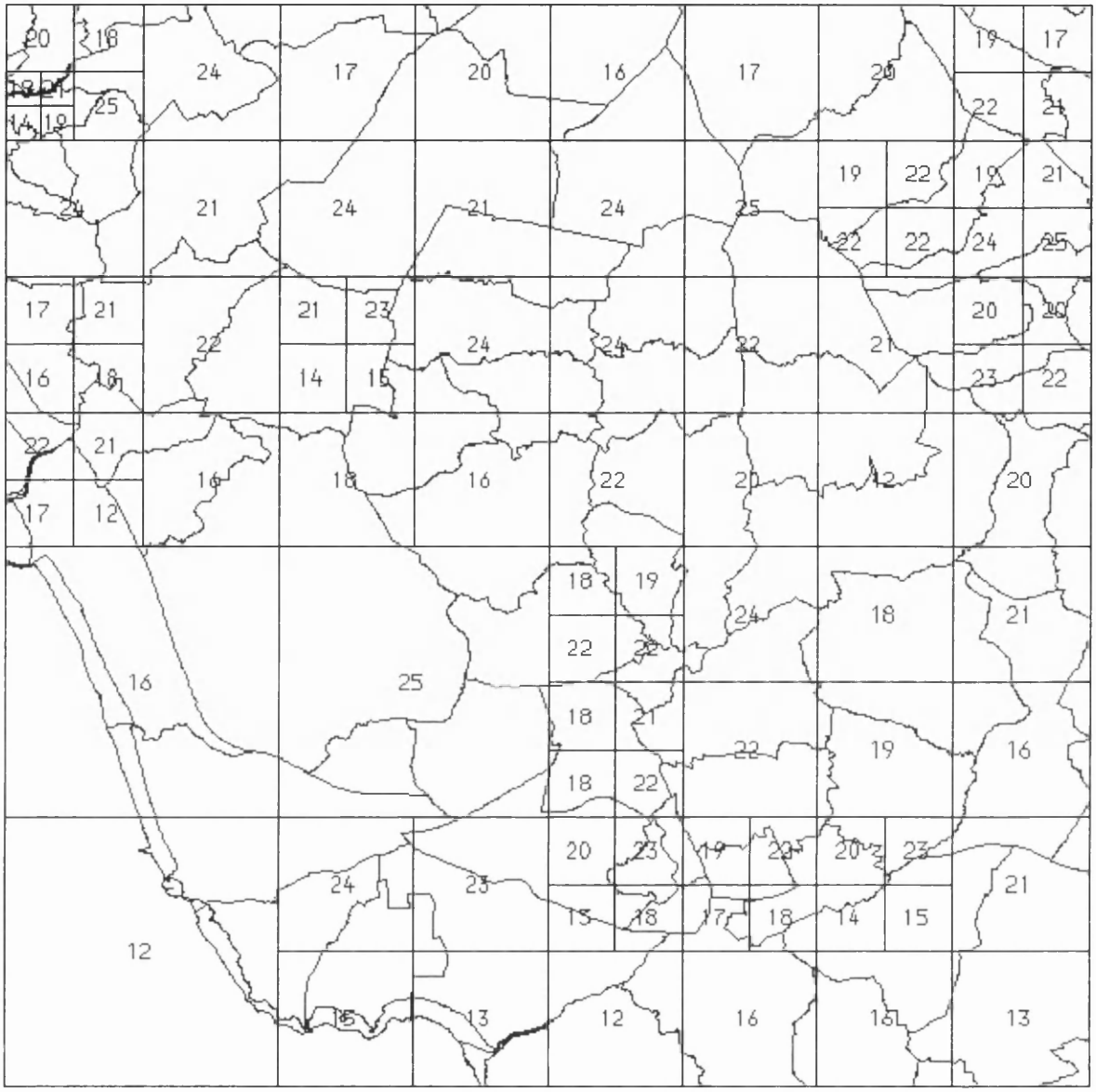
The second step of this exact indexing approach is a quite time consuming process because a polygon often comprises a number of line segments. Furthermore, each polygon needs to be tested four times, *i.e.* a test has to be conducted against each quadrant. As a result, the quartering operation for polygons will become rather lengthy and inefficient. In order to overcome this drawback, a simple but approximate indexing approach has been employed in the persistent IGIS. The quartering operation of this approach is based on the mbrs of polygons rather the boundaries of polygons. This approximate indexing approach can be described as follows: -

(1) Retrieve the mbr of a polygon from the *poly_mbr* table and determine the mbr of the overlap area between the polygon and the quadrant.

(2) Use the mbr of the overlap area to determine in which quadrant(s) the polygon identifier should be indexed.

Using this approach, polygon B in Fig. 7.8 will also be indexed to quadrant 3, where in fact they do not overlap. Hence, several redundant polygon identifiers may be added to a quadrant. However, they can easily be eliminated during the operation of spatial queries for polygons using the point-in-mbr test, the boxing test, *etc.* Although this approach slightly increases the storage space that is necessary, it can significantly reduce the time required for indexing polygons. Therefore, the approximate indexing approach must be regarded as being more efficient than the exact indexing method.

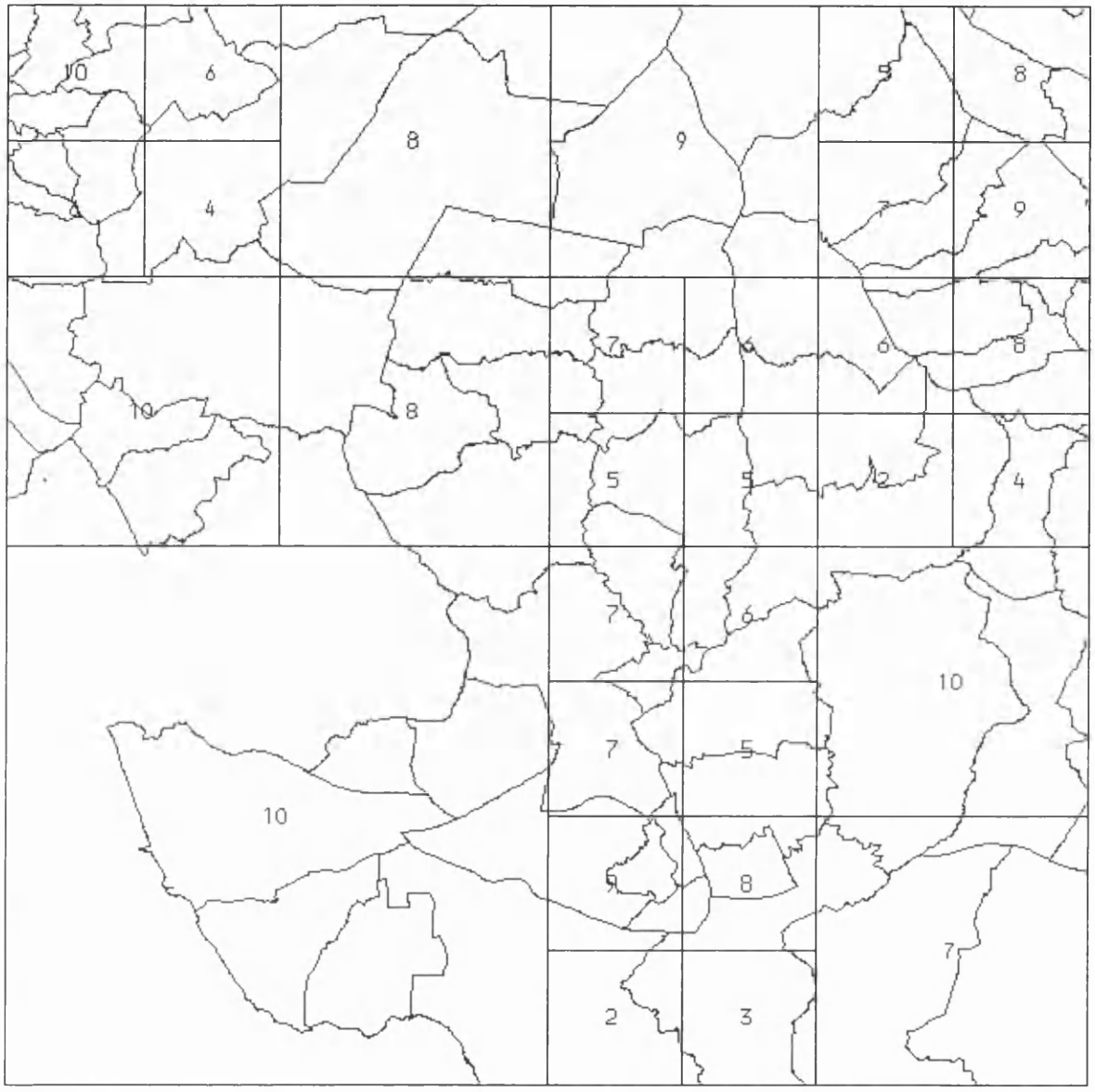
Based on the approximate indexing approach, the procedure *lqtNdxPoly* (See Appendix E) has been developed for the indexing of polygons. Fig. 7.9 shows the linear quadtree diagram after indexing all the polygons in OS map 2144. The numbers in the diagram indicate the number of polygon mbrs being indexed in each quadrant (n.b. the mbrs themselves are not shown). The accompanying table gives a summary of the results of this particular spatial indexing operation similar to those that have been described for Fig. 7.6 in the previous subsection. Because this map employs the polygon-based data model (See Subsection 5.3.3) and consists of complex polygons (*i.e.* polygons that can be nested within each other), so it is not possible to count visually the number of polygon mbrs occurring in each quadrant. Therefore, a set of simple polygons whose feature code = 3901 (District Ward) has been extracted from map 2144. This data set has then been indexed and is shown in Fig. 7.10. As such, the number of polygon mbrs in each quadrant can be counted. Also, it can be seen that some quadrants contain the mbrs but not the boundaries of polygons; this is the case more particularly for small quadrants.



Level	Side Length (m)	Total No. of Quadrants	Greater than Threshold
0	25000.000	1	1
1	12500.000	4	4
2	6250.000	16	13
3	3125.000	55	13
4	1562.500	94	1
5	781.250	97	0

The number of polygons = 237; Threshold = 25
The minimum side length = 781.25 m; The number of quadrants = 97

Figure 7.9 The linear quadtree diagram for the polygons of OS map 2144



Level	Side Length (m)	Total No. of Quadrants	Greater than Threshold
0	25000.000	1	1
1	12500.000	4	3
2	6250.000	13	6
3	3125.000	31	0

The number of polygons = 75; Threshold = 10
The minimum side length = 3125 m; The number of quadrants = 31

Figure 7.10 The linear quadtree diagram for the polygons of the district wards (feature code 3901) of OS map 2144

7.3.5 Spatial Indexing of Lines

The concepts involved in the spatial indexing of lines are basically the same as those which have been described above for points and polygons. However, the line entity requires more complex processing in its quartering operation than the point and polygon entities. After the extent of the coverage of a square block has been established and a list of the line identifiers contained within the square has been retrieved, each line is then checked against its four quadrants to see whether the line passes through some or all of them. If the line passes through a quadrant, the line identifier is indexed by that quadrant. The process of checking whether a line passes through a quadrant can be described as follows: -

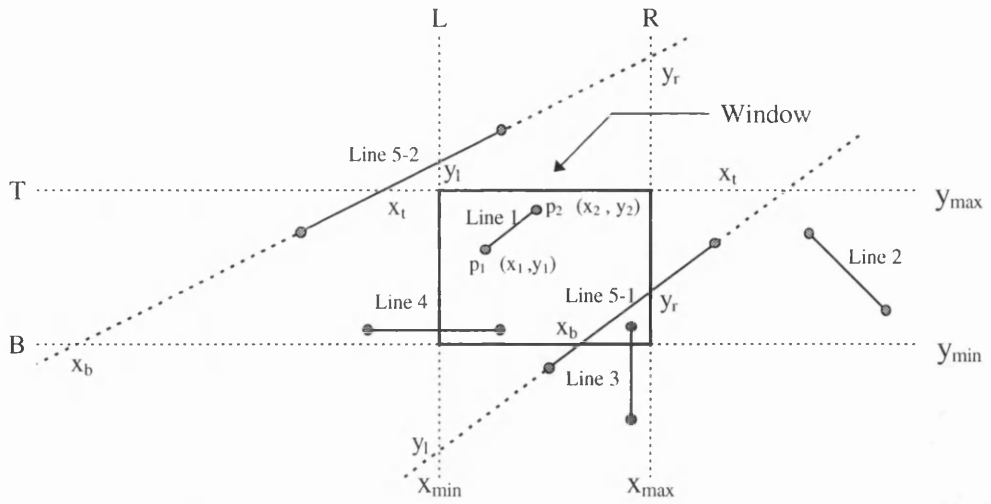
(1) Check whether the mbr of the line intersects the quadrant.
   - if the mbr lies completely within the quadrant, then the line identifier is indexed by this quadrant and the process will stop; otherwise the program will proceed to step 2.

(2) Check each monotonic section of the line to see if any part of it intersects the quadrant.
   - if none of the monotonic sections intersects the quadrant, the process is stopped; else it proceeds to step 3.

(3) Check each line segment in the monotonic section which intersects the quadrant to see whether a line segment intersects the quadrant.
   - if so, the line identifier is indexed by the quadrant.

Steps 1 and 2 require the use of the boxing test between the mbr and the quadrant - which has been described in the previous subsection. The monotonic sections of a line used in the step 2 can be retrieved from the *line_key_pts* table (See Subsection 7.3.2). Step 3 requires a line containment test to determine whether a line segment is either completely contained or partially contained within a rectangle. The algorithm of the line containment test is illustrated in Fig. 7.11 [Harrington, 1987; Mortenson, 1989].

Fig. 7.12 illustrates the concept of indexing lines. The mbr of line A lies completely within quadrant 1 which ensures that the whole line must be located inside the quadrant. The mbr of the line C overlaps quadrant 3 but the line is actually located outside the quadrant. This can be determined by using the boxing test on the mbrs of its monotonic sections. As for line B, it passes through three quadrants (0, 2 and 3). However, both the mbr of the whole line and the mbr of the monotonic section 2 overlap quadrant 1. Furthermore, the mbr of the second line string in the monotonic section 2 still overlaps quadrant 1. Therefore, the



Test whether a line is visible (either completely or partially contained) or invisible in a window
if $x_{min} <= x_1, x_2 <= x_{max}$ and $y_{min} <= y_1, y_2 <= y_{max}$
then the line is completely visible → **Case 1**. eg. Line 1
else the line is either partially visible or invisible
if $x_1, x_2 < x_{min}$ or $x_1, x_2 > x_{max}$ or $y_1, y_2 < y_{min}$ or $y_1, y_2 > y_{max}$
then the line is invisible → **Case 2**. eg. Line 2
else the line is either partially visible or invisible
when the line is vertical, i.e. $x_1 = x_2$ → **Case 3**
the line is visible. eg. Line 3.
when the line is horizontal, i.e. $y_1 = y_2$ → **Case 4**
the line is visible. eg. Line 4.
when the line is neither vertical nor horizontal → **Case 5**
compute intersections with $x = x_{min}$, $x = x_{max}$, $y = y_{min}$ and $y = y_{max}$ to get y_l, y_r, x_b and x_t respectively.
if $x_{min} <= x_b <= x_{max}$ or $x_{min} <= x_t <= x_{max}$ or $y_{min} <= y_l <= y_{max}$ or $y_{min} <= y_r <= y_{max}$
then the line is visible. eg. Line 5-1.
else the line is invisible. eg. Line 5.-2.

Figure 7.11 The line containment test

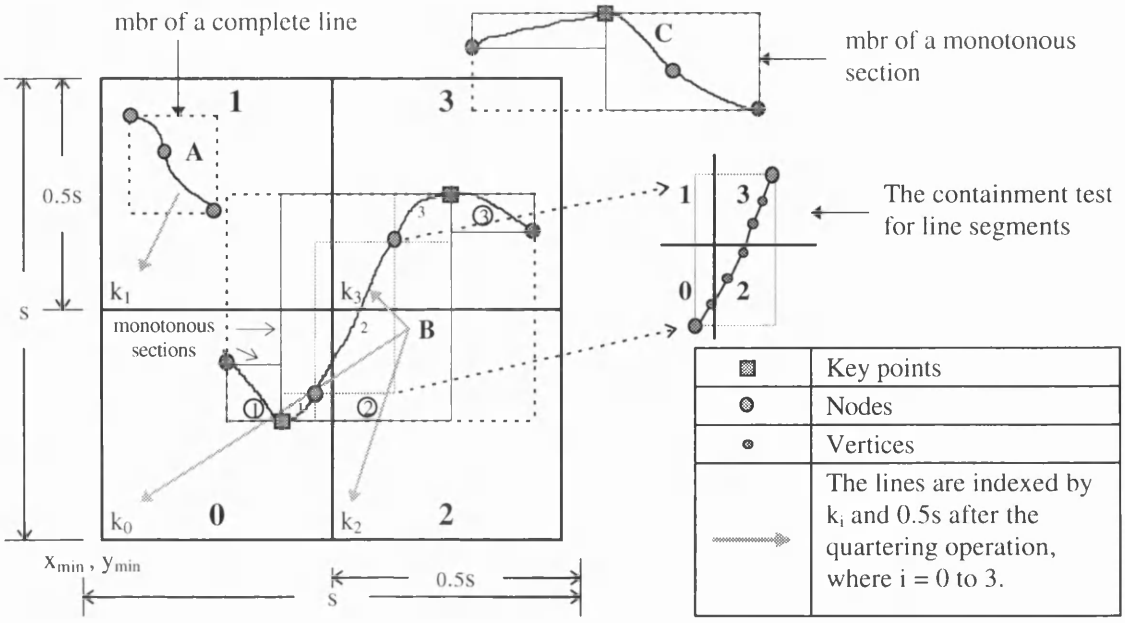


Figure 7.12 The concept of indexing lines in the quartering operation

line containment test is needed to find out if any line segment in this line string passes through quadrant 1. As a result, line A is indexed by quadrant 1, whereas line B is indexed by quadrants 0, 2 and 3.

The procedure *lqtNdxLine* (See Appendix E) has been developed based on the quartering approach outlined above. The OS map 270190 has been used for the test of spatial indexing. Fig. 7.13 shows the linear quadtree diagram resulting from the operation of spatial indexing for all the lines contained in the map. The number in each quadrant indicates the number of lines which intersect that quadrant. The accompanying table provides a summary of the number of quartering operations carried out at each tree level. It should be noted that most of the lines shown in Fig. 7.13 are polylines, *i.e.* they may contain several simple lines going from one end point to the other end point. Therefore, it is not possible to count visually the number of lines indexed in a quadrant from the diagram.

7.4 The Construction of Combined Spatial Indices

Using the spatial indexing method discussed in the previous section, an index table can be constructed for each entity type in a map data set. In this way, three types of index tables (*point_index*, *line_index*, *polygon_index*) may be created for each map data set. However, depending on the data model employed, a map data set may not contain all the entity types, *e.g.* a DEM data set only comprises point entities. In terms of the particular data models used in the persistent IGIS, the spaghetti and the link and node data models contain point and line entities, while the polygon-based data model includes all three entity types. Therefore, the spatial indexing of a basemap will always create two index tables - *point_index* and *line_index*, and may also produce an additional index table - *polygon_index*. Thus two compulsory elements and one optional element are required in the construction of a composite index table. The optional element may easily be defined by a **variant** type supported by Napier88. However, considering the possibility that a basemap may be restructured into other data models for particular applications, this operation may create polygon entities and, in this case, it may also require the construction of a polygon index table. Therefore, an empty polygon index table is always included by default in the operation of creating index tables for a basemap using the spaghetti or the link and node data model. Thus the polygon index table is treated as a compulsory element in the composite index table.

On the other hand, knowledge of the minimum side length of the quadrants produced by the spatial indexing procedure is essential for the use of these index tables, especially in the search of geographical data. Because the spatial indexing for each entity is carried out independently, so three minimum side lengths will be created and will need to be combined into a single composite index table.

Level	Side Length (m)	Total No. of Quadrants	Greater than Threshold
0	5000.000	1	1
1	2500.000	4	3
2	1250.000	13	6
3	625.000	31	13
4	312.500	70	14
5	156.250	112	1
6	78.125	115	0

The number of lines = 865; Threshold = 25;
The minimum side length = 78.125 m; The number of quadrants = 115

Figure 7.13 The linear quadtree diagram for the lines of OS map 270190

Based on the above discussion, the data types required for the construction of a composite index table can be declared as follows: -

```
type Min_quad_sl is structure(point, line, polygon: real)
type Entity_index is structure(point: Point_index;
                               line: Line_index;
                               polygon: Polygon_index;
                               min_quad_sl: Min_quad_sl)
type Entity_indices is Map[Map_id, Entity_index]
```

Using these data types, a composite index table *entity_index* of the data type *Entity_index* will be created for each basemap and then inserted into an aggregate table *entity_indices* of the data type *Entity_indices*. The map identifier of the basemap is used as a key to access this table. Thereafter, the *entity_indices* table is stored in the *Index* database and is available for those applications involving spatial queries which will be discussed below.

7.5 Spatial Queries of Geographical Data

In a GIS, query functions are essential for the retrieval and the analysis of geographical data. The query functions deal with the search of geographical data using the conditions specified by the operator. The search conditions may be quite varied in different applications. However, a geographical search can be classified into one of three categories: it will either be attribute-based, location-based or a combination of both, *i.e.* involving the search of a database using attribute keys, spatial keys or both. The methods of attribute-based query have been well established in conventional DBMSs. They are also widely used in GIS software. For example, SQL (**S**tructured **Q**uery **L**anguage) has been a commonly-used database interface for a GIS that employs relational tables to handle attribute data [Egenhofer, 1992; 1994]. In other words, the spatial component of a geographical feature can easily be acquired and displayed through the search of its attribute component. By contrast, the methods of location-based (or spatial) query involve the complexities of data models and data structures as well as the considerable use of computational geometry. As a result, spatial queries are generally quite complicated both in their algorithms and in their implementations.

Various kinds of spatial queries may be defined to meet the needs of different applications. Several are regarded as important query types in GIS applications, including point queries (or query by pointing), zone queries, path queries and buffer zone queries [Laurini and Thompson, 1992]. Each query type may be carried out through the use of one or several geometric algorithms. Therefore the provision of spatial query functions for the persistent

IGIS mainly involves the selection of an efficient geometric algorithm for each query type and the implementation of these algorithms in conjunction with the spatial indexing structures which have been constructed beforehand.

The development of a complete set of spatial query functions requires a great deal of effort. This results mainly from the complexities of the computational geometry. Therefore, in this section, only several simpler query types will be discussed. Nevertheless, the concepts used in these simple query types can also be applied or extended to other more complex query types.

7.5.1 Queries and Searches by Pointing

The simplest type of spatial querying is to access and retrieve the descriptive information of a geographical entity by pointing to its spatial location on the map displayed on the screen using a cursor. Such a query by pointing includes two key operations: -

(1) The search for the entity identifier in the entity index table using the spatial key derived from the specified search point.
(2) The display of thematic and attribute information of the entity using the entity identifier found in the first operation.

The second (display) operation simply accesses the geometry and attribute tables and displays their information, so it will not be discussed further. Instead, the following discussion will be concentrated on the search methods employed in the persistent IGIS. Fig. 7.14 illustrates the flowchart covering the operation of searching for an entity in a basemap. Before carrying out the search for the entity, the relevant *entity_index* table and its associated data have to be retrieved from the *Index* database. The *entity_indices* table is first checked to see if it contains the *entity_index* table. If it does, the *entity_index* table, the minimum side length and the map length will all be retrieved. Also, the initial side length of the search block will be set to the minimum side length.

The coordinates (x, y) of a digitised or search point can be used to determine a square cell which has the size of the smallest quadrant, *i.e.* a quadrant at the leaf level, in which the search point may be contained. However, the square cell may not be contained in the *entity_index* table, because the quadrant which actually contains the search point has already been determined in the spatial indexing operation. Therefore, the purpose of determining the square cell is to provide an initial search block for the operation of looking up the actual quadrant which contains the search point in the *entity_index* table. The coordinates (x_0, y_0) of the south-west corner of the square cell containing a search point can be computed by the following formulae.

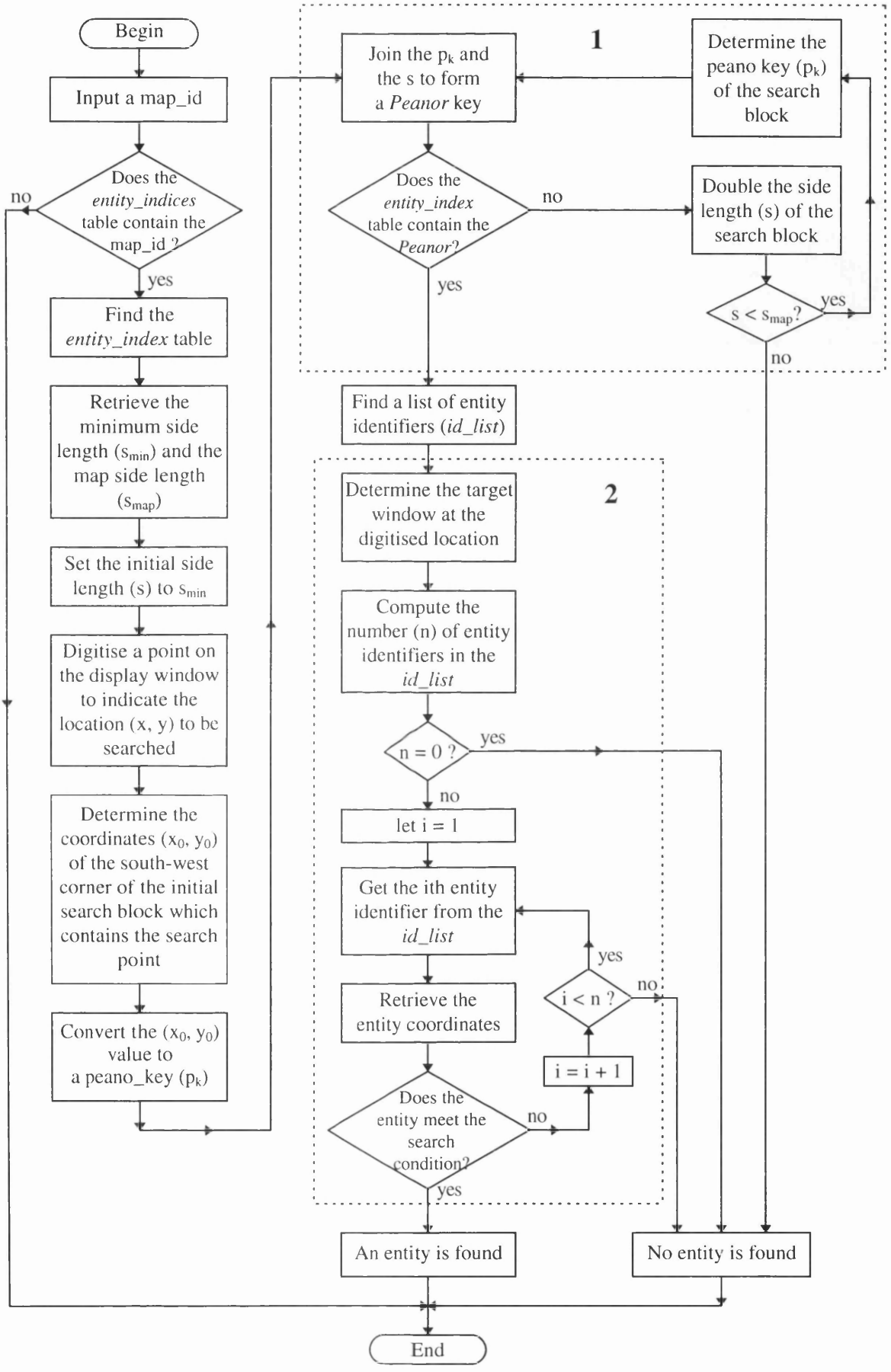


Figure 7.14 The flowchart covering the operation of searching for a specific entity in a basemap

$x_0 = \text{int}[x / s_{min}] * s_{min}$

$y_0 = \text{int}[y / s_{min}] * s_{min}$

where x and y are the coordinates of the digitised (or search) point;

s_{min} is the minimum side length of the smallest quadrant determined in the *entity_index* table; and

int[ ] represents the integer part of a real value.

The coordinate pair (x_0, y_0) is then converted to the corresponding peano key (p_k). Thereafter, the p_k key joins the s_{min} to form the *Peanor* key of the initial search block. This is followed by the quadrant search (dashed block **1**) and the "entity-meet-condition" test (dashed block **2**) which will be described below.

The quadrant search is used to find the quadrant which contains the search point in the *entity_index* table. The quadrant search starts with the initial search block being set to the square cell, *i.e.*, the search direction is from the leaf nodes towards the root of the quadtree. The *peanor* key of the search block is used to check whether the *entity_index* table contains this block. If the search block finds the corresponding quadrant in the *entity_index* table, then the test to satisfy the entity-meet-condition is carried out. Otherwise the side length of the search block is doubled to find its parent block which is used to redefine the search block required for the next search. The same process will be repeated until either the search block is found in the *entity_index* table or the side length of search block reaches the map length.

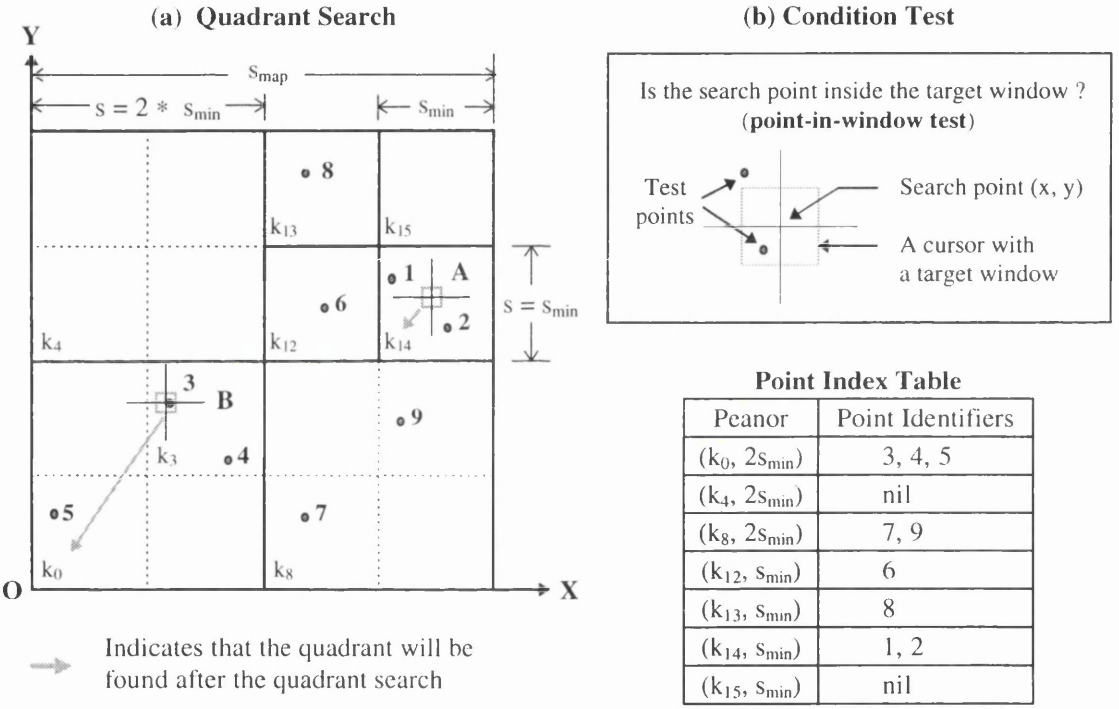
If the quadrant containing the search point is found in the *entity_index* table, then a list of the entity identifiers existing in this quadrant can be retrieved. These entity identifiers will be used to retrieve the corresponding sets of coordinates from the geometry table. Thereafter, each entity is tested against the search condition specified for this particular search - since three types of query by pointing are possible, *i.e.* querying by pointing to either a point, line or polygon entity. Since each entity type employs a quite different geometric algorithm for the search condition, therefore each of these will be discussed separately in the sub-subsections that follow.

7.5.1.1 Searching for a Point Entity

The search for a point entity requires an operator to identify it using a screen pointer or cursor. Because it is rather difficult for a user to control a point-like cursor to pick out a graphical entity, therefore the cursor is usually associated with a small target window to ease the picking or pointing operation. In terms of computational geometry, the use of a target window with the cursor is to replace the "point-match-point" test and the "point-on-line" test by the "point-in-window" test and the "window-contain-line" test (*i.e.* employing the line containment test described in Fig. 7.11). As such, the point-in-window test and the

line containment test will be the search conditions required for the search for a point and a line respectively. In fact, the point-in-window test is the only search condition required for searching out a point entity.

Fig. 7.15 illustrates the concept of searching for a point entity through the use of two examples of search cases (A and B). After the quadrant search has been carried out, the quadrant which contains the search point will be obtained. In case A, the first quadrant search will find the required quadrant because the *point_index* table contains its *Peanor* (k_{14}, s_{min}) key. In case B, the first search will fail (since key(k_3, s_{min}) is not in the table) but the second search will succeed (since key(k_0, $2s_{min}$) is in the table).



Point Index Table

Peanor	Point Identifiers
(k_0, $2s_{min}$)	3, 4, 5
(k_4, $2s_{min}$)	nil
(k_8, $2s_{min}$)	7, 9
(k_{12}, s_{min})	6
(k_{13}, s_{min})	8
(k_{14}, s_{min})	1, 2
(k_{15}, s_{min})	nil

Figure 7.15 The concept of searching for a point entity

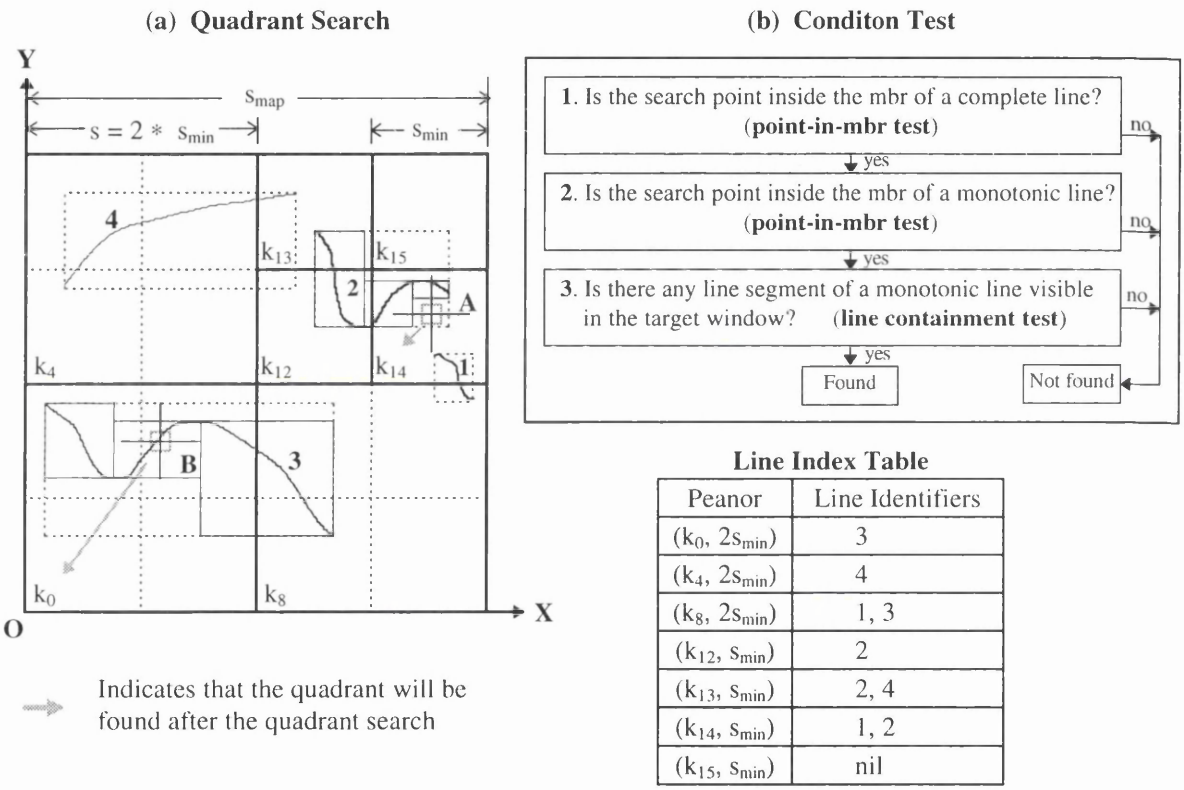
Having obtained the quadrant containing the search point, a list of the points indexed within this quadrant can be retrieved to perform the point-in-window test. Each point is tested against the target window of the search point to see whether the point lies inside or outside the window. If a point lies inside the window, then the point is found. For example, in case A, neither point 1 nor point 2 are to be found within the target window; whereas in case B, three points, 3, 4 and 5, may be tested but only point 3 will be found to lie within the window.

It is worthwhile mentioning that the size (or aperture) of a target window needs to be adjusted appropriately. The larger the given size, the easier it is for an operator to select a point of interest. On the other hand, the use of a smaller size of window can significantly reduce the chance of irrelevant points being included in the target window, but it will

become more difficult to select a specific point. Thus the size of the target window should be adjusted to the complexity of the map and to the zooming scale of the display.

7.5.1.2 Searching for a Line Entity

Fig. 7.16 illustrates the concept of searching for a line entity through the use of examples of two different search cases (A and B). The quadrant search operations required when searching for a line entity are exactly the same as those which have already been discussed in the search for a point entity. In case A, lines 1 and 2 will be found; whereas in case B, line 3 can be retrieved.



Line Index Table

Peanor	Line Identifiers
(k_0, $2s_{min}$)	3
(k_4, $2s_{min}$)	4
(k_8, $2s_{min}$)	1, 3
(k_{12}, s_{min})	2
(k_{13}, s_{min})	2, 4
(k_{14}, s_{min})	1, 2
(k_{15}, s_{min})	nil

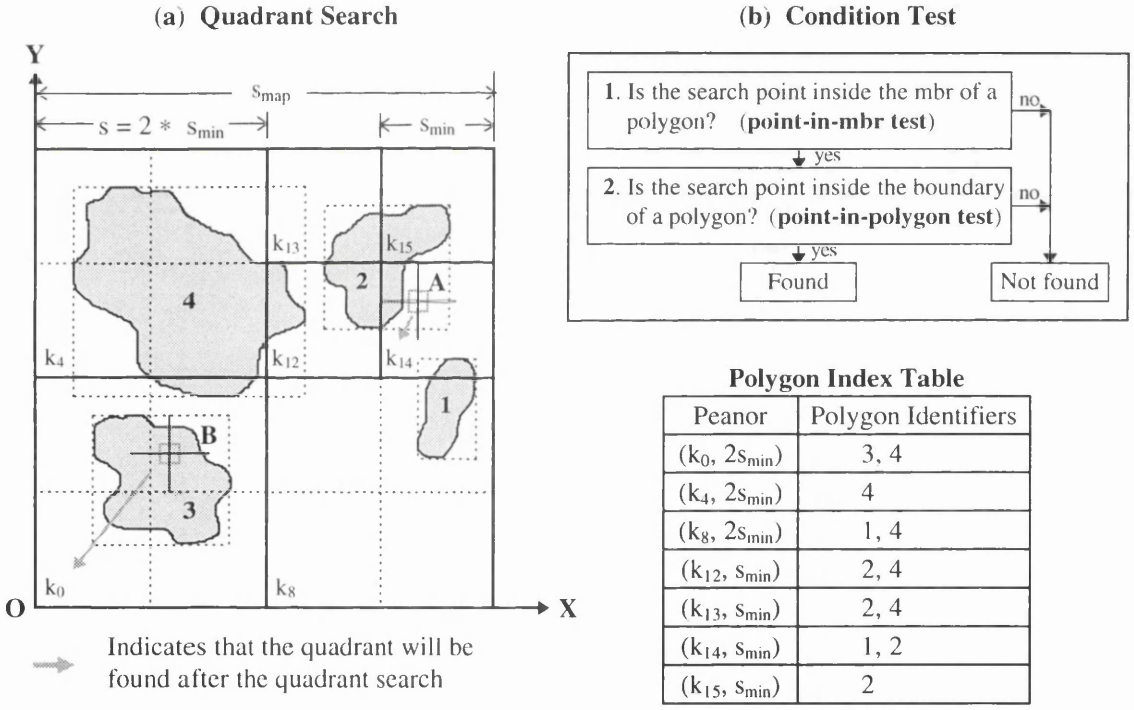
Figure 7.16 The concept of searching for a line entity

The tests required in the search for a specific line include the point-in-mbr test and the line containment test. The major steps of the condition test for a line are shown in Fig. 7.16(b). If a line passes one test, then the other test will be implemented; otherwise the line is excluded from this search. The first step checks whether the search point is inside the mbr of the whole line. The second step then checks whether or not the search point is contained in the mbr of a monotonic section of the line. Both the first and the second steps use the point-in-mbr test. If a line passes these two tests, then the line containment test will be carried out for the line segments of the particular monotonic line which passes the second test. If any line segment lies completely inside or passes through the target window, then the required line has been found.

The lines shown in Fig. 7.16 give the various situations that may occur in which a line will have to pass different test conditions. Considering case A, line 1 will be rejected after the first test because the search point lies outside the mbr of the line 1. Line 2 passes the first test but will fail in the second test because the search point will not be inside the mbr of its monotonic sections. Turing to case B, line 3 passes the first and second tests, therefore the third test will be carried out for the line segments of its second monotonic section. Since the line passes through the target window, so it will pass the line containment test. Line 4 will not be tested because it will not be found in the quadrant search.

7.5.1.3 Searching for a Polygon Entity

Basically, the principle of searching for a polygon entity is the same as those described for point and line entities. Fig. 7.17 gives the concept of searching for a polygon entity.



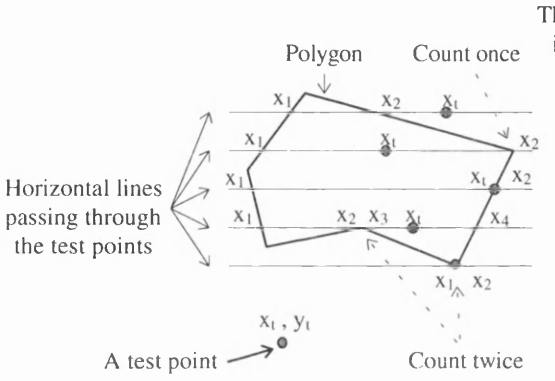
Polygon Index Table

Peanor	Polygon Identifiers
(k_0, $2s_{min}$)	3, 4
(k_4, $2s_{min}$)	4
(k_8, $2s_{min}$)	1, 4
(k_{12}, s_{min})	2, 4
(k_{13}, s_{min})	2, 4
(k_{14}, s_{min})	1, 2
(k_{15}, s_{min})	2

Figure 7.17 The concept of searching for a polygon entity

The tests which will be applied during the search for a polygon include the point-in-mbr test and the point-in-polygon test. Fig. 7.17(b) shows the two major steps required for the condition test. The first step tests whether the search point lies inside the mbr of a polygon. If it does, then the search point is further tested to ascertain whether it lies inside or outside the polygon. The first step uses the point-in-mbr test, whereas the second step employs the point-in-polygon test. The point-in-polygon test is an important geometric operation which is included in every GIS and may be implemented by any one of several algorithms. A particular form of a point-in-polygon algorithm has been adopted and slightly modified for

use in the persistent IGIS [Harrington, 1987; Mortenson, 1989]. Fig. 7.18 illustrates the basic concept of this particular point-in-polygon algorithm. The algorithm works well for both convex and concave polygons, including polygons with holes (or island polygons). Based on it, the procedure *pointInPolygon* (see Appendix E) has been developed for the search of a polygon.



The number of intersection points	Examples: Test	Result
2	$x_1 < x_2 < x_t$	outside
2	$x_1 < x_t < x_2$	inside
2	$x_1 < x_t = x_2$	on (the boundary)
4	$x_3 < x_t < x_4$	inside
2	$x_1 = x_t = x_2$	on (the vertex)

! Given a test point, determine if it lies inside, outside or on the boundary of a polygon.
If the test point lies inside the mbr of the polygon,
then compute the intersections of $y = y_t$ (*i.e.* the horizontal line passing through the test point) with the edges of the polygon
and count the number of intersection points.
If the horizontal line intersects a vertex of the polygon and both edge lines are on the same side of the horizontal line,
then count the number of intersections twice,
else count only once.
There is always an even number of intersections.
Sort the x-coordinates of the intersections in ascending order and form them into pairs, *i.e.* (x_1, x_2), (x_3, x_4) and so on.
If the x_t falls inside an interval of a pair, *e.g.* $x_1 < x_t < x_2$
then the test point lies inside the polygon,
else if the x_t is identically equal to one or both of the interval limits, *e.g.* $x_1 = x_2$ or $x_1 = x_t = x_2$,
then it lies on the boundary or on a vertex,
else it lies outside the polygon.

Figure 7.18 The concept of the point-in-polygon algorithm

In Fig. 7.17, the quadrant search will find polygons 1 and 2 in case A. After the first condition test, polygon 1 will be rejected. Although polygon 2 passes the first test, it will fail the second test. In case B, polygons 3 and 4 will be found in the quadrant search. Polygon 4 will then be rejected in the first test, but polygon 3 will pass both tests. Thus only polygon 3 will actually be found in these two cases of the polygon search.

7.5.2 Queries and Searches by Zones

Queries and searches by zones are perhaps the most commonly-used methods adopted in the querying of geographical information. Typical tasks carried out via a query by zones are the retrievals of points, lines, polygons or a combination of them contained within a zone. A search zone is usually identified on the screen using a mouse-controlled cursor. The shape of the zone may be quite varied, including such possibilities as a rectangle, circle, regular polygon, irregular polygon, *etc.* In this section, only the search by a rectangular zone is discussed.

Basically the principle of searches by zones is the same as that described for searches by pointing. Thus two main operations - the quadrant search and the entity-meet-condition test - are required. However, searches by zones may involve quite complicated geometric computations, especially when performing the condition test for the determination of the entities occurring in an irregular search zone. The condition test required for a rectangular search zone includes the point-in-mbr test, the line containment test and the overlay test of two rectangles. The searches for both point and line entities will use the same test procedures as those discussed in sub-subsections 7.5.1.1 and 7.5.1.2 respectively, except that the target window of a cursor is replaced by a rectangular zone. The search for a polygon entity requires, firstly, to test whether the mbr of the polygon entity overlaps the mbr of the search zone. If both rectangles overlap, then a second test will be carried out to find out whether the polygon boundaries (*i.e.* line strings) are contained within the search zone. All these geometric algorithms required for the search by a rectangular zone have already been described in previous subsections, therefore the condition test need not be discussed further.

Because a search zone may cover an arbitrary number of quadrants, this means that the quadrant search operation associated with the querying of a zone will require an individual quadrant search for each square cell (a square cell has the size of the smallest quadrant) covered by the search zone. For a rectangular search zone, the coverage of the square cells occurring within the rectangle can be determined by the following formulae: -

$$x_1 = \text{int}[x_{min} / s_{min}] * s_{min}$$
$$y_1 = \text{int}[y_{min} / s_{min}] * s_{min}$$
$$x_2 = \text{int}[x_{max} / s_{min}] * s_{min}$$
$$y_2 = \text{int}[y_{max} / s_{min}] * s_{min}$$
$$m = (x_2 - x_1) / s_{min}$$
$$n = (y_2 - y_1) / s_{min}$$

where (x_{min}, y_{min}) and (x_{max}, y_{max}) are the coordinate pairs of the SW and the NE corners of a rectangular search zone respectively;

s_{min} is the minimum side length of the smallest quadrant determined in the *entity_index* table;

int[ ] represents the integer part of a real value;

(x_1, y_1) and (x_2, y_2) are the minimal and the maximal coordinate pairs of the origins (the SW corners) of the square cells respectively; and

m and n are the number of square cells covered in both the x- and the y-directions respectively.

Thereafter, a quadrant search can be carried out by setting an initial search block to each square cell to determine which quadrant in its search path is contained in the *entity_index* table. All the unique quadrants found in the quadrant search can be saved as a list for the use in the subsequent condition test. Fig. 7.19 illustrates the flowchart of the quadrant search required for a rectangular zone. The rectangle bounded by the coordinates (x_{min}, y_{min}, x_{max}, y_{max}) represents a search zone. The quadrants within the shaded area will be found after the quadrant search.

Searches by zones can be used to retrieve different (point, line and polygon) entity types. Because each entity type has been individually indexed to create an index table, so the quadrant search has to be carried out for each of them. However, all the procedures required for the quadrant search can be combined together as a single operation, thus queries by zones allow users to acquire geographical information without needing to refer to their entity types. In a way, queries by zones may be viewed as an integrated form of the various types of queries by pointing.

7.6 Summary

Spatial querying is an important function in a GIS, and in turn, spatial indexing is a fundamental facility essential for the operation of spatial querying. This chapter has described the approach taken to provide these capabilities for the persistent IGIS. A spatial indexing method based on a linear quadtree and Peano ordering has been described and employed to index geographical data. The quartering and the indexing operations used for the construction of an index table for each entity type have also been discussed. Also, a set of procedures has been developed for indexing the basemaps held in the *Processed* database described in Chapter 5. Two map datasets have been selected for the tests of these indexing procedures. The linear quadtree diagrams generated by these tests show that each entity type can be indexed properly using this particular approach.

On the other hand, the functions required for the querying of geographical data are also described. Two primary query or search methods - querying by pointing and querying by zones - have been used to study the provision of querying functions for the persistent IGIS.

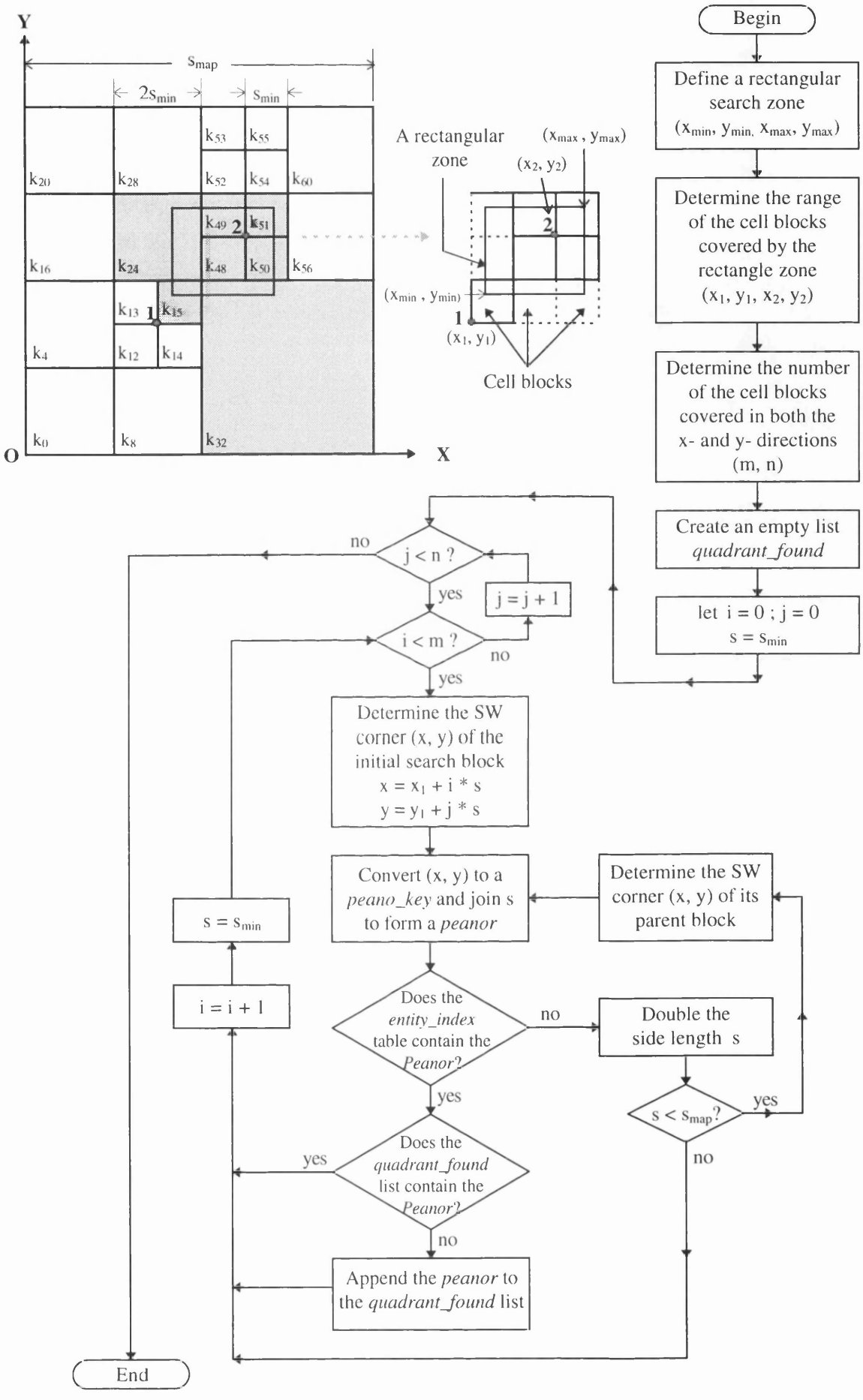


Figure 7.19 Quadrant search for a rectangular zone

The quadrant search and the entity condition test are the two basic operations required for each query type. Based on the querying approach presented in this chapter, several functions which can be used for the querying of an entity type as well as a rectangular zone have been developed for use in the persistent IGIS.

It is worthwhile mentioning that the Bulk Type Libraries *Maps* and *Lists* have been used extensively in the design and the construction of the spatial index tables used for the geographical data. These library procedures have proved to be very useful for carrying out the indexing task. In other words, the support of the Bulk Type Libraries by Napier88 has greatly facilitated the development of the facilities made available for spatial indexing and queries.

With the provision of these facilities, the persistent IGIS is able to achieve the process level of integration. When taken together with the storage level and the display level of integration reported in Chapters 5 and 6, the persistent IGIS has now realised a full degree of integration. Thus the persistent IGIS can be regarded as being a fully integrated GIS (or FIGIS), *i.e.* it is a true IGIS.

CHAPTER 8 : THE IMPLEMENTATION OF THE PROTOTYPE IGIS

8.1 Introduction

The previous three chapters (5, 6 and 7) have already discussed the design of the essential facilities required for the development of an IGIS. Each chapter has concentrated on the provision of a specific level of integration, namely the data modelling and organisation of various geographical data to achieve the storage level of integration in Chapter 5; the superimposition and interrelation of vector maps and raster images to provide the display level of integration in Chapter 6; and the spatial indexing and querying of geographical data to reach the process level of integration in Chapter 7. Each of these facilities has been implemented individually and has also been tested successfully on a specially selected dataset. These preliminary tests have indicated that the integration approach based on the persistent programming language Napier88 is very promising for the development of a true IGIS. However, the suitability of this approach still cannot be fully justified without providing more evidence from practical experiments.

In order to carry out a comprehensive series of tests, a prototype IGIS has been formed by joining the three essential facilities together and by further extending several of the basic functions required for an operational prototype. In addition, large volumes of geographical data have been acquired to evaluate the capabilities and the performance of the prototype IGIS. As such, two main aspects will be discussed in this chapter - the development of the prototype IGIS and the trials and tests carried out using this prototype.

This chapter first describes the user interface design employed in the development of the prototype IGIS. Then the functions and the linkage of the software modules implemented in the prototype IGIS are discussed briefly. This is followed by a description of the platform (hardware configuration and system software) and the data used in the trials and tests. Based on the resulting system which combines the software, platform and data, both a functional test and a performance test (which evaluate respectively the capability and the efficiency of the prototype IGIS) have been carried out. Finally, the findings derived from the trials and the test results will be presented and discussed.

8.2 User Interface Design

Quite apart from the construction of the various procedures required for the core modules, it is essential that a suitable user interface be included in an IGIS. The user interface, which is situated between the core system and the user, plays a particularly important role in GIS operations. In principle, the design of the user interface should implement and feature three

characteristics: ease of learning, ease of use and functionality. In practice, it is quite difficult to ensure that all three features can be provided in a single system without conflicting with one another. For example, an extensive emphasis in the interface design on ease of learning for novice users may result in a system that is not easy to use and lacks the capabilities which are desirable for skilled users. On the other hand, a system which supports the features of full functionality and ease of use for skilled and knowledgeable users may need to sacrifice those features required for ease of learning. Furthermore, the user interface design involves the consideration of a very large number of aspects, such as the layout of the display formats, the use of colours, the dialogue design, the menu design, the use of icons, the display of error messages and on-line help, and the use of various control and display devices, *etc.* [Brown, 1988]. The particular requirements needed to satisfy each of these can often be in conflict with one another. Therefore, the design of the optimal user interface is quite a complex issue to be resolved in the development of a GIS, particularly in the situation where the multi-media representation of geographical information is needed.

Graphical User Interfaces (GUIs) are probably the most widely used type of interface implemented in GIS software packages. The GUI makes use of graphical objects such as pull-down or pop-up menus, dialogue boxes, icons, scroll bars, *etc.* within a window system to provide the "look and feel" of the display screen presented to users of the system. In a proprietary window system such as Microsoft Windows, IBM's Presentation Manager (PM), the GUI is closely tied to the operating system and the underlying display hardware. In other words, the GUI is a specific software built into the window system. On the other hand, in a non-proprietary window system such as the X-window system, the GUI and the window manager that supports it are quite independent of the operating system and the display hardware. The X-window system is a software standard that provides a framework for window-managed applications to run over a network. The X-window system doesn't have a graphical user interface built in, but it provides a structure that supports many different types of GUIs. Commonly-used GUIs operating under the X-window system are OSF/Motif, OpenLook, DECWindow and Xview [Tektronix, 1991]. Within the X-window environment, application programs running on either a local or a remote system are called X-clients, and the special software running in the display system which provides services for client applications is known as an X-server. It is worth mentioning here that this concept is the opposite to the normal sense of the client/server architecture used in a networking environment, in which the display system is the client (on which the user sends requests and gets the display of the results) and the computer which processes the user's requests is the server (compute server, application server, print server, file server, *etc.*).

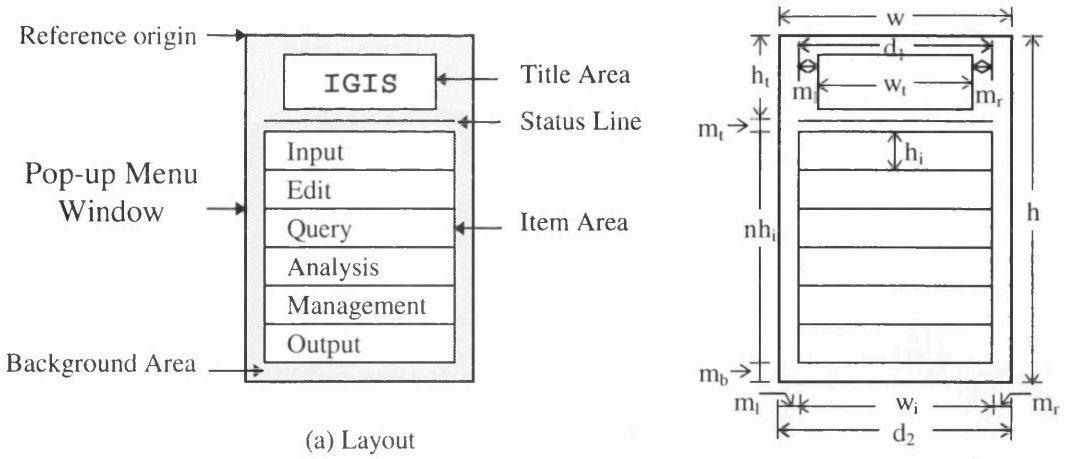
Napier88 provides its own window management system WIN which uses a proprietary graphical user interface. The concept of window management in Napier88 is quite different to that of a conventional window management system. In the WIN system, windows are

independent objects which are created by a separate window creation procedure rather than by a window manager. A window manager can be created to display and manipulate a set of (possibly overlapping) windows by supplying a parent window to the window manager creation procedure. Because windows may be organised into groups in a hierarchical manner, thus window managers can be nested recursively to any depth. The recursion is grounded by the root window which operates directly on the display device [Cutts *et al.*, 1989; Kirby *et al.*, 1994]. All windows may be held in the persistent store. Also all these window management operations can be carried out either in an X-window using an X-server or in a special graphical window utilising the local frame buffer of a workstation. Since WIN has already provided the basic facilities for programmers to create their graphic user interfaces, originally it was planned that the development of the prototype IGIS would make use of these facilities.

However, a number of problems have been encountered in the attempt to use WIN for the creation of the GUI. On the one hand, the implementation of WIN in Napier88 Release 1 does not support colours, but it can use colours in X-windows displayed on a local host or on a remote X-server running any conventional window manager. On the other hand, Release 2 provides colours for WIN, but neither WIN nor X-windows can properly display colours on a remote X-server. As an aside, Release 1 also had the problem of displaying colours in an X-window on a remote X-server at the very early stage of this research. The problem was first discovered by the author, then identified by the technical support staff in Glasgow and eventually solved by the system developer in St. Andrews. Since the author used a PC equipped with an X-server to run Napier88 programs over the network during the development of the prototype IGIS, (- the hardware configuration will be described in Section 8.4 -) it was not possible to make use of the WIN library procedures supported by either of the two Releases. Therefore in order to provide the GUI facility as well as the display of colours for the prototype IGIS, a built-in GUI has been developed based on Release 1. As a result, much effort has had to be expended to develop this GUI using the standard library procedures provided by Napier88. This particular GUI includes several graphical procedures which can be used to pop up a menu, generate a dialogue box and display a message in an X-window. The designs of the pop-up menu and the dialogue box are described in the following subsections.

8.2.1 The Pop-up Menu Design

A pop-up menu is a window that appears rapidly on the display device to provide options for a list of functions, from which the user can select an action to be executed. Depending on its internal design, the layout of pop-up menus and the operations available on the menu may be quite varied. The layout of one of the pop-up menus implemented in the prototype IGIS is illustrated in Fig. 8.1(a).



(a) Layout

(b) Dimensions

Figure 8.1 The design of a pop-up menu

The pop-up menu contains five components; the functions of each of these components are as follows: -

1. The title area is used to display the menu title.
2. The status line indicates whether the cursor is inside or outside the menu window.
3. The item area displays a list of menu entries.
4. The background area represents the coverage of the menu window.
5. The reference origin is used to determine the relative displacement when moving the menu window and to find out in which area the cursor is located.

The operations available on a pop-up menu depend on the use of the mouse and how the mouse buttons are actually used within each area. They can be categorised into two possible cases: -

(I) If there is no pop-up menu being displayed, holding down mouse button 3 will result in a pop-up menu being displayed with the reference origin at the location of the cursor.

- Continuously holding down mouse button 3 and moving the cursor within the menu window will result in the corresponding line or area being highlighted, *i.e.*
  - if the cursor lies within the menu window, then the status line will appear;
  - if the cursor is inside the title area, the characters of the title will be highlighted via a colour change of the text ;
  - if the cursor is inside the item area, the background area of the corresponding item will be highlighted.
- The release of mouse button 3 will result in the following actions: -
  - if the cursor is inside the title area, the menu window will be retained;
  - if the cursor is inside the item area, the command of the corresponding item will be executed and the menu window will not be retained;
  - if the cursor lies outside both the title and item areas, then the menu window will be dismissed, *i.e.* it will disappear.

(II) If there is a pop-up menu being displayed, moving the cursor within the menu window will highlight the corresponding area. The highlighted situations are the same as those described in case I except that, in this case, there is no necessity for the mouse button to be pressed.

- Clicking mouse button 2 in the title area will dismiss the menu window which will disappear;
- Clicking mouse button 1 in the item area will execute the command of the corresponding menu item;
- Holding down mouse button 1 in the title area de-highlights the title and the movement of the cursor controlled by the mouse will show the frame of the menu window being dragged to a new location, *i.e.* to the position where the cursor has been moved. The release of mouse button 1 will result in the movement of the menu window to the new location.

It should be noted that when using a two-button mouse, *e.g.* a Microsoft mouse, a third button can be emulated by clicking both mouse buttons at once. In the current implementation, the left and right buttons are assigned as buttons 1 and 3 respectively; the simultaneous depression of both the left and right buttons acts as button 2.

In order to accommodate the use of (i) various font types, (ii) the variable lengths of a text string and item strings; and (iii) the variable number of menu items in a pop-up menu, the size of the menu window should not be fixed. Instead, the width (w) and the height (h) of a pop-up menu will be determined by the following formulae (See Fig 8.1(b)): -

$$d_1 = m_l + w_t + m_r$$

$$d_2 = m_l + w_i + m_r$$

$$w = \max(d_1, d_2)$$

$$h = h_t + m_t + n * h_i + m_b$$

where d_1 and d_2 are the widths of the windows of the title and item areas respectively. N.B. although the diagram shows $d_2 > d_1$, the situation $d_1 > d_2$ can also be encountered;

m_l and m_r are the left and right margins of both the title and item areas;

m_t and m_b are the top and bottom margins of the item area;

w_t is the width of the title area;

h_t is a constant value which is defined by the maximal height of the available fonts plus the top and bottom margins;

w_i is the maximal item width;

h_i is the item height;

n is the number of menu items; and

max is a function giving maximum dimension of the two values d_1 and d_2.

The detection of the specific situation as to whether the cursor lies in a particular area or not can be carried out using the procedure *locator* provided by the Napier88 Standard Library. This procedure will continuously generate the information (cursor coordinates, status of buttons, *etc.*) about the pointing device in the form of a one-dimensional array which is represented as a vector of data type **int**, *e.g.* **let** data := **vector** 1 **to** 8 **of** 0. In the situation of using a mouse being associated with an X-window, the information contained in the vector elements is described in Table 8.1 [Morrison *et al.*, 1989; 1993].

Element No.	Value generated by Release 1	Value generated by Release 2
1	0	0
2	0	0
3	x	x
4	y	y
5	the status of button 1	a date stamp
6	the status of button 2	the status of button 1
7	the status of button 3	the status of button 2
8	0	the status of button 3

Notes: 1. x and y are the absolute coordinates of the cursor in terms of pixels with reference to the lower-left corner of an X-window.
2. The status of a mouse button is either down (1) or up (0).

Table 8.1 The vector elements generated by the procedure *locator* which is available from both Releases when using a mouse.

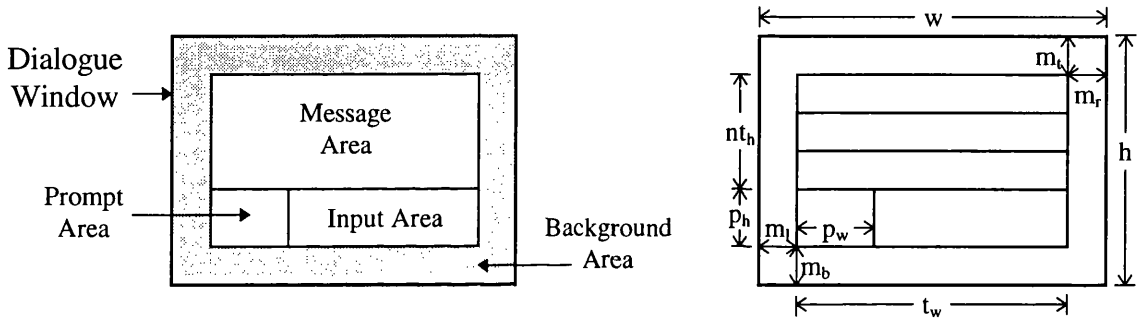
The absolute coordinates of the reference origin can be obtained from the cursor location where the menu is being popped up and these will also be updated when moving the menu window to a new location. Furthermore, the relative coordinates of the title and item areas which refer to the reference origin can be determined during the construction of a pop-up menu window. Therefore, the mbr of each rectangle in the menu window can be determined. Thus the cursor location can be easily tested to find out whether it is inside or outside a specific area. For example, if $x_{min\ ,\ item(i)} < data(3) < x_{max\ ,\ item(i)}$ and if $y_{min\ ,\ item(i)} < data(4) < y_{max\ ,\ item(i)}$, then the cursor lies within the area of the *i*th menu item (- it should be noted that the current implementation of the pop-up menu design is based on Release 1). The status of the mouse buttons will then be tested to perform the predefined functions. For example, if data (7) changes from 1 to 0, this means that mouse button 3 has been released from the holding state. This will activate the operation described in case I, *i.e.* the execution of the command corresponding to this particular item.

Using the above principle, the procedure *popupMenu* has been developed for use in the prototype IGIS (See Appendix D). The procedure takes a vector of text strings (including a menu title and several menu items), a vector of item actions and other parameters to generate a pop-up menu. The relationship between the menu items and the item actions is a

one-to-one mapping. A menu action is a procedure which may perform a GIS function or which may pop up a submenu. The use of pop-up menus can be nested to any depth. Thus the *popupMenu* procedure can be used to create a menu tree which comprises a hierarchy of available commands.

8.2.2 The Dialogue Box Design

A dialogue box is a pop-up window which displays a message to the user and into which data may be entered. Various forms of dialogue box are possible. For example, a dialogue window may contain a simple prompt with several check boxes allowing the user to make a selection or it may provide a message, a prompt and an input field requesting the user to input data. In general, the design of a dialogue box is simpler than that of a pop-up menu. Only one form of dialogue box has been developed for the current implementation of the prototype IGIS. The design of this particular dialogue box is illustrated in Fig. 8.2.



(a) Layout (b) Dimensions

Figure 8.2 The design of a dialogue box

The dialogue window contains four components; the functions of each of these components are as follows: -

1. The message area is used to display textual information, such as a text string or a list of map identifiers;
2. The prompt area is used to display a few words which suggest how to enter data in the input area;
3. The input area allows the user to enter data; and
4. The background area indicates the coverage of the dialogue window.

The operations available within the dialogue box are quite simple and can be described as follows: -

- Any keystroke of the displayable ASCII codes (lying in the range between 31 and 127, but excluding 31 and 127) will add a character into the input area starting after the prompt string;
- The "Delete" key (ASCII code = 127) can backspace one character at a time;
- The "Return" key (ASCII code = 10) is used to end the input of a string; and

- The "Esc" key (ASCII code = 27) or a click of mouse button 2 (*i.e.* data(6) = 1) will dismiss the dialogue window and discard the text string while entering data.

The number of text strings in the message area and the number of the characters in a text string are both variable, as is the size of a dialogue box. The width (w) and the height (h) of a dialogue box can be determined by the following formulae: -

$$w = m_l + t_w + m_r$$

$$h = m_t + n * t_h + p_h + m_b$$

where m_l, m_r, m_t and m_b are the left, right, top and bottom margins of the dialogue area - which is a combination of the message, the prompt and the input areas;

t_w is the width of the maximal text string;

t_h is the height of the text string;

p_h is the height of the prompt string; and

n is the number of text string.

Based on the above design, the procedure *dialogueBox* has been developed to provide the textual I/O capability within an X-window for the prototype IGIS (See Appendix D). The dialogue box can also be used to display a message only, *i.e.* when $p_h = 0$. In this instance, the dialogue box will act as a message display box without providing the prompt and the input areas.

8.3 Designing and Building the Prototype IGIS

Having described the provision of the basic GUI facility in the previous section, a truly integrated GIS can be developed with the feature that it allows most operations to be carried out in a common X-window. As has been discussed in the design of the system architecture (Chapter 4), the functionality of the persistent IGIS is intended to deal with the various forms of data required in different types of geographical data processing. However, the development of such a system with a complete set of functions will take at least several man-years. Since this research is aimed at the integration of geographical data, and in particular, the construction of a framework for vector and raster data integration, thus the feasibility of a persistent IGIS is the major concern at this stage. In order to test and evaluate the suitability of using the Napier88-based persistent IGIS in practical applications regarding information integration, the implementation of a prototype IGIS has been carried out to provide the basic facilities required to conduct this proof of the basic concept.

This prototype IGIS comprises five main modules (**View & Query**, **Spatial Indexing**, **Management**, **Pre-processing** and **Import/Export**). These five modules are represented as five optional items in the main menu. Each main module is composed of several sub-modules. Each sub-module may contain additional small modules or executable commands

(procedures) or a combination of both. Based on the use of pop-up menus, a menu hierarchy can be designed for the prototype IGIS. Fig. 8.3 illustrates the menu hierarchy employed in the persistent IGIS.

The construction of each main module involves the design of the data flow between its constituent parts (sub-modules or procedures) and the development of relevant procedures. The data flow required for each main module can be designed using a data flow diagram which is a graphical notation used to describe how data flows between processes in a system. The primary data flows in each of the main modules are shown in the flowcharts contained in Fig. 8.4 to 8.8.

Before carrying out software development, a fresh persistent store should first be obtained. This initial store contains the Standard Library, the Bulk Libraries, other utilities, *etc.* Next the data types (See Appendix A) needed in the prototype IGIS are designed and saved in the persistent store. This is followed by the operations of creating the store environments (See Appendix B) used for the storage of the GIS software library and geographical databases.

Having prepared the persistent store necessary for IGIS software development, every program procedure required in the development of the prototype IGIS had to be individually designed and tested. After this had been done, these various procedures have been organised into three program libraries and have been stored in the software *Library* within the *User* environment of the persistent store. Appendices C, D and E show the source listings of the *General*, *Graphical* and *GIS* libraries respectively. It should be noted that the methods involved in compiling and executing a Napier88 program and in developing a program procedure in the stable store by the incremental construction method have been described in Subsections 3.4.1 and 3.4.6 respectively.

Using these library procedures in combination with the flow diagrams and the GUI facilities, the overall prototype IGIS program has been designed and composed. Appendix F gives the source code of the prototype IGIS program. The compilation of the IGIS source program has been carried out against the persistent store in which the three libraries described above have been installed. After all the processes of debugging and testing (which are normally required in software development) had been completed successfully, an executable file in object code - the IGIS executable program - was generated by the Napier88 system. Both the executable program and the persistent store are simply two files available under the Unix operating system. However, they will act as an integrated GIS software and as an integrated geographical database respectively when running under the Napier88 system. Thus the program and the persistent store form the prototype IGIS system.

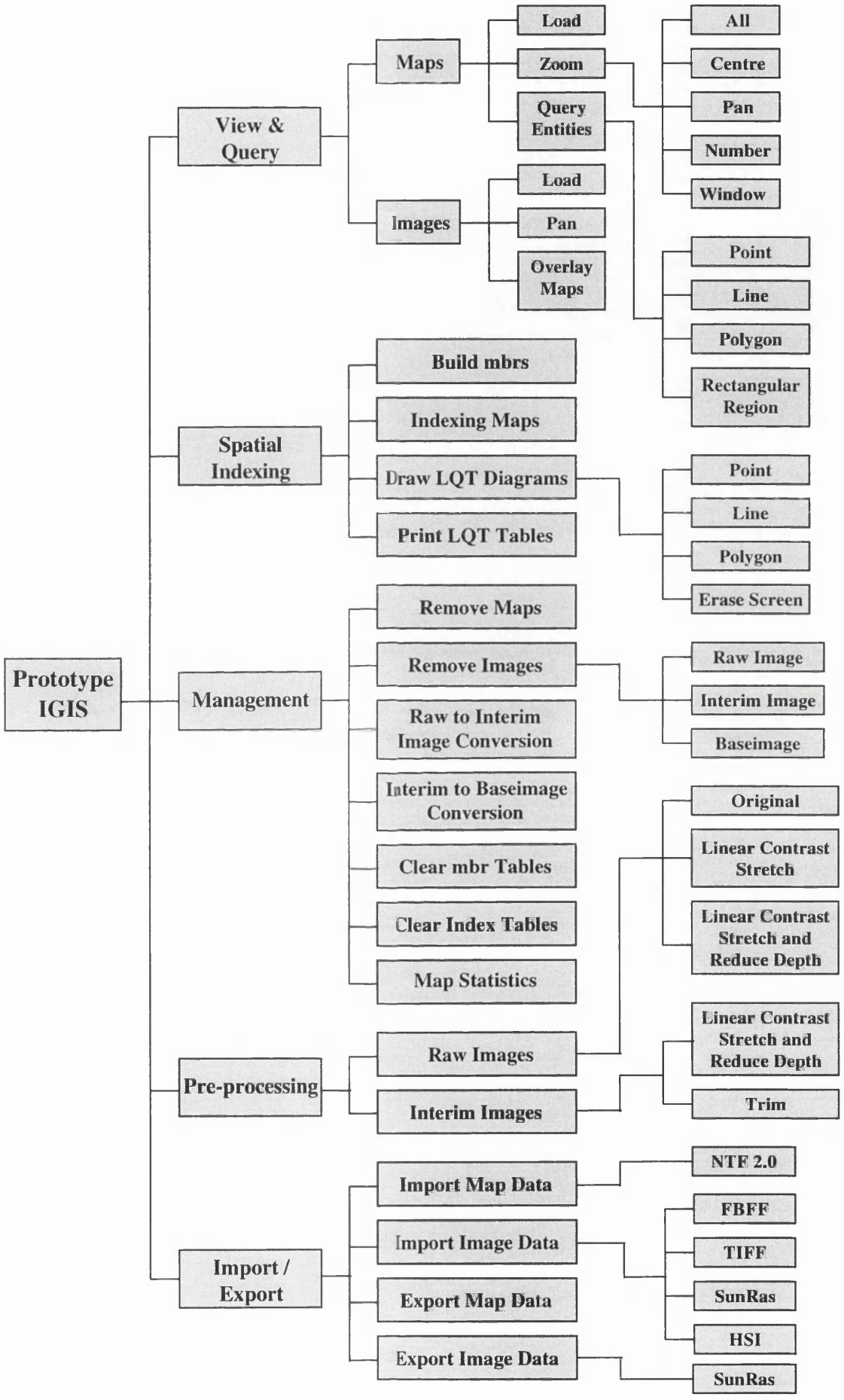


Figure 8.3 The menu hierarchy of the prototype IGIS

In the current implementation, the primary functions of the prototype IGIS can be summarised by each module as follows: -

The **View & Query** module deals with the display of vector maps or raster images or a superimposition of them, and the search for graphical entities.

- it displays maps or images independently
- maps can be superimposed on an image backdrop
- a map can be zoomed in or out
- an image can be panned
- a search can be made for points, lines or polygons by pointing to entities or a rectangular region

The **Spatial Indexing** module deals with the construction of index tables for vector map data.

- mbr tables can be built for line and polygon entities
- index tables can be constructed for point, line and polygon entities.
- linear quadtree index diagrams can be drawn
- linear quadtree index tables can be printed
- a summary of spatial indexing process can be printed

The **Management** module deals with the data management of vector maps and raster images.

- a map or an image can be removed from the persistent store
- a raw image can be converted to an interim image
- an interim image can be converted to a baseimage
- mbr or entity index tables can be erased
- a statistical summary of the elements contained in basemaps can be printed

The **Pre-processing** module handles the pre-processing of raw and interim images.

- a raw or an interim image can be previewed
- a linear contrast stretch can be performed on a raw or an interim image
- the depth of a raw or an interim image can be reduced
- an interim image can be trimmed

The **Import & Export** module is used to import and export vector map data and raster image data.

- vector map data stored in the NTF format can be imported
- raster image data stored in the FBFF, TIFF, SunRas or HSI [HSI, 1993] formats can be imported
- an image can be exported to a file in the SunRas format

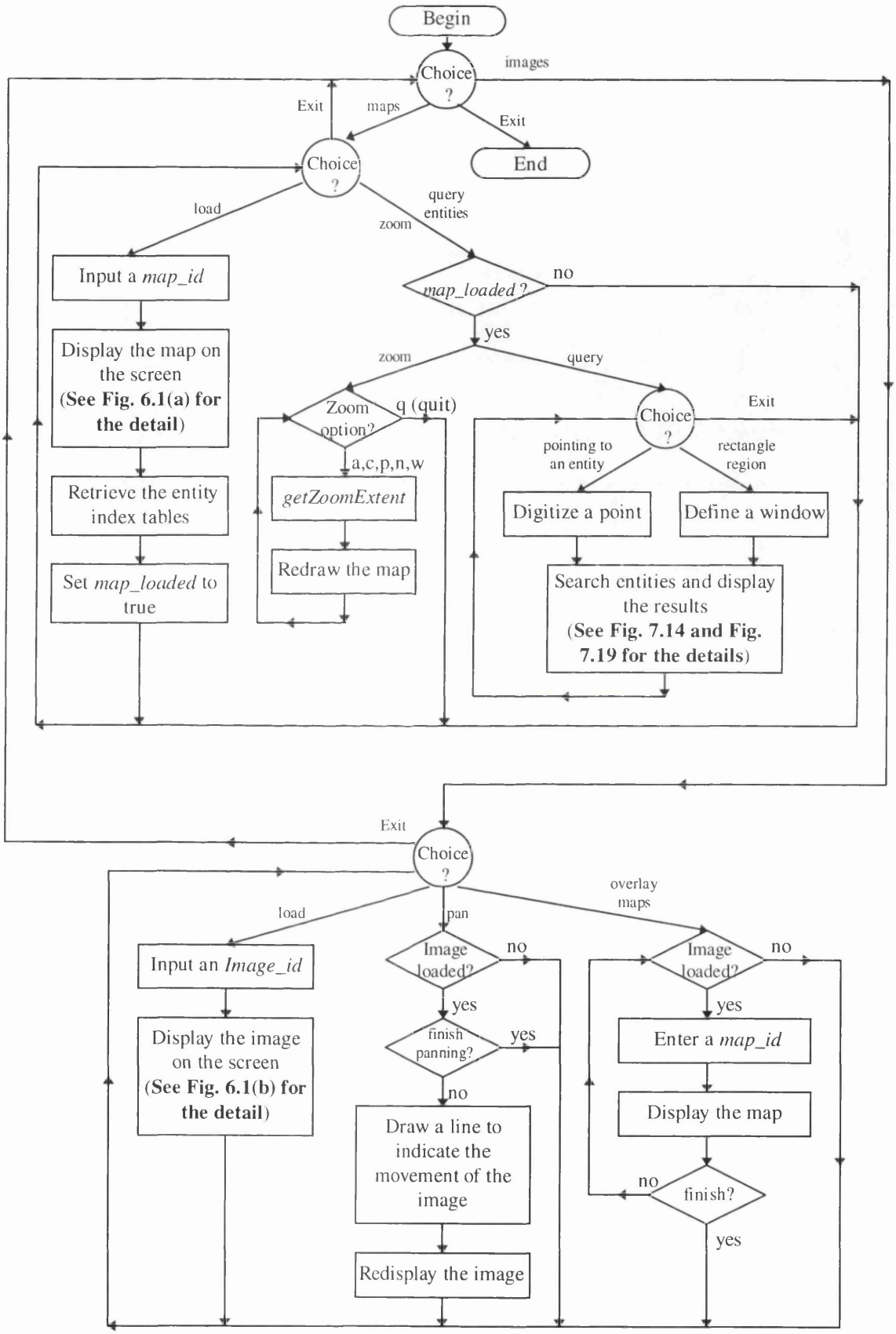


Figure 8.4 The flowchart of the **View & Query** module

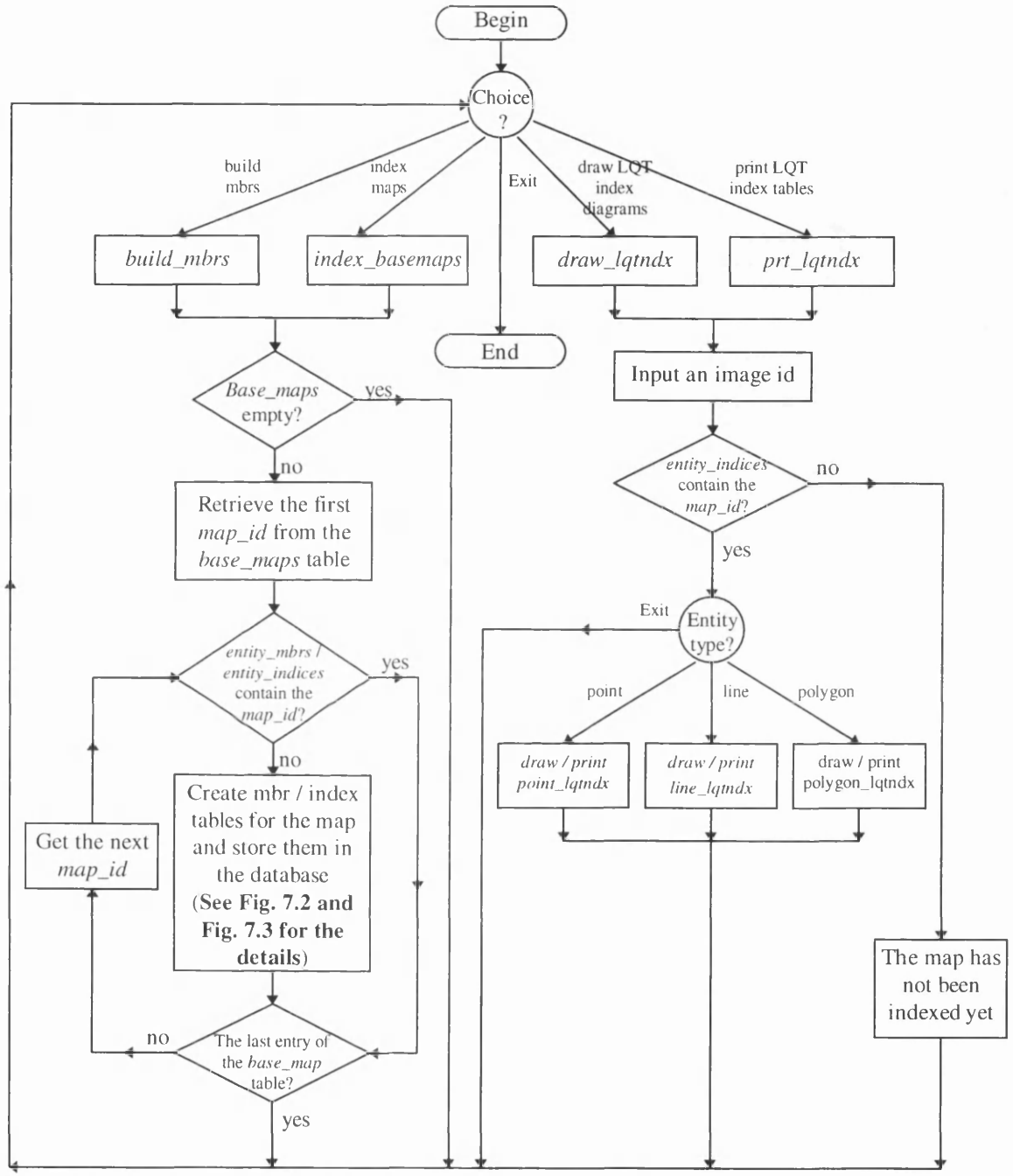


Figure 8.5 The flowchart of the **Spatial Indexing** module

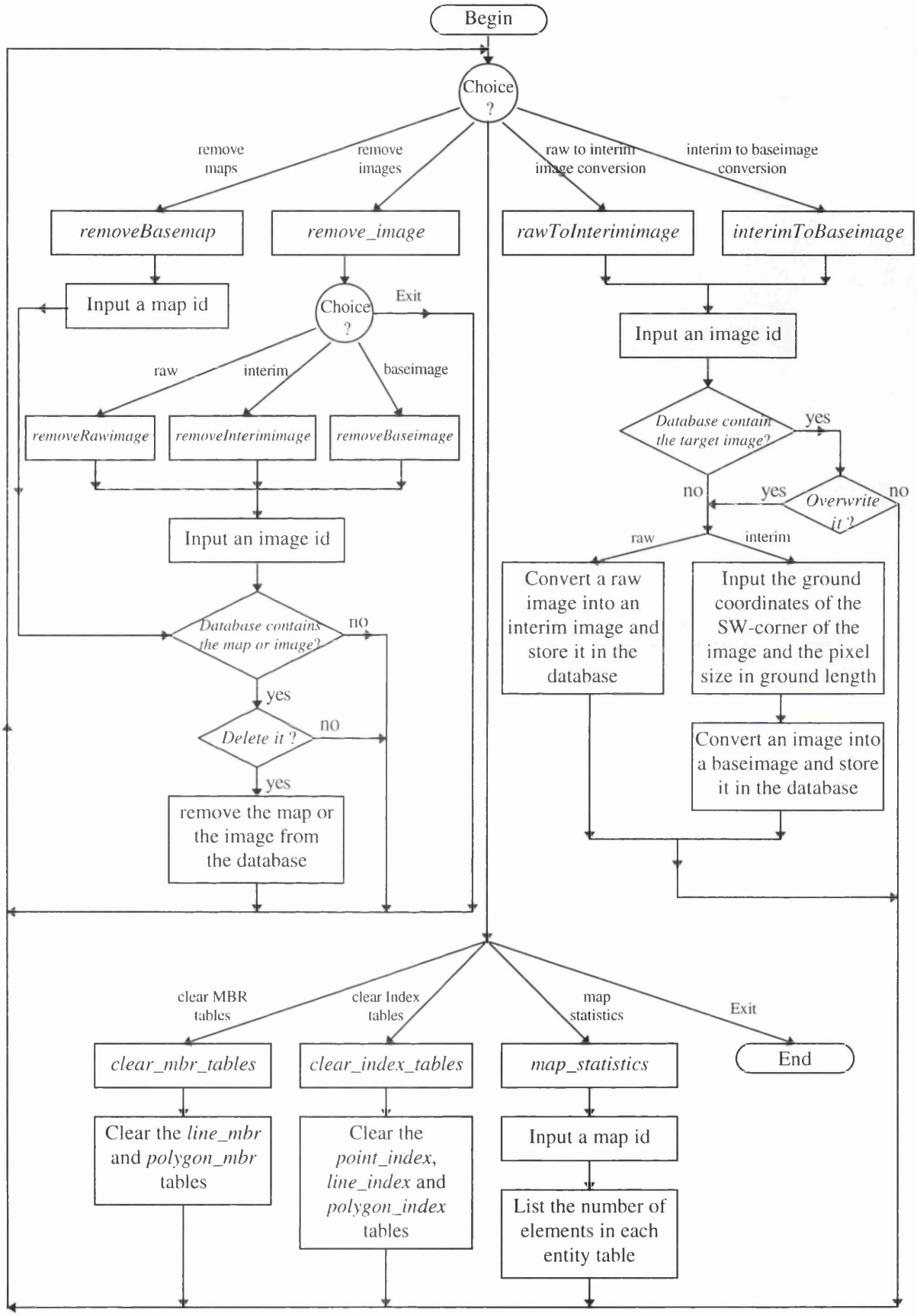


Figure 8.6 The flowchart of data management in the **Management** module

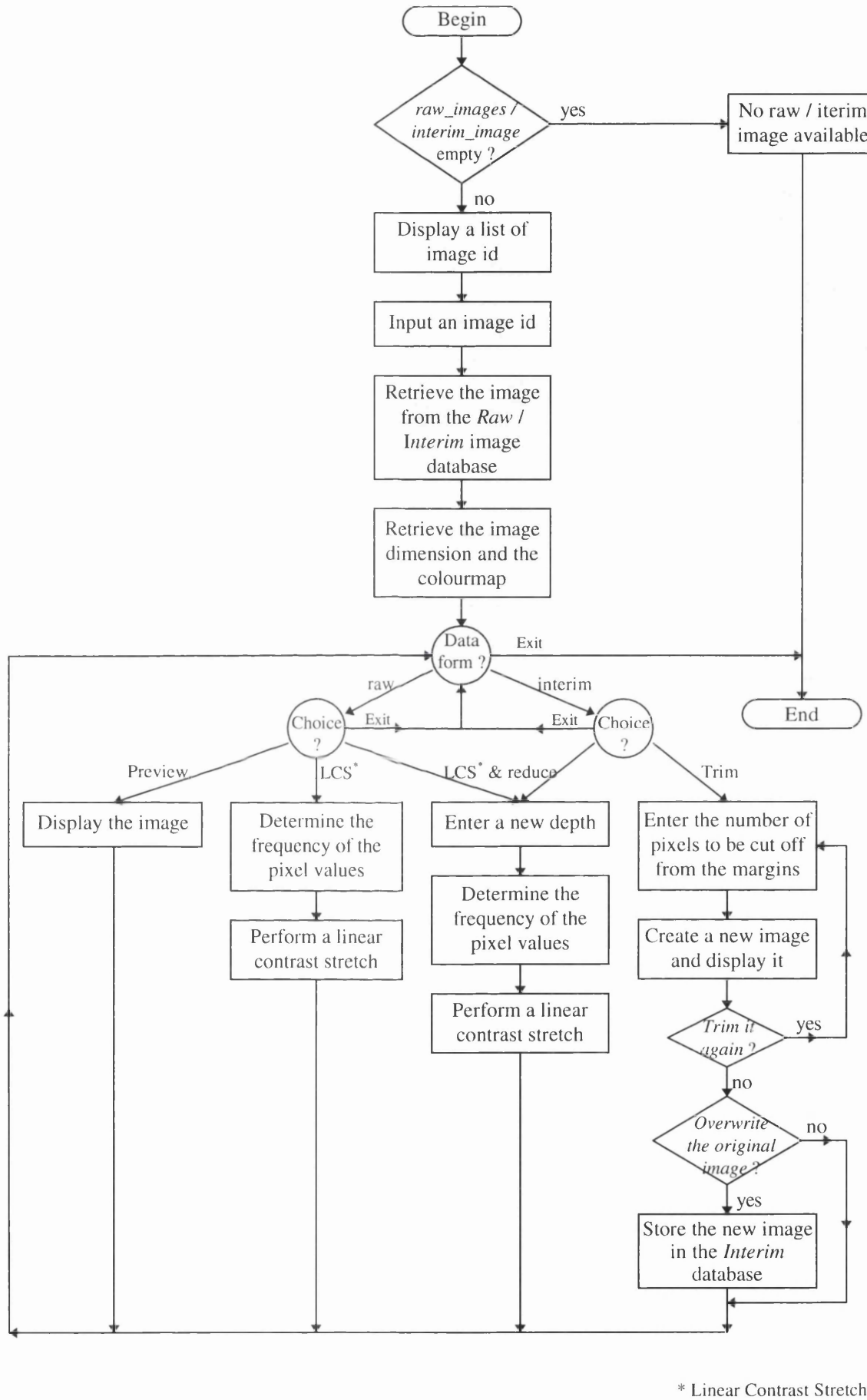
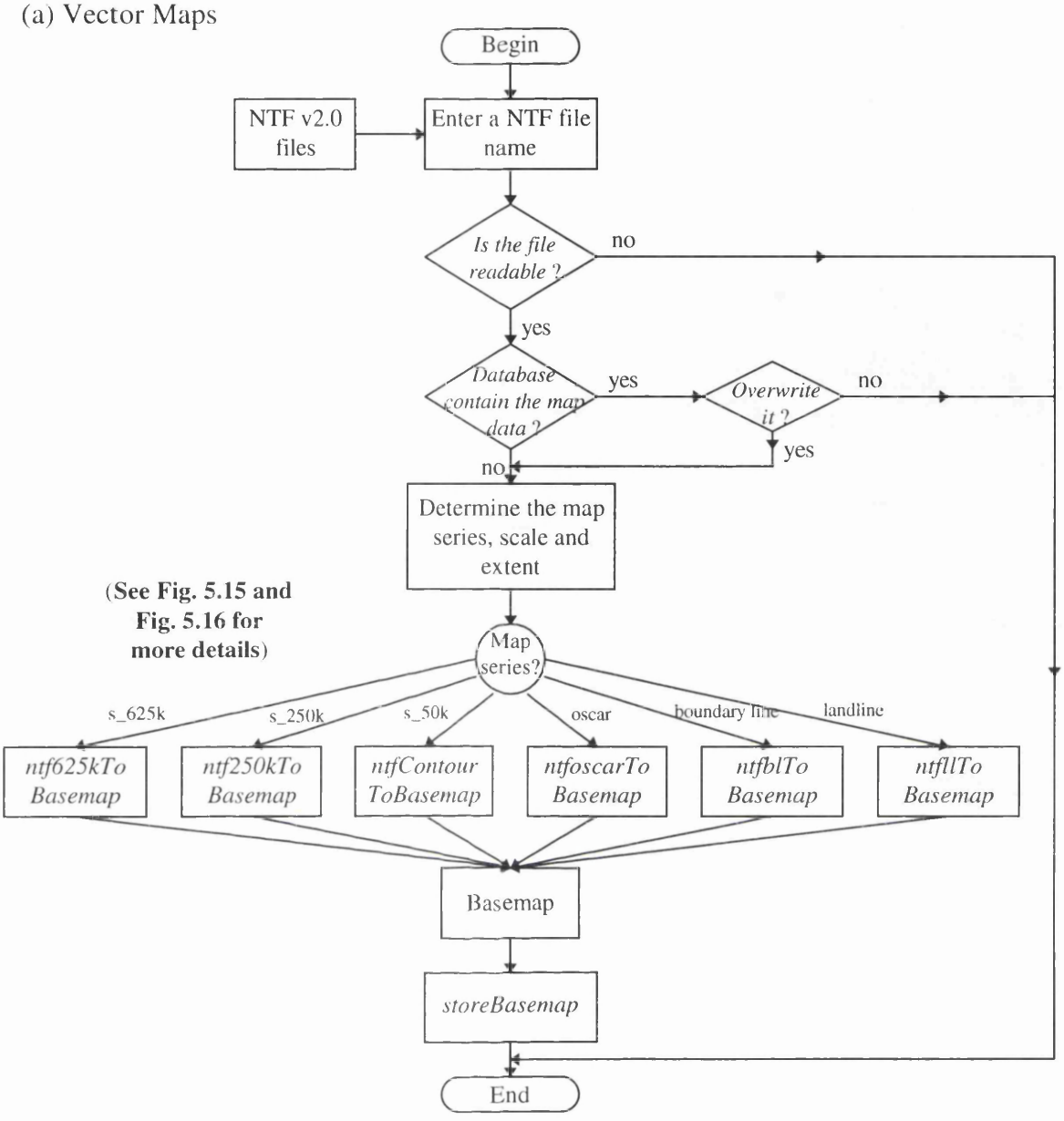


Figure 8.7 The flowchart for processing a raw or interim image in the **Pre-processing** module

(a) Vector Maps



(b) Raster Images

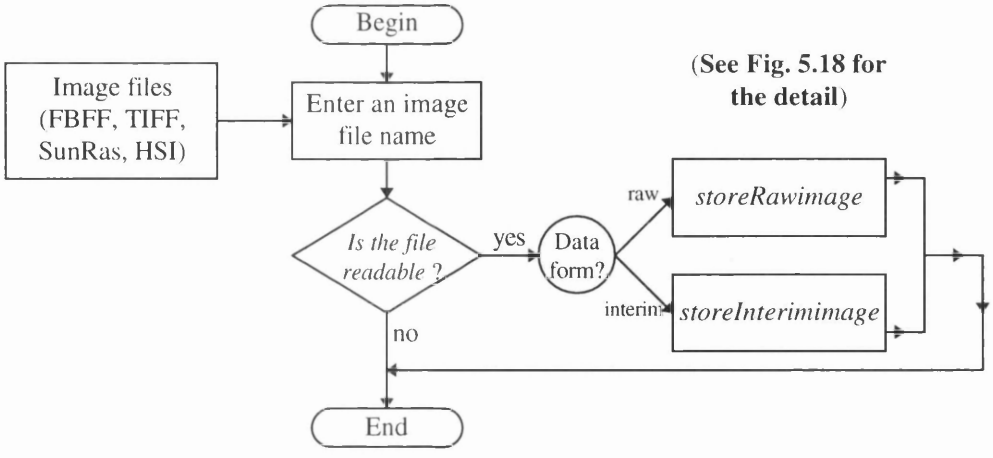


Figure 8.8 The flowcharts for the import of vector maps and raster images in the **Import/Export** module

All of these sub-modules and/or commands existing within a module are organised as a pop-up menu. Fig 8.9 gives a specific example of the pop-up menus employed in the prototype IGIS. The **View & Query** menu which is popped up from the main menu contains the **Maps** and **Images** modules; the **Maps** menu comprises a single command (**Load**) and two modules (**Zoom** and **Query Entities**); while the **Query Entities** module comprises a collection of commands relevant to the search for and querying of different geographical entities. Each menu also includes an **Exit** item which can be used to close the current menu and return to the previous menu in the hierarchy.

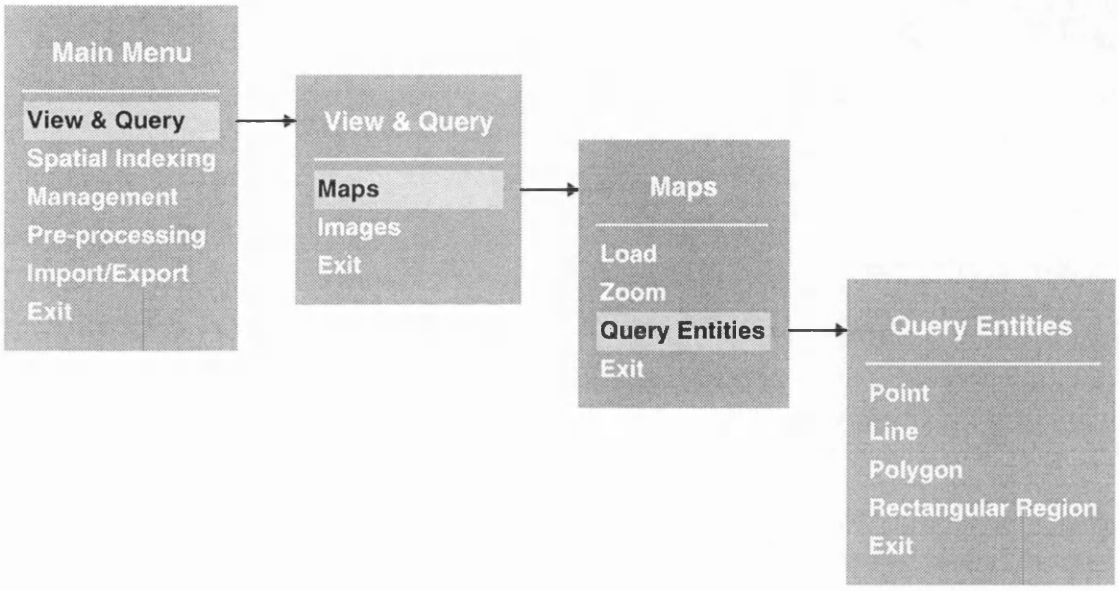


Figure 8.9 An example of pop-up menus used in the prototype IGIS

8.4 The Prototype IGIS Platform

At the time of writing, the Napier88 database programming language only runs under certain very specific versions of the Unix operating system. In practice, this confines the candidate hardware platform to being either a Sun workstation running SunOS or Solaris or a DEC workstation running Ultrix [Kirby *et al.*, 1994]. The system configurations required for both Releases 1 and 2 are different because their persistent environments are unlike in structure - as has already been discussed in Chapter 3. Both Napier88 Releases are installed on Sun workstations in the Department of Computing Science. Since these workstations are connected to the Departmental LAN (CS LAN), so the Napier88 system can be accessed from any computer within the Department. The CS LAN is further linked to the University's high-speed (FDDI/Ethernet) network (campus backbone); thus the machines can also be accessed from other Departments.

The design and the development of the persistent IGIS were conducted in the Department of Geography & Topographic Science. The implementation of the prototype IGIS is based

on Napier88 Release 1. The software development and the tests were mainly carried out using a 486PC equipped with X-window software connected to the Sun machines in Computing Science over the campus network. The overall configuration used for most of the development of the prototype IGIS and the subsequent trials and tests is illustrated in Fig. 8.10.

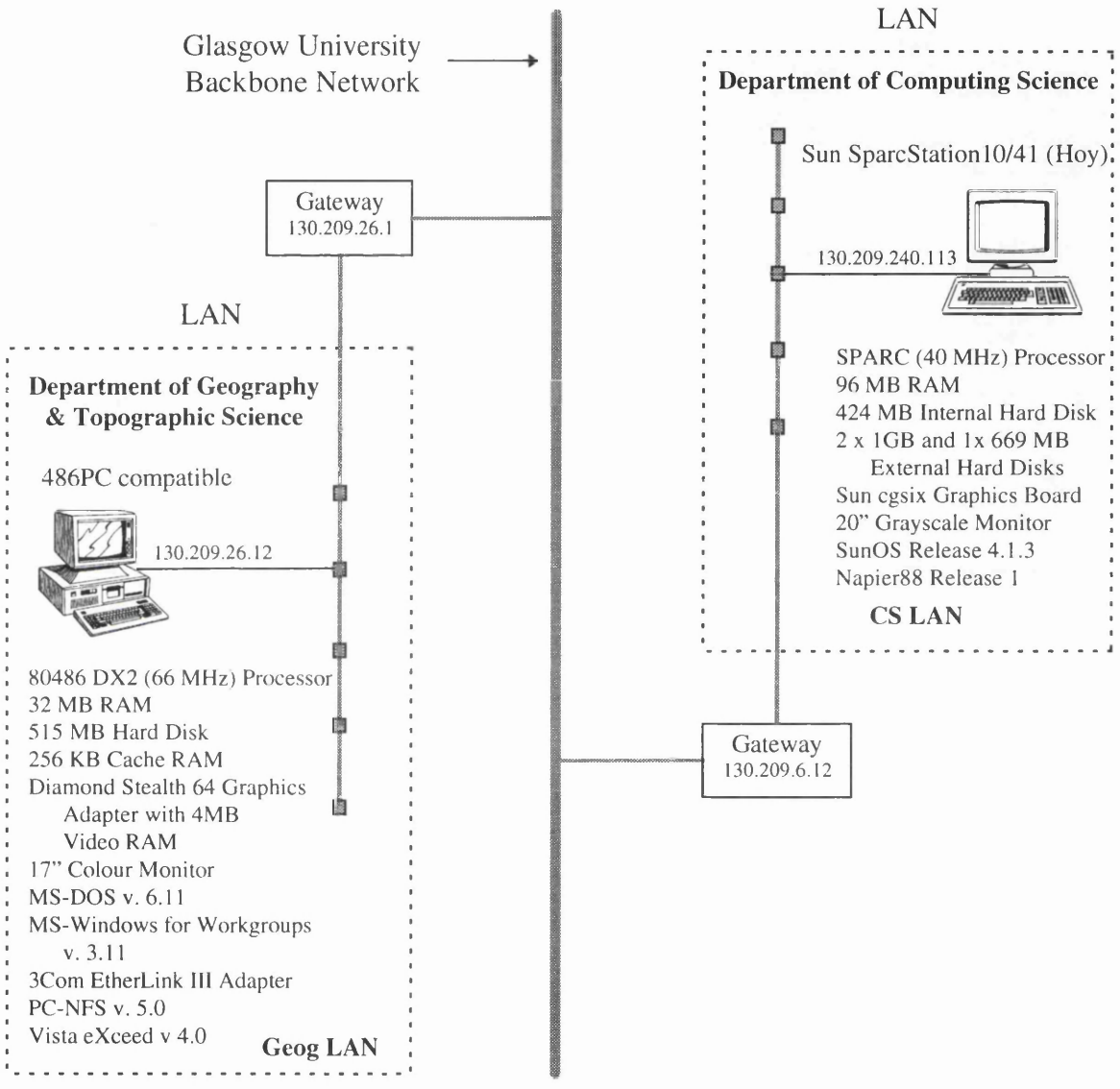


Figure 8.10 The prototype IGIS platform

In fact, a variety of hardware and software system components have been employed at different times during the period of the research. Two workstations - a Sun 4/75 (Albatross) and a SparcStation-10 (Hoy) - in the Department of Computing Science have been used as host computers running Napier88. Also several personal computers or terminals acted as local X-servers in the Department of Geography & Topographic Science. These local hardware devices include two PCs (Viglen 486 DX; Viglen PCI 486 DX2); two X-terminals (Tektronix XP-25; Pericom X-LINE 200) and one workstation (Sun SparcStation 1+). In addition, various types of network and X-window software systems had been used

with the PCs, including two network transport systems (SunSelect PC-NFS v. 4.0 and 5.0 and Trumpet Winsock v. 1.0 Rev A) and two X-window systems (Vista eXceed/W v. 3.2, 3.3 and 4.0 and StarNet MicroX v.2.8.6 and 2.8.8 (Demo version)).

It should be noted that the utilisation of various X-servers resulted from some difficulties in using X-windows with Napier88. Apart from the problem of displaying colours in an X-window on a remote X-server - which has already been mentioned in Section 8.2, the research also came across the twin problems of being "unable to open multiple X-windows" and of "often failing to open an X-window". Once again, these problems only occur when using a remote X-server. In other words, the operations of opening X-windows will appear quite normal when using any suitable workstation located in the Department of Computing Science. These problems caused a lot of inconvenience in the course of developing the prototype IGIS and will be discussed in more detail later in this chapter. Since making good use of X-windows is essential for the provision of the graphical capabilities required for the prototype IGIS, therefore many attempts were made to solve the above problems by using various X-servers. As a result, the performance of various X-servers used with Napier88 has also been tested and will be reported later in this thesis. This result has particular relevance to the future development of an IGIS with a distributed database.

8.5 Test Data

Based on the prototype IGIS described above, a number of trials and tests have been carried out using substantial amounts of data in order to examine the feasibility of applying the idea of the persistent IGIS to a real world situation. The results from these tests have been used to analyse the functionality and the performance of the prototype IGIS. The results were intended to provide some evidence as to whether the persistent-based approach to an IGIS is really a practical proposition or not.

In order to obtain objective results, real geographical data sets were used in the trials and tests. These datasets were acquired from Ordnance Survey, Taywood Data Graphics (now MR-Data Graphics) and NRSC. All of the data covers the same geographical region comprising the Port Talbot area in South Wales, UK. Fig. 8.11 illustrates the geographical location of the test area. The datasets consist of vector map data at different scales and raster image data of different resolutions. The largest coverage of the test data sets is that of the OS 1:625,000 scale map (SS) which is shown in Fig 8.11. The ground coverages of the other vector maps and raster images are illustrated in Fig. 8.12. The shaded areas show the coverages of the raster images, whereas the square blocks with alphanumeric identifiers indicate the coverages of the various vector maps used in the tests.

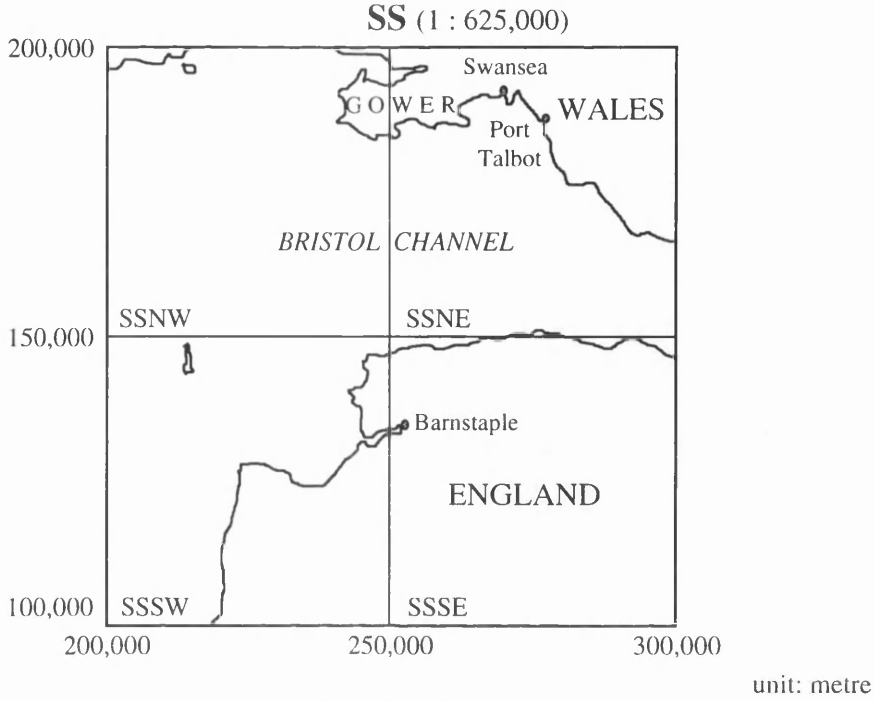


Figure 8.11 Geographical location of the test area

8.5.1 Vector Data Sets

The vector data sets are all in NTF v. 2.0 format [Ordnance Survey, 1993a]. These data sets use three kinds of data models, *i.e.* the spaghetti, the link and node and the polygon-based models, utilised in different OS map series. The principal characteristics of the vector test data are summarised in Table 8.2(a) [Ordnance Survey, 1994].

All the vector map files used in this experiment had already been edited and validated by the suppliers. Hence, they could be regarded as 'clean data' and were imported directly into the *Processed* database, *i.e.* storing them as basemaps. Thus no further processing was required for the vector map data.

8.5.2 Raster Data Sets

The raster data sets use either a TIFF or FBFF format. These data sets only use one data model, *i.e.* the grid cell model. The principal characteristics of the raster test data are summarised in Table 8.2(b). Although the image files SS88SW and PTBAND (1,2,3,4,5,7) - comprising respectively a scanned 1:50,000 scale map and a Landsat TM image - had also been pre-processed by the suppliers to be 'clean data', the raster image data could not be used directly as baseimages without performing a prior manipulation. In particular, the depth (intensity range) of these images needed to be reduced from 8 bits to 4 bits in order to utilise them as backdrops on which vector map data could be overlaid (See Section 6.4).

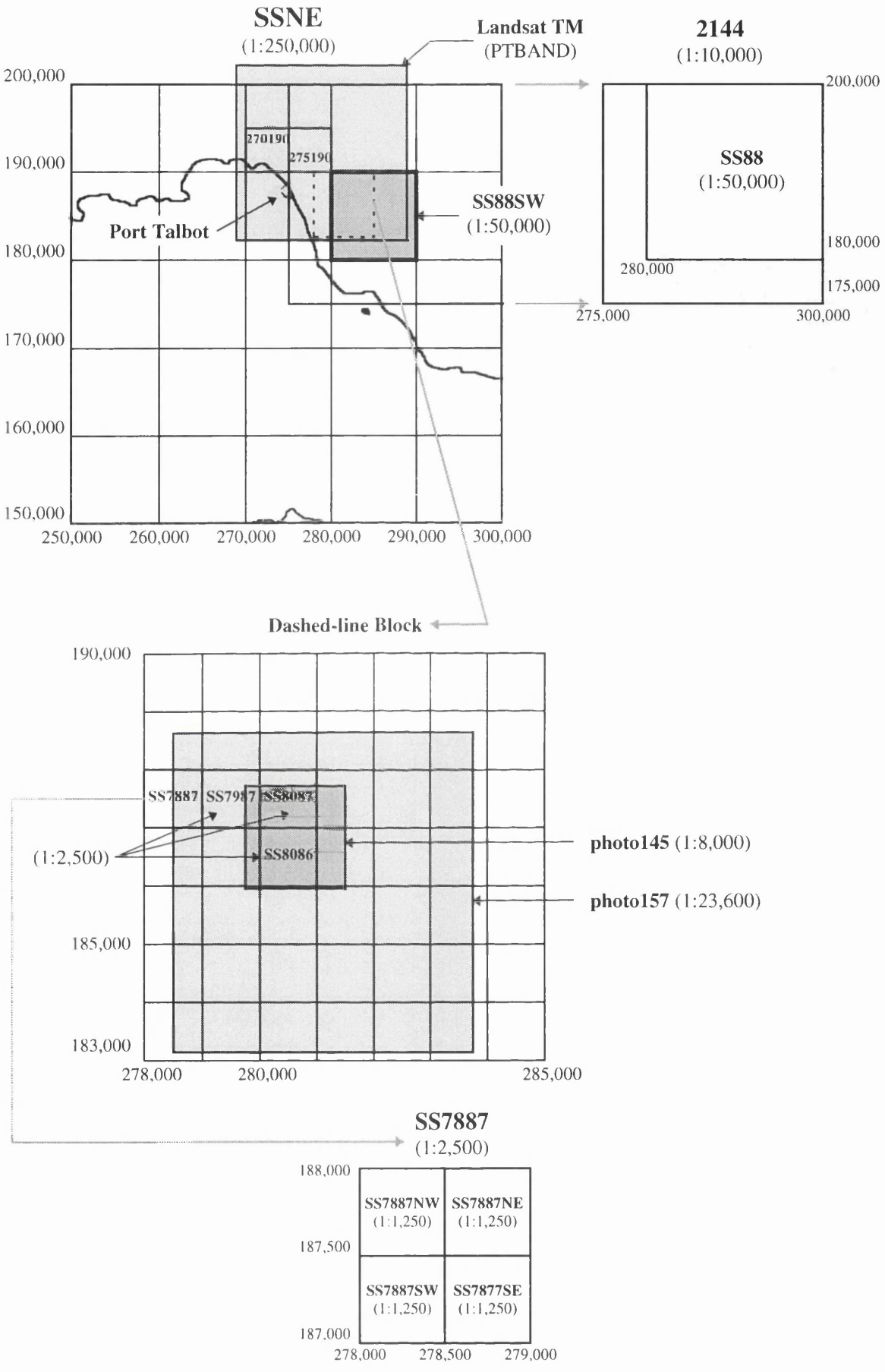


Figure 8.12 The coverage of vector map data and raster image data in the test area

(a) Vector Maps

File Name	Map Series	Map Scale	Coverage (km^2)	Coord. Resol. (m)	Data Model	File Format	File Size (bytes)
SS	BaseData. GB	1:625,000	100 x 100	50	LN	NTF	339,100
SSNE	Strategi	1:250,000	50 x 50	25	LN	NTF	1,185,624
SS88	Land-Form PANORAMA	1:50,000	20 x 20	3 (ht.accur.)	SP	NTF	4,205,279
2144	Boundary-line	1:10,000	25 x 25	0.1	PB	NTF	390,665
270190	OSCAR	1:10,000	5 x 5	1	LN	NTF	210,271
275190	OSCAR	1:10,000	5 x 5	1	LN	NTF	210,483
SS7987	Landline	1:2,500	1 x 1	0.10	SP	NTF	187,920
SS8086	Landline	1:2,500	1 x 1	0.10	SP	NTF	276,866
SS8087	Landline	1:2,500	1 x 1	0.10	SP	NTF	133,400
SS7887NE	Landline	1:1,250	0.5 x 0.5	0.05	SP	NTF	158,275
SS7887NW	Landline	1:1,250	0.5 x 0.5	0.05	SP	NTF	299,225
SS7887SE	Landline	1:1,250	0.5 x 0.5	0.05	SP	NTF	278,217
SS7887SW	Landline	1:1,250	0.5 x 0.5	0.05	SP	NTF	241,419

(b) Raster Images

File Name	Image Series	Image Size (pixels)	Coverage (km^2)	Pixel Size (m)	Data Model	File Format	File Size (bytes)
PTBAND (1, 2, 3, 4, 5, 7)	Landsat TM	800 x 800	20 x 20	25	GC	FBFF	640,000 (x 6)
SS88SW	Colour Raster[1]	2,000 x 2,000	10 x 10	5	GC	TIFF	4,005,880
157	Scanned Airphoto[2]	2,480 x 3,507	5.2 x 5.7	2.1	GC	TIFF	8,706,904
145	Scanned Airphoto[3]	2,480 x 3,507	1.7 x 1.8	0.68	GC	TIFF	8,706,904

Notes: 1 : Scale of Scanned Map = 1:50,000;
2 : Photo Scale = 1 : 23,600;
3 : Photo Scale = 1 : 8,000;
NTF : National Transfer Format;
FBFF : Flat Binary File Format;
TIFF : Tagged Image File Format.
SP : SPaghetti;
LN : Link and Node;
PB : Polygon-Based;
GC : Grid Cell;

Table 8.2 The characteristics of the test data

In addition, a contrast stretch is required for the PTBAND (1,2,3,4,5,7) images to improve or enhance their visual interpretability. The PTBAND (1,2,3,4,5,7) images comprise multispectral data of an extracted Landsat TM (Thematic Mapper) scene. The Landsat TM sensors detect and measure reflected solar energy from the Earth in discrete portions (or wavelength bands) of the electromagnetic spectrum. Each band is useful for a particular application field. For example, Band 1 covers the wavelength range 0.45 to 0.52 μm (visible blue), so the image PTBAND1 is most likely to distinguish geographical features such as those found in coastal water [NRSC, 1992]. Because the sensors mounted on Landsat are capable of detecting the very wide range of radiance levels likely to be found on an Earth wide scale, so it is unlikely that the full dynamic range of any of these sensors will be utilised when sensing a particular area. Hence the pixel intensity values recorded in an image are usually clustered in a narrow section of the full range of available values. As a result, when the original PTBAND images are displayed on a monitor screen, they appear dull, *i.e.* they appear either too dark or over-bright using their original pixel values, due to a lack of contrast. The linear contrast stretching technique is the most commonly-used contrast enhancement method allowing the user to improve the interpretability of the images. This technique involves the "mapping" of the pixel values (PVs) from the observed range PV_{min} to PV_{max} to the full range of the display device, *i.e.* over the range 0 to 255 when using an 8-bit graphic display [Mather, 1987].

On the other hand, image files 145 and 157 were acquired from the scanning of aerial photographs using the Cannon CLC-10 scanner at the Department of Computing Science. These photographs had been scanned at the resolution of 300 dpi, using the A4 size of the scanner and an 8-bit depth. Neither of the resulting image files are 'clean data' because they contain the image deformations resulting from the systematic distortions produced by the scanner itself and from the tilt and relief displacements produced by the aerial photographic geometry. Since each of the image files involves different forms of raster data required in the various stages of data handling, they have been imported into either the *Raw* or the *Interim* database depending on the requirements of their particular operations.

It should be noted that the grid coordinates shown in Fig. 8.11 and Fig. 8.12 are represented in the OSGB National Grid coordinate system, *i.e.* all the map and image data have been referenced to the same ground coordinate system.

8.6 Tests and Results

The purpose of exercising the prototype IGIS is to ensure that it is working properly and to test and identify differences between its expected and its actual behaviour. The expected behaviour of the prototype IGIS is that it will be able to perform effectively the functions

set out in the design discussed in Section 8.3. The testing of the actual behaviour of the system may involve two key aspects: -

1. The functionality test - which examines whether all the functions are working correctly; and
2. The performance test - which measures the efficiency of some critical properties such as the response time required for some complex queries, the storage space needed for databases, and so forth.

The testing of a software package is typically a bottom-up process, usually comprising a unit test, an integration test and finally a system test. Generally speaking, a standard test for a software package is often based on its performance utilising standardised "benchmarks". A benchmark is a program or a set of procedures devised to enable comparisons to be made between two software or hardware systems. In addition, testing should be carried out by a number of real users operating on a prototype long enough to provide feedback for an assessment of its capabilities when used in practical applications. However, in the case of the tests carried out with the prototype IGIS, these cannot be regarded as a rigorous or formal test since the full capabilities required for an IGIS have not yet been implemented. Furthermore, no benchmarks were employed in the tests and the whole test procedure was carried out solely by the author. However, the results derived from this series of tests have shown that a number of unexpected situations may occur during the actual implementation of the persistent IGIS.

Many factors may affect the results of the tests carried out on the prototype IGIS, including the use of specific hardware (hosts and local computers); the configuration of the system software (operating systems, network transport systems and X-window systems); the density of the network traffic; the algorithms employed in the IGIS software; the optimisation of the IGIS software; and so on. In order to minimise the influence of various factors on the evaluation of functionality and performance, all the tests were carried out on the same platform, *i.e.* using the hardware and the software configuration shown in Fig. 8.10, except the test comparing the performance of different X-servers (See Subsection 8.6.6). The testing process has placed an emphasis on the integration of vector and raster data and concerned the following specific aspects: -

1. The construction of vector and raster databases;
2. The spatial indexing of vector map data;
3. The pre-processing of various forms of raster image data;
4. The management of vector and raster databases;
5. The display of vector maps and/or raster images;
6. Comparisons of various X-servers.

Each of these is described in a subsection that follows.

8.6.1 Constructing Databases

This subsection deals with the construction of an integrated geographical database for the test area. The database construction involved the input of the vector map and raster image files listed in Table 8.2 into the persistent store using the import functions provided in the **Import/Export** module. Before the database construction, an initial store was obtained from the "public" directory where an initial store had been prepared for general users. Once this had been obtained, it was followed by the operations of saving the GIS data types, creating the database environments and the building GIS-related Libraries into the persistent store as described above in Section 8.3. All of these operations were organised as a batch job and carried out using a single command.

Having prepared the persistent store, the database construction was carried out by importing each map or image file using the appropriate menu item. Thus all the vector map files used the **NTF 2.0** item available within the **Import Map Data** menu, while the PTBAND files and other raster images (SS88SW, 145 and 157) employed the **FBFF** and **TIFF** items available within the **Import Image Data** menu respectively. Each map file was constructed as a basemap and stored in the *Processed* database. All of the image files except the SS88SW file - which was saved in the *Interim* database - were imported into the *Raw* database. It should be noted that, since the prototype IGIS currently does not have proper image processing capabilities implemented, both of the scanned airphoto images (145 and 157) were simply used to simulate the movement of image data between the different databases. The time required for importing each file into the persistent store was recorded and is shown in Table 8.3.

Vector File Name	SS	SSNE	SS88	2144	270190	275190	SS7987	SS8086	SS8087
Time (m:s)	2:32	7:33	22:18	2:22	1:23	1:27	1:08	1:39	0:51
File Size (MB)	0.339	1.186	4.205	0.391	0.210	0.210	0.188	0.277	0.133

SS7887 NE	SS7887 NW	SS7887 SE	SS7887 SW
0:57	1:33	1:28	1:14
0.158	0.299	0.278	0.241

Raster File Name	PTBAND 1,2,3,4,5,7	SS88SW	157	145
Time (m:s)	0:10	7:55	15:05	15:11
File Size (MB)	0.640	4.006	8.707	8.707

Table 8.3 The times required for the import of each of the various test data sets

During the course of database construction, the input files were intentionally divided into several groups for the import operation, *i.e.* the import of each group of data files was carried out in a separate session. In order to investigate the behaviour of the persistent store, the size of the store before starting each session and the time required to start or launch the prototype IGIS program for that session were recorded. These results are shown in Table 8.4.

Store Size (MB)	16.4	24.8	40.7	45.2	48.5	52.9	57.0	61.2	64.7	68.2
Launch Time (m:s)	0:40	0:46	0:49	0:52	0:54	0:57	0:58	1:00	1:02	1:04
Names of Input Files	SS SSNE	SS88	2144 270190 275190	SS7987 SS8086 SS8087	SS7887NE SS7887NW SS7887SE SS7887SW	PTBAND 1,2,3,4,5,7	SS88SW	157	158	-

Table 8.4 The store size and the launch time relevant to the import of each group of data files

It can be seen quite clearly that the launch time of the prototype IGIS program increases with the size of the persistent store. However, it was very disconcerting to discover that other operations such as the spatial indexing of vector map data and the pre-processing of raster image data also significantly expanded the store size. Therefore a similar summary table (Table 8.9) related to this particular aspect will be provided and analysed later in the next section (8.7).

8.6.2 The Spatial Indexing of Vector Map Data

This subsection is concerned with the construction of the entity index tables associated with the vector map data. This function can be carried out by the two items provided within the **Spatial Indexing** menu. The **Build mbrs** item is used to create mbr tables of the lines and polygons contained in a basemap, whereas the **Indexing Map** option will construct point, line and polygon entity tables for each basemap. In this test, the mbr tables were only built once, but the entity index tables were constructed several times to find out the appropriate threshold for each entity type (which is described in Subsection 8.7.4). In practice, the construction of entity index tables need only be carried out once and can be combined with the building of mbr tables in a single operation.

The indexing operation was tested by adopting different threshold values for the sub-division of the data set into smaller quadrants each time the program was run. The total time required for indexing all of the basemaps was recorded for each operation. Also, the time needed to search for a point, line or polygon was tested after each indexing operation.

These entity search tests were carried out on three particular basemaps, namely the basemaps SSNE, SS88 and 2144, containing respectively the most complex elements of each of the three entity types held in the database. Each of these three basemaps was displayed and the entity search test carried out for points, lines and polygons in turn. The search for a specific entity type was performed 20 times by randomly pointing the cursor to any location on the map display. The time required for each entity search was then recorded. Thus the maximum time needed to search for each entity type was determined. The results of these indexing operations and entity search tests carried out for different threshold values are shown in Table 8.5.

Threshold	Points	1,000	500	400	300	250	200	150	100
	Lines	500	250	200	150	125	100	75	50
	Polygons	200	100	80	60	50	40	30	20
Indexing Time (m:s)		10:20	13:05	15:30	16:07	17:01	20:39	25:50	48:24
Entity Search Time (sec)	Point	2	1	1	1	1	1	1	1
	Line	14	9	7	6	4	2	1	1
	Polygon	1	1	1	1	1	1	1	1

Note: The time used for the construction of the mbr tables = 8 min 3 sec.

Table 8.5 The results of the indexing operations and the entity search tests for different threshold values

The table shows the threshold values used for each entity type gradually decreasing from an initial large value. Thus the number of quadrants resulting from the use of smaller threshold values steadily increased. For the indexing operation, the initial threshold for each entity type was given a large specific value to start with, *i.e.* a different initial value was used for each entity type depending on its nature in spatial indexing. As has been discussed in Chapter 7, the complexity of the spatial indexing likely to encountered with each of the three entity types increases in the order point, polygon and line. Thus originally the threshold values 1,000, 600 and 500 were provided for the numbers of point, polygon and line entities respectively. However, the database contains only one polygon-based basemap (map 2144) in which the maximum number of polygons is 237. Therefore, the initial threshold value for the number of polygon entities was reduced to 200. From the results contained in Table 8.5, it should be noted that the continuous reduction of the threshold values was not really necessary for the indexing of the polygon and point entities (The reason is discussed in Section 8.7.4). Nevertheless, the continual reduction of the threshold values was necessary in order to investigate its effect on the indexing times.

8.6.3 The Pre-processing of Raster Image Data

As already mentioned in Subsection 8.6.1, the raster images PTBAND (1,2,3,4,5,7), 145 and 157 were imported into the *Raw* database. These images require pre-processing operations comprising linear contrast stretching, image depth reduction and trimming. These functions are provided in the **Pre-processing** module.

Each band of the Landsat PTBAND images was retrieved from the *Raw* database to perform the contrast enhancement and the reduction of image depth. These two normally quite separate operations were implemented as a single combined operation in the prototype IGIS. The first step in this operation involved the collection of the brightness values (BVs) across the luminance range, resulting in the construction of a histogram of intensity values. The upper and lower bounding values were chosen automatically through the examination of the histogram for the relatively low pixel counts occurring near the high and low ends of the luminance range. Table 8.6 shows the upper and lower bounding values of each band produced by this operation. Also, the median value indicates that 50% of the total number of pixels have values that are smaller (or bigger) than this particular value. These upper and lower bounding values were then used by a linear transform function to compute a new value for each pixel to a specified image depth (4 bits was used in this test). After this operation, the resultant images were saved in the *Interim* database.

Band / Bounds	1	2	3	4	5	7
low	61	21	16	8	2	0
median	78	31	27	71	67	25
high	111	52	59	124	123	62

Table 8.6 The bounding values of the intensity values in the luminance range for each PTBAND image

On the other hand, the two aerial photographic images 145 and 157 were retrieved from the *Raw* database and transferred into the *Interim* database without performing any processing operation. Then these images were retrieved from the *Interim* database and a trimming operation carried out since this facility is not available at the moment for raw images. Because the aerial photographs had been scanned at the A4 (210 mm x 297 mm) size, which is slightly narrower in one direction but is much longer in the other direction than the corresponding side lengths of a square (229 mm x 229 mm) photograph, therefore the blank portions occurring at both ends of the longer side of the image can be clipped (Fig. 8.13). The trimming function was used to cut off the useless blank areas of each of the images 145 and 157 from the size of 2,480 x 3,507 to 2,480 x 2,720 pixels. After the trimming operation, the resultant images were saved in the *Interim* database. Afterwards, the depths

of these images were also reduced to 4 bits and again the results were placed in the same database.

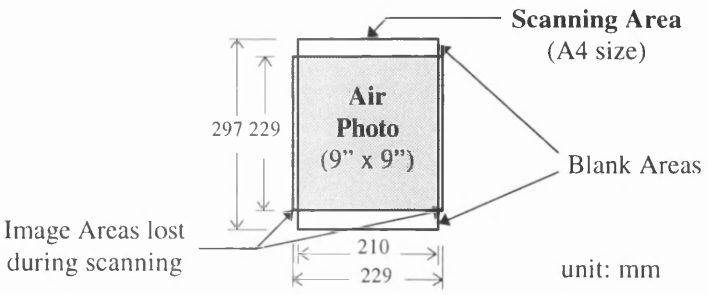


Figure 8.13 Dimensional relationship between an air photo and an A4-size scanner

In the prototype IGIS, the resultant image created by the trimming function is designed to overwrite the original image in the *Interim* database using the fundamental feature of automatic data persistence supported by Napier88. Since the size of the modified image is always smaller than that of its original image, so it should be possible to replace the old image object in the persistent store without increasing the store size. However, the author then came across a problem of "unaccountable store growth". For example, the trimming of image 145 ended up with an increase of the store size by 6.4 MB which is just about the size of the resultant image (2,480 x 2,720 x 8 bits). Similarly, the reduction of the image depth also increased the store size by 3.2 MB (2,480 x 2,720 x 4 bits). The same situation occurred during the pre-processing of the image 157. It is obvious that the original image has not been overwritten and was simply made inaccessible by the Napier88 system. This problem has caused an unnecessary waste of storage space when conducting the operations of updating the databases.

8.6.4 The Management of Vector and Raster Databases

Having performed the necessary pre-processing of the images described in the previous subsection, all of the images stored in the *Interim* database can be used to create baseimages for GIS applications. In this series of tests, all the interim images have been used to create baseimages associated with the grid cell data model. This was carried out by the **Interim to Baseimage Conversion** function contained in the **Management** module. Each interim image was used to construct a baseimage by providing the ground coordinates of the image origin and the pixel size in terms of its corresponding ground dimensions. For example, the scanned map image SS88SW was supplied with the coordinates E = 280,000, N = 180,000 (See Fig. 8.12) and the pixel size 5 x 5m (See Table 8.2) to convert it from the interim form (*interim_image*) to the derived form (*baseimage*). Using this operation, all the baseimages created in the *Processed* database have been referenced to the OSGB National Grid coordinate system. It will be recalled that all the basemaps contained in the *Processed*

database had already been referenced to the ground coordinate system. Thus the same geographical features identified on both baseimages and basemaps can be geometrically registered in an overlay operation if this is required.

Apart from the conversions of images from one form to the other, unused maps or images can be removed from the databases. For example, all the interim images were removed from the *Interim* database using the **Interim Image** item within the **Remove Images** menu after their baseimages had been created. However, the size of the persistent store had not actually been reduced because this operation simply "dropped" (*i.e.* removed the pointers to) the image objects from their bound environment. Napier88 provides two system commands (`nprgc` and `nprcompact`) to deal with the garbage collection and the compression of the persistent store. However, store compression is a very time-consuming process. For example, the author experienced a store compression which reduced the size from 171.3 MB to 86.0 MB and took 1 hour 23 minutes to execute. Furthermore, the time required for launching the prototype IGIS remained almost unchanged in spite of the store size having been significantly reduced. These so-far unexplained features of actual Napier88 operations have been a disappointing characteristic of the language. The occurrence of this problem has of course resulted in difficulties in the management and the maintenance of the integrated geographical database.

8.6.5 The Display of Vector Maps and/or Raster Images

The simple display of a map or image is probably the most commonly-used operation in any GIS. The time required to produce such a display is an important factor in determining the usefulness of a system. Because the GIS database is structured to suit analytic applications, the data may not be optimised for display purposes. Thus it is quite usual to find that the time needed for the display of a vector map on a GIS is slower than on a CAD system. Furthermore, the use of certain specific algorithms and the optimisation of software may also improve the display time significantly. Nevertheless, in order to give an appraisal of the current display capability provided by the prototype IGIS, the times required for the display of the test data have been recorded. Table 8.7 shows the display time required for each basemap or each baseimage and the number of elements being displayed using the hardware configuration shown in Fig. 8.10.

It should be noted that only the essential entities (points and lines) were displayed for each basemap, other optional entities (text, polygon identifier, symbols, *etc.*) have not been used in the comparison. Since a line string may comprise an arbitrary number of straight lines (line segments), so the number of elements for both line strings and line segments is shown in Table 8.7. Another important point is that the size of the X-window used for the display test was set to 800 x 600 pixels.

(a) Vector Map Data

Map Name	SS	SSNE	SS88	2144	270190	275190
Number of Points	1,208	3,681	3	0	0	0
Number of Line Strings (Line Segments)	1,062 (7,402)	5,994 (24,482)	2,051 (182,028)	237 (22,428)	865 (3,871)	810 (4,550)
Time (m:s)	0:34	2:36	7:44	0:35	0:13	0:19

SS7987	SS8086	SS8087	SS7887NE	SS7887NW	SS7887SE	SS7887SW
50	117	27	136	758	492	526
375 (10,468)	495 (15,686)	169 (7,967)	352 (8,299)	1,534 (9,136)	1,239 (10,785)	1,213 (7,917)
0:24	0:34	0:18	0:19	0:26	0:33	0:23

(b) Raster Image Data

Image Name	PTBAND 1,2,3,4,5,7	SS88SW	157	145
Width (Pixels)	800	2,000	2,480	2,480
Height (Pixels)	800	2,000	2,720	2,720
Time (m:s)	0:06	0:26	0:47	0:46

N.B. All images are 4 bits.

Table 8.7 The times required for the display of the test basemap and baseimage data

Apart from the individual display of a basemap or baseimage, the overlay capability of the prototype of IGIS has also been examined. This test was carried out by first displaying each baseimage and then superimposing the related basemaps on it using the **Overlay Maps** function available in the **Images** menu of the **View & Query** module. All the possibilities of superimposition between basemaps and baseimages (See Fig. 8.12) have been performed. The results show that the raster images PTBAND (1,2,3,4,5,7) and SS88SW can be correctly overlaid with the corresponding basemaps. Also, the colours of the baseimages and the basemaps can be adjusted independently of one another. Despite the facts that neither of the images 145 and 157 had been rectified and that their image origin and pixel size had only been defined roughly, the simulation of overlying large scale basemaps such as SS8086 and SS8087 on these images also gave satisfactory results.

8.6.6 Comparisons of Various X-servers

As mentioned in the previous subsection, the display capability is one of the major concerns in the evaluation of the prototype IGIS. In particular, the display of a complex basemap

such as SSNE or SS88 requires a considerable amount of time. In order to find out how an X-server may affect the display speed, various X-servers have been used to carry out a performance test.

The test was split into two parts. The first part involved the use of two X-terminals, two PCs associated with different X-window systems and one workstation located in the Department of Geography & Topographic Science, whereas the second part encompassed the use of several workstations available in the Department of Computing Science. Two basemaps, SSNE and SS88, were selected for the performance test of the various X-servers. The time required for the display of all lines contained in both basemaps on each X-server was recorded. In order to minimise the effect caused by the network traffic, the experiment was carried out at weekends when traffic was very low. Also, the display of a basemap on each X-server was tested several times so as to determine an average display time. The results of these tests are summarised in Table 8.8.

Remote Site

X-server at the Department of Geography & Topographic Science	The Display Time of Map SSNE (m:s)	The Display Time of Map SS88 (m:s)	Window Manager
Tektronix XP-25	2:16	11:56	twm*
Viglen 486 DX2/66 & MicroX v.2.8.8	2:08	10:53	MS-Windows
Viglen 486 DX/33 & eXceed/W v.3.3	2:06	10:18	MS-Windows
Viglen 486 DX2/66 & eXceed/W v.4.0	1:28	7:44	MS-Windows
Pericom X-Line 200	1:08	3:41	twm*
Sun SPARCstation 1+	1:08	3:36	olwm

Local Site

X-server at the Department of Computing Science	The Display Time of Map SSNE (m:s)	The Display Time of Map SS88 (m:s)	Window Manager
Sun SPARCstation 1+ (Weddell)	1:06	3:21	twm
Sun SPARCstation IPC (Barren)	1:04	3:09	twm
Sun SPARCstation 10 (Hoy) **Host Computer**	1:09	3:21	twm

* running on the host computer (Hoy).

Table 8.8 The performance of various X-servers used for the display of basemaps

During these tests, the selection of a window manager had also been considered. Whenever it was possible, the window manager and the X-server running on the same local computer were used for the test. In principle, this arrangement can reduce the network traffic and improve the response time of X-clients. For example, on a PC-based X-server, if the choice of a window manger was either MS-Windows running on the PC itself or any available window manager running on the host, then MS-Windows was used. In fact, only the two X-terminals (Tektronix XP-25 and Pericom X-LINE 200) used in this test had to depend on the window manager (twm was used) running on the host computer. It should be noted that an X-server running on the host computer itself was also tested. In this particular situation, all the software systems including the X-server, the window manager and the X-client were mounted and executed in the host computer (Hoy). In addition, the persistent store was also existing in the same computer. In other words, in this particular case, the prototype IGIS was run in a stand-alone mode without involving the use of the network for data transfer. This particular result is also shown in Table 8.8 and will be further discussed in Subsection 8.7.5.

8.7 Analyses and Discussions

In this section, the results of testing the prototype IGIS are analysed. Based on these analyses, some findings can be derived. These are discussed below.

8.7.1 The Functionality of the Prototype IGIS

On the whole, the capabilities designed and implemented in the prototype IGIS do function properly. In particular, the tests have shown that the functions provided by three key facilities - the construction of an integrated geographical database; the superimposition of vector maps and raster images; and the spatial indexing of vector map data - can work very well. However, several functions such as the pre-processing of image data and the management of databases require further enhancement. In addition, many other essential functions which have already been described in Section 4.5 - the functional design of the persistent IGIS - need to be included.

In general, the display speeds (See Tables 8.7 and 8.8) of a map or an image are quite appropriate. In fact, already they are not dissimilar to those encountered in a commercially sold GIS. However, the indexing operations required about 20 minutes for all 13 basemaps to achieve a reasonable search time (See Table 8.5). This seems quite moderate but undoubtedly can be improved. The times required for the construction of various databases (See Table 8.3) from the existing digital files also seems a bit slow, particularly the import of image files in TIFF format. In terms of overall system performance, the prototype IGIS program still needs to be optimised. Also a thorough and rigorous investigation needs to be

mounted to ensure that efficient algorithms have been implemented for each procedure or function provided by the IGIS.

8.7.2 The Launch Time vs. The Store Size

It is recognised that, in any software system, the larger the file size that is used, the longer the processing time that will be needed for any particular operation. This is the case with Napier88 system. However, the persistent store which aggregates all data objects into a single Unix file behaves in a quite different fashion to that of normal file systems. In order to access any object held in the persistent store, a Napier88 program which instructs the system how to retrieve and manipulate this object, needs to be executed. This running program checks against the persistent store and makes the persistent environment related to this particular object available to the user. As a result, the size of the persistent store has a noticeable effect on the time required for launching a Napier88 program.

During the period of this research, the author has suffered from the long waits associated with the launch of test programs. It is quite usual for a wait of 1 to 2 minutes to occur before a test program became operational. The situation became even worse when the tests had to be carried out on a large-size persistent store. For example, a 3-minute wait will be unavoidable if the store size is about 250 MB. Arising from this experience, a test has been carried out to investigate how the store size affects the launch time.

Table 8.4 has shown that the launch time steadily increased when the store size grew from 16.4 to 68.2 MB. In order to find out more about the relationship between the launch time and the store size, the launch time was regularly recorded for several major operations until the store size was about 190 MB in size. These results are summarised in Table 8.9.

Store Size (MB)	92.3	107.4	112.5	114.0	122.7	131.4	140.0
Launch Time (m:s)	1:25	1:38	1:43	1:44	1:48	1:59	2:04
Operations	Spatial Indexing	Pre-processing	Pre-processing	Pre-processing	Trim image 145	Trim image 145	Trim image 145

148.5	157.0	165.4	173.6	181.7	189.7
2:10	2:15	2:21	2:26	2:30	2:34
Trim image 145	Trim image 145	Trim image 145	Trim image 145	Trim image 145	Trim image 145

Table 8.9 The store size and the launch time relevant to the operations of the prototype IGIS

Using Tables 8.4 and 8.9, the relationship between the launch time and the store size may be derived by plotting a scatter diagram using the store size as the X-axis and the launch time as the Y-axis. Fig. 8.14 shows that a characteristic curve (the solid line with cross marks) can be obtained from the measured data. From this, it is very obvious that the launch time increases quite steadily with the store size. This characteristic curve can be used to determine the response time required for a specific store size when running the prototype IGIS on the particular platform illustrated in Fig. 8.10. According to the above analysis, a summary description of launch time vs. store size can be generated and is shown in Table 8.10. It can be seen that, at its present stage of development, a user would need to exercise patience when running the prototype IGIS, if it was really to be put into practical use.

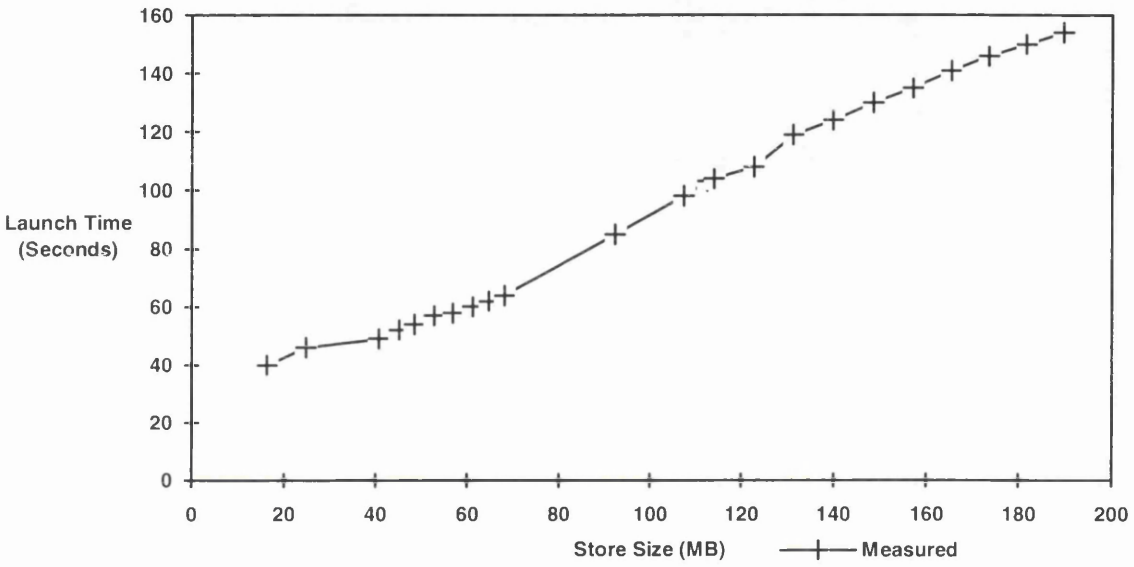


Figure 8.14 The characteristic curve of launch time vs. store size when running the prototype IGIS with the particular platform illustrated in Fig. 8.10.

Store Size (MB)	< 60	60 ~ 130	130 ~ 180	> 180
Launch Time (min)	< 1	1 ~ 2	2 ~ 2.5	> 2.5

Table 8.10 A summary description of launch time vs. store size

It should be noted that the several operations used for trimming the image 145 in Table 8.9 were not actually necessary in terms of the examination of the prototype IGIS functionality. However, these operations again demonstrated the problem of "unaccountable store growth" described earlier in Subsection 8.6.3. Each operation retrieved the image 145 from the *Interim* database, then trimmed 20 pixels off from the top and bottom margins of the image, and finally saved the resultant image and transferred it back to the *Interim* database. It can be seen from Table 8.9 that the store size had increased by between 8.0 to 8.7 MB after each operation. The cause of the unaccountable store growth is probably due to the

fact that, in some situations, Napier88 cannot properly perform the "in situ updates" function on objects held in the persistent store.

This problem also occurred at an earlier stage in this research when the author attempted to develop an updating capability for the prototype IGIS. The problem had then appeared and been reported to and identified by the Napier88's system developers. As a palliative, it was suggested that the `nprgc` command should be regularly executed against the persistent store. In general, the regular use of the `nprgc` command can eliminate the unexpected growth of size of the persistent store. This approach may be very useful for handling situations where the store size is quite small (*e.g.* < 50 MB). However, it would be very inefficient to use this approach to deal with a persistent store of substantial size because the time required for executing this command is rather long and the store size may also increase significantly after this operation. It should be noted that Release 2 has also been tested concerning this feature and has also exhibited the same problem. This problem causes a great difficulty during the operations of the persistent IGIS with a large-size geographical database. As a result, on several occasions during the research period, the author has been forced to discard a large persistent store (~ 250 MB) and has had to restart from an initial store simply because of the unbearable situation resulting from the slow launch time and the continual store growth.

8.7.3 The Optimal Thresholds for Indexing Vector Map Data

A discussion has already been conducted in Chapter 7 pointing out that a number of factors will affect the choice of an optimal threshold. Considering the fact that the prototype IGIS is based on a particular platform, all the IGIS facilities, including the required computing power, the data models, the data structures and the spatial indexing method, have been fixed. Therefore, the optimal threshold for each entity type can be determined.

According to the response time criteria described in Brown [1988], the maximum response time recommended for a simple query such as searching for an entity by pointing is 2 seconds, while that needed for a complex query such as a region or buffer query is recommended as 10 seconds. In this particular test, the simple query method was used to test the response time for different threshold values. From the test results shown in Table 8.5, the optimal thresholds for point and line entities can be determined. Their values are 1,000 and 100 respectively. However, the optimal threshold for the polygon entity still cannot be resolved because the basemap database used in the tests does not contain sufficient polygons. The best estimate is that its optimal threshold is larger than 200 individual items.

Having set up the optimal threshold values in the prototype IGIS, the **Rectangular Region** function in the **Query Entities** menu was used to examine the response time required to execute several complex queries on any basemap. The results can be regarded as very satisfactory since the maximum response time was also 2 seconds.

In Table 8.5, the indexing time represents the total time that was used to index all 13 basemaps for the three entity types utilising a set of threshold values. With the optimal threshold values (1,000, 100, 200) determined above, the indexing time was further recorded as 19 minutes 42 seconds. Comparing this with the indexing time (20 minutes 39 seconds) used for the previous set of threshold values (200, 100, 40), it is obvious that the indexing of the lines contained in the database had consumed the largest part of the time. In order to understand how much time is really needed for indexing each entity type, the indexing time required to index each basemap for each entity type have been recorded for the set of smallest threshold values (100, 50, 20). The results are shown in Table 8.11.

Map Name		SS	SSNE	SS88	2144	270190	275190	SS7987
Indexing Time (m:s)	**Points**	0:7	0:27	0:1	0:0	0:0	0:0	0:2
	Lines	0:37	21:57	18:20	0:2	0:34	0:30	0:10
	Polygons	0:0	0:0	0:0	0:8	0:0	0:0	0:0

SS8086	SS8087	SS7887NE	SS7887NW	SS7887SE	SS7887SW	All Maps
0:3	0:2	0:2	0:4	0:3	0:3	0:54
0:8	0:9	0:19	2:51	0:56	0:49	47:22
0:0	0:0	0:0	0:0	0:0	0:0	0:8

Table 8.11 The indexing times required for the threshold values (100, 50, 20)

The times required for the indexing of points, lines and polygons take 1.9 % (57s), 97.8 % (47m 22s) and 0.3 % (8s) of the total time respectively. Obviously, the indexing of points and polygons is quite insignificant in terms of the whole spatial indexing operation. This is because the number of lines is almost always far larger than the numbers of points and polygons (See Table 8.7) and the indexing of lines is inherently of greater complexity than that required with points and polygons. Therefore, an improvement of the efficiency with which the indexing of lines can be carried out is a complete necessity. Also, more polygon data should be acquired to carry out a further test on the indexing of polygons to ensure that its high performance has indeed been achieved, even when applied to larger polygon data sets.

8.7.4 The Use of X-windows

When running the prototype IGIS, one of the essential steps within the program is to open an X-window. With an X-window opened, all the GUI facilities and the graphical functions can then be made available to the user. During the various tests, the Napier88 system often failed to open an X-window on the remote X-server. According to author's statistics, the probability of this failure case occurring is about 30 % when running the program in the Department of Geography & Topographic Science. However, the probability of this problem occurring on the machines in the Department of Computing Science is almost negligible - although it does happen occasionally. Since network communication is required when using a remote X-server, the author has tried several ways to find out whether the cause of these failures arises from the network connection. These trials include the setting of the maximum time slice on the X-server to ensure that one X-client can run uninterrupted; the running of the program at night or on weekends to reduce the network traffic and to ensure that the host computer is itself operating in the circumstance of a minimal load; and the use of various X-servers. In spite of these various arrangements and trials, the problem of failing to respond to an X-window remains the same. It is worth mentioning that this problem only occurs with Napier88 X-clients and does not occur when using other X-clients. Therefore, the possibility that the problem is caused by the network seems very unlikely. After all these efforts, the author has come to the view that the problem may result from the inappropriate implementation of the X-window features utilised in the Napier88 system. Once again. the problem has been reported to the system developers but it has still not been resolved at the time of writing.

The second major problem relevant to the use of X-windows is that the Napier88 system cannot open multiple X-windows on a remote X-server. If the author had been able to use the WIN window management system (the difficulty of doing so has been described in Section 8.2), the GUI facility and the hyper-programming environment supported by Napier88 could and would have been used in the IGIS development. Once the hyper-programming system is running, the display of a map or an image can be carried out using a Napier WIN window. Also, it allows the user to compose and execute Napier88 programs and to examine their results in an integrated programming environment [Kirby *et al.*, 1994]. Since multiple windows can be created within the single X-window that is running the hyper-programming system, therefore it is not necessary to open another X-window when WIN is used.

Regarding the current implementation of the prototype IGIS, all the operations are also based on an X-window but without the desirable capability of multiple windows. In terms of practical GIS applications, it would be very useful (- some would say essential -) to have a system that allows several windows to be opened concurrently so as to view or compare

geographical features in different windows, *e.g.* displaying a map in one X-window and a part of it at a larger (zoomed) scale in another X-window. In fact, the Napier88 system has the capability to open multiple X-windows running a conventional window manager. With this facility, the multiple X-window feature may be developed for the persistent IGIS. However, it was not possible to open multiple X-windows on any of the several X-servers which have been employed during the tests carried out in the Department of Geography & Topographic Science. Peculiarly, multiple X-windows can be opened on the workstations at the Department of Computing Science. However, once again there is a strong possibility of a failure occurring when an attempt is made to open multiple X-windows. This problem is of course closely related to the first problem described above.

It is also worth mentioning that, with a fixed amount of video memory, the maximum size of an X-window that can be opened by a Napier88 X-client is quite substantially smaller than that which can be opened by a normal X-client. For example, the author used a Viglen PC 486 DX/33, which only has 1 MB video RAM, with eXceed/W v. 3.2 as the X-server at the early research stage. A normal X-client can open an X-window with the size of 1024 x 768 x 8 on this X-server, but the maximum X-window size that can be opened by a Napier88 X-client is 830 x 630 x 8. This situation did cause confusion when attempts were made to open a larger-sized X-window on this particular X-server. Through the use of various X-servers equipped with different sizes of video RAM, the fact that a Napier88 X-client demands more video memory than a normal X-client was discovered. As a result, the PC X-server configured for use as the prototype IGIS platform (See Fig. 8.10) has been equipped with 4 MB video RAM to ensure that it can display maps and images with an X-window of the size of 1024 x 768 x 8. Thus it provides the potential that the display of maps or images in a larger-sized X-window may be included in the future when this is required in the design of the persistent IGIS.

8.7.5 The Performance of Various X-servers

Although the use of an X-server to run Napier88 programs remotely in a remote site produced and brought to light several problems, it should be said that it provided the author based in the Topographic Science Department with a very convenient way to carry out the IGIS development. The use of various X-servers to investigate the problem of opening X-windows also ended up with a by-product - in the shape of figures giving the performance of Napier88 vector graphics on various X-servers. Since real-world data has been used in the tests, so the results are quite meaningful and can be used as a guideline for choosing an X-server.

Table 8.8 showed that the use of a different X-server has a significant effect on the display time experienced at the remote site, in particular that needed for the display of the contour

map SS88. All the Sun workstations, whether at the local or remote site, give similar results. It is quite interesting to note that running the prototype IGIS in the standalone mode does not produce the best result. This is probably caused by having the X-server and the X-client running on the same computer.

The two X-terminals - Pericom X-200 and Tektronix XP-25 - gave the best and the worst results respectively at the remote site. The main reason for this is that their graphics processors are quite different. The Pericom X-200 makes use of an AMD 29k RISC processor, whereas the Tektronix XP-25 embodies a TI 34020 processor. In fact, the use of AMD 29k RISC processor has ensured that the Pericom X-200 terminal performed as well as the Sun workstation hosts whose Sparc processors are also RISC-based.

On the other hand, the PC X-servers produced results in the middle range of performance. Obviously, the X-window software plays the most critical part in determining their performance. This can be demonstrated by the results given in Table 8.8 which show that the 486 DX2/66 running MicroX is obviously much poorer in performance than the 486 DX/66 with eXceed/W and indeed it is even slower than the 486 DX/33 used with eXceed/W. It should also be noted the 486 DX2/66 uses a S3 Vision 928 64-bit graphics processor which is much faster than the WD Paradise 90C30 processor employed in the 486 DX/33. Also, both the graphics adapter (Diamond Stealth 64) and the X-window software (eXceed/W v. 4.0) used in the 486 DX2/66 are generally regarded as being representative of the current state-of-the-art in PC X-server technology. However, the performance of PC X-servers still cannot compete with the RISC-based X-servers, though of course they still offer a very favourable price : performance ratio.

Another important point to note is that the networking between the two Departments over the campus network (See Fig. 8.10) does not have a significant effect on the display speed. This can easily be proved by comparing the results (See Table 8.8) obtained utilising the same type of X-server (*i.e.* the Sun SPARCstation 1+) at both sites. In other words, if a RISC-based X-server can be used at the remote site over the campus network to access Napier88, the graphics response time is almost as good as that performed directly at the local site.

8.7.6 General Aspects of Using Napier88 in the Development and Implementation of the Prototype IGIS

The development of the prototype IGIS has already exploited a large number of the features and system facilities provided by Napier88. In particular, those (essential) features of the language that have been used to provide a full degree of integration for the prototype IGIS have been intensively tested and examined. Thus some general points based on the author's

experience of Napier88 obtained while implementing and evaluating the features or facilities contained in the prototype IGIS can be made as follows.

Pros

- The facility of automatic data persistence is very convenient for the storage of geographical data.
- The Bulk Type libraries, *Lists* and *Maps* in particular, are excellent tools for use in the modelling and structuring of geographical data.
- The same consistent representation of the data used in both programs and databases simplifies the handling and manipulation of geographical data.
- The built-in graphical data types (**pic**, **pixel** and **image**) were extremely useful for the design and development of the graphical functions provided by the prototype IGIS.
- The feature of the language that procedures are first class objects proved to be very useful for the modular design of the prototype IGIS.
- The incremental loading mechanisms were found to be suitable for the construction of the voluminous software libraries needed for use in the prototype IGIS.
- Because of type completeness and strongly-typed check, the composition, compilation and debugging of the prototype IGIS program was generally a smooth process.

Cons

- Several important facilities cannot function properly when using a remote X-server, including the use of colours, the opening of an X-window (See 8.7.4), the use of multiple X-windows (See 8.7.4), *etc.*
- The size of the persistent store may grow unexpectedly when carrying out the update operations for large-size objects such as images (See 8.7.2).
- The Standard Library does not support some essential procedures such as the matrix manipulations and graphics primitives required for GIS software development.
- The Napier88 system does not provide facilities for interfacing digitisers and plotters, neither can an external utility program be embedded for the development of digital mapping functions.
- The persistent store cannot be used concurrently by several programs.
- Several functions are not described clearly in the reference manuals, *e.g.* the instructions concerning the use of the *colourMap* procedure do not point out that the colour intensity values should be in the sequence of B-G-R rather than R-G-B.
- The execution of a Napier88 program based on Release 2 causes a sluggish response (a delay of 5 to 10 seconds) to other jobs - including other Napier88 programs and non-Napier88 applications - running on the same workstation.

On the whole, Napier88 proved to be a powerful and flexible database programming tool in the design and development of the prototype IGIS program. However, quite a number of

problems were encountered in the implementation of the prototype IGIS in real-world situations which have their roots in the Napier88 system.

8.8 Summary

In order to evaluate the suitability of the implementation of a truly IGIS using Napier88, a prototype IGIS has been developed for a series of trials and tests. Because of the difficulty of displaying colours with the WIN facility provided by Napier88 Release 2 using a remote X-server, so the development of the prototype IGIS has had to be based on Release 1 which can correctly display colours in an X-window on the remote X-server. As a result, a basic GUI facility has had to be developed for the prototype IGIS prior to the construction of the IGIS itself. The main functions actually incorporated in the prototype IGIS have emphasised the integration capabilities of geographical information available in both vector and raster formats. The prototype itself has been successfully built and run on the Sun machine hosting the Napier88 system. A large set of vector map data at various scales and raster image data of different resolutions covering the same area has then been used for a wide-ranging series of tests of the prototype IGIS. A PC X-server was configured and mainly used for these tests, but other X-servers have also been provided for an investigation of the problems encountered when using X-windows in conjunction with the Napier88 system.

The tests have examined the functionality and the performance of the persistent IGIS. The main points derived from the test results can be outlined as follows: -

- In general terms, the Napier88 system can support the features and facilities used in the design and development of the prototype IGIS, but it lacks several essential facilities required for the further development of specific GIS functions.

- The prototype IGIS can function properly providing for the full integration of geographical data. This result again confirms the preliminary findings associated with each integration level discussed individually in Chapters 5, 6 and 7.

- The launch time of the persistent IGIS increments with the growth of the store size. In term of running the prototype IGIS, a wait of 1 to 3 minutes for starting the program is unavoidable.

- The optimal threshold values for indexing points and lines are 1,000 and 100 respectively. The corresponding optimal value for polygon entities cannot be determined because the test data set did not contain sufficient polygons.

- The size of the persistent store may be increased unexpectedly when "in situ updates" on the objects held in the persistent store are implemented. The problem has caused difficulty in the operations of the persistent IGIS when used with a large-sized geographical database.

- The Napier88 system often fails to open an X-window and furthermore it cannot open multiple X-windows on a remote X-server.

- A Napier X-client demands much more video memory than a normal X-client.

- The use of different types of X-server has a significant effect on the performance of the graphics display. A RISC-based X-server performs better than other type of X-server. The use of a certain type of X-window software on PC X-servers is also a critical factor in determining the performance.

Based on the results obtained from and the experience gained by carrying out the implementation, trials and tests of the prototype IGIS discussed in this chapter, as well as the results of the tests performed for each level of integration discussed in Chapters 5, 6 and 7, some general conclusions and recommendations will be given in the next chapter.

CHAPTER 9 : CONCLUSIONS AND RECOMMENDATIONS

9.1 Introduction

A true IGIS deals with the storage, processing and display of various types of geographical data in a fully integrated system. The key component in the design of a true IGIS is the capability to integrate various types of geographical data, particularly vector and raster data, together with their functions into a single system. The nature of persistent programming languages is such that they have the potential to meet such a requirement. These languages are based on a novel system architecture, including the provision of a persistent environment to integrate both the program workspace and the data store. This thesis explores the feasibility of developing a fully integrated GIS (FIGIS) using the persistent programming language Napier88. The research focuses on the construction of the basic framework which is essential for building a FIGIS. In particular, this requires the integration of vector, raster and attribute data to be achieved at the storage, process and display levels to the fullest possible extent. The language features provided by Napier88 have been investigated both specifically with a view to them providing the required integration facilities and, more generally, to their application to real-world situations. This chapter presents the general conclusions that can be drawn from this research project and also gives recommendations for future research into this area of integrating geographical data within a GIS environment.

9.2 General Conclusions

Three main objectives have been set out in the Introduction (Section 1.4) for the development of a FIGIS. These were: -

(1) the provision of appropriate GIS software and of a geographical database which is integrated at the storage, process and display levels;
(2) the multiple modelling of geographical data; and
(3) the construction of an object-oriented geographical database.

The requirements to satisfy the first and the second of these objectives, which aimed to provide the support for the storage, process and display levels of integration for the FIGIS, formed the core of this research. The need to satisfy the third objective was to allow object-oriented data management to be implemented in the FIGIS. Six major tasks have been defined (See also Section 1.4) and have been carried out with a view to reaching and satisfying these objectives. The details of the methodologies used and the results obtained from the present research project have been described and discussed in the relevant chapters (4 to 8). Based on these results and the experience gained from the research work, some

general conclusions regarding the development of an IGIS using Napier88 can be set out as follows: -

1. ***The Provision of Data Persistence Eases the Storage and Use of Geographical Data.***
   The unique feature of orthogonal persistence which is supported by persistent programming languages faithfully maps the modelled data of the real world to the data store. The Napier88 system is able to provide longevity for the values of all the geographical data types used in the IGIS program and does not require the explicit organisation of movements of data to and from the storage system by the programmer. Significantly, this provision of data persistence eliminates the need to translate data formats. Furthermore, it reduces considerably the amount of source code which is required since it provides a consistent treatment of the data used both in the IGIS program and in the geographical database.

2. ***The Bulk Type Libraries Can Simplify the Tasks of Geographical Data Structuring.***
   Napier88 provides abundant data types so that all kinds of geographical data can be easily denoted by available data types without effort or by newly constructed data types with very little effort. In particular, the Bulk Type Libraries provide data types with regular data structures and support the operations associated with them. This facility has simplified the task of structuring geographical data significantly since the same representation of data types is used in both the IGIS program and the geographical database. The Bulk Type Libraries have proved to be extremely useful both for data modelling and for the spatial indexing of geographical data.

3. ***The Persistent Store Can Amalgamate Various Types of Geographical Data and Related Library Procedures into an Integrated Storage Unit.***
   The Napier88 system provides a persistent store in which the vector, raster and attribute data describing geographical features are stored as an integrated geographical database side by side with an integrated software library where all the procedures relevant to GIS operations are stored. This facility has achieved the storage integration forming part of Objective 1. It has also resulted in a certain ease of programming because far less use is made of pointers and identifiers. More importantly, the data and the procedures are always fully type-checked by the system to ensure that they are consistent within the persistent store. This feature is especially useful for the maintenance of an integrated geographical database and the implementation of updates of the integrated GIS software over a long period.

4. ***Multi-scale Map Data and Multi-resolution Image Data Held within the Persistent Store Can be Represented by Multiple Data Models.***

   Multiple-scale map data may require different data models for various applications and the same requirement holds true for multiple-resolution image data. In Napier88, type systems are used to represent the data models of various kinds of data. Thus a spatial (vector or raster) data model can be represented as a type system. Since there is no limit to the number of type systems that can exist within Napier88, therefore various data models can coexist within the FIGIS. The provision of this facility means that Objective 2 has been accomplished. In conjunction with Conclusion 2 set out above, a data model can easily be implemented with different data structures to suit specific applications. Thus a condition required for the process integration forming part of Objective 1 has also been achieved.

5. ***The Spatial Indexing Method Based on the Combination of a Linear Quadtree and Peano Ordering Can Index and Query Vector Map Data in an Effective Way.***

   This spatial indexing method has been adopted successfully in order to index vector map data for the prototype IGIS. The provision of this indexing facility allows the selective retrieval of geographical features to be carried out for further manipulation. Therefore it has partially satisfied another condition required for the process integration forming part of Objective 1. It should be emphasised that, generally speaking, the provision of a spatial indexing facility is regarded as a difficult task in the development of GISs. Thus potentially the implementation of this indexing facility in the prototype IGIS could have caused considerable difficulty to the author, if the Napier88 system had not supported the Bulk Type Libraries. In this latter situation, the provision was made relatively easy.

6. ***The Basic Graphical Types Provided by Napier88 Facilitated the Development of the Capability for the Superimposition of Vector Maps and Raster Images and their Subsequent Display.***

   The availability of the three basic graphical data types (**pixel**, **image** and **pic**) proved to be very useful when dealing with the graphical display of geographical data. With the appropriate allocation of bit planes and the arrangement of the colours required for the handling of vector and raster data, the capability of overlaying vector maps and raster images could be realised. The implementation of this function has also helped to lay the foundation for the display integration required by Objective 1. However, at present, the Napier88 system lacks the basic graphical utilities needed for the development of a FIGIS.

7. ***The Spatial Indexing Method Based on Peano Ordering Can Effectively Interrelate Maps and Images Held in the Persistent Store.***

   With the support of the feature given in Conclusion 2, an indexing mechanism based on the Peano ordering of a uniform grid cell has been developed to interrelate vector maps and raster images within the prototype IGIS. This mechanism facilitates the superimposition of maps and images covering an area within a common window. The provision of this indexing mechanism has fulfilled a partial condition for the display integration specified in Objective 1.

8. ***The Prototype IGIS has Realised a Full Degree of Integration; i.e. it is Truly an IGIS.***

   The prototype IGIS can support the storage level (Conclusion 3), the process level (Conclusions 4 and 5) and the display level (Conclusions 6 and 7) of integration needed for various types of geographical data. Furthermore, all the GIS-related procedures have been integrated into the software library being stored along with the integrated geographical database in the persistent store. Since the GIS software and the geographical database have been tightly integrated as a single system, thus the prototype IGIS can be regarded as a FIGIS.

9. ***Several Defects have been Found in the Napier88 System which are Inimical to the Design and Implementation of a FIGIS.***

   Apart from the lack of some essential facilities required for the development of a fully-fledged IGIS (See 8.7.6), the Napier88 system has two major drawbacks which have come to light as a result of this research. First of all, the size of persistent store may grow unexpectedly and, as a result, the launch time of the prototype IGIS program increments with the growth of the store size. Consequently it is very difficult to deal with a large-sized geographical database. Secondly, the X-window facilities cannot function properly on a remote X-server. This problem has caused great difficulties in the design, implementation and execution of the prototype IGIS. In the long run, it will also restrict the development of a distributed IGIS if the problem is not solved.

10. ***The Integrated Geographical Database in the Current Implementation of the Prototype IGIS is Object-based rather than Object-oriented.***

    Since the Napier88 system allows the possibility for the researcher or system developer to develop software and databases with an object-oriented approach, it was planned that an object-oriented data management system would be developed to support the geographical database. However, due to the problems encountered in the use of Napier88 for the IGIS development (as set out in Conclusion 9), the author was forced to divert much effort to the investigation of the unexpected features and the inappropriate functions of several of the facilities provided in Napier88. As a result, the

whole of Objective 3 has not been reached in this research. Nevertheless, all the geographical data have been structured and organised in the form of manageable data objects held within the persistent store. This arrangement allows the possibility for object-oriented data management to be developed in the future. At the present time, the integrated geographical database provided in the prototype IGIS can only be regarded as being object-based rather than object-oriented.

Taking an overall view, the persistent programming language Napier88 can provide a sound integrated database/programming environment for the design and development of a FIGIS. The persistent programming system removes the semantic gap and the impedance mismatch between the program domain and the database domain occurring in conventional (or even object-oriented) programming languages. The underlying principle of orthogonal persistence on which the Napier88 language is based is unique when compared to other programming languages and is novel to GIS researchers and developers. Most benefits arising from the use of Napier88 can be attributed to this particular feature (Conclusion 1). In other words, it is the key factor that makes the full integration of GIS software and geographical databases possible.

Furthermore, the Napier88 system can support the facilities required for the development of the basic functions required in the FIGIS, including multiple modelling of vector map data and raster image data; the superimposition of the tandem processing of the vector maps and raster images; and the spatial indexing of geographical data. Based on the results of this research, it may be concluded that a FIGIS with complete levels of integration can be designed and developed using the Napier88 system, more especially if the various defects encountered and described above can be cured or rectified.

At this moment, it would be inappropriate to say that the Napier88 system is in a mature state and can cope with practical GIS applications. Based on the several difficulties outlined in Sections 8.7.6 and 8.8, and in particular, the problems relevant to the persistent store encountered in the implementation of the prototype IGIS, there are a number of distinct limitations and shortcomings in the language. It can be seen that currently the Napier88 system is not efficient in terms of launching an application program, in this particular case, involving a "large" persistent store. The launch time of the prototype IGIS against a store size of 130 MB requires 2 minutes (See Table 8.10), which is quite unacceptable in real-world applications. In this situation, GIS users would have to wait for at least two minutes in order to perform a simple query which may only require a few seconds to execute. Furthermore, the file size of the data sets used in the trials and tests of the prototype IGIS only had a total size of 33.4 MB, comprising 8.1 MB of vector data and 25.3 MB of raster data (See Table 8.2). By the standards of real-world GIS applications, this is fairly small. The datasets used in an operational GIS are usually quite substantially larger. For example,

the file size of the OS 1:250,000 scale map data required to provide coverage of the whole of Great Britain is 180 MB [Ordnance Survey, 1994]. This figure only reflects the storage requirement of a small-scale map series which has a quite small total file size when compared to other larger-scale map series, not to mention other map series and voluminous image data.

It should be noted that, in an organisation where a workstation can be dedicated to the IGIS operation, the long time required for launching the persistent IGIS may not be too much of a problem since the system is only started once and all the operations are then carried in the persistent environment. However, in many organisations, it may not be possible to provide a dedicated system for GIS work, in which case, the persistent IGIS will have to start and stop as users carry out other tasks using other software systems mounted on the same workstation. Therefore, currently it is not practical to apply the persistent IGIS to real-world situations. In the author's view, the provision of a shorter time in which to launch a Napier88 program is one of the main factors which is critical for the successful implementation of the persistent IGIS using the language. Since Napier88 is still a research language, these problems may be resolved through the further development of the language. If these problems can be solved, the Napier88 system may be regarded as an ideal database programming tool for the design and development of a FIGIS or a true IGIS.

9.3 Recommendations for Future Research

The prototype IGIS developed during this research has already provided the fundamental framework for a true IGIS. Thus it could form a very good basis for further developments and enhancements so as to achieve a fully-fledged IGIS. These could include all the primary functions outlined in Fig. 4.5 and the further development of an object-oriented geographical database. With respect to the detailed design and development of the persistent IGIS using the Napier88 language, the following specific points may be made regarding the work which could be undertaken in the future: -

1. ***The Development of Continuous Map Databases and Image Databases.***
   The prototype IGIS has carried out the integration of multi-scale maps and multi-resolution images in the "vertical" direction. Further development is required to integrate each map series and each image series in the "horizontal" direction. That is, the database of each map/image series should be seamless both in geometry and topology rather than being discretely partitioned into tiles as at present.

2. ***The Development of a High-level Universal Data Model for the Integrated Database.***
   Three kinds of vector data models - the spaghetti, the link and node and the polygon-based models - have been used to represent various series of OS map data. A higher-

level universal data model could be developed to allow the use of these data models in a transparent way. This concept is similar to that implemented in the IGN GeO_2 (See 2.8.2.1). This development would involve the construction of a semantic data model for the three data models and the implementation of an inheritance mechanism between them using an object-oriented approach.

3. ***The Development of Other Spatial Indexing Methods for the Persistent IGIS.***
   Each spatial indexing method has its own particular advantages or characteristics when handling geographical data. Since the Bulk Type Libraries supported by Napier88 can facilitate the implementation of complex data structures, a further development to include one or several spatial indexing methods can be considered for use in the persistent IGIS. This arrangement would allow the selection of an optimal indexing method for a specific dataset. For example, the R-tree indexing method could be implemented in the prototype IGIS for polygon-intensive applications. Furthermore, if Recommendation 1 is implemented, then the spatial indexing should also be based on the use of continuous databases rather on the map sheet or image tile as at present.

4. ***The Development of a Scheme that Can Dynamically Optimise and Allocate Bit Planes for the Display and Manipulation of Dual Format Data.***
   In the current implementation of the prototype IGIS, the number of bit planes allocated for the handling of both vector and raster data is fixed, *i.e.* only 4 bit planes are available for each data format. The number of bit planes actually used by each data format depends on the number of colours needed and the requirements of specific applications. As a result, one data format may not fully use the allocated bit planes, while another format may have to limit the display of information due to insufficient bit planes being available or allocated. Therefore, a further development to provide the capability of allocating the optimal number of bit planes for handling dual-format data is required for the prototype IGIS.

5. ***The Development of a Distributed Geographical Database and a Multi-media IGIS.***
   The use of geographical databases over a communication network is undoubtedly something that will be needed in the future. A geographical database covering a specific theme created and maintained by one organisation could then be shared by others. The integration of various databases into a distributed geographical database could also be developed further. This integration requires the construction of a "logical" persistent store which organises disparate "physical" persistent stores distributed over the network. Furthermore, since the presentation of geographical information in multi-media form is becoming prevalent, the inclusion of video and sound data within the geographical database to form a multi-media IGIS could also be developed.

All these recommendations can be regarded as extended features of the present persistent IGIS. They can be developed based on the existing framework of the prototype IGIS. It is also recommended that further research work should be carried out using Napier88 Release 2 in order to make use of its more advanced features or facilities in the IGIS development.

9.4 Final Remarks

The persistent programming language Napier88 has been developed as a database programming tool to deal with the construction and maintenance of large, long-lived, object-based application systems. The design and implementation of a true IGIS carried out in this research is just one possible application in the fields of Geography and Topographic Science. In the author's view, the Napier88 system could be a good tool to exploit other fields within these disciplines requiring database construction and management. Thus, for example, the Napier88 system might be found useful in other types of system development, such as a digital photogrammetric system (or workstation); a geographical modelling and simulation system; a cartographic expert system; *etc.* Hopefully further investigations into the applications of the Napier88 language for these other types of system development can be conducted by interested researchers.

BIBLIOGRAPHY

Abdallah, A.A., 1990. *The Design and Implementation of a Prototype Geographic Information System Using A Novel Architecture Based on PS-Algol.* Ph.D. Thesis. Department of Geography & Topographic Science, University of Glasgow. 255 pages. [Also published under the same title as *Computing Science Technical Report*, CSC91/R4, University of Glasgow.]

Abel, D.J., 1989. SIRO-DBMS: A Database Tool-Kit for Geographical Information Systems. *International Journal of Geographical Information Systems*, 3(2): 103-116.

AGI, 1994. GIS Standards. In: Green, D.R. and Rix, D. (eds), *The AGI Source Book for GIS 1995*. The Association for Geographic Information: 346-364.

Aldus & Microsoft, 1988. *An Aldus/Microsoft Technical Memorandum: TIFF v.5.0.* 52 pages.

Arnaud, A.M., Craglia, M., Masser, I., Salgé, F. and Scholten H., 1993. The Research Agenda of the European Science Foundation's GISDATA Scientific Programme. *International Journal of Geographical Information Systems*, 7(5): 463-470.

Aronoff, S., 1989. *Geographic Information Systems: A Management Perspective*. WDL Publications, Ottawa. 294 pages: 164-187.

Artz, M., 1991. ArcCAD: The Integration of ARC/INFO and AutoCAD: ArcCAD builds on the Complementary Aspects of the Leaders of the GIS and CAD Industries. *ARC News*, 13(4): 1-2.

Atkinson, M.P., 1978. Programming Languages and Databases. In: Yao, S.B. (ed.) *The Fourth International Conference on Very Large Data Bases*. Berlin, West Germany. 408-419.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott,W.P. and Morrison, R., 1983. An Approach to Persistent Programming. *The Computer Journal*, 26(4): 360-365.

Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R., 1984. Progress with Persistent Programming. *Persistent Programming Research Report*, 8. Universities of St. Andrews and Glasgow. 67 pages

Atkinson, M.P. and Buneman, Q.P., 1987. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2): 105-190.

Atkinson, M.P., Bancilhon, F., Dewitt, D., Dittrich, K., Maier, D. and Zdonik, S., 1989. The Object-Oriented Database System Manifesto. In: *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*. Kyoto, Japan: 40-57.

Atkinson, M.P. and Morrison, R., 1990. Persistent System Architectures. In: Rosenberg, J. and Koch, D. (eds.), *Proceedings of the Third International Workshop on Persistent Object Systems*. Newcastle, Australia. 405 pages: 73-97.

Atkinson, M.P., Lecluse, C. and Philbrow, P., 1991. Maps as Bulk Types for Data Base Programming Languages. *Technical Report Series*, FIDE/91/24, Department of Computing Science, University of Glasgow. 27 pages.

Atkinson, M.P., 1992a. Persistent Foundation for Scaleable Multi-Paradigmal Systems. *International Workshop on Distributed Object Management*, Edmonton, Canada. 29 pages.

Atkinson, M.P., 1992b. *Programming in Napier88*. Lecture Notes on CS4H Databases. Department of Computing Science, University of Glasgow. 51 pages.

Atkinson, M.P., Bailey, P., Christie, D., Cropper, K. and Philbrow, P., 1993a. *Towards Bulk Types Libraries for Napier88* (Release 1). Department of Computing Science, University of Glasgow. 36 pages.

Atkinson, M.P., Trinder, P.W. and Watt, D.A., 1993b. Bulk Type Constructors. *Technical Report Series*, FIDE/93/61. Department of Computing Science, University of Glasgow. 74 pages.

Aybet, J., 1990. Integrated Mapping Systems: Data Conversion and Integration. *Mapping Awareness*, 4(6): 18-23.

Batty P., 1992. Object-Orientation - Some Objectivity Please! *Smallworld Technical Paper*, 18. Smallworld Systems Ltd. 16 pages.

Bennis, K., David, B., Quilio, I. and Viémont, Y., 1990. GéoTropics Database Support Alternatives for Geographic Applications. In: *4th International Symposium on Spatial Data Handling*. Zürich: 599-610.

BSI, 1992. Specification for NTF Structures. *Electronic Transfer of Geographic Information (NTF), Part 1*. British Standard Institution, BS7567. 66 pages.

Brown, A.L., 1989. Persistent Object Stores. *Persistent Programming Research Report*, 71. Universities of St. Andrews and Glasgow. 145 pages.

Brown, C.M., 1988. *Human-Computer Interface Design Guidelines*. Ablex Publishing Corporation, Norwood, New Jersey. 256 pages.

Burrough, P.A., 1986. *Principles of Geographical Information Systems for Land Resources Assessment*. Clarendon Press, Oxford. 193 pages.

Cattell, R.G.G., 1991. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley Publishing Company, New York. 318 pages.

Chance, A., Newell, R.G. and Theriault, D.G., 1989. An Overview of Smallworld Magik. *Smallworld Technical Paper*, 9. Smallworld Systems Ltd. 9 pages.

Chance, A., Newell, R.G. and Theriault, D.G., 1990. An Object-Oriented GIS - Issues and Solutions. *Smallworld Technical Paper*, 7. Smallworld Systems Ltd. 11 pages.

Cogan, L., Luhmann, T. and Walker, A.S., 1991. Digital Photogrammetry at Leica, Arrau. *Digital Photogrammetric Systems*. Wichmann, Karlsruhe: 155-166.

Coleman, D.J. and McLaughlin, J.D., 1992. Standards for Spatial Information Interchange: A Management Perspective. *CISM Journal ACSGC*, 46(2): 133-141.

Connor, R., Cutts, Q., Dearle, A., Kirby, G. and Morrison, R., 1991. *Programmers' Guide to the Napier88 Standard Library, Edition 2*. Universities of St. Andrews and Adelaide. 87 pages.

Cooper, R.L. (ed), 1991. Data Modelling Research 1990-1991. *Computing Science Technical Report*, CSC91/R14, University of Glasgow. 56 pages.

Cooper, R.L. and Kirby, G.N.C., 1994. Type-safe Linguistic Reflection: A Practical Perspective. In: Atkinson, M.P., Maier, D. and Benzaken, V. (eds), *Proceedings of the 7th International Workshop on Persistent Object Systems*. Tarascon. Springer Verlag Workshops in Computer Science: 355-373.

Cowen, D.J., 1988. GIS vs. CAD vs. DBMS: What are the Differences? *Photogrammetric Engineering & Remote Sensing*, 54(11): 1551-1556.

Cromley, R.G., 1992. *Digital Cartography*. Prentice-Hall, Inc., London. 317 pages: 63-174.

Cutts, Q.I., Dearle, A., Kirby, G.N.C. and Marlin, C.D., 1989. WIN: A Persistent Window Management System. *Persistent Programming Research Report*, 73. Universities of St. Andrews and Glasgow. 25 pages.

Cutts, Q.I., Kirby, G.N.C., Connor, R.C.H., Dearle, A. and Marlin, C.D., 1989. An Object-Oriented Approach to Window-Based Applications. *Persistent Programming Research Report*, 72. Universities of St. Andrews and Glasgow. 31 pages.

David, B., Raynal, L., Schorter, G. and Mansart, V., 1993. GeO_2: Why Objects in a Geographical DBMS? In: *SSD'93: The 3rd International Symposium on Large Spatial Databases*. Singapore. 13 pages.

Davis, F.W. and Simonett, D.S., 1991. GIS and Remote Sensing. In: Maguire, D.J., Goodchild, M.F., Rhind, D.W. (eds), *Geographical Information Systems: Principles and Applications*. Longman, London. vol. 1: 191-213.

Davis, F.W., Quadttrochi, D.A., Ridd, M.K., Lam, N.S-N., Walsh, S.J., Michaelsen, J.C., Franklin, J., Stow, D.A., Johannesen, C.J. and Johnston, C., 1991. Environmental Analysis Using Integrated GIS and Remotely Sensed Data: Some Research Needs and Priorities. *Photogrammetric Engineering & Remote Sensing*, 57(6): 689-697.

Dearle, A., Connor, R.C.H., Brown, A.L. and Morrison, R., 1989. Napier88 - A Database Programming Language? *Proceedings of 2nd International Workshop on Database Programming Languages*: 179-195.

Derenyi, E.E., 1991. Design and Development of a Heterogeneous GIS. *CISM Journal ACSGC*, 45(4): 561-567.

Derenyi, E.E. and Pollock, R., 1990. Extending a GIS to Support Image-Based Map Revision. *Photogrammetric Engineering & Remote Sensing*, 56(11): 1493-1496.

DGIWG, 1992. *The Digital Geographic Information Exchange Standard (DIGEST). Part 1: General Description.* Edition 1.1. Digital Geographic Information Working Group. 32 pages.

Dowman, I., 1991. Design of Digital Photogrammetric Workstations. In: Ebner, H., Fritsch, D. and Heipke, C. (eds), *Digital Photogrammetric Systems*. Wichmann, Karlsruhe: 28-38.

Egenhofer, M.J. and Herring, J.R., 1991. High-Level Spatial Data Structures for GIS. In: Maguire, D.J., Goodchild, M.F. and Rhind, D.W. (eds), *Geographical Information Systems: Principles and Applications*. Longman, London. vol. 1: 227-237.

Egenhofer, M.J., 1992. Why not SQL! *International Journal of Geographical Information Systems*, 6(2): 71-85.

Egenhofer, M.J., 1994. Pre-Processing Queries with Spatial Constraints. *Photogrammetric Engineering & Remote Sensing*, 60(6): 783-790.

Ehlers, M., Edwards, G. and Bédard, Y., 1989. Integration of Remote Sensing with Geographic Information Systems: A Necessary Evolution. *Photogrammetric Engineering & Remote Sensing*, 55(11): 1619-1627.

Ehlers, M. and Blesius, L., 1991. Progress in Image Processing Workstations for Remote Sensing. In: Ebner, H., Fritsch, D. and Heipke, C. (eds), *Digital Photogrammetric Systems*. Wichmann, Karlsruhe: 291-294.

ESRI, 1992. ESRI - Tomorrow's Technology, Today? *Mapping Awareness & GIS in Europe*, 6(4): 3-7.

Estes, J.E., 1992. Remote Sensing and GIS Integration: Research Needs, Status and Trends. *ITC Journal*, 1992(1): 2-10.

Evangelatos, T.V., 1991. Digital Geographic Interchange Standards. In: Taylor, D.R.F. (ed), *Geographical Information Systems: The Microcomputer and Modern Cartography*. Pergamon Press, Oxford. 251 pages: 151-166.

Fabbri, A.G., 1992. Remote Sensing, Geographical Information Systems and the Environment: A Review of Interdisciplinary Issues. *ITC Journal*, 1992(2): 119-126.

Fegeas, R.G., Cascio, J.L. and Lazar, R.A., 1992. An Overview of FIPS 173, The Spatial Data Transfer Standard. *Cartography and Geographic Information Systems*, 19(5): 278-293.

Fisher, P., Dykes, J. and Wood, J., 1993. Map Design and Visualization. *The Cartographic Journal*, 30(2): 136-148.

Flowerdew, R., 1991. Spatial Data Integration. In: Maguire. D.J., Goodchild, M.F. and Rhind, D.W. (eds), *Geographic Information Systems: Principles and Applications.* Longman, London. vol. 1: 375-387.

Gahegan, M.N., 1989. An Efficient Use of Quadtrees in a Geographical Information System. *International Journal of Geographical Information Systems*, 3(3): 201-214.

Genasys II, 1991. Genamap - the Open Systems GIS. *Mapping Awareness*, 5(1): 3 -6.

Genasys II, 1993. Genasys - An I^{2}Mage for the Future. *Mapping Awareness & GIS in Europe*, 7(6): 3-7.

Genasys II, 1994. *Geographical Information Systems: System Overview.* Genasys II Ltd. 36 pages.

Gong, J. 1994. An Object-Oriented GIS Software - GEOSTAR. *Proceedings of ISPRS Commission VI Symposium.* Beijing, China: 53-59.

Green, R., 1992. GIS Counters Recession and Builds for Growth. *Mapping Awareness & GIS in Europe*, 6(6): 2-6.

Grenzdörffer, G. and Bill, R., 1994. Digital Orthophotos for Mapping and Interpretation in Hybrid GIS-Environment. *Proceedings of the 5th European Conference on Geographical Information System (EGIS'94).* Paris, France. vol. 2: 1845-1856.

Gunston, M., 1993. *Geographic Information Systems: A Buyer's Guide.* CCTA, London: HMSO: 25-42.

Hamon, C. and Créhange, M., 1991. Object Models and Methodology for Object-Oriented Database Design. In: Harper, D.J. and Norrie, M.C. (eds), *Specification of Database Systems.* Glasgow: 135-153.

Harrington, S., 1987. *Computer Graphics: A Programming Approach.* McGraw-Hill.

Hartnall, T.J., 1993a. *British National Space Centre Integrated Geographical Information System: Prototype System Requirements Specification.* Laser-Scan Ltd., Cambridge, UK. 19 pages.

Hartnall, T.J., 1993b. *British National Space Centre Integrated Geographical Information System: Top Level Design.* Laser-Scan Ltd, Cambridge, UK. 9 pages.

Hartnall, T.J., 1993c. *British National Space Centre Integrated Geographical Information System: Prototype System (Stage 3) Functional Specification.* Laser-Scan Ltd, Cambridge, UK. 86 pages.

Helokunnas, T., 1994. Object-Oriented Geographic Data Management. *Proceedings of the 5th European Conference on Geographical Information System (EGIS'94).* Paris, France. vol. 2: 1194-1203.

Herring, J.R., 1987. TIGRIS: Topologically Integrated Geographic Information Systems. *Proceedings of the AutoCarto 8 Conference.* Baltimore, Maryland: 282-291.

HSI, 1993. *Image Alchemy v. 1.7 User's Manual.* Handmade Software Inc. 210 pages: 195 - 198.

Ibbs, T.J. and Stevens, A., 1988. Quadtree Storage of Vector Data. *International Journal of Geographical Information Systems*, 2(1): 43-56.

Intera Tydac, 1993. SPANS: Product Overview. Intera Tydac Technologies Inc. 30 pages.

Intergraph, 1989a. *Technical Overview - MicroStation GIS Environment.* Intergraph Corporation, Huntsville, Alabama. 6 pages.

Intergraph, 1989b. The Next Generation of GIS. *Mapping Awareness*, 3(5): 3-8.

Intergraph, 1990. *MGE - The Modular GIS Environment.* Intergraph Corporation, Huntsville, Alabama. 25 pages.

IS, 1995. *IMAGINE - Image Processing News from I.S. Ltd.* I.S. Ltd. 4 pages.

Jackson, M.J. and Mason, D.C., 1986. The Development of Integrated Geo-Information Systems. *International Journal of Remote Sensing*, 7(6): 723-740.

Janssen, L.L.F., Jaarsma, M.N. and van der Linden, E.T.M., 1990. Integrating Topographic Data with Remote Sensing for Land-Cover Classification. *Photogrammetric* Engineering *& Remote Sensing*, 56(11): 1503-1506.

Jensen, J.R., Cowen, D.J., Halls, J., Narumalani, S., Schmidt, N.J., Davis, B.A. and Burgess, B., 1994. Improved Urban Infrastructure Mapping and Forecasting for BellSouth Using Remote Sensing and GIS Technology. *Photogrammetric Engineering & Remote Sensing*, 60(3): 339-346.

Kaehler, M.R. and U. Theissing, 1989. Cartographic Raster Archives -The First Step of an Hybrid Geographic Information System Concept. *Proceedings of GIS/LIS'89.* Orlando, Florida. vol. 2: 804-813.

Kaiser, D., 1991. ImageStation: Intergraph's Digital Photogrammetric Workstation. In: Ebner, H., Fritsch, D. and Heipke, C. (eds), *Digital Photogrammetric Systems.* Wichmann, Karlsruhe: 188-197.

Khoshafian, S. and Abnous, R., 1990. *Object-Orientation: Concepts, Languages, Databases, User Interfaces.* John Wiley & Son, Inc. 434 pages: 1-36, 257-322.

Kim, W., 1990. *Introduction to Object-Oriented Databases.* The MIT Press, Cambridge. 234 pages.

Kirby, G.N.C., 1992. *Reflection and Hyper-Programming in Persistent Programming Systems.* Ph.D. Thesis. Department of Computational Science, University of St. Andrews. 175 pages.

Kirby, G., Brown, F., Connor, R., Cutts, Q., Dearle, A., Moore, V., Morrison, R. and Munro, D., 1994. *The Napier88 Standard Library Reference Manual.* Universities of St. Andrews and Adelaide. 132 pages.

Korte, G.B., 1994. *The GIS Book.* OnWord Press, Santa Fe, USA: 16-25, 44-67.

Kraak, M.J., 1993. Cartographical Terrain Modelling in a Three-Dimensional GIS Environment. *Cartography and Geographical Information Systems*, 20(1): 13-18.

Kuo, Y.J., 1994a. The Development of an Integrated GIS Based on a Persistent Programming Language. *Proceedings of the 5th European Conference on Geographical Information System (EGIS'94).* Paris, France. vol. 1: 112-121.

Kuo. Y.J., 1994b. Organizing Geographical Data in a Persistent Store. *Proceedings of the 2nd UK Conference on GIS Research (GISRUK).* Leicester, UK: 204-215.

Langran, G., 1992. *Time in Geographic Information Systems.* Taylor & Francis Inc. 189 pages: 104-119.

Laser-Scan, 1989. Laser-Scan's Metropolis Maps the Way Ahead for Local Authorities. *Mapping Awareness*, 3(5): 3-8.

Laser-Scan, 1990. HORIZON - GIS for the Environment. *Mapping Awareness & GIS in Europe*, 4(7): 3-6.

Laser-Scan, 1991. Implementing GIS in Local Authorities - How to Plan for Success. *Mapping Awareness & GIS in Europe*, 5(8): 3-7.

Laser-Scan, 1992. Laser-Scan: A World of Applications. *Mapping Awareness & GIS in Europe*, 6(8): 3-5.

Laser-Scan, 1993a. *GOTHIC: Developing Applications for Tomorrow's World.* Product Description. Laser-Scan Ltd. 4 pages.

Laser-Scan, 1993b. *IGIS: The Natural Way to Manage the Environment.* Product Description. Laser-Scan Ltd. 4 pages.

Laser-Scan, 1994. *IGIS (Integrated Geographical Information System) - Technical Product Description.* Laser-Scan Ltd. 29 pages.

Lauer, D.T., 1991. The Integration of Remote Sensing and Geographic Information Systems: An Overview of Institutional Issues. In: Star, J.L. (ed.) *Proceedings of the Conference on the Integration of Remote Sensing and Geographic Information Systems*: 33-38.

Laurini, R. and Thompson, D., 1992. *Fundamentals of Spatial Information Systems.* Academic Press Limited, London. 680 pages.

Laurini, R., 1994. Distributed Geographic Databases: An Overview. In: Green, D.R. and Rix, D. (eds), *The AGI Source Book for GIS 1995.* The Association for Geographic Information: 45-55.

Logan, T.L. and Bryant, N.A., 1987. Spatial Data Software Integration: Merging CAD/CAM/Mapping with GIS and Image Processing. *Photogrammetric Engineering & Remote Sensing*, 53(10): 1391-1395.

Maguire, D.J., 1991. An Overview and Definition of GIS. In: Maguire. D.J., Goodchild, M.F. and Rhind, D.W. (eds), *Geographic Information Systems: Principles and Applications.* Longman, London. vol. 1: 9-20.

Maguire, D.J. and Dangermond, J., 1991. The Functionality of GIS. In: Maguire. D.J., Goodchild, M.F. and Rhind, D.W. (eds), *Geographic Information Systems: Principles and Applications.* Longman, London. vol. 1: 319-335.

Maguire, D.J., Kimber, B. and Chick, J., 1991. Integrated GIS: The Importance of Raster. *Technical Papers of the 1991 ACSM-ASPRS Annual Convention.* Baltimore. 107-116.

Maguire, D.J. and Dangermond, J., 1994. Future GIS Technology. In: Green, D.R. and Rix, D. (eds), *The AGI Source Book for GIS 1995.* The Association for Geographic Information: 113-120.

Marble, D.F., 1994. An Introduction to the Structured Design of Geographic Information Systems. In: Green, D.R. and Rix, D. (eds), *The AGI Source Book for GIS 1995.* The Association for Geographic Information: 31-38.

Mark, D.M., Lauzon, J.D. and Gebrian, J.A., 1989. A Review of Quadtree-based Strategies for Interfacing Coverage Data with Digital Elevation Models in Grid Form. *International Journal of Geographical Information Systems*, 3(1): 3-14.

Mason, D.C., Corr., D.G., Cross, A., Hogg, D.C., Lawrence, D.H., Detrou, M. and Tailor, A.M., 1988. The Use of Digital Map Data in the Segmentation and Classification of Remotely-Sensed Images. *International Journal of Geographical Information Systems*, 2(3): 195-215.

Mather, P.M., 1987. *Computer Processing of Remotely-Sensed Images - An Introduction.* John Wiley & Sons, Chichester. 352 pages.

Matambanadzo, P., 1992. *Single and Stereo-Pair Methods Using Digital Imagery in Photogrammetry.* M.App.Sci. Dissertation. Department of Geography & Topographic Science, University of Glasgow. 208 pages.

McRae, S.D., 1989. GIS Design and the Questions Users Should be Asking.. *Proceedings of GIS/LIS'89*. Orlando, Florida. vol. 2: 528-537.

Menon, S., Gao, P. and Zhan, C., 1991. GRID: A Data Model and Functional Map Algebra for Raster Geo-Processing. *Proceedings of GIS/LIS'91*. Atlanta, Georgia. vol. 2: 551-561.

Milne, P., Milton, S. and Smith J.L., 1993. Geographical Object-Oriented Databases - A Case Study. *International Journal of Geographical Information Systems*, 7(1): 39-55.

Morehouse, S., 1989. The Architecture of ARC/INFO. *Proceedings of the AutoCarto 9 Conference*. Baltimore, Falls Church, Virginia, USA: 266-277.

Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P., 1986. An Integrated Graphics Programming Environment. *Persistent Programming Research Report*, 14. Universities of St. Andrews and Glasgow. 15 pages.

Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. and Atkinson, M.P., 1987. Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment. *Software Engineering Journal*, 2(6):199-204.

Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A.J. and Livesey, M.J., 1989. Language Design Issues in Supporting Process-Oriented Computation in Persistent Environments. *Proceedings of 22nd International Conference on System Sciences*. Hawaii. 736-744.

Morrison, R., Brown, F., Connor, R., Dearle, A., 1989. The Napier88 Reference Manual. *Persistent Programming Research Report*, 77. Universities of St. Andrews and Glasgow. 96 pages

Morrison, R., Brown, F., Carrick, R., Connor, R., Dearle, A. and Atkinson, M.P., 1990. The Napier Type System. *Proceedings of the Third International Workshop on Persistent Object Systems*. Newcastle, Australia. 405 pages: 3-17.

Morrison, R., Baker, C., Connor, R.C.H., Cutts, Q.I. and Kirby, G.N.C., 1993a. Approaching Integration in Software Environment. *Technical Report*, CS/93/10. Department of Computational Science, University of St. Andrews. 11 pages.

Morrison, R., Brown, F., Connor, R., Cutts, Q., Dearle, A., Kirby, G. and Munro, D., 1993b. The Napier88 Reference Manual (Release 2.0). *Technical Report*, CS/93/15. Department of Computational Science, University of St. Andrews. 67 pages.

Mortension, M.E., 1989. *Computer Graphics: An Introduction to the Mathematics and Geometry*. Heinemann Newnes, Oxford. 381 pages.

Newell, R.G. and Theriault, D.G., 1989. Ten Different Problems in Building a GIS. *Smallworld Technical Paper*, 1. Smallworld Systems Ltd. 9 pages.

Newell, R.G. and Sancha, T.L., 1990. The Difference Between CAD and GIS. *Smallworld Technical Paper*, 5. Smallworld Systems Ltd. 10 pages.

Newell, R.G., 1992. Practical Experience of Using Object-Orientation to Implement a GIS. *Smallworld Technical Paper*, 16. Smallworld Systems Ltd. 8 pages.

Newell, R.G., 1994. Where is GIS Technology going? In: Green, D.R. and Rix, D. (eds), *The AGI Source Book for GIS 1995*. The Association for Geographic Information: 19-23.

Newton, P.W., Zwart, P.R. and Cavill, M.E. (eds), 1992. *Networking Spatial Information Systems*. Belhaven Press, London and New York: 49-88, 205-250.

Nordstrand, E., 1990. Using Raster Data with ARC/INFO. *ARC News*, 12(1): 32-33.

NRSC, 1992. *Satellites and Maps - Port Talbot*. NRSC. 2 pages.

Oosterhoff, A., 1993. *The Integration of GIS, Remote Sensing and Image Processing Systems: An Annotated Bibliography*. Department of Geographic Information Systems, School of Computing, Curtin University of Technology. 36 pages.

Ordnance Survey, 1993a. *A Technical Guide for Sample Digital Map Data in National Transfer Format, Version 2.0*. Ordnance Survey. 270 pages.

Ordnance Survey, 1993b. *Digital Map Data - 1:50,000 Scale Colour Raster Technical Information*. Ordnance Survey. 3 pages.

Ordnance Survey, 1994. *Digital Map Data Catalogue*. Ordnance Survey. 32 pages.

PCI, 1993. *Using PCI Software, Volume 1*. PCI Remote Sensing Corp. (3): 2-13.

Peloux, J.P., St. Michel, G.R., Scholl, M., 1993. Evaluation of Spatial Indices Implemented with the DBMS O_2 (Draft). *Technical Paper of ESPRIT Basic Research Action Program 6881 (AMUSING)*. 21 pages.

Petrie, G., 1989a. Networking for Digital Mapping. Part I - Past and Present Networking Solutions for Digital Mapping. *Mapping Awareness*, 3(3): 9-16.

Petrie, G., 1989b. Networking for Digital Mapping. Part II - The OSI Network Model and its Application in Digital Mapping. *Mapping Awareness*, 3(4): 38-42.

Petrie, G., 1994. Photogrammetry and Remote Sensing. In: Green, D.R. and Rix, D. (eds), *The AGI Source Book for GIS 1995*. The Association for Geographic Information: 73-85.

Peuquet D.J., 1981a. An Examination of Techniques for Reformatting Digital Cartographic Data. Part 1: The Vector-to-Raster Process. *Cartographica*, 18(1): 34-48.

Peuquet D.J., 1981b. An Examination of Techniques for Reformatting Digital Cartographic Data. Part 2: The Raster-to-Vector Process. *Cartographica*, 18(3): 21-33.

Peuquet D.J., 1984. A Conceptual Framework and Comparison of Spatial Data Models. *Cartographica*, 21(4): 66-113.

Peuquet, D.J., 1991. Methods for Structuring Digital Cartographic Data. In: Taylor, D.R.F. (ed), *Geographical Information Systems: The Microcomputer and Modern Cartography.* Pergamon Press, Oxford: 67-95.

Piwowar, J.M. and LeDrew, E.F., 1990. Integrating Spatial Data: A User's Perspective. *Photogrammetric Engineering & Remote Sensing*, 56(11): 1497-1502.

Piwowar, J.M., LeDrew, E.F. and Dudycha, D.J., 1990. Integration of Spatial Data in Vector and Raster Formats in a Geographical Information System Environment. *International Journal of Geographical Information Systems*, 4(4): 429-444.

Rado, B.Q., Bury, A.S., Smith, C.C., 1991. Raster-Vector Integration: Real World Solutions. *Technical Papers of the 1991 ACSM-ASPRS Annual Convention.* Baltimore. 166-172.

Sacchi, C. and Sbattella, L., 1994. An Object-Oriented Approach to Spatial Databases. *Proceedings of the 5th European Conference on Geographical Information System (EGIS'94).* Paris, France. vol. 2: 1204-1213.

Samet, H., 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2): 220-248.

Samet, H., 1989. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, Mass.: 1-56.

Schumacher, D., 1992. General Filtered Image Rescaling. In: Kirk, D. (ed), *Graphics Gems III.* Academic Press, San Diego London: 8-16.

Shaffer, C.A., 1990. QUILT: A Geographic Information System Based on Quadtrees. *International Journal of Geographical Information Systems*, 4(2): 103-131.

Shepherd, I.D.H., 1991. Information Integration and GIS. In: Maguire. D.J., Goodchild, M.F. and Rhind, D.W. (eds), *Geographic Information Systems: Principles and Applications.* Longman, London. vol. 1: 337-360.

Siemens, 1989. Siemens - Making GIS Work. *Mapping Awareness*, 3(3): 3-7.

Sinha, A.K. and Waugh, T.C., 1988. Aspects of the Implementation of the GEOVIEW Design. *International Journal of Geographical Information Systems*, 2(2): 91-99.

Smallworld, 1992. Smallworld: GIS Innovation and Vision. *Mapping Awareness & GIS in Europe*, 6(3): 3-7.

Sproull, R. F., Sutherland, W. R. and Ullner, M.K., 1985. *Device-Independent Graphics with Examples from IBM Personal Computers.* McGraw-Hill, Inc., 485-491.

Star, J. and Estes, J., 1990. *Geographic Information Systems: An Introduction.* Prentice Hall, New Jersey. 303 pages.

Stow, D., Westmoreland, S., McKinsey, D., Mertz, F., Nathanson, J., Sperry, S. and Nagel D., 1990. Efficient Creation, Correction and Updating of Vector-Coded GIS Coverages Using Remotely Sensed Data. *Proceedings of GIS/LIS'90.* Anaheim, California. vol. 1: 209- 218.

Strachey, C., 1967. *Fundamental Concepts in Programming Languages.* Oxford University Press, Oxford.

Stuart, N., 1990. Quadtrees GIS - Pragmatics for the Present, Prospects for the Future. *Proceedings of GIS/LIS'90.* Anaheim, California. vol. 1: 373-382.

Tektronix, 1991. *X Windows Primer.* Tektronix UK Ltd. 27 pages.

The Committee for Advanced DBMS Functions, 1990. Third-Generation Database System Manifesto. *SIGMOD Record*, 19(3): 31-45.

Theodoridis, Y. and Sellis, T., 1993. Optimization Issues in R-tree Construction. *Technical Paper of ESPRIT Basic Research Action Program 6881 (AMUSING).* 20 pages.

Theriault, D.G., 1989. An Overview of Geographical Information Systems - The Technology and its Users. *Smallworld Technical Paper*, 2. Smallworld Systems Ltd. 8 pages.

Tydac, 1989. Tydac Technologies: Thinking Spatially. *Mapping Awareness*, 2(6): 3-7.

Tydac, 1990. *SPANS: SPatial ANalysis System.* Tydac Technologies. 16 pages.

Tydac, 1991. From Mapping to GIS: Tydac Offers More. *Mapping Awareness & GIS in Europe*, 5(10): 3-7.

Tydac, 1994. *SPANS Explorer.* Tydac Technologies. 2 pages.

Usery, E.L., 1993. Category Theory and the Structure of Features in Geographic Information Systems. *Cartography and Geographic Information Systems*, 20(1): 5-12.

van Eck, J.W. and Uffer, M., 1989. A Presentation of System 9. *Photogrammetry and Land Information Systems*: 139-178.

van Oosterom, P., 1993. *Reactive Data Structures for Geographic Information Systems.* Oxford University Press Inc., New York. 198 pages.

Verts, W., 1989. Quadtree Meshes. *Proceedings of the AutoCarto 9 Conference.* Baltimore, USA: 406-415.

Vijbrief, T., and van Oosterom, P., 1992. The GEO System: An Extensible GIS. *Proceedings of the 5th International Symposium on Spatial Data Handling.* Charleston, South Carolina: 40-55.

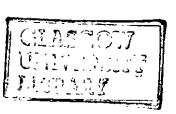
Wallace, T. and Clark, S.R., 1988. Raster and Vector Data Integration: Past Techniques, Current Capabilities, and Future Trends. *Proceedings of GIS/LIS'88*. San Antonio, Texas. vol. 1: 418-426.

Waugh, T.C. and Healey, R.G., 1987. The GEOVIEW Design: A Relational Data Base Approach to Geographical Data Handling. *International Journal of Geographical Information Systems*, 1(2): 101-118.

Worboys, M.F., Hearnshaw, H.W. and Maguire, D.J., 1990. Object-Oriented Data Modelling for Spatial Databases. *International Journal of Geographical Information Systems*, 4(4): 369-383.

Williams, R.J., 1993. Digital Geographic Data Exchange Standards and Products: Descriptions, Comparisons and Opinions. *Cartography: Journal of the Australian Institute of Cartographers*, 22(1): 15-53.

Yearsley, C., Worboys, M.F., Story, P., Jayawardena, D.P.W. and Bofakos, P., 1994. Computational Support for Spatial Information Handling: Models and Algorithms. In: Worboys, M.F. (ed), *Innovation in GIS*. Taylor & Francis Ltd: 75-88.



THE DESIGN AND IMPLEMENTATION

OF

A TRULY INTEGRATED GIS

USING THE PERSISTENT PROGRAMMING LANGUAGE NAPIER88

BY

YING JEAN KUO

VOLUME II

A Thesis Submitted for the Degree of Doctor of Philosophy (Ph.D.)
of the Faculty of Science at the University of Glasgow

Department of Geography &
Topographic Science

Department of Computing Science

June 1995

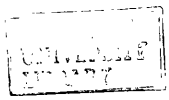


Table of Contents

VOLUME II

APPENDIX A : THE CREATION OF GIS DATA TYPES

```
!
! System types
!
     type Map[A, Z] is MapRep1to1[A, Z]
     rec type List[T] is variant(full: Cell[T]; empty: null)
     &        Cell[T] is structure(hd: T; tl: List[T])
!
!-------------------------------------------------------------------------
!
! General types
!
! 1. Identifers
!
     type Point_id is int
     type Line_id is int
     type Text_id is int
     type Link_id is int
     type Node_id is int
     type Geom_id is int
     type Attr_id is int
     type Poly_id is int
     type Chain_id is int
     type Cpoly_id is int
     type Coll_id is int
     type Grid_id is int
     type Peano_key is int

     type Map_id is string
     type Image_id is string

!
! 2. Geometry
!
     type Pos is structure(x, y: int)
     type XY is structure(x, y: real)
     type MBR is structure(x_min, y_min, x_max, y_max: real)
     type Extent is structure(x_min, y_min, x_range, y_range: real)
     type Circle is structure(center: XY; radius: real)
     type Win_size is structure(width, height: int)
     type Transient_image is structure(raster: image; pos: Pos;
                                                      start_bp: int)
     type Frequency is structure(count: *int; lower, median, upper,
                                                      vmin, vmax: int)
     type Freq_chart is structure(original_chart, stretch_chart: pic;
                                                           level: int)
!
! 3. Feature Code & Description
!
     type FC is string    ! Feature Code
     type FD is string    ! Feature Description
     type FCD is Map[FC, FD]

!
! 4. Spatial index key
!
     type Peano is structure(peano_key: int; side_length: real)
     type Peanor is structure(peano_key, side_length: real)
     type Map_index is Map[Peano, Map_id]
     type Image_index is Map[Peano, List[Image_id]]
!
!
     type OS_map_info is structure(name: string; series: string;
                                   mapscale: real; extent: Extent)
     type OS_map_name is structure(s_625k, s_250k, s_50k, s_10k, s_2500,
                                   s_1250, oscar, boundary_line: string)
!
```

```
! Data structures
!
! 1. Point
!
!    Spaghetti structure
!
     type Landline_point_attr is structure(orientation: real; distance: int)
     type Contour_point_attr is real  ! height
     type SP_point_attr is variant(landline: Landline_point_attr;
                                   contour: Contour_point_attr)
     type SP_point is structure(xy: XY; fc: FC; attribute: SP_point_attr)
     type SP_pid_point is Map[Point_id, SP_point]
!
!    Link and Node structure
!
     type LN_point is structure(geom_id: Geom_id; attr_id: Attr_id)
     type LN_pid_point is Map[Point_id, LN_point]
!
!    Square Grid DTM
!
     type SG_point is structure(x_first, y_first: real; z: List[real])
     type SG_gid_point is Map[Grid_id, SG_point]
!
! 2. Line
!
!    Spaghetti structure
!
     type Contour_line_attr is real  ! height
     type SP_line_attr is variant(landline: null; contour: Contour_line_attr)
     type SP_line is structure(xy_list: List[XY]; fc: FC;
                               attribute: SP_line_attr)
     type SP_lid_line is Map[Line_id, SP_line]
!
!    Link and Node structure
!
     type LN_line is structure(geom_id: Geom_id; attr_id: Attr_id)
     type LN_lid_line is Map[Line_id, LN_line]
!
! 3. Text
!
!    Spaghetti structure
!
     type SP_text is structure(xy: XY; text_code, text_body: string;
                               text_ht, orient: real; font, dig_postn: int)
     type SP_tid_text is Map[Text_id, SP_text]
     type SP_tid_txt is variant(sp_tid_text: SP_tid_text; none: null)
!
!    Link and Node structure
!
     type LN_text is structure(geom_id: Geom_id; attr_id: Attr_id;
                               text_ht, orient: real; font, dig_postn: int)
     type LN_tid_text is Map[Text_id, LN_text]
     type LN_tid_txt is variant(ln_tid_text: LN_tid_text; none: null)
!
! 4. Geometry
!
!    Link and Node structure
!
     type LN_geometry is structure(gtype, num_coord: int; xy_list: List[XY])
     type LN_gid_geometry is Map[Geom_id,LN_geometry]
!
!    Polygon-based structure
!
     type PB_geometry is structure(gtype, num_coord: int; xy_list: List[XY];
                                   attr_id: Attr_id)
     type PB_gid_geometry is Map[Geom_id, PB_geometry]
```

```
!
! 5. Attribute
!
!    Link and Node structure
!
     type LN_attr_ssm is structure(RB, RU: bool; OR: real; PN, NU: string)
     type LN_attr_oscar is structure(SY: int; LL: real;
                                     SC, PN, RN, FW: string)
     type LN_attr is variant(small_scale_map: LN_attr_ssm;
                             oscar: LN_attr_oscar)
     type LN_attribute is structure(fc: FC; attribtue: LN_attr)
     type LN_aid_attribute is Map[Attr_id, LN_attribute]
!
!    Polygon-based structure
!
     type PB_attribute is structure(AI, LK, PI, HW, LV: int; HA: real;
                                    fc: FC; NM, OP, SD, CT: string)
     type PB_aid_attribute is Map[Attr_id, PB_attribute]
!
! 6. Link
!
!    Link and Node structure
!
     type LN_link is structure(from_node, to_node: Node_id;
                               geom_id_of_link: Geom_id)
     type LN_kid_link is Map[Link_id, LN_link]
!
! 7. Node
!
!    Link and Node structure
!
     type Link is structure(direction: int; geom_id_of_link: Geom_id;
                            orient: real; level: int)
     type LN_node is structure(geom_id_of_node: Geom_id; num_links: int;
                               link_list: List[Link])
     type LN_nid_node is Map[Node_id, LN_node]
!
! 8. Polygon
!
!    Polygon-based structure
!
     type PB_polygon is structure(chain_id: Chain_id; geom_id: Geom_id;
                                  attr_id: Attr_id)
     type PB_polyid_polygon is Map[Poly_id, PB_polygon]
!
! 9. Chain
!
!    Polygon-based structure
!
     type PB_link is structure(direction: int; geom_id_of_link: Geom_id)
     type PB_chain is structure(num_parts: int; link_list: List[PB_link])
     type PB_cid_chain is Map[Chain_id, PB_chain]
!
!10. Cpolygon
!
!    Polygon-based structure
!
     type PB_polyid_sign is structure(poly_id: Poly_id; sign: string)
     type PB_cpolygon is structure(num_parts: int;
                                   polyid_sign_list: List[PB_polyid_sign];
                                   geom_id: Geom_id; attr_id: Attr_id)
     type PB_cpolyid_cpolygon is Map[Cpoly_id, PB_cpolygon]
!
!11. Collection
!
!    Polygon-based structure
```

```
!
     type PB_polyid is structure(poly_id: Poly_id; poly_type: int)
     type PB_collection is structure(num_parts: int;
                                     polyid_list: List[PB_polyid];
                                     attr_id: Attr_id)
     type PB_collid_collection is Map[Coll_id, PB_collection]
!
!
! Data Models
!
!
! 1. Spaghetti data model
!
     type SP_DM is structure(point: SP_pid_point;
                             line: SP_lid_line;
                             txt: SP_tid_txt;
                             fcd: FCD)
!
! 2. Link and Node data model
!
     type LN_DM is structure(point: LN_pid_point;
                             line: LN_lid_line;
                             geometry: LN_gid_geometry;
                             attribute: LN_aid_attribute;
                             link: LN_kid_link;
                             node: LN_nid_node;
                             txt: LN_tid_txt;
                             fcd: FCD)
!
! 3. Polygon-based data model
!
     type PB_DM is structure(collection: PB_collid_collection;
                             cpolygon: PB_cpolyid_cpolygon;
                             polygon: PB_polyid_polygon;
                             chain: PB_cid_chain;
                             geometry: PB_gid_geometry;
                             attribute: PB_aid_attribute;
                             fcd: FCD)
!
! 4. Grid Cell data model
!
     type GC_attribute is string
     type GC_aid_attribute is Map[Attr_id, GC_attribute]
     type GC_DM is structure(raster: image;
                             extent: Extent;
                             colourmap: **int;
                             attribute: GC_aid_attribute)
!
! 5. Run-Length Encoding data model
!      (Value Point Encoding)
!
     type Value_point is int
     type RLE_attr is string
     type RLE_DM is Map[Value_point, RLE_attr]
!
! 6. Linear Quadtree data model
!
     type LQT_attribute is string
     type LQT_quadrant is structure(side_length: int; attr_id: Attr_id)
     type LQT_peano_quadrant is Map[Peano_key, LQT_quadrant]
     type LQT_aid_attribute is Map[Attr_id, LQT_attribute]
     type LQT_DM is structure(quadrant: LQT_peano_quadrant;
                              attribute: LQT_aid_attribute;
                              extent: Extent;
                              colourmap: **int;
```

```
                                pixel_size: real;
                                depth: int)
!
! Databases
!
!
! 1. Basemap database
!
     type Basemap_DM is variant(spaghetti: SP_DM;
                                link_node: LN_DM;
                                polygon_based: PB_DM)
     type Basemap is structure(data_model: Basemap_DM)
     type Base_Maps is Map[Map_id,Basemap]
!
! 2. Baseimage database
!
     type Baseimage_DM is variant(grid_cell: GC_DM; linear_quadtree: LQT_DM)
     type Baseimage is structure(data_model: Baseimage_DM)
     type Base_Images is Map[Image_id, Baseimage]
!
! 3. Raw image database
!
     type Rawimage is structure(data: *int;
                                width, height, depth: int;
                                colourmap: **int)
     type Raw_Images is Map[Image_id, Rawimage]
!
! 4. Interim image database
!
     type Interim_image is structure(raster: image; colourmap: **int)
     type Interim_Images is Map[Image_id, Interim_image]
!
!
! Spatial Indexing
!
! 1. Entity MBR
!
     type Line_mbr is Map[Line_id, MBR]
     type Polygon_mbr is Map[Poly_id, MBR]
     type Line_key_pts is Map[Line_id, List[XY]]
     type Entity_mbr is structure(line: Line_mbr;
                                  polygon: Polygon_mbr;
                                  key_pts: Line_key_pts)
     type Entity_MBRs is Map[Map_id, Entity_mbr]
!
! 2. Entity Indices
!
     type Point_index is Map[Peanor, List[Point_id]]
     type Line_index is Map[Peanor, List[Line_id]]
     type Polygon_index is Map[Peanor, List[Poly_id]]
     type Min_quad_sl is structure(point, line, polygon: real)
     type Entity_index is structure(point: Point_index;
                                    line: Line_index;
                                    polygon: Polygon_index;
                                    min_quad_sl: Min_quad_sl)
     type Entity_indices is Map[Map_id, Entity_index]

!
! 3. Basemap & Baseimage Indices
!
     type Basemap_indices is Map[Peano, Basemap]
     type Baseimage_indices is Map[Peano, Baseimage]
```

APPENDIX B : THE CREATION OF LIBRARY AND DATABASE ENVIRONMENT

```
!-------------------------------------------------------------------------!
!                                                        [mk_library_env.N] !
!                                                                         !
!      Set up Library environment in User environment                     !
!                                                                         !
!  PS() --|                                                              !
!         |-- User --|                                                   !
!                    |-- Library                                        !
!                                                                         !
!-------------------------------------------------------------------------!
```

```
use PS() with IO, User: env; environment: proc(-> env) in
use IO with writeString: proc(string) in
begin
   if User contains Library then
      writeString("User already contains Library, no action taken.'n")
   else
      begin
         in User let Library := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                        [mk_general_env.N] !
!                                                                         !
!      Set up General environment in Library environment                  !
!                                                                         !
!  PS() --|                                                              !
!         |-- User --|                                                   !
!                    |-- Library --|                                     !
!                                  |-- General                          !
!                                                                         !
!-------------------------------------------------------------------------!
```

```
use PS() with IO, User: env; environment: proc( -> env) in
use User with Library: env in
use IO with writeString: proc(string) in
begin
   if Library contains General then
      writeString("Library already contains General, no action taken.'n")
   else
      begin
         in Library let General := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                      [mk_graphical_env.N] !
!                                                                         !
!      Set up Graphical environment in Library environment                !
!                                                                         !
!  PS() --|                                                              !
!         |-- User --|                                                   !
!                    |-- Library --|                                     !
!                                  |-- Graphical                        !
!                                                                         !
!-------------------------------------------------------------------------!
```

```
use PS() with IO, User: env; environment: proc( -> env) in
use User with Library: env in
use IO with writeString: proc(string) in
begin
   if Library contains Graphical then
      writeString("Library already contains Graphical, no action taken.'n")
   else
      begin
```

```
         in Library let Graphical := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                            [mk_gis_env.N] !
!                                                                           !
!      Set up GIS environment in Library environment                        !
!                                                                           !
!  PS() --|                                                                 !
!         |-- User --|                                                      !
!                    |-- Library --|                                       !
!                                  | -- GIS                                !
!                                                                           !
!-------------------------------------------------------------------------!
```

```
use PS() with IO, User: env; environment: proc( -> env) in
use User with Library: env in
use IO with writeString: proc(string) in
begin
   if Library contains GIS then
      writeString("Library already contains GIS, no action taken.'n")
   else
      begin
         in Library let GIS := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                                         !
!                                                       [mk_database_env.N] !
!      Set up Database environment in User environment                    !
!                                                                         !
!  PS() --|                                                               !
!         |-- User --|                                                    !
!                    |-- Database                                        !
!                                                                         !
!-------------------------------------------------------------------------!
use PS() with IO, User: env; environment: proc( -> env) in
use IO with writeString: proc(string) in
begin
   if User contains Databse then
      writeString("User already contains Database, no action taken.'n")
   else
      begin
         in User  let Database := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                            [mk_raw_env.N] !
!                                                                           !
!      Set up Raw environment in Database environment                       !
!                                                                           !
!  PS() --|                                                                 !
!         |-- User --|                                                      !
!                    |-- Database --|                                      !
!                                   |-- Raw                                 !
!                                                                           !
!-------------------------------------------------------------------------!
use PS() with IO, User: env; environment: proc( -> env) in
use User with Database: env in
use IO with writeString: proc(string) in
begin
   if Database contains Raw then
      writeString("Database already contains Raw, no action taken.'n")
```

```
   else
      begin
         in Database let Raw := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                          [mk_raw_images.N] !
!                                                                          !
!      Set up raw_images in Raw environment                                !
!                                                                          !
!  User --|                                                               !
!         |-- Database --|                                                !
!                        |-- Raw --|                                      !
!                                  |-- raw_images: Map[Image_id,Rawimage]  !
!                                                                          !
!-------------------------------------------------------------------------!
type Map[A, Z] is MapRep1to1[A,Z]
use PS() with IO, User, GlasgowLibraries:env; environment: proc( -> env) in
use IO with writeString: proc(string) in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps: env in
use Maps with
    m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) -> Map[A,Z]) in
use User with Library, Database: env in
use Library with General: env in
use General with eq_str: proc(string, string -> bool);
                 lt_str: proc(string, string -> bool)  in
use Database with Raw: env in
begin
   if Raw contains raw_images then
      writeString("Raw already contains raw_images, no action taken.'n")
   else
      begin
         in Raw let raw_images := m_empty[Image_id,Rawimage](eq_str,lt_str)
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                          [mk_interim_env.N] !
!                                                                          !
!      Set up Interim environment in Database environment                  !
!                                                                          !
!  PS() --|                                                               !
!         |-- User --|                                                    !
!                    |-- Database --|                                     !
!                                   |-- Interim                          !
!                                                                          !
!-------------------------------------------------------------------------!
use PS() with IO, User: env; environment: proc( -> env) in
use User with Database: env in
use IO with writeString: proc(string) in
begin
   if Database contains Inerim then
      writeString("Database already contains Interim, no action taken.'n")
   else
      begin
         in Database let Interim := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                      [mk_interim_images.N] !
!                                                                          !
!      Set up interim_images in Interim environment                        !
```

```
!                                                                              !
!  Database -|                                                                 !
!            |- Interim -|                                                    !
!                        |- interim_images: Map[Image_id,Interim_image]       !
!                                                                              !
!------------------------------------------------------------------------------!
type Map[A, Z] is MapRep1to1[A,Z]
use PS() with IO, User, GlasgowLibraries: env; environment: proc( -> env) in
use IO with writeString: proc(string) in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps: env in
use Maps with
    m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) -> Map[A,Z]) in
use User with Library, Database: env in
use Library with General: env in
use General with eq_str: proc(string,string -> bool);
                 lt_str: proc(string,string -> bool) in
use Database with Interim: env in
begin
   if Interim contains interim_images then
      writeString("Interim already contains interim_images, no action
taken.'n")
   else
      begin
         in Interim let interim_images :=
                           m_empty[Image_id, Interim_image](eq_str, lt_str)
      end
end
```

```
!------------------------------------------------------------------------------!
!                                                          [mk_processed_env.N] !
!                                                                              !
!     Set up Maps environment in Data environment                              !
!                                                                              !
!  PS() --|                                                                    !
!         |-- User --|                                                         !
!                    |-- Database --|                                         !
!                                   |-- Processed                              !
!                                                                              !
!------------------------------------------------------------------------------!
use PS() with IO, User: env; environment: proc( -> env) in
use User with Database: env in
use IO with writeString: proc(string) in
begin
   if Database contains Processed then
      writeString("Database already contains Processed, no action taken.'n")
   else
      begin
         in Database let Processed := environment()
      end
end
```

```
!------------------------------------------------------------------------------!
!                                                            [mk_base_maps.N] !
!                                                                              !
!      Set up base_maps in Processed environment                               !
!                                                                              !
! User --|                                                                     !
!        |-- Database --|                                                      !
!                       |-- Processed --|                                     !
!                                       |-- base_maps: Map[Map_id, Basemap]  !
!                                                                              !
!------------------------------------------------------------------------------!
type Map[A, Z] is MapRep1to1[A,Z]
use PS() with IO, User, GlasgowLibraries: env; environment: proc( -> env) in
use GlasgowLibraries with BulkTypes: env in
```

```
use BulkTypes with Maps: env in
use Maps with
    m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) -> Map[A,Z]) in
use User with Library, Database: env in
use Library with General: env  in
use General with eq_str, lt_str:  proc(string,string -> bool) in
use Database with Processed: env in
use IO with writeString: proc(string) in
begin
   if Processed contains base_maps then
      writeString("The Processed environemnt already contains base_maps, no
action taken.'n")
   else
   begin
      in Processed let base_maps := m_empty[Map_id,Basemap](eq_str,lt_str)
   end
end
```

```
!-------------------------------------------------------------------------!
!                                                          [mk_base_images.N] !
!                                                                         !
!      Set up base_images in Processed environment                        !
!                                                                         !
!    Database --|                                                         !
!               |-- Processed --|                                        !
!                               |-- base_images: Map[Image_id, Baseimage]  !
!                                                                         !
!-------------------------------------------------------------------------!
```

```
type Map[A,Z] is MapRep1to1[A,Z]
use PS() with IO, User, GlasgowLibraries: env; environment: proc( -> env) in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps: env in
use Maps with
    m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) -> Map[A,Z]) in
use User with Library, Database: env in
use Library with General: env in
use General with eq_str: proc(string,string -> bool);
                 lt_str: proc(string,string -> bool) in
use Database with Processed: env in
use IO with writeString: proc(string) in
use Database with Processed: env in
use IO with writeString: proc(string) in
begin
   if Processed contains base_images then
      writeString("The Processed environment already contains base_images, no
action taken.'n")
   else
      begin
         in Processed let base_images :=
                              m_empty[Image_id, Baseimage](eq_str, lt_str)
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                          [mk_derived_env.N] !
!                                                                         !
!      Set up Derived environment in Database environment                 !
!                                                                         !
!   PS() --|                                                             !
!          |-- User --|                                                  !
!                     |-- Database --|                                   !
!                                    |-- Derived                        !
!                                                                         !
!-------------------------------------------------------------------------!
```

```
use PS() with IO, User: env; environment: proc( -> env) in
use User with Database: env in
```

```
use IO with writeString: proc(string) in
begin
   if Database contains Derived then
      writeString("Database already contains Derived, no action taken.'n")
   else
      begin
         in Database let Derived := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                          [mk_index_env.N] !
!                                                                           !
!      Set up Index environment in Database environment                     !
!                                                                           !
!  PS() --|                                                                 !
!         |-- User --|                                                     !
!                    |-- Database --|                                      !
!                                   |-- Index                             !
!                                                                           !
!-------------------------------------------------------------------------!
use PS() with IO, User: env; environment: proc( -> env) in
use User with Database: env in
use IO with writeString: proc(string) in
begin
   if Database contains Index then
      writeString("Database already contains Index, no action taken.'n")
   else
      begin
         in Database let Index := environment()
      end
end
```

```
!-------------------------------------------------------------------------!
!                                                        [mk_entity_mbrs.N] !
!                                                                           !
!      Set up entity_mbrs in Index environment                              !
!                                                                           !
!  User --|                                                                 !
!         |-- Data --|                                                     !
!                    |-- Index -|                                          !
!                               |-- entity_mbrs: Map[Map_id, Entity_mbr]    !
!                                                                           !
!                                                                           !
!-------------------------------------------------------------------------!
use PS() with IO, User, GlasgowLibraries: env; environment: proc( -> env) in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps: env in
use Maps with
    m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) -> Map[A,Z]) in
use User with Database, Library: env in
use Library with General: env in
use General with  eq_str, lt_str:  proc(string,string -> bool) in
use Database with Index: env in
use IO with writeString: proc(string) in
begin
   if Index contains entity_mbrs then
      writeString("Index already contain entity_mbrs, no action taken.'n")
   else
      in Index let entity_mbrs := m_empty[Map_id,Entity_mbr](eq_str,lt_str)
end
```

```
!-------------------------------------------------------------------------!
!                                                      [mk_entity_indices.N] !
!                                                                           !
!      Set up entity_indices in Index environment                           !
```

```
!                                                                              !
!  User -|                                                                     !
!        |- Data -|                                                           !
!                 |- Index -|                                                 !
!                           |-- entity_indices: Map[Map_id, Entity_index]     !
!                                                                              !
!------------------------------------------------------------------------------!
use PS() with IO, User, GlasgowLibraries: env; environment: proc( -> env) in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps: env in
use Maps with
    m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) -> Map[A,Z]) in
use User with Database, Library: env in
use Library with General: env in
use General with  eq_str, lt_str:  proc(string,string -> bool) in
use Database with Index: env in
use IO with writeString: proc(string) in
begin
   if Index contains entity_indices then
      writeString("Index already contain entity_indices, no action taken.'n")
   else
      in Index let entity_indices :=
                              m_empty[Map_id, Entity_index](eq_str, lt_str)
end

!------------------------------------------------------------------------------!
!                                                        [mk_basemap_indices.N] !
!                                                                              !
!       Set up basemap_indices in Index environment                            !
!                                                                              !
!  User -|                                                                     !
!        |- Database -|                                                       !
!                     |- Index -|                                             !
!                               |-- basemap_indices: Map[Peano, Map_id]        !
!                                                                              !
!------------------------------------------------------------------------------!
type Map[A, Z] is MapRep1to1[A,Z]
use PS() with IO, User, GlasgowLibraries: env; environment: proc( -> env) in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps: env in
use Maps with
    m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) -> Map[A,Z]) in
use User with Library, Database: env in
use Library with General: env in
use General with eq_peano: proc(Peano,Peano -> bool);
                 lt_peano: proc(Peano,Peano -> bool) in
use Database with Index: env in
use IO with writeString: proc(string) in
begin
   if Index contains basemap_indices then
      writeString("Index already contains basemap_indices, no action
taken.'n")
   else
      begin
         in Index let basemap_indices :=
                                  m_empty[Peano, Map_id](eq_peano, lt_peano)
      end
end

!------------------------------------------------------------------------------!
!                                                      [mk_baseimage_indices.N] !
!                                                                              !
!      Set up baseimage_indices in Index environment                           !
!                                                                              !
!  Database --|                                                                !
!             |-- Index --|                                                    !
```

```
!                          |-- baseimage_indices: Map[Peano, List[Image_id]] !
!                                                                             !
!-----------------------------------------------------------------------------!
type Map[A, Z] is MapRep1to1[A,Z]
use PS() with IO, User, GlasgowLibraries: env; environment: proc( -> env) in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps: env in
use Maps with
    m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) -> Map[A,Z]) in
use User with Library, Database: env in
use Library with General: env in
use General with eq_peano: proc(Peano,Peano -> bool);
                 lt_peano: proc(Peano,Peano -> bool) in
use Database with Index: env in
use IO with writeString: proc(string) in
begin
   if Index contains baseimage_indices then
      writeString("Index already contains baseimage_indices, no action
taken.'n")
   else
      begin
         in Index let baseimage_indices :=
                          m_empty[Peano, List[Image_id]](eq_peano, lt_peano)
      end
end
```

APPENDIX C : THE GENERAL LIBRARY PROCEDURES

```
!-------------------------------------------------------------------------!
!                                                                         !
!                                                          [general_stubs.N]  !
!                                                                         !
!                                                                         !
!     This program sets up the variable stubs in the General environment  !
!                                                                         !
!                                                                         !
!                                                                         !
!-------------------------------------------------------------------------!

use PS() with IO, Time, GlasgowLibraries,
              User: env; environment: proc( -> env) in
use User with Library: env in
use Library with General:env in
use IO with writeString: proc(string) in
use Time with date: proc(-> string) in
use GlasgowLibraries with Miscellany: env in
use Miscellany with  uninitialised: proc[T](string -> T);
                     uninitialised_void: proc(string)  in
begin
   let date = date()
   if General contains stringToInt then
      writeString("'nGeneral already contains stringToInt, no changes
made.'n")
   else
      begin
      ! keep a record of date when the General library was last updated

      in General let changedOn := date

      in General let stringToInt := proc(s:string -> int)
                     uninitialised[int]("stringToInt")

      in General let stringToReal := proc(s:string -> real)
                     uninitialised[real]("stringToReal")

      in General let errorAbort := proc(s:string)
                     uninitialised_void("errorAbort")

      in General let waitSymbol := proc(count: int)
                     uninitialised_void("waitSymbol")

      in General let newline := proc(n:int)
                     uninitialised_void("newline")

      in General let space := proc(n:int)
                     uninitialised_void("space")

      in General let intToBits := proc(value, n: int -> *int)
                     uninitialised[*int]("intToBits")

      in General let bitsToInt := proc(bits: *int -> int)
                     uninitialised[int]("bitsToInt")

      in General let power_2_k := proc(k: int -> int)
                     uninitialised[int]("power_2_k")

      in General let vector_isu_sort := proc(v: *real)
                     uninitialised_void("vector_isu_sort")

      in General let eq_int := proc(a, b: int -> bool)
                     uninitialised[bool]("eq_int")

      in General let lt_int := proc(a, b: int -> bool)
                     uninitialised[bool]("lt_int")
```

```
    in General let eq_str := proc(a, b: string -> bool)
                  uninitialised[bool]("eq_str")

    in General let lt_str := proc(a, b: string -> bool)
                  uninitialised[bool]("lt_str")

    in General let eq_peano := proc(a, b: Peano -> bool)
                  uninitialised[bool]("eq_peano")

    in General let lt_peano := proc(a, b: Peano -> bool)
                  uninitialised[bool]("lt_peano")

    in General let eq_peanor := proc(a, b: Peanor -> bool)
                  uninitialised[bool]("eq_peanor")

    in General let lt_peanor := proc(a, b: Peanor -> bool)
                  uninitialised[bool]("lt_peanor")

    in General let xyToPK := proc(xy: XY -> int)
                  uninitialised[int]("xyToPK")

    in General let pkToXY := proc(pk: int -> XY)
                  uninitialised[XY]("pkToXY")

    in General let xyToPKR := proc(xy: XY -> real)
                  uninitialised[real]("xyToPKR")

    in General let pkrToXY := proc(pk: real -> XY)
                  uninitialised[XY]("pkrToXY")

    in General let getQuadExtent := proc(peano: Peanor -> Extent)
                  uninitialised[Extent]("getQuadExtent")


    writeString("'n'"General'" environment stubs set up on ")
    writeString(date)
    writeString("'n")
    end
end
```

```
!-------------------------------------------------------------------------!
!                                                                         !
!                                                          [generalLib.N] !
!                                                                         !
!        This program sets up the libraries in the General environment    !
!                                                                         !
!                                                                         !
!                                                                         !
!-------------------------------------------------------------------------!

use PS() with Arithmetical, String, IO, System, Vector, User:env  in
use Arithmetical with float: proc(int -> real);
                      truncate: proc(real -> int)  in
use String with stringToAscii: proc(string -> int);
                length: proc(string -> int)  in
use IO with writeString: proc(string)  in
use System with abort: proc()  in
use Vector with upb, lwb: proc[W](*W -> int) in
use User with Library:env  in
use Library with General:env  in
use General with
            stringToInt: proc(string -> int);
            stringToReal: proc(string -> real);
            errorAbort: proc(string);
            waitSymbol: proc(int);
            newline: proc(int);
            space: proc(int);
            intToBits: proc(int,int -> *int);
            bitsToInt: proc(*int -> int);
            power_2_k: proc(int -> int);
            vector_isu_sort: proc(*real);
            eq_int, lt_int: proc(int,int -> bool);
            eq_str, lt_str: proc(string,string -> bool);
            eq_peano, lt_peano: proc(Peano,Peano -> bool);
            eq_peanor, lt_peanor: proc(Peanor,Peanor -> bool);
            xyToPK: proc(XY -> int);
            pkToXY: proc(int -> XY);
            xyToPKR: proc(XY -> real);
            pkrToXY: proc(real -> XY);
            getQuadExtent: proc(Peanor -> Extent)   in
begin

!-------------------------------------------------------------------------!
!                                                                         !
!   convert a string to an integer                                        !
!                                                                         !
!-------------------------------------------------------------------------!
  stringToInt := proc(s:string -> int)
  begin
     let start := 1; let finish = length(s)
     let x := 0; let sign := 1
     while stringToAscii(s(start|1)) = 32 and start <= finish do
        start := start + 1
     if s(start|1) = "-" do
     begin
        sign := -1
        start := start + 1
     end
     for i = start to finish do
     x := 10 * x + stringToAscii(s(i|1))-48
     x := x * sign
     x
  end

!-------------------------------------------------------------------------!
!                                                                         !
```

```
!   convert a string to a real                                                  !
!                                                                               !
!-------------------------------------------------------------------------------!
  stringToReal := proc(s:string -> real)
  begin
     let start := 1 ; let finish = length(s)
     let x := 0.0; let sign := 1.0
     let s1 := 0; let s2 := 0; let divisor := 1.0
     while stringToAscii(s(start|1)) = 32 and start <= finish do
        start := start + 1
     if s(start|1) = "-" do
     begin
        sign := -1.0
        start := start + 1
     end
     while s(start|1) ~= "." do
     begin
        s1 := 10 * s1 + stringToAscii(s(start|1))-48
        start := start + 1
     end
     start := start + 1
     while start < finish do
     begin
        s2 := 10 * s2 + stringToAscii(s(start|1))-48
        start := start + 1
        divisor := divisor * 10.0
     end
     x := float(s1) + float(s2) / divisor
     x := x * sign
     x
  end

!-------------------------------------------------------------------------------!
!                                                                               !
!   write an error message and quit the program                                 !
!                                                                               !
!-------------------------------------------------------------------------------!
  errorAbort := proc(s: string)
  begin
     writeString("'n Error: " ++ s ++ "'n Aborting .....'n'n")
     abort()
  end

!-------------------------------------------------------------------------------!
!                                                                               !
!   generate a symbol that represents waiting for a job to be completed         !
!                                                                               !
!-------------------------------------------------------------------------------!
  waitSymbol := proc(count: int)
  begin
     case count rem 4 of
     0 : { writeString("'b'b- ") }
     1 : { writeString("'b'b\ ") }
     2 : { writeString("'b'b| ") }
     3 : { writeString("'b'b/ ") }
     default : { }
  end

!-------------------------------------------------------------------------------!
!                                                                               !
!   create a number of newlines                                                 !
!                                                                               !
!-------------------------------------------------------------------------------!
  newline := proc(n: int)
  begin
     for i = 1 to n do  writeString("'n")
```

```
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   create a number of spaces                                             !
!                                                                         !
!-------------------------------------------------------------------------!
  space := proc(n: int)
  begin
     for i = 1 to n do   writeString(" ");
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   convert an integer to an n-bit representation                         !
!                                                                         !
!-------------------------------------------------------------------------!
  intToBits := proc(value, n: int -> *int)
  begin
     let bits := vector 0 to n-1 of 0
     for i = 0 to n-1 do
     begin
        bits(i) := value rem 2
        value := value div 2
     end
     bits
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   convert an n-bit representation to an integer                         !
!                                                                         !
!-------------------------------------------------------------------------!
  bitsToInt := proc(bits: *int -> int)
  begin
     let value := bits(0)
     let k := 1
     let n = upb[int](bits) - lwb[int](bits)
     for i = 1 to n do
     begin
         k := k * 2
         value := value + k * bits(i)
     end
     value
  end

!-------------------------------------------------------------------------!
!                                                                         !
!     k                                                                   !
!   2   :  2 to the power of k                                            !
!                                                                         !
!-------------------------------------------------------------------------!
   power_2_k := proc(k: int -> int)
   begin
       let x := 1
       if k = 0 then x := 1 else
          begin
             for i = 1 to k do
                x := x * 2
          end
     x
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   sort a vector of real values                                          !
```

```
!                                                                              !
!------------------------------------------------------------------------------!
  vector_isu_sort := proc(v: *real)
  begin
     let l = lwb[real](v); let u = upb[real](v)
     for i = l to u do
     for i = l + 1 to u do
     begin
        if v(i-1) > v(i) do { let tmp = v(i-1); v(i-1) := v(i); v(i) := tmp }
     end
  end

!------------------------------------------------------------------------------!
!                                                                              !
!   Several predicates for creating Map bulk types                             !
!                                                                              !
!------------------------------------------------------------------------------!
  eq_int := proc(a,b:int -> bool); { if a = b then true else false }
  lt_int := proc(a,b:int -> bool); { if a < b then true else false }

  eq_str := proc(a,b:string -> bool); { if a = b then true else false }
  lt_str := proc(a,b:string -> bool); { if a < b then true else false }

  eq_peano := proc(a,b:Peano -> bool)
  begin
     if a(peano_key) = b(peano_key) and
        a(side_length) = b(side_length) then true else false
  end

  lt_peano := proc(a,b:Peano -> bool)
  begin
     if a(peano_key) < b(peano_key) then true
     else if a(peano_key) = b(peano_key) and
           a(side_length) < b(side_length) then true
     else false
  end

  eq_peanor := proc(a,b:Peanor -> bool)
  begin
     if a(peano_key) = b(peano_key) and
        a(side_length) = b(side_length) then true else false
  end

  lt_peanor := proc(a,b:Peanor -> bool)
  begin
     if a(peano_key) < b(peano_key) then true
     else if a(peano_key) = b(peano_key) and
           a(side_length) < b(side_length) then true
     else false
  end

!------------------------------------------------------------------------------!
!                                                                              !
!   Convert an xy-coordinate pair to a peano key of integer value              !
!   -  Each coordinate value ranges from 0. to 9999.                           !
!      i.e. a maximum of 4 digits for each coordinate                          !
!   -  The value of the peano key ranges from 0 to 1073741824 (2 ^ 30)         !
!                                                                              !
!------------------------------------------------------------------------------!
  xyToPK := proc(xy: XY -> int)
  begin
     let n = 15
     let x_bits = intToBits(truncate(xy(x)),n)
     let y_bits = intToBits(truncate(xy(y)),n)
     let xy_bits := vector 0 to 2*n-1 of 0
     ! perform bit interleaving
```

```
    for i = 0 to n-1 do
    begin
       xy_bits(2*i) := y_bits(i)
       xy_bits(2*i+1) := x_bits(i)
    end
    let pk = bitsToInt(xy_bits)
    pk
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Convert a peano key of integer value to an xy-coordinate pair         !
!   - The value of the peano key (pk) ranges from 0 to 1073741824  (2 ^ 30) !
!                                                                         !
!-------------------------------------------------------------------------!
  pkToXY := proc(pk: int -> XY)
  begin
    let n = 15
    let xy_bits = intToBits(pk,2*n)
    let x_bits := vector 0 to n-1 of 0
    let y_bits := vector 0 to n-1 of 0
    for i = 0 to n-1 do
    begin
       y_bits(i) := xy_bits(2*i)
       x_bits(i) := xy_bits(2*i+1)
    end
    let x = float(bitsToInt(x_bits))
    let y = float(bitsToInt(y_bits))
    let xy = XY(x,y)
    xy
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Convert an xy-coordinate pair to a peano key of real value            !
!   -  Each coordinate value ranges from 0. to 999999999.                 !
!      i.e. a maximum of 9 digits for each coordinate                     !
!   -  The value of the peano key ranges from 0. to  2. ^ 62              !
!                                                    (max. 19 digits).    !
!                                                                         !
!-------------------------------------------------------------------------!
  xyToPKR := proc(xy: XY -> real)
  begin
    let n = 31
    let x_bits := intToBits(truncate(xy(x)),n)
    let y_bits := intToBits(truncate(xy(y)),n)
    let xy_bits := vector 0 to 2*n-1 of 0
    ! perform bit interleaving
    for i = 0 to n-1 do
    begin
       xy_bits(2*i) := y_bits(i)
       xy_bits(2*i+1) := x_bits(i)
    end
    ! decompose to two positive integer values of 31-bit representation and
    ! form into a positive real value of 62-bit representaion
    for i = 0 to n-1 do
    begin
       y_bits(i) := xy_bits(i)
       x_bits(i) := xy_bits(i+n)
    end
    let pk = float(bitsToInt(x_bits)) * 2147483648. +
                                              float(bitsToInt(y_bits))
    pk
  end

!-------------------------------------------------------------------------!
```

```
!                                                                              !
!   convert a peano key of real value to an xy-coordinate pair                 !
!   - the value of the peano key (pk) ranges from 0. to 2. ^ 62                !
!                                                        (max. 19 digits)      !
!   -  each coordinate value ranges from 0. to 999999999.                      !
!      i.e. a maximum of 9 digits for each coordinate                          !
!                                                                              !
!------------------------------------------------------------------------------!
  pkrToXY := proc(pk: real -> XY)
  begin
     let n = 31
     ! convert a positive real value to two positive integers of 31-bit
     ! representation
     let pkl = truncate(pk / 2147483648.)      ! 2^31 = 2147483648
     let pkr = truncate(pk - float(pkl) * 2147483648.)
     let x_bits := intToBits(pkl,n)
     let y_bits := intToBits(pkr,n)
     let xy_bits := vector 0 to 2*n-1 of 0
     ! joint two parts to a value of 62-bit representaion
     for i = 0 to n-1 do
     begin
        xy_bits(i) := y_bits(i)
        xy_bits(i+n) := x_bits(i)
     end
     ! decompose and perform bit interleaving to two positive real values of
     ! 31-bit representation.
     for i = 0 to n-1 do
     begin
        y_bits(i) := xy_bits(2*i)
        x_bits(i) := xy_bits(2*i+1)
     end
     let x = float(bitsToInt(x_bits))
     let y = float(bitsToInt(y_bits))
     let xy = XY(x,y)
     xy
  end

!------------------------------------------------------------------------------!
!                                                                              !
!   determine the coverage extent of a quadrant from the peano key             !
!                                                                              !
!------------------------------------------------------------------------------!
  getQuadExtent := proc(peano: Peanor -> Extent)
  begin
     let range = peano(side_length)
     let xy = pkrToXY(peano(peano_key))
     let x = xy(x)
     let y = xy(y)
     let xmin = if (x / range) - float(truncate(x / range)) = 0.
                then float(truncate(x / range)) * range
                else float(truncate(x / range) + 1) * range
     let ymin = if (y / range) - float(truncate(y / range)) = 0.
                then float(truncate(y / range)) * range
                else float(truncate(y / range) + 1) * range
     let quad_extent := Extent(xmin, ymin, range, range)
     quad_extent
  end


end
```

APPENDIX D : THE GRAPHICAL LIBRARY PROCEDURES

```
!-----------------------------------------------------------------------!
!                                                                       !
!                                                    [graphical_stubs.N]!
!                                                                       !
!     This program sets up the variable stubs in Graphical environment  !
!                                                                       !
!                                                                       !
!-----------------------------------------------------------------------!

use PS() with IO, Time, GlasgowLibraries,
              User: env; environment: proc( -> env) in
use User with Library: env in
use Library with Graphical: env in
use IO with writeString: proc(string) in
use Time with date: proc(-> string) in
use GlasgowLibraries with Miscellany: env in
use Miscellany with uninitialised: proc[T](string -> T);
                    uninitialised_void: proc(string)  in
begin
   let date = date()
   if Graphical contains drawPoint then
     writeString("'nGraphical already contains drawPoint, no changes
made.'n")
   else
   begin
   ! keep a record of date when the Graphical library was last updated

   in Graphical let changedOn := date

   in Graphical let drawPoint := proc(point: XY; pt_col: pixel;
                                      window: image; draw_extent: Extent)
                    uninitialised_void("drawPoint")

   in Graphical let drawLineString := proc(ln: List[XY]; line_col: pixel;
                                           window: image;
                                           draw_extent: Extent)
                    uninitialised_void("drawLineString")

   in Graphical let drawText := proc(txt: string; txt_ht: real;
                                     txt_orient: real; txt_col: pixel;
                                     insert_pt: XY; window: image;
                                     draw_extent: Extent)
                    uninitialised_void("drawText")

   in Graphical let drawRectangle := proc(rectangle: MBR;
                                          rectangle_col: pixel;
                                          window: image; draw_extent: Extent)
                    uninitialised_void("drawRectangle")

   in Graphical let makeCircle := proc(cp: XY; r: real -> List[XY])
                    uninitialised[List[XY]]("makeCircle")

   in Graphical let pointInWindow := proc(test_pt: XY; win_mbr: MBR -> bool)
                    uninitialised[bool]("pointInWindow")

   in Graphical let lineVisibleInWindow := proc(p1, p2: XY;
                                                win_mbr: MBR -> bool)
                    uninitialised[bool]("lineVisibleInWindow")

   in Graphical let lineStrThroughWindow := proc(line_str: List[XY];
                                                 line_str_mbr: MBR;
                                                 target_pt: XY;
                                                 aperture: real -> bool)
                    uninitialised[bool]("lineStrThroughWindow")

   in Graphical let getPoint := proc(fd: file; window: image;
```

```
                                    win_size: Win_size; draw_extent: Extent;
                                    start: int -> XY)
                    uninitialised[XY]("getPoint")

in Graphical let xHairGetPoint := proc(fd: file; window: image;
                                    win_size: Win_size;
                                    draw_extent: Extent;
                                    start: int -> XY)
                    uninitialised[XY]("xHairGetPoint")


in Graphical let dynaGetWinCornersA := proc(fd: file; window: image;
                                    win_size: Win_size;
                                    draw_extent: Extent;
                                    start: int -> *XY)
                    uninitialised[*XY]("dynaGetWinCornersA")

in Graphical let dynaGetWinCornersB := proc(fd: file; window: image;
                                    win_size: Win_size;
                                    draw_extent: Extent;
                                    start: int -> *XY)
                    uninitialised[*XY]("dynaGetWinCornersB")

in Graphical let dynaGetCircle := proc(fd: file; window: image;
                                    win_size: Win_size;
                                    draw_extent: Extent;
                                    start: int -> Circle)
                    uninitialised[Circle]("dynaGetCircle")

in Graphical let getDragDxy := proc(fd: file; window: image;
                                win_size: Win_size;
                                start: int -> XY)
                    uninitialised[XY]("getDragDxy")

in Graphical let getZoomExtent := proc(zoom_opt: string; fd: file;
                                    window: image; win_size: Win_size;
                                    draw_extent: Extent;
                                    map_extent: Extent;
                                    start: int -> Extent)
                    uninitialised[Extent]("getZoomExtent")

in Graphical let getLineStrKeyPts := proc(xy_list: List[XY] -> List[XY])
                    uninitialised[List[XY]]("getLineStrKeyPts")

in Graphical let getLineStrMBR := proc(xy_list: List[XY] -> MBR)
                    uninitialised[MBR]("getLineStrMBR")


in Graphical let defaultPixel := proc(dp: pixel; depth: int -> pixel)
                    uninitialised[pixel]("defaultPixel")

in Graphical let colourToPixel := proc(c, depth: int -> pixel)
                    uninitialised[pixel]("colourToPixel")

in Graphical let pixelToColour := proc(p: pixel; depth: int -> int)
                    uninitialised[int]("pixelToColour")

in Graphical let rgb := proc(nc: int -> **int)
                    uninitialised[**int]("rgb")

in Graphical let grayLevel := proc(nc: int -> **int)
                    uninitialised[**int]("grayLevel")

in Graphical let invGrayLevel := proc(nc: int -> **int)
                    uninitialised[**int]("invGrayLevel")
```

```
   in Graphical let remap16 := proc(rv, gv, bv: int;
                                    pixel_table: *pixel -> pixel)
                    uninitialised[pixel]("reamp16")

   in Graphical let viewImage := proc(raster: image; shift_pt: XY;
                                      window: image; win_size: Win_size)
                    uninitialised_void("viewImage")


   in Graphical let popupMenu := proc(items: *string; actions: *proc();
                                      init: bool; fd: file; window: image;
                                      win_size: Win_size; start: int)
                    uninitialised_void("popupMenu")

   in Graphical let dialogueBox := proc(message, prompt: string; fd: file;
                                        window: image; win_size: Win_size;
                                        start: int -> string)
                    uninitialised[string]("dialogueBox")

   in Graphical let writeMessage := proc(message: string; fd: file;
                                         window: image; win_size: Win_size;
                                         start: int -> Transient_image)
                    uninitialised[Transient_image]("writeMessage")

   in Graphical let eraseMessage := proc(msg_img: Transient_image;
                                         window: image)
                    uninitialised_void("eraseMessage")


   writeString("'n'"Graphical'" environment stubs set up on ")
   writeString(date)
   writeString("'n")
   end
end
```

```
!-------------------------------------------------------------------------!
!                                                                         !
!                                                        [graphicalLib.N] !
!                                                                         !
!       This program sets up the libraries in Graphical environment       !
!                                                                         !
!                                                                         !
!-------------------------------------------------------------------------!

type Font is structure(constant characters: *image; constant fontHeight:int;
                       constant descender: int; constant info: string)
type FontPack is structure(font: Font; stringToTile,
                           charToTile: proc(string -> image))
type drawFunction is variant(imageDraw: proc(image,pic,real,real,real,real);
                             fileDraw: proc(file,pic,real,real,real,real);
                             fail: null)

use PS() with Arithmetical, String, IO, Vector, System, Format, Font,
              Graphical, Device, GlasgowLibraries, User: env  in
use Arithmetical with
                 abs: proc(int -> int);
                 rabs: proc(real -> real);
                 sqrt: proc(real -> real);
                 float: proc(int -> real);
                 truncate: proc(real -> int);
                 bitwiseOr: proc(int,int -> int)  in
use String with
           stringToAscii: proc(string -> int);
           asciiToString: proc(int -> string);
           letter, digit: proc(string -> bool);
           length: proc(string -> int)  in
use IO with
       PrimitiveIO: env;
       makeReadEnv: proc(file -> env);
       readLine: proc(-> string);
       readReal: proc(-> real);
       writeInt: proc(int);
       writeString: proc(string)  in
use PrimitiveIO with
                create: proc(string,int -> file);
                open: proc(string,int -> file);
                seek: proc(file,int,int -> int);
                close: proc(file -> int);
                readBytes: proc(file,*int,int,int -> int);
                writeBytes: proc(file,*int,int,int -> int);
                getByte: proc(int,int -> int);
                setByte: proc(int,int,int -> int);
                errorNumber: proc( -> int)  in
use Vector with
           lwb, upb: proc[t](*t -> int)  in
use System with
           abort: proc()  in
use Format with
           iformat: proc(int -> string);
           fformat: proc(real,int,int -> string)  in
use Font with screenR12,screenR14,screenB14,serifR16,gallantR19: FontPack in
use Graphical with
              Raster, Outline: env  in
use Outline with
            makeDrawFunction: proc(string -> drawFunction)  in
use Raster with
           getPixel: proc(image,int,int -> pixel);
           setPixel: proc(image,int,int,pixel);
           xDim: proc(image -> int);
           yDim: proc(image -> int);
           zDim: proc(image -> int);
```

```
            line: proc(image,int,int,int,int,pixel,int)  in
use Device with
            getScreen: proc(file -> image);
            colourMap: proc(file,pixel,int);
            locator: proc(file,*int);
            getCursor: proc(file -> image);
            getCursorInfo: proc(file,*int);
            setCursor: proc(file,image);
            colourOf: proc(file,pixel -> int)  in
use GlasgowLibraries with BulkTypes: env  in
use BulkTypes with Maps, Lists: env  in
use Maps with
          m_empty: proc[A,Z](proc(A,A -> bool), proc(A,A -> bool) ->
Map[A,Z]);
          m_contains: proc[A,Z](Map[A,Z],A -> bool);
          m_isu_assign: proc[A,Z](Map[A,Z],A,Z);
          m_isu_insert: proc[A,Z](Map[A,Z],A,Z);
          m_find: proc[A,Z](Map[A,Z],A -> Z)  in
use Lists with
           l_make: proc[T]( -> List[T]);
           l_first: proc[T](List[T] -> T);
           hd: proc[T](List[T] -> T);
           tl: proc[T](List[T] -> List[T]);
           l_length: proc[T](List[T] -> int );
           l_prepend: proc[T](T,List[T]->List[T]);
           l_map: proc[T,X](List[T], proc(T->X) -> List[X]);
           l_join: proc[T](List[T],List[T] -> List[T]);
           l_isu_join: proc[T](List[T],List[T] -> List[T]);
           l_reverse: proc[T](List[T] -> List[T]);
           l_contains: proc[T](List[T],T->bool);
           l_nth: proc[T](List[T],int -> T)  in
use User with
          Library: env  in
use Library with
             General,
             Graphical: env  in
use General with
             stringToInt: proc(string -> int);
             errorAbort: proc(string);
             newline: proc(int);
             space: proc(int);
             intToBits: proc(int, int -> *int);
             bitsToInt: proc(*int -> int);
             power_2_k: proc(int -> int);
             vector_isu_sort: proc(*real)  in
use Graphical with
               drawPoint: proc(XY, pixel, image, Extent);
               drawLineString: proc(List[XY], pixel, image, Extent);
               drawText: proc(string, real, real, pixel, XY, image, Extent);
               drawRectangle: proc(MBR, pixel, image, Extent);
               makeCircle: proc(XY, real -> List[XY]);
               pointInWindow: proc(XY, MBR -> bool);
               lineVisibleInWindow: proc(XY, XY, MBR -> bool);
               lineStrThroughWindow: proc(List[XY], MBR, XY, real -> bool);
               getPoint: proc(file, image, Win_size, Extent, int -> XY);
               xHairGetPoint: proc(file, image, Win_size, Extent, int -> XY);
               dynaGetWinCornersA: proc(file, image, Win_size, Extent,
                                                              int -> *XY);
               dynaGetWinCornersB: proc(file, image, Win_size, Extent,
                                                              int -> *XY);
               dynaGetCircle: proc(file, image, Win_size, Extent,
                                                          int -> Circle);
               getDragDxy: proc(file, image, Win_size, int -> XY);
               getZoomExtent: proc(string, file, image, Win_size, Extent,
                                                  Extent, int -> Extent);
               getLineStrKeyPts: proc(List[XY] -> List[XY]);
```

```
                getLineStrMBR: proc(List[XY] -> MBR);
                defaultPixel: proc(pixel, int -> pixel);
                colourToPixel: proc(int, int -> pixel);
                pixelToColour: proc(pixel, int -> int);
                rgb: proc(int -> **int);
                grayLevel: proc(int -> **int);
                invGrayLevel: proc(int -> **int);
                remap16: proc(int, int, int, *pixel -> pixel);
                viewImage: proc(image, XY, image, Win_size);
                popupMenu: proc(*string, *proc(), bool, file, image,
                                                        Win_size, int);
                dialogueBox: proc(string, string, file, image, Win_size,int
                                                            -> string);
                writeMessage: proc(string, file, image, Win_size, int
                                                    -> Transient_image);
                eraseMessage: proc(Transient_image, image)   in
begin

!-------------------------------------------------------------------------!
!                                                                         !
!   initialize required variables                                         !
!                                                                         !
!-------------------------------------------------------------------------!

  let draw = makeDrawFunction("image")'imageDraw

!-------------------------------------------------------------------------!
!                                                                         !
!   draw a point                                                          !
!                                                                         !
!-------------------------------------------------------------------------!
  drawPoint := proc(point: XY; pt_col: pixel; window: image;
                                              draw_extent: Extent)
  begin
     let x_min = draw_extent(x_min)
     let y_min = draw_extent(y_min)
     let x_max = x_min + draw_extent(x_range)
     let y_max = y_min + draw_extent(y_range)
     let pt = colour [point(x), point(y)] in pt_col
     draw(window, pt , x_min, x_max, y_min, y_max)
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   draw a line string                                                    !
!                                                                         !
!-------------------------------------------------------------------------!
  drawLineString := proc(ln: List[XY]; line_col: pixel; window: image;
                                                    draw_extent: Extent)
  begin
     let x_min = draw_extent(x_min)
     let y_min = draw_extent(y_min)
     let x_max = x_min + draw_extent(x_range)
     let y_max = y_min + draw_extent(y_range)
     let line_str := [l_first[XY](ln)(x),l_first[XY](ln)(y)]
     ln := tl[XY](ln)
     while ln isnt empty do
     begin
        let xy = hd[XY](ln)
        line_str := line_str ^ [xy(x),xy(y)]
        ln := tl[XY](ln)
     end
     line_str := colour line_str in line_col
     draw(window, line_str , x_min, x_max, y_min, y_max)
  end
```

```
!-------------------------------------------------------------------------!
!                                                                         !
!   draw a text                                                           !
!                                                                         !
!-------------------------------------------------------------------------!
  drawText := proc(txt: string; txt_ht: real; txt_orient: real;
                   txt_col: pixel; insert_pt: XY; window: image;
                   draw_extent: Extent)
  begin
     let width = xDim(window)
     let x_min = draw_extent(x_min)
     let y_min = draw_extent(y_min)
     let x_range = draw_extent(x_range)
     let x_max = x_min + x_range
     let y_max = y_min + draw_extent(y_range)
     let nc = length(txt)
     let result := colour text txt from insert_pt(x), insert_pt(y) to
                   insert_pt(x) + txt_ht * float(nc) ,insert_pt(y) in txt_col
     if txt_orient ~= 0.0 do
     begin
        result := shift rotate shift result by -insert_pt(x),-insert_pt(y) by
                  -txt_orient by insert_pt(x),insert_pt(y)
     end
     draw(window, result , x_min, x_max, y_min, y_max)
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   draw a rectangle                                                      !
!                                                                         !
!-------------------------------------------------------------------------!
  drawRectangle := proc(rectangle: MBR; rectangle_col: pixel;
                                        window: image; draw_extent: Extent)
  begin
     let x_min = draw_extent(x_min)
     let y_min = draw_extent(y_min)
     let x_max = x_min + draw_extent(x_range)
     let y_max = y_min + draw_extent(y_range)
     let result = colour [rectangle(x_min),rectangle(y_min)] ^
                         [rectangle(x_min),rectangle(y_max)] ^
                         [rectangle(x_max),rectangle(y_max)] ^
                         [rectangle(x_max),rectangle(y_min)] ^
                         [rectangle(x_min),rectangle(y_min)] in rectangle_col
     draw(window, result, x_min, x_max, y_min, y_max)
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   make a circle                                                         !
!                                                                         !
!-------------------------------------------------------------------------!
  makeCircle := proc(cp: XY; r: real -> List[XY])
  begin
     let xc = cp(x); let yc = cp(y)
     let circle := l_make[XY]()
  !  let n = truncate(pi / 2. * (r / pxl_resol) )  !
     let n = 10
     if n >= 1 do
     begin
        let symToYaxis = proc(pt: XY -> XY)
        begin
           let sym_pt = XY(xc - pt(x) + xc, pt(y))
           sym_pt
        end
        let symToXaxis = proc(pt: XY -> XY)
        begin
```

```
            let sym_pt = XY(pt(x), yc - pt(y) + yc)
            sym_pt
         end
         let dt := 1. / float(n)
         let t := dt / 2.
         while t < 1.0 do
         begin
            let d = 1. + t * t
            let x = xc + r * (2. - d) / d
            let y = yc + r * 2. * t / d
            circle := l_prepend[XY](XY(x,y),circle)
            t := t + dt
         end
         let quad_circle = l_map[XY,XY](circle,symToYaxis)
         circle := l_isu_join[XY](l_reverse[XY](circle),quad_circle)
         let half_circle = l_map[XY,XY](circle,symToXaxis)
         circle := l_isu_join[XY](circle,l_reverse[XY](half_circle))
      end
      circle
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   point in a window test                                                !
!                                                                         !
!-------------------------------------------------------------------------!
   pointInWindow := proc(test_pt: XY; win_mbr: MBR -> bool)
   begin
      let x = test_pt(x); let y = test_pt(y)
      let in_win = if x >= win_mbr(x_min) and x <= win_mbr(x_max) and
                      y >= win_mbr(y_min) and y <= win_mbr(y_max)
                   then true else false
      in_win
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   test whether a line segment is either completely visible or only      !
!   partially visible in a window                                         !
!                                                                         !
!-------------------------------------------------------------------------!
   lineVisibleInWindow := proc(p1, p2: XY; win_mbr: MBR -> bool)
   begin
      let x1 = p1(x); let y1 = p1(y)
      let x2 = p2(x); let y2 = p2(y)
      let XL = win_mbr(x_min)
      let XR = win_mbr(x_max)
      let YB = win_mbr(y_min)
      let YT = win_mbr(y_max)
      let visible := false
      if x1 < XL or x1 > XR or x2 < XL or x2 > XR or
         y1 < YB or y1 > YT or y2 < YB or y2 > YT then
      begin
         ! the line is not totally visible
         if (x1 < XL and x2 < XL) or (x1 > XR and x2 > XR) or
            (y1 > YT and y2 > YT) or (y1 < YB and y2 < YB) then
            { visible := false }  ! the line is invisible
         else
            ! the line is partially visible or diagonally crosses the corner
            begin
               ! determine the intersections
               if x2 - x1 = 0. then          ! the line is vertical
               begin
                  if x1 <= XR and x1 >= XL then
                  begin
                     if (y1 >= YT and y2 >= YT) or
```

```
                                                    (y1 <= YB and y2 <= YB) then
                        { visible := false }
                     else { visible := true }
                  end
                  else
                     { visible := false }
               end
               else if y2 - y1 = 0. then     ! the line is horizontal
               begin
                  if y1 <= YT and y1 >= YB then
                  begin
                     if (x1 <= XL and x2 <= XL) or
                                                    (x1 >= XR and x2 >= XR) then
                        { visible := false }
                     else { visible := true }
                  end
                  else
                      { visible := false }
               end
               else
               begin
                  let m = (y2 - y1) / (x2 - x1)
                  let int_y_left = m * (XL - x1) + y1
                  let int_y_right = m * (XR - x1) + y1
                  let int_x_top = x1 + (YT - y1) / m
                  let int_x_bottom = x1 + (YB - y1) / m
                  if (int_y_left <= YT and int_y_left >= YB) or
                     (int_y_right <= YT and int_y_right >= YB) or
                     (int_x_top <= XR and int_x_top >= XL) or
                     (int_x_bottom <= XR and int_x_bottom >= XL)  then
                     { visible := true }
                  else
                     { visible := false }
               end
            end
      end
      else
         { visible := true }  ! the line is visible
      visible
   end

!-------------------------------------------------------------------------!
!                                                                         !
!  test whether a line string passes through a window                     !
!                                                                         !
!-------------------------------------------------------------------------!
   lineStrThroughWindow := proc(line_str: List[XY]; line_str_mbr: MBR;
                                target_pt: XY; aperture: real -> bool)
   begin
      let found := false
      let hf_size = aperture / 2.
      let x = target_pt(x); let y = target_pt(y)
      let target_win = MBR(x - hf_size, y - hf_size,
                                                    x + hf_size, y + hf_size)
      if pointInWindow(target_pt,line_str_mbr) do
      begin
         let pt1 := hd[XY](line_str)
         line_str := tl[XY](line_str)
         while line_str isnt empty and ~found do
         begin
            let pt2 = hd[XY](line_str)
            found := lineVisibleInWindow(pt1,pt2,target_win)
            pt1 := pt2
            line_str := tl[XY](line_str)
         end
      end
```

```
    found
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   locate a point in the display window and return a pair of coordinates !
!                                                                         !
!-------------------------------------------------------------------------!
  getPoint := proc(fd: file; window: image; win_size: Win_size;
                   draw_extent: Extent; start: int -> XY)
  begin
     let x_min = draw_extent(x_min)
     let y_min = draw_extent(y_min)
     let x_range = draw_extent(x_range)
     let y_range = draw_extent(y_range)
     let x_max = x_min + x_range
     let y_max = y_min + y_range
     let data := vector 1 to 7 of 0
     let x := 0. ; let y := 0.
     let xy := XY(0.,0.)
     let last_x := 0. ; let last_y := 0.
     ! create a new cursor image
     let new_cursor := image 16 by 16 of off
     setCursor(fd,new_cursor)
     let flag := 0
     let pxl_size = x_range / float(win_size(width))
     ! cross size = 10 pixels
     let hfsize = 5.0 * pxl_size
     let old_cross := nilpic
     while flag ~= 1 do
     begin
        locator(fd,data)
        x := x_min + float(data(3)) * pxl_size
        y := y_min + float(data(4)) * pxl_size
        if last_x ~= x and last_y ~= y do  ! prevent crosshair from flashing
        begin
           let new_cross = colour [x - hfsize, y] ^ [x + hfsize, y] ++
                                  [x, y - hfsize] ^ [x, y + hfsize] in on
           draw(window(start|1),new_cross ++ old_cross,
                                              x_min, x_max, y_min, y_max)
           old_cross := colour new_cross in off
           last_x := x; last_y := y
        end
        if data(5) = 1 do { flag := 1; xy := XY(x,y) }
        if data(6) = 1 do
        begin
           flag := 1
           draw(window(start|1), old_cross, x_min, x_max, y_min, y_max)
        end
     end
     line(new_cursor, 0,15,8,0, on, 12)
     line(new_cursor, 0,15,0,9, on, 12)
     line(new_cursor, 0,15,5,12, on, 12)
     setCursor(fd,new_cursor)
     xy
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   locate a point in the display window with a cross-hair cursor and     !
!   return a pair of coordinates                                          !
!                                                                         !
!-------------------------------------------------------------------------!
  xHairGetPoint := proc(fd: file; window: image; win_size: Win_size;
                        draw_extent: Extent; start: int -> XY)
  begin
```

```
    let x_min = draw_extent(x_min)
    let y_min = draw_extent(y_min)
    let x_range = draw_extent(x_range)
    let y_range = draw_extent(y_range)
    let x_max = x_min + x_range
    let y_max = y_min + y_range
    let pxl_size = x_range / float(win_size(width))
    let data := vector 1 to 7 of 0
    let x := 0. ; let y := 0.
    let xy := XY(0.,0.)
    let last_x := 0. ; let last_y := 0.
    ! create a new cursor image
    let new_cursor := image 16 by 16 of off
    setCursor(fd, new_cursor)

    let flag := 0
    let old_cross := nilpic
    while  flag ~= 1 do
    begin
       locator(fd,data)
       x := x_min + float(data(3)) * pxl_size
       y := y_min + float(data(4)) * pxl_size
       if last_x ~= x and last_y ~= y do  ! prevent crosshair from flashing
       begin
          let new_cross = colour [x_min, y] ^ [x_max, y] ++
                                              [x, y_min] ^ [x, y_max] in on
          draw(window(start|1),new_cross ++ old_cross,
                                                x_min, x_max, y_min, y_max)
          old_cross := colour new_cross in off
          last_x := x; last_y := y
       end
       if data(5) = 1 do { flag := 1 ; xy := XY(x,y)}
       if data(6) = 1 do
       begin
          flag := 1
          draw(window(start|1), old_cross, x_min, x_max, y_min, y_max)
       end
    end
    line(new_cursor, 0,15,8,0, on, 12)
    line(new_cursor, 0,15,0,9, on, 12)
    line(new_cursor, 0,15,5,12, on, 12)
    setCursor(fd,new_cursor)
    xy
end

!-------------------------------------------------------------------------!
!                                                                         !
!   determine the min-max coordinates of a viewing window defined by      !
!   dynamically moving a mouse-controlled cursor                          !
!   - press and hold the mouse button 1 at the first point, drag the      !
!     cursor to the second point and release it.                          !
!                                                                         !
!-------------------------------------------------------------------------!
dynaGetWinCornersA := proc(fd: file; window: image; win_size: Win_size;
                           draw_extent: Extent; start: int -> *XY)
begin
    let x_min = draw_extent(x_min)
    let y_min = draw_extent(y_min)
    let x_range = draw_extent(x_range)
    let y_range = draw_extent(y_range)
    let x_max = x_min + x_range
    let y_max = y_min + y_range
    let pxl_size = x_range / float(win_size(width))
    let data := vector 1 to 7 of 0
    let xy := vector 1 to 2 of XY(0.,0.)
    let x1 := 0.; let y1 := 0.; let x2 := 0.; let y2 := 0.
```

```
    let last_x := data(3); let last_y := data(4)
    let failed := false
    let flag := 0
    while flag ~= 1 do
    begin
       locator(fd, data)
       x1 := x_min + float(data(3)) * pxl_size
       y1 := y_min + float(data(4)) * pxl_size
       if data(5) = 1 do { flag := 1 }
       if data(6) = 1 do { flag := 1 ; failed := true }
    end

    let old_box := nilpic
    if ~failed do
    begin
       while flag ~= 2 do
       begin
          locator(fd,data)
          if data(3) ~= last_x and data(4) ~= last_y do
          begin
             x2 := x_min + float(data(3)) * pxl_size
             y2 := y_min + float(data(4)) * pxl_size
             let new_box = colour [x1, y1] ^ [x1, y2] ^ [x2, y2] ^
                                  [x2, y1] ^ [x1, y1] in on
             let result = old_box ++ new_box
             draw(window(start|1), result, x_min, x_max, y_min, y_max)
             old_box := colour new_box in off
             last_x := data(3); last_y := data(4)
          end
          if data(5) = 0 do flag := 2
       end
    end

    if ~failed do
    begin
       let wx_min = if x1 < x2 then x1 else x2
       let wy_min = if y1 < y2 then y1 else y2
       let wx_max = if x2 > x1 then x2 else x1
       let wy_max = if y2 > y1 then y2 else y1
       xy(1) := XY(wx_min,wy_min)
       xy(2) := XY(wx_max,wy_max)
    end
    xy
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   determine the min-max coordinates of a viewing window defined by      !
!   dynamically moving a mouse-controlled cursor                          !
!   - click the mouse button 1 at the first point, move the cursor to the !
!     second point and click the mouse button 1 again.                    !
!                                                                         !
!-------------------------------------------------------------------------!
  dynaGetWinCornersB := proc(fd: file; window: image; win_size: Win_size;
                             draw_extent: Extent; start: int -> *XY)
  begin
     let x_min = draw_extent(x_min)
     let y_min = draw_extent(y_min)
     let x_range = draw_extent(x_range)
     let y_range = draw_extent(y_range)
     let x_max = x_min + x_range
     let y_max = y_min + y_range
     let pxl_size = x_range / float(win_size(width))
     let data := vector 1 to 7 of 0
     let xy := vector 1 to 2 of XY(0.,0.)
     let x1 := 0.; let y1 := 0.; let x2 := 0.; let y2 := 0.
```

```
    let failed := false
    let flag := 0
    while flag ~= 1 do
    begin
       locator(fd,data)
       x1 := x_min + float(data(3)) * pxl_size
       y1 := y_min + float(data(4)) * pxl_size
       if data(5) = 1 do { flag := 1 }
       if data(6) = 1 do { flag := 1 ; failed := true }
    end

    let old_box := nilpic
    if ~failed do
    begin
       while flag ~= 2 do
       begin
          locator(fd,data)
          x2 := x_min + float(data(3)) * pxl_size
          y2 := y_min + float(data(4)) * pxl_size
          if x2 = x1 and y2 = y1 then { }
          else
          begin
             let new_box = colour [x1,y1] ^ [x1,y2] ^ [x2,y2] ^
                                              [x2,y1] ^ [x1,y1] in on
             let result = old_box ++ new_box
             draw(window(start|1), result, x_min, x_max, y_min, y_max)
             old_box := colour new_box in off
             if data(5) = 1 do flag := 2
             if data(6) = 1 do { flag := 2 ; failed := true }
          end
       end
    end

    if ~failed do
    begin
       let wx_min = if x1 < x2 then x1 else x2
       let wy_min = if y1 < y2 then y1 else y2
       let wx_max = if x2 > x1 then x2 else x1
       let wy_max = if y2 > y1 then y2 else y1
       xy(1) := XY(wx_min,wy_min)
       xy(2) := XY(wx_max,wy_max)
    end
    xy
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   dynamically get a circle                                              !
!                                                                         !
!-------------------------------------------------------------------------!
  dynaGetCircle := proc(fd: file; window: image; win_size: Win_size;
                        draw_extent: Extent; start: int -> Circle)
  begin
     !   convert a line string to a picture
     let lineStrToPic := proc(l: List[XY] -> pic)
     begin
        let first_pt := hd[XY](l)
        let result := [first_pt(x), first_pt(y)]
        while l isnt empty do
        begin
           let pt = hd[XY](l)
           result := result ^ [pt(x), pt(y)]
           l := tl[XY](l)
        end
        result := result ^ [first_pt(x), first_pt(y)]
```

```
   result
end
let x_min = draw_extent(x_min)
let y_min = draw_extent(y_min)
let x_range = draw_extent(x_range)
let y_range = draw_extent(y_range)
let x_max = x_min + x_range
let y_max = y_min + y_range
let new_cursor := image 16 by 16 of off
setCursor(fd,new_cursor)
let pxl_size = x_range / float(win_size(width))
! cross size = 10 pixels
let hfsize = 5.0 * pxl_size
let data := vector 1 to 7 of 0
let xc := 0.; let yc := 0.; let r := 0.
let x := 0.; let y := 0.
let last_x := data(3); let last_y := data(4)
let old_cross := nilpic
let failed := false
let flag := 0
while  flag ~= 1 do
begin
   locator(fd,data)
   if data(3) ~= last_x and data(4) ~= last_y do
   begin
      xc := x_min + float(data(3)) * pxl_size
      yc := y_min + float(data(4)) * pxl_size
      let new_cross = colour [xc - hfsize, yc] ^ [xc + hfsize, yc] ++
                           [xc, yc - hfsize] ^ [xc, yc + hfsize] in on
      draw(window(start|1), new_cross ++ old_cross,
                                          x_min, x_max, y_min, y_max)
      old_cross := colour new_cross in off
      last_x := data(3); last_y := data(4)
   end
   if data(5) = 1 do { flag := 1 }
   if data(6) = 1 do
   begin
      flag := 1
      failed := true
      xc := 0.; yc := 0.
      draw(window(start|1), old_cross, x_min, x_max, y_min, y_max)
   end
end
let cp = XY(xc,yc)

let old_circle := nilpic
if ~failed do
begin
   while flag ~= 2 do
   begin
      locator(fd,data)
      if data(3) ~= last_x and data(4) ~= last_y do
      begin
         x := x_min + float(data(3)) * pxl_size
         y := y_min + float(data(4)) * pxl_size
         r := sqrt((x - xc) * (x - xc) + (y - yc) * (y - yc))
         let circle = makeCircle(cp, r)
         let new_circle = colour lineStrToPic(circle) in on
         let result = old_circle ++ new_circle
         draw(window(start|1), result, x_min, x_max, y_min, y_max)
         old_circle := colour new_circle in off
         last_x := data(3); last_y := data(4)
      end
      if data(5) = 0 do flag := 2
   end
end
```

```
    line(new_cursor, 0,15,8,0, on, 12)
    line(new_cursor, 0,15,0,9, on, 12)
    line(new_cursor, 0,15,5,12, on, 12)
    setCursor(fd,new_cursor)
    let cir_param = Circle(cp,r)
    cir_param
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   get the shift amount by dragging a rubber line                        !
!                                                                         !
!-------------------------------------------------------------------------!
  getDragDxy := proc(fd: file; window: image; win_size: Win_size;
                     start: int -> XY)
  begin
    let x_max = float(win_size(width))
    let y_max = float(win_size(height))
    let data := vector 1 to 7 of 0
    let cursor_data := vector 1 to 3 of 0
    let x1 := 0.; let y1 := 0.; let x2 := 0.; let y2 := 0.
    let last_x := 0. ; let last_y := 0.
    let flag := 0
    let failed := false
    let dxy := XY(0.,0.)
    let new_cursor := image 16 by 16 of off
    setCursor(fd,new_cursor)     ! disable the default cursor
    let hfsize = 5.0   ! cross size = 10 pixels
    let old_cross1 := nilpic
    while flag ~= 1 do
    begin
      locator(fd, data)
      x1 := float(data(3))
      y1 := float(data(4))
      if last_x ~= x1 and last_y ~= y1 do
      begin
        let new_cross1 = colour [x1 - hfsize, y1] ^ [x1 + hfsize, y1] ++
                         [x1, y1 - hfsize] ^ [x1, y1 + hfsize] in on
        draw(window(start|1), new_cross1 ++ old_cross1,
                                          0., x_max, 0., y_max)
        old_cross1 := colour new_cross1 in off
        last_x := x1; last_y := y1
      end
      if data(5) = 1 do { flag := 1 }
      if data(6) = 1 do { flag := 1; failed := true }
    end

    old_cross1 := colour old_cross1 in on
    let old_cross2 := nilpic
    let old_line := nilpic
    if ~failed do
    begin
      while  flag ~= 2 do
      begin
        locator(fd,data)
        x2 := float(data(3))
        y2 := float(data(4))
        if last_x ~= x2 and last_y ~= y2 do
        begin
          if rabs(x2 - x1) > hfsize and rabs(y2 - y1) > hfsize do
          begin
            let new_cross2 = colour [x2 - hfsize, y2] ^
                                    [x2 + hfsize, y2]++
                                    [x2, y2 - hfsize] ^
                                    [x2, y2 + hfsize] in on
            draw(window(start|1), new_cross2 ++ old_cross2,
```

```
                                                    0., x_max, 0., y_max)
                  old_cross2 := colour new_cross2 in off
               end
               let new_line = colour [x1,y1]^[x2,y2] in on
               draw(window(start|1), old_cross1 ++ old_line ++ new_line,
                                                    0., x_max, 0., y_max)
               old_line := colour new_line in off
               last_x := x2; last_y := y2
            end
            if data(5) = 0 do { flag := 2 }
         end
         dxy := XY(x2 - x1 ,y2 - y1)
      end
      old_cross1 := colour old_cross1 in off
      draw(window(start|1), old_cross1 ++ old_line ++ old_cross2,
                                                    0., x_max, 0., y_max)
      dxy
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   determine the drawing extent for various zooming options              !
!                                                                         !
!-------------------------------------------------------------------------!
   getZoomExtent := proc(zoom_opt: string; fd: file; window: image;
                         win_size: Win_size; draw_extent, map_extent: Extent;
                         start: int -> Extent)
   begin
      let x_min := draw_extent(x_min)
      let y_min := draw_extent(y_min)
      let x_range := draw_extent(x_range)
      let y_range := draw_extent(y_range)
      let x_max := x_min + x_range
      let y_max := y_min + y_range
      let hw_ratio = float(win_size(height)) / float(win_size(width))
      case zoom_opt of
      "A","a" : begin
                   writeString("Fitting the map to the entire window'n")
                   let map_xrange := map_extent(x_range)
                   let map_yrange := map_extent(y_range)
                   if map_xrange * hw_ratio <= map_yrange then
                   begin
                      x_range := map_yrange / hw_ratio
                      y_range := map_yrange
                   end
                   else
                   begin
                      x_range := map_xrange
                      y_range := map_xrange * hw_ratio
                   end
                   x_min := map_extent(x_min)
                   y_min := map_extent(y_min)
                   x_max := x_min + x_range
                   y_max := y_min + y_range
                end
      "C","c" : begin
                   writeString("Digitise the centre point of an intended zoom-
in/out area.'n")

                   let cp = getPoint(fd,window,win_size,draw_extent,start)
                   if cp(x) = 0. and cp(y) = 0. then { } else
                   begin
                      writeString("'nEnter a magnification/reduction factor:
");let mag_fac = readReal()
                      let trash = readLine()
                      if mag_fac ~= 0. do
```

```
                    begin
                        x_min := cp(x) - x_range / 2./ mag_fac
                        x_max := cp(x) + x_range / 2./ mag_fac
                        y_min := cp(y) - y_range / 2./ mag_fac
                        y_max := cp(y) + y_range / 2./ mag_fac
                    end
                end
            end
    "P","p" : begin
                writeString("Draw a line to indicate the direction and the
distance of panning. 'n")
                let dxy = getDragDxy(fd,window,win_size,start)
                if dxy(x) = 0. and dxy(y) = 0. then { } else
                begin
                    x_min := x_min - dxy(x) * x_range /
                                                    float(win_size(width))
                    y_min := y_min - dxy(y) * y_range /
                                                    float(win_size(height))
                    x_max := x_min + x_range
                    y_max := y_min + y_range
                end
                let new_cursor := image 16 by 16 of off
                line(new_cursor, 0,15,8,0, on, 12)
                line(new_cursor, 0,15,0,9, on, 12)
                line(new_cursor, 0,15,5,12, on, 12)
                setCursor(fd,new_cursor)
            end
    "X","x" : begin
                writeString("Enter a magnification/reduction factor: ");let
mag_fac = readReal()
                let trash = readLine()
                if mag_fac ~= 0. do

                begin
                    let map_xrange := map_extent(x_range)
                    let map_yrange := map_extent(y_range)
                    if map_xrange * hw_ratio <= map_yrange then
                    begin
                        x_range := map_yrange / hw_ratio
                        y_range := map_yrange
                    end
                    else
                    begin
                        x_range := map_xrange
                        y_range := map_xrange * hw_ratio
                    end
                    let x_cent = map_extent(x_min) + map_xrange / 2.
                    let y_cent = map_extent(y_min) + map_yrange / 2.
                    x_min := x_cent - x_range / 2./ mag_fac
                    x_max := x_cent + x_range / 2./ mag_fac
                    y_min := y_cent - y_range / 2./ mag_fac
                    y_max := y_cent + y_range / 2./ mag_fac
                end
            end
    "W","w" : begin
                writeString("Digitise the diagonal corners of the intended
zoom-in area.'n")
                let cp =
dynaGetWinCornersA(fd,window,win_size,draw_extent,start)

                if cp(1)(x) = 0. and cp(1)(y) = 0. and
                    cp(2)(x) = 0. and cp(2)(y) = 0. then { } else
                begin
                    let w_xrange = cp(2)(x) - cp(1)(x)
                    let w_yrange = cp(2)(y) - cp(1)(y)
                    if w_xrange * hw_ratio <= w_yrange then
```

```
                    begin
                       x_range := w_yrange / hw_ratio
                       y_range := w_yrange
                    end
                    else
                    begin
                       x_range := w_xrange
                       y_range := w_xrange * hw_ratio
                    end
                    x_min := cp(1)(x)
                    y_min := cp(1)(y)
                    x_max := x_min + x_range
                    y_max := y_min + y_range
                 end
              end
     "Q","q" : {}
     default : {}
     draw_extent := Extent(x_min, y_min, x_max - x_min, y_max - y_min)
     draw_extent
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Construct a list of critical points that breaks a line string into    !
!   several components of monotonic lines                                 !
!                                                                         !
!-------------------------------------------------------------------------!
  getLineStrKeyPts := proc(xy_list: List[XY] -> List[XY])
  begin
     let old_dir_code := 0
     let key_pts_list := l_make[XY]()
     let pt1 := hd[XY](xy_list)
     xy_list := tl[XY](xy_list)
     while xy_list isnt empty do
     begin
        let pt2 = hd[XY](xy_list)
        let dx = pt2(x) - pt1(x)
        let dy = pt2(y) - pt1(y)
        let new_dir_code = if dx >= 0. and dy > 0. then   1
                           else if dx > 0. and dy <= 0. then  2
                           else if dx <= 0. and dy < 0. then  3
                           else if dx < 0. and dy >= 0. then  4
                           else  0
        if new_dir_code > 0 do
        begin
           if new_dir_code ~= old_dir_code  do
           begin
              key_pts_list := l_prepend[XY](pt1,key_pts_list)
              old_dir_code := new_dir_code
           end
           pt1 := pt2
        end
        xy_list := tl[XY](xy_list)
     end
     key_pts_list := l_prepend[XY](pt1,key_pts_list)
     key_pts_list := l_reverse[XY](key_pts_list)
     key_pts_list
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Determine the MBR of a linestring                                     !
!                                                                         !
!-------------------------------------------------------------------------!
  getLineStrMBR := proc(xy_list: List[XY] -> MBR)
  begin
```

```
    let xy := hd[XY](xy_list)
    let x_min := xy(x); let y_min := xy(y)
    let x_max := x_min; let y_max := y_min
    xy_list := tl[XY](xy_list)
    while xy_list isnt empty do
    begin
       xy := hd[XY](xy_list)
       let x = xy(x); let y = xy(y)
       if x < x_min do x_min := x
       if y < y_min do y_min := y
       if x > x_max do x_max := x
       if y > y_max do y_max := y
       xy_list := tl[XY](xy_list)
    end
    let mbr = MBR(x_min, y_min, x_max, y_max)
    mbr
  end

!-------------------------------------------------------------------------!
!                                                                         !
!    set default pixel (on or off) for a specified depth                  !
!                                                                         !
!-------------------------------------------------------------------------!
   defaultPixel := proc(dp: pixel; depth: int -> pixel)
       begin
          let pix := dp
          for i = 1 to depth - 1 do  { pix := pix ++ dp }
          pix
       end

!-------------------------------------------------------------------------!
!                                                                         !
!   convert a colour-index value to its corresponding pixel representaion !
!                                                                         !
!-------------------------------------------------------------------------!
  colourToPixel := proc(c, depth: int -> pixel)
  begin
     let pnew := off
     if c rem 2 = 0 then pnew := off else pnew := on
     c := c div 2
     for i = 1 to depth - 1 do
        begin
           if c rem 2 = 0
           then pnew := pnew ++ off
           else  pnew :=  pnew ++ on
           c := c div 2
        end
     pnew
  end

!-------------------------------------------------------------------------!
!                                                                         !
!  convert a pixel representation to its corresponding colour-index value !
!                                                                         !
!-------------------------------------------------------------------------!
   pixelToColour := proc(p: pixel; depth: int -> int)
   begin
      let v := if p(0|1) = on then 1 else 0
      let s := 1
      for i = 1 to depth-1 do
         begin
            s := s * 2
            if p(i|1) = on  do v :=  v + s
         end
      v
   end
```

```
!-------------------------------------------------------------------------!
!                                                                         !
!   default n-colour RGB intensities                                      !
!   -  n = 8 or 16                                                        !
!                                                                         !
!-------------------------------------------------------------------------!
!
   rgb := proc(nc: int -> **int)
   begin
      let rgb := vector 0 to nc - 1 using proc(i:int -> *int);
                                                    vector 1 to 3 of 0
      case nc of
      8: begin          ! default 8-colour RGB intensity
                                                              ! black  (0)
            rgb(1,1) := 255                                   ! red    (1)
                              rgb(2,2) := 255                 ! green  (2)
            rgb(3,1) := 255;  rgb(3,2) := 255                 ! yellow (3)
                                                rgb(4,3) := 255 ! blue   (4)
            rgb(5,1) := 255;                    rgb(5,3) := 255 ! magenta(5)
                              rgb(6,2) := 255;  rgb(6,3) := 255 ! cyan   (6)
            rgb(7,1) := 255;  rgb(7,2) := 255;  rgb(7,3) := 255 ! white  (7)
         end
       16: begin          ! default 16-colour RGB intensity
                                                              ! black  ( 0)
            rgb( 1,1) := 127; rgb( 1,2) := 127                ! olive  ( 1)
            rgb( 2,1) := 127;                   rgb( 2,3) := 127! purple ( 2)
            rgb( 3,1) := 255;                                 ! red    ( 3)
                              rgb( 4,2) := 127; rgb( 4,3) := 127! aqua   ( 4)
                              rgb( 5,2) := 255                ! green  ( 5)
                                                rgb( 6,3) := 255! blue   ( 6)
            rgb( 7,1) :=  85; rgb( 7,2) :=  85; rgb( 7,3) :=  85! dk gray( 7)
            rgb( 8,1) := 170; rgb( 8,2) := 170; rgb( 8,3) := 170! lt gray( 8)
            rgb( 9,1) := 255; rgb( 9,2) := 255                ! yellow ( 9)
            rgb(10,1) := 255;                   rgb(10,3) := 255! magenta(10)
            rgb(11,1) := 255; rgb(11,2) := 127; rgb(11,3) := 127! pink   (11)
                              rgb(12,2) := 255; rgb(12,3) := 255! cyan   (12)
            rgb(13,1) := 127; rgb(13,2) := 255; rgb(13,3) := 127! lime   (13)
            rgb(14,1) := 127; rgb(14,2) := 127; rgb(14,3) := 255! sky    (14)
            rgb(15,1) := 255; rgb(15,2) := 255; rgb(15,3) := 255! white  (15)
         end
       default: { }
       rgb
   end

!-------------------------------------------------------------------------!
!                                                                         !
!    create an n-level intensity of gray scales                           !
!                                                                         !
!-------------------------------------------------------------------------!
   grayLevel := proc(nc: int -> **int)
   begin
      let gray := vector 0 to nc-1 using proc(i:int -> *int);
                                                    vector 1 to 3 of 0
      let intensity := 0; let diff = 255 div (nc - 1)
         for j = 1 to 3 do
            for i = 0 to nc - 1 do
               begin
                  gray(i,j) := intensity
                  intensity := intensity + diff
                  if intensity > 255 do intensity := 0
               end
      gray
   end

!-------------------------------------------------------------------------!
```

```
!                                                                              !
!    create an n-level inverse intensity of gray scales                        !
!                                                                              !
!------------------------------------------------------------------------------!
   invGrayLevel := proc(nc: int -> **int)
   begin
      let gray := vector 0 to nc-1 using proc(i:int -> *int);
                                                    vector 1 to 3 of 0
      let intensity := 255; let diff = 255 div (nc - 1)
         for j = 1 to 3 do
            for i = 0 to nc - 1 do
               begin
                  gray(i,j) := intensity
                  intensity := intensity - diff
                  if intensity < 0  do intensity := 255
               end
      gray
   end

!------------------------------------------------------------------------------!
!                                                                              !
!    remap RGB (24 bits) to 16 colours (4bits)                                 !
!                                                                              !
!------------------------------------------------------------------------------!
   remap16 := proc(rv,gv,bv: int; pixel_table: *pixel -> pixel)
   begin
      let code := 0
      let dist := 127
      let mask := 0
      if  rv + gv - bv > dist do code := bitwiseOr(code,1)
      if  rv - gv + bv > dist do code := bitwiseOr(code,2)
      if -rv + gv + bv > dist do code := bitwiseOr(code,4)
      dist := 382; mask := 8
      if code = 0 then { dist := 127 ; mask := 7 }  else
         if code = 7 do { code := 8; dist := 637 ; mask := 7 }
      if rv + gv + bv > dist do { code := bitwiseOr(code,mask) }
      let newp := pixel_table(code)
      newp
   end


!------------------------------------------------------------------------------!
!                                                                              !
!   procedure for drawing an image in a window                                 !
!                                                                              !
!------------------------------------------------------------------------------!
  viewImage := proc(raster: image; shift_pt: XY; window: image;
                                                  win_size: Win_size)
  begin
     let ras_width = xDim(raster); let ras_height = yDim(raster)
     let win_width = win_size(width); let win_height = win_size(height)
     !  project image on the window
     let xr := 0; let yr :=0; let xw := 0; let yw := 0
     let x = truncate(shift_pt(x)); let y = truncate(shift_pt(y))
     if x >= 0 and x < win_width
     then { xw := x ; xr := 0 }
     else if x < 0 and x >= - ras_width
          then { xw := 0; xr := -x - 1  }
          else if x < - ras_width
               then  { xw := 0; xr := ras_width - 1 }
               else  { xw := win_width - 1; xr := 0 }
     if y >= 0 and y < win_height
     then { yw := y ; yr := 0 }
     else if y < 0 and y >= - ras_height
          then { yw := 0; yr := -y - 1 }
          else if y < - ras_height
```

```
            then  { yw := 0; yr := ras_height - 1 }
            else  { yw := win_height - 1; yr := 0 }
    copy limit raster at xr,yr onto limit window at xw,yw
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   popup a menu in an X-window                                           !
!                                                                         !
!-------------------------------------------------------------------------!
  popupMenu := proc(items: *string; actions: *proc(); init: bool;
                    fd: file; window: image; win_size: Win_size; start: int)
  begin
    ! define dimension of the menu
    let title_height = 30 ; let item_height = 20
    let top := 0; let bottom := 10
    let left := 15; let right := 15
    let menu_depth = 4

    let fg_col = 9
    let bg_col = 8
    let mv_bp = start + menu_depth - 1 ! the bit plane of the menu frame
    let win_width = win_size(width)
    let win_height = win_size(height)

    let nc = power_2_k(menu_depth)
    let default_pixel = defaultPixel(off,menu_depth)
    let colour_index := vector 0 to nc-1 of default_pixel
    for i = 0 to nc-1 do colour_index(i) := colourToPixel(i,menu_depth)
    let title = items(0)
    let title_image = screenB14(stringToTile)(title)
    let title_width = xDim(title_image)
    let title_font_height = yDim(title_image)
    let max_item_width := title_width
    let no_items = upb[string](items) - lwb[string](items)
    for i = 1 to no_items do
    begin
      let item_width = xDim(screenR14(stringToTile)(items(i)))
      if item_width > max_item_width do max_item_width := item_width
    end
    let menu_width = left + max_item_width + right
    let menu_height = title_height + top + no_items * item_height + bottom
    let menu_image := image menu_width by menu_height of
                                                    colour_index(bg_col)
    let font_height = yDim(screenR14(stringToTile)(items(1)))
    let half_space = (item_height - font_height) div 2
    let dy = menu_height - title_height - top - half_space - font_height
    let c := fg_col
    for i = 1 to no_items do
    begin
      for k = 0 to menu_depth - 2 do
      begin
        if c rem 2 = 1 do { copy screenR14(stringToTile)(items(i)) onto
                            limit menu_image(k|1) at
                                      left, dy - (i-1) * item_height }
        c := c div 2
      end
      c := fg_col
    end
    c := fg_col
    for k = 0 to menu_depth - 2 do  ! the highest bit is reserved
                                    ! for foreground colour.
    begin
      ! Note: font tiles are one-bit images.
      !       0 = background, 1 = foreground.
      if c rem 2 = 1 do
```

```
      { copy title_image onto limit menu_image(k|1) at
        left + (max_item_width - title_width) div 2, menu_height -
        (title_height - title_font_height) div 2 - title_font_height }
   c := c div 2
end
let menu_tmp_img := image menu_width by menu_height of
                                          defaultPixel(off,menu_depth)
let title_tmp_img := image title_width by title_font_height of
                                            defaultPixel(off,menu_depth)
let item_tmp_img := image max_item_width by item_height of
                                            defaultPixel(off,menu_depth)
! use one of the bit planes of the window image for the manipulation
! of moving menu frame.
let bg_tmp_img := image xDim(window) by yDim(window) of off
copy window(mv_bp | 1) onto bg_tmp_img
let finished := false
let popuped := false
let retained := if init then true else false
let highlight_line := false
let highlight_title := false
let highlight_item := false
let visited := false
let moved := false
let anchor_x := win_width - menu_width - 15
let anchor_y := win_height - 15
let anchor_x_save := 0 ; let anchor_y_save := 0
let last_x := 0; let last_y := 0
let last_item := 0
let menu_frame := image menu_width by menu_height of on
line(menu_frame,0,0,menu_width-1,0,off,12)
line(menu_frame,menu_width-1,0,menu_width-1,menu_height-1,off,12)
line(menu_frame,menu_width-1,menu_height-1,0,menu_height-1,off,12)
line(menu_frame,0,menu_height-1,0,0,off,12)
line(menu_frame,2,2,menu_width-3,2,off,12)
line(menu_frame,menu_width-3,2,menu_width-3,menu_height-3,off,12)
line(menu_frame,menu_width-3,menu_height-3,2,menu_height-3,off,12)
line(menu_frame,2,menu_height-3,2,2,off,12)
let data := vector 1 to 7 of 0
let do_menu = proc()
begin
   while ~finished do
   begin
      locator(fd, data)
      if (data(7) = 1 or popuped or retained) and  ~moved do
      begin
         !
         ! popup the menu, if it does not appear.
         !
         if ~popuped do
         begin
            if data(7) = 1 do { anchor_x := data(3);
                                                anchor_y := data(4) }
            if win_width - anchor_x < menu_width do
               { anchor_x := win_width - menu_width }
            if anchor_y < menu_height do { anchor_y := menu_height }
            copy limit window(start | menu_depth) at
                 anchor_x,anchor_y - menu_height onto menu_tmp_img
            copy menu_image onto limit window(start | menu_depth) at
                                anchor_x, anchor_y - menu_height
            popuped := true
            highlight_item := false
         end
         !
         ! highlight the menu title, title underline and items.
         !
         ! if the cursor is inside the menu template area, then
```

```
! highlight the underline of the menu title.
if data(3) > anchor_x and data(3) - anchor_x < menu_width and
   data(4) < anchor_y and data(4) > anchor_y - menu_height
then
begin
   if ~highlight_line do
   begin
      let line_image = image max_item_width by 1 of
                                              colour_index(fg_col)
      copy line_image onto limit window(start | menu_depth) at
                  anchor_x + left, anchor_y - title_height + 5
      highlight_line := true
   end
end
else if highlight_line do
begin
   let line_image = image max_item_width by 1 of
                                              colour_index(bg_col)
   copy line_image onto limit window(start | menu_depth) at
                anchor_x + left, anchor_y - title_height + 5
   highlight_line := false
end
! if the cursor is inside the menu tile area, then highlight
! the menu title.
if data(3) > anchor_x + left and
   data(3) - anchor_x < menu_width -right and
   data(4) > anchor_y - title_height and
   data(4) < anchor_y  then
begin
   if ~highlight_title do
   begin
      copy limit window(start | menu_depth) at
         anchor_x + (menu_width - title_width) div 2,
         anchor_y - (title_height - title_font_height) div 2 -
          title_font_height onto title_tmp_img
      let new_title_image := image title_width by
                     title_font_height of
                     defaultPixel(off, menu_depth - 1) ++ on
      for k = 0 to menu_depth - 2 do  ! a white title
         { copy title_image onto new_title_image(k|1) }
      copy new_title_image onto limit
         window(start | menu_depth) at
         anchor_x + (menu_width - title_width) div 2,
         anchor_y - (title_height - title_font_height) div 2 -
         title_font_height
      highlight_title := true
   end
end
else if highlight_title do
begin
   copy title_tmp_img onto limit window(start | menu_depth) at
      anchor_x + (menu_width - title_width) div 2,
      anchor_y - (title_height - title_font_height) div 2 -
      title_font_height
   highlight_title := false
end
! if the cursor is inside the item area, then highlight the
! selected item.
if data(3) > anchor_x + left and
   data(3) - anchor_x < menu_width - right and
   data(4) < anchor_y - top - title_height and
   data(4) > anchor_y - menu_height + bottom  then
begin
   let y_offset = data(4) - anchor_y + title_height + top
   let item = - y_offset div ((menu_height - title_height -
                                  top - bottom) div no_items)
```

```
      if ~highlight_item do
      begin
         copy limit window(start | menu_depth) at
             anchor_x + 15,anchor_y - title_height -
             top - (item + 1) * item_height onto item_tmp_img
         let light_image := image max_item_width by
                            item_height of colour_index(fg_col)
         ! reverse the foreground and background colours of the
         ! image of the selected item.
         xor item_tmp_img(0 | menu_depth - 1) onto
                                    light_image(0 | menu_depth - 1)
         copy light_image onto limit
              window(start | menu_depth) at anchor_x + 15,
              anchor_y - title_height - top -
              (item + 1) * item_height
         highlight_item := true
         last_item := item
      end
      if item ~= last_item do
      begin
         copy item_tmp_img onto limit
              window(start | menu_depth) at anchor_x + 15,
              anchor_y - title_height - top -
                                 (last_item + 1) * item_height
         highlight_item := false
      end
   end
   else if highlight_item do
   !  if the cursor is outside the item area and an item is
   !  already highlighted, then turn it off.
   begin
      copy item_tmp_img onto limit window(start | menu_depth) at
           anchor_x + 15, anchor_y - title_height - top -
                                    (last_item + 1) * item_height
      highlight_item := false
   end
end
!
! execute the command of an item
!
if popuped and data(7) = 0 do
begin
   ! when the cursor is inside the menu title area
   ! if the menu is just popuped, then release mouse button 3
   ! will retain the menu.
   ! if the menu is already retained, then click mouse button 2
   ! will dismiss the menu.
   if data(3) > anchor_x + left and
      data(3) - anchor_x < menu_width - right and
      data(4) > anchor_y - title_height and
      data(4) < anchor_y  do
   begin
      retained := true
      ! click button 2 -- dismiss the menu.
      if data(6) = 1 do { retained := false }
      ! intialize the conditions for the movement of the menu
      if data(5) = 0 then
         { last_x := data(3); last_y := data(4) }
      else if data(5) = 1 do
      begin
         if ~moved do
         begin
            moved := true
            ! dehighlight the menu title
            copy menu_image onto limit
                 window(start | menu_depth) at
```

```
                anchor_x, anchor_y - menu_height
            ! stand out the menu frame
            copy menu_frame onto limit window(mv_bp | 1) at
                            anchor_x, anchor_y - menu_height
            anchor_x_save := anchor_x
            anchor_y_save := anchor_y
         end
      end
   end
   ! drag the menu -- hold down the button 1 and drag the menu
   !                  to a new location, then release the button.
   if data(5) = 1 and moved then
   begin
      let dx = data(3) - last_x
      let dy = data(4) - last_y
      if  dx ~= 0 or dy ~= 0  do
      begin
         let anchor_x_tmp = anchor_x + dx
         let anchor_y_tmp = anchor_y + dy
         if anchor_x_tmp > 0 and
            anchor_x_tmp < win_width - menu_width and
            anchor_y_tmp > menu_height and
            anchor_y_tmp < win_height  do
         begin
            let tmp_img_ht = menu_height + abs(dy)
            let tmp_image := image (menu_width + abs(dx)) by
                                                tmp_img_ht of off
            let qd = if dx >= 0 and dy >= 0 then 1
                     else if dx >= 0 and dy < 0 then 2
                     else if dx <  0 and dy >= 0 then 3
                     else  4
            case qd of
            1 : { copy limit bg_tmp_img at
                        anchor_x, anchor_y - menu_height onto
                        tmp_image
                  copy menu_frame onto limit tmp_image at dx,dy
                  copy tmp_image onto limit window(mv_bp | 1) at
                        anchor_x, anchor_y - menu_height }
            2 : { copy limit bg_tmp_img at
                        anchor_x, anchor_y - tmp_img_ht onto
                        tmp_image
                  copy menu_frame onto limit tmp_image at dx,0
                  copy tmp_image onto limit window(mv_bp | 1) at
                        anchor_x, anchor_y - tmp_img_ht }
            3 : { copy limit bg_tmp_img at anchor_x_tmp,
                        anchor_y_tmp - tmp_img_ht onto tmp_image
                  copy menu_frame onto limit tmp_image at 0,dy
                  copy tmp_image onto limit window(mv_bp | 1) at
                        anchor_x_tmp, anchor_y_tmp - tmp_img_ht }
            4 : { copy limit bg_tmp_img at anchor_x_tmp,
                        anchor_y - tmp_img_ht onto tmp_image
                  copy menu_frame onto tmp_image
                  copy tmp_image onto limit window(mv_bp | 1) at
                        anchor_x_tmp, anchor_y - tmp_img_ht }
            default: {}
            anchor_x := anchor_x_tmp
            anchor_y := anchor_y_tmp
            last_x := data(3); last_y := data(4)
         end
      end
   end
   else
   begin
      if moved do
      begin
         if anchor_x = anchor_x_save and anchor_y = anchor_y_save
```

```
        then { }    ! clicked but actually not moved
        else
        begin
           ! refresh the image of the bit plane which has been
           ! changed by the movement of the menu frame
           copy bg_tmp_img onto window(mv_bp | 1)
           copy menu_tmp_img onto
                limit window(start | menu_depth) at
                anchor_x_save, anchor_y_save - menu_height
           copy limit window(start | menu_depth) at anchor_x,
                anchor_y - menu_height onto menu_tmp_img
        end
        copy menu_image onto limit
             window(start | menu_depth) at anchor_x,
                                        anchor_y - menu_height
        highlight_line := false
        highlight_title := false
        moved := false
     end
  end
  ! when cursor is inside the item area
  ! if the menu is just popuped, then release mouse button 3
  ! wlll execute the command of a menu item.
  ! if the menu is already retained, click mouse button 1
  ! will execute the command of a menu item.
  if data(3) > anchor_x + left and
     data(3) - anchor_x < menu_width - right and
     data(4) < anchor_y - top - title_height and
     data(4) > anchor_y - menu_height + bottom  do
  begin
     if ~retained or (retained and data(5) = 1 and ~moved) do
     begin
        if ~retained then       ! dismiss the menu
        begin
           popuped := false
           copy menu_tmp_img onto limit
                window(start | menu_depth) at anchor_x,
                                        anchor_y - menu_height
        end
        else         ! dehighlight the menu after the selection
        begin
           copy menu_image onto limit
                window(start | menu_depth) at
                anchor_x, anchor_y - menu_height
        end
        let y_offset = data(4) - anchor_y + title_height + top
        let c = -y_offset div ((menu_height - title_height -
                                   top - bottom) div no_items)
        visited := false

        ! execute the procedure of the selected item
        if c >= 0 and c <= no_items - 2 then
        begin
           ! restore the original window image before doing any
           ! action
           copy menu_tmp_img onto limit
                window(start | menu_depth) at anchor_x,
                                        anchor_y - menu_height
           actions(c)()
           visited := true
           ! update the content of the moving bit plane of the
           ! menu frame
           copy window(mv_bp | 1) onto bg_tmp_img
           copy limit window(start | menu_depth) at anchor_x,
                     anchor_y - menu_height onto menu_tmp_img
        end
```

```
                    else if c = no_items - 1  do  { finished := true }

                    if retained and visited do
                    begin
                       copy menu_image onto limit
                            window(start | menu_depth) at
                            anchor_x, anchor_y - menu_height
                       highlight_line := false
                       highlight_title := false
                       highlight_item := false
                       visited := false
                    end
                 end
              end
              if ~retained or finished do
              begin
                 popuped := false
                 copy menu_tmp_img onto limit window(start | menu_depth) at
                                          anchor_x, anchor_y - menu_height
              end
           end
        end
     end
     do_menu()
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   display and get message in a dialogue box                             !
!                                                                         !
!-------------------------------------------------------------------------!
  dialogueBox := proc(message, prompt: string; fd: file; window: image;
                      win_size: Win_size; start: int -> string)
  begin
     let top := 15; let bottom := 15
     let left := 20;
     let right := if message = "" then 75 else 20
     let box_depth = 4
     let msg_height = 20
     let max_msg_width := 0
     let bg_col = 14
     let fg_col = 9
     let char_col = 9
     let win_width = win_size(width)
     let win_height = win_size(height)
     let nc = power_2_k(box_depth)
     let default_pixel = defaultPixel(off,box_depth)
     let colour_index := vector 0 to nc-1 of default_pixel
     for i = 0 to nc-1 do colour_index(i) := colourToPixel(i,box_depth)
     let message_len = length(message)
     let n := 1
     for i= 1 to message_len do       ! 'n = LF (ASC = 10)
     begin
        if stringToAscii(message(i | 1)) = 10 do { n := n + 1 }
     end
     if prompt ~= "" and message ~= "" do  n := n + 2
     let msg := vector 1 to n of ""
     msg(n) := prompt
     let k := 1
     for i = 1 to message_len do
     begin
        if stringToAscii(message(i | 1)) = 10 then { k := k + 1 }
        else  { msg(k) := msg(k) ++ message(i | 1) }
     end
     for i = 1 to n do
     begin
```

```
   let msg_width = xDim(screenR14(stringToTile)(msg(i)))
   if msg_width > max_msg_width do max_msg_width := msg_width
end
let box_width = left + max_msg_width + right
let box_height = top + n * msg_height + bottom
let box_image := image box_width by box_height of colour_index(bg_col)
let box_tmp_img := image box_width by box_height of
                                              defaultPixel(off,box_depth)
let font_height = yDim(screenR14(stringToTile)(" "))
let half_space = (msg_height - font_height) div 2
let dy = box_height - top - half_space - font_height
let c := fg_col
for i = 1 to n do
begin
   for k = 0 to box_depth - 2 do
   begin
      if c rem 2 = 1 do { copy screenR14(stringToTile)(msg(i)) onto
                                 limit box_image(k|1) at
                                 left, dy - (i-1) * msg_height }
      c := c div 2
   end
   c := fg_col
end
let xr = (win_width - box_width) div 2
let yr = (win_height - box_height) div 2
copy limit window(start | box_depth) at xr,yr onto box_tmp_img
copy box_image onto limit window(start | box_depth) at xr,yr
let result := ""
use makeReadEnv (fd) with
    inputPending : proc (-> bool);
    readChar : proc (-> string) in
begin
   ! clear characters remaining in the input buffer
   repeat { } while inputPending() do { let char = readChar() }
   let data := vector 1 to 7 of 0
   let finished := false
   let i := 0
   while ~finished do
   begin
      ! keep waiting for keyboard input and mouse click
      while ~inputPending() and data(6) = 0 do { locator(fd,data) }
      if data(6) = 1 then { finished := true }
      else
      begin
         let char = readChar()
         let asc = stringToAscii(char)
         if asc = 10 or asc = 27 then       ! if LF or ESC then stop.
            { finished := true }
         else
         begin
            if prompt ~= "" do
            begin
               if asc > 31 and asc < 127 then
               begin
                  result := result ++ char
                  let c := char_col
                  let char_img := image
                                  xDim(screenR14(stringToTile)(" ")) by
                                  yDim(screenR14(charToTile)(" ")) of
                                  colour_index(bg_col)
                  for k = 0 to box_depth - 2 do
                  begin
                     if c rem 2 = 1 do
                        { copy screenR14(stringToTile)(char) onto
                                                limit char_img(k|1) }
                     c := c div 2
```

```
                        end
                        copy char_img onto limit window(start | box_depth) at
                             xr+left+xDim(screenR14(stringToTile)(msg(n)))+
                             i*xDim(screenR14(charToTile)(char)),
                             yr+bottom+half_space
                        i := i + 1
                     end
                     else if asc = 127 and prompt ~= "" do
                     begin
                        if i > 0 do
                        begin
                           if i > 1 then
                                    {result := result(1|length(result) - 1)}
                                  else { result := ""}
                           i := i - 1
                           let delete := image
                                       xDim(screenR14(stringToTile)(" ")) by
                                       yDim(screenR14(charToTile)(" ")) of
                                       colour_index(bg_col)
                           copy delete onto limit window(start | box_depth) at
                              xr+left+xDim(screenR14(stringToTile)(msg(n)))+
                              i*xDim(screenR14(charToTile)(" ")),
                              yr+bottom+half_space
                        end
                     end
                  end
               end
            end
         end
      end
      copy box_tmp_img onto limit window(start | box_depth) at xr,yr
      result
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   write a message in an X-window                                        !
!                                                                         !
!-------------------------------- ----------------------------------------!
   writeMessage := proc(message: string; fd: file; window: image;
                        win_size: Win_size; start: int -> Transient_image)
   begin
      let top := 15; let bottom := 15
      let left := 20; let right := 20
      let box_depth = 4
      let msg_height = 20
      let max_msg_width := 0
      let bg_col = 14
      let fg_col = 9
      let char_col = 9
      let win_width = win_size(width)
      let win_height = win_size(height)
      let nc = power_2_k(box_depth)
      let default_pixel = defaultPixel(off,box_depth)
      let colour_index := vector 0 to nc-1 of default_pixel
      for i = 0 to nc-1 do colour_index(i) := colourToPixel(i,box_depth)
      let message_len = length(message)
      let n := 1
      for i= 1 to message_len do       ! 'n = LF (ASC = 10)
      begin
         if stringToAscii(message(i | 1)) = 10 do { n := n + 1 }
      end
      let msg := vector 1 to n of ""
      let k := 1
      for i = 1 to message_len do
      begin
```

```
      if stringToAscii(message(i | 1)) = 10 then { k := k + 1 }
      else  { msg(k) := msg(k) ++ message(i | 1) }
   end
   for i = 1 to n do
   begin
      let msg_width = xDim(screenR14(stringToTile)(msg(i)))
      if msg_width > max_msg_width do max_msg_width := msg_width
   end

   let box_width = left + max_msg_width + right
   let box_height = top + n * msg_height + bottom
   let box_image := image box_width by box_height of colour_index(bg_col)
   let box_tmp_img := image box_width by box_height of
                                                defaultPixel(off,box_depth)
   let font_height = yDim(screenR14(stringToTile)(" "))
   let half_space = (msg_height - font_height) div 2
   let dy = box_height - top - half_space - font_height
   let c := fg_col
   for i = 1 to n do
   begin
      for k = 0 to box_depth - 2 do
      begin
         if c rem 2 = 1 do { copy screenR14(stringToTile)(msg(i)) onto
                             limit box_image(k|1) at
                             left, dy - (i-1) * msg_height }
         c := c div 2
      end
      c := fg_col
   end
   let xr = (win_width - box_width) div 2
   let yr = (win_height - box_height) div 2
   copy limit window(start | box_depth) at xr,yr onto box_tmp_img
   copy box_image onto limit window(start | box_depth) at xr,yr
   let pos = Pos(xr,yr)
   let transient_img = Transient_image(box_tmp_img,pos,start)
   transient_img
end

!-------------------------------------------------------------------------!
!                                                                         !
!   erase a message in an X-window                                        !
!                                                                         !
!-------------------------------------------------------------------------!
  eraseMessage := proc(msg_img: Transient_image; window: image)
  begin
     let raster = msg_img(raster)
     let depth = zDim(raster)
     let pos = msg_img(pos)
     copy raster onto limit window(msg_img(start_bp) | depth) at
                                                        pos(x), pos(y)
  end


end
```

APPENDIX E : THE GIS LIBRARY PROCEDURES

```
!-------------------------------------------------------------------------!
!                                                                         !
!                                                          [gis_stubs.N]  !
!                                                                         !
!                                                                         !
!    This program sets up the variable stubs in the GIS environment       !
!                                                                         !
!                                                                         !
!                                                                         !
!-------------------------------------------------------------------------!

use PS() with IO, Time, GlasgowLibraries,
              User:env; environment: proc( -> env) in
use User with Library: env in
use Library with GIS: env in
use IO with writeString: proc(string) in
use Time with date: proc(->string) in
use GlasgowLibraries with Miscellany: env in
use Miscellany with  uninitialised: proc[T](string->T);
                     uninitialised_void: proc(string)  in
begin
   let date = date()
   if GIS contains getOSmapInfo then
     writeString("'nGIS already contains getOSmapInfo, no changes made.'n")
   else
   begin
   ! keep a record of date when the GIS library was last updated

   in GIS let changedOn := date

   in GIS let getOSmapInfo := proc(map_name: string -> OS_map_info)
                    uninitialised[OS_map_info]("getOSmapInfo")

   in GIS let getOSmapName := proc(point: XY -> OS_map_name)
                    uninitialised[OS_map_name]("getOSmapName")


   in GIS let ntf625kToBasemap := proc(fn: string -> Basemap)
                    uninitialised[Basemap]("ntf625kToBasemap")

   in GIS let ntf250kToBasemap := proc(fn: string -> Basemap)
                    uninitialised[Basemap]("ntf250kToBasemap")

   in GIS let ntfcontourToBasemap := proc(fn: string -> Basemap)
                    uninitialised[Basemap]("ntfcontourToBasemap")

   in GIS let ntfblToBasemap := proc(fn: string -> Basemap)
                    uninitialised[Basemap]("ntfblToBasemap")

   in GIS let ntfllToBasemap := proc(fn: string -> Basemap)
                    uninitialised[Basemap]("ntfllToBasemap")

   in GIS let ntfoscarToBasemap := proc(fn: string -> Basemap)
                    uninitialised[Basemap]("ntfoscarToBasemap")


   in GIS let storeBasemap := proc(map_id: Map_id; basemap: Basemap;
                                   map_extent: Extent)
                    uninitialised_void("storeBasemap")

   in GIS let removeBasemap := proc()
                    uninitialised_void("removeBasemap")


   in GIS let getPolyMBR := proc(poly_id: Poly_id;
                                 pb_cid_chain: PB_cid_chain;
```

```
                                pb_gid_geometry: PB_gid_geometry -> MBR)
                uninitialised[MBR]("getPolyMBR")


in GIS let pointInPolygon := proc(test_pt: XY;
                                pb_polygon: PB_polygon;
                                pb_cid_chain: Map[Chain_id, PB_chain];
                                pb_gid_geometry: Map[Geom_id, PB_geometry];
                                chain_mbr: Map[Geom_id, MBR] -> bool)
                uninitialised[bool]("pointInPolygon")

in GIS let gridNdxPoly := proc(poly_id: Poly_id; mbr: MBR;
                                 polygon_index: Map[Peanor,List[Poly_id]];
                                 sl: real)
                uninitialised_void("gridNdxPoly")

in GIS let lqtNdxPoint := proc(peano: Peanor;
                                 pointid_list: List[Point_id];
                                 pid_point: Map[Point_id, XY];
                                 point_index: Map[Peanor, List[Point_id]])
                uninitialised_void("lqtNdxPoint")

in GIS let lqtNdxLine := proc(peano: Peanor; lid_list: List[Line_id];
                                lid_line: Map[Line_id, List[XY]];
                                line_mbr: Map[Line_id, MBR];
                                line_key_pts: Map[Line_id, List[XY]];
                                line_index: Map[Peanor, List[Line_id]])
                uninitialised_void("lqtNdxLine")

in GIS let lqtNdxPoly := proc(peano: Peanor; pid_list: List[Poly_id];
                                poly_mbr: Map[Poly_id, MBR];
                                polygon_index: Map[Peanor, List[Poly_id]])
                uninitialised_void("lqtNdxPoly")


in GIS let fbffToRaw := proc(fn: string;
                               width, height, depth: int -> Rawimage)
                uninitialised[Rawimage]("fbffToRaw")

in GIS let tiffToRaw := proc(fn: string -> Rawimage)
                uninitialised[Rawimage]("tiffToRaw")

in GIS let rawToInterim := proc(rawimage: Rawimage -> Interim_image)
                uninitialised[Interim_image]("rawToInterim")

in GIS let hsiToInterim := proc(fn: string -> Interim_image)
                uninitialised[Interim_image]("hsiToInterim")

in GIS let sunrasToInterim := proc(fn: string -> Interim_image)
                uninitialised[Interim_image]("sunrasToInterim")

in GIS let tiffToInterim := proc(fn: string -> Interim_image)
                uninitialised[Interim_image]("tiffToInterim")

in GIS let interimToSunras := proc(interim_image: Interim_image;
                                     fn: string)
                uninitialised_void("interimToSunras")


in GIS let previewRaw := proc(rawimge: Rawimage;
                                win_x,win_y: int -> image)
                uninitialised[image]("previewRaw")

in GIS let previewStretchedRaw := proc(rawimge: Rawimage;
                                         win_x,win_y: int -> image)
                uninitialised[image]("previewStretchedRaw")
```

```
    in GIS let freqCount := proc(rawimage: Rawimage -> Frequency)
                    uninitialised[Frequency]("freqCount")

    in GIS let linearStretch := proc(rawimage: Rawimage;
                                     frequency: Frequency;
                                     new_depth: int -> image)
                    uninitialised[image]("linearStretch")

    in GIS let freqCount2 := proc(interim: image -> Frequency)
                    uninitialised[Frequency]("freqCount2")

    in GIS let linearStretch2 := proc(interim: image;
                                      frequency: Frequency;
                                      new_depth: int -> image)
                    uninitialised[image]("linearStretch2")

    in GIS let getFreqChart := proc(frequncy: Frequency;
                                    new_depth: int -> Freq_chart)
                    uninitialised[Freq_chart]("getFreqChart")

    in GIS let plotFreqChart := proc(window: image; frequncy: Frequency;
                                     freq_chart: Freq_chart;
                                     bg_col, fg_col: pixel)
                    uninitialised_void("plotFreqChart")


    in GIS let rawToInterimImage := proc()
                    uninitialised_void("rawToInterimImage")

    in GIS let interimToBaseimage := proc()
                    uninitialised_void("interimToBaseimage")


    in GIS let storeRawimage := proc(image_id: Image_id; rawimage: Rawimage)
                    uninitialised_void("storeRawimage")

    in GIS let storeInterimImage := proc(image_id: Image_id;
                                         interim_image: Interim_image)
                    uninitialised_void("storeInterimImage")

    in GIS let removeRawimage := proc()
                    uninitialised_void("removeRawimage")

    in GIS let removeInterimImage := proc()
                    uninitialised_void("removeInterimImage")

    in GIS let removeBaseimage := proc()
                    uninitialised_void("removeBaseimage")


    in GIS let loadBaseimage := proc(image_id: Image_id;
                                     window_file: file -> image)
                    uninitialised[image]("loadBaseimage")


    writeString("'n'"GIS'" environment stubs set up on ")
    writeString(date)
    writeString("'n")
    end
end
```

```
!-------------------------------------------------------------------------!
!                                                                         !
!                                                              [gisLib.N] !
!                                                                         !
!      This program sets up the libraries in the GIS environment          !
!                                                                         !
!                                                                         !
!                                                                         !
!-------------------------------------------------------------------------!

type drawFunction is variant(imageDraw: proc(image,pic,real,real,real,real);
                             fileDraw: proc(file,pic,real,real,real,real);
                             fail: null)

use PS() with Arithmetical, String, IO, Vector, System, Format,
              Graphical, Device, GlasgowLibraries, User: env  in
use Arithmetical with
                 float: proc(int -> real);
                 truncate: proc(real -> int);
                 bitwiseOr: proc(int,int -> int)  in
use String with
           stringToAscii: proc(string -> int);
           asciiToString: proc(int -> string);
           letter,digit: proc(string -> bool);
           length: proc(string -> int)  in
use IO with
       PrimitiveIO: env;
       makeReadEnv: proc(file -> env);
       readLine: proc(-> string);
       readReal: proc(-> real);
       writeInt: proc(int);
       writeString: proc(string)  in
use PrimitiveIO with
                 create: proc(string,int -> file);
                 open: proc(string,int -> file);
                 seek: proc(file,int,int -> int);
                 close: proc(file -> int);
                 readBytes: proc(file,*int,int,int -> int);
                 writeBytes: proc(file,*int,int,int -> int);
                 getByte: proc(int,int -> int);
                 setByte: proc(int,int,int -> int);
                 errorNumber: proc(-> int)  in
use Vector with
           lwb,
           upb: proc[t](*t -> int)  in
use System with
           abort: proc()  in
use Format with
           iformat: proc(int -> string);
           fformat: proc(real,int,int -> string)  in
use Graphical with
              Raster,
              Outline: env  in
use Outline with
            makeDrawFunction: proc(string -> drawFunction)  in
use Raster with
           getPixel: proc(image,int,int -> pixel);
           setPixel: proc(image,int,int,pixel);
           xDim: proc(image -> int);
           yDim: proc(image -> int);
           zDim: proc(image -> int);
           line: proc(image,int,int,int,int,pixel,int)  in
use Device with
           getScreen: proc(file -> image);
           colourMap: proc(file,pixel,int);
           locator: proc(file,*int);
```

```
            getCursor: proc(file -> image);
            getCursorInfo: proc(file,*int);
            setCursor: proc(file,image);
            colourOf: proc(file,pixel -> int)  in
use GlasgowLibraries with BulkTypes: env  in
use BulkTypes with Maps, Lists: env  in
use Maps with
          m_empty: proc[A,Z](proc(A,A -> bool),proc(A,A -> bool)
                              -> Map[A,Z]);
          m_isEmpty: proc[A,Z](Map[A,Z] -> bool);
          m_isu_insert: proc[A,Z](Map[A,Z],A,Z);
          m_isu_remove: proc[A,Z](Map[A,Z],A);
          m_isu_assign: proc[A,Z](Map[A,Z],A,Z);
          m_find: proc[A,Z](Map[A,Z],A -> Z);
          m_length: proc[A,Z](Map[A,Z] -> int);
          m_contains: proc[A,Z](Map[A,Z],A -> bool);
          m_copy: proc[A,Z](Map[A,Z] -> Map[A,Z]);
          m_isu_union: proc[A,Z](Map[A,Z], Map[A,Z]);
          m_isu_clear: proc[A,Z](Map[A,Z]);
          m_filter: proc[A,Z](Map[A,Z],proc(A,Z -> bool) -> Map[A,Z]);
          m_app: proc[A,Z](Map[A,Z],proc(A,Z)) in
use Lists with hd: proc[T](List[T] -> T);
               tl: proc[T](List[T] -> List[T]);
               l_make: proc[T](->List[T]);
               l_length: proc[T](List[T] -> int);
               l_append: proc[T](List[T],T -> List[T]);
               l_prepend: proc[T](T,List[T] -> List[T]);
               l_reverse: proc[T](List[T] -> List[T]);
               l_app: proc[T](List[T],proc(T));
               l_map: proc[T,X](List[T],proc(T->X) -> List[X]);
               l_contains: proc[T](List[T],T -> bool);
               l_first: proc[T](List[T] -> T);
               l_last: proc[T](List[T] -> T);
               l_nth: proc[T](List[T],int -> T);
               l_rest: proc[T](List[T] -> List[T] );
               l_isu_remove: proc[T](T,List[T] -> List[T])  in

use User with
          Library, Database: env  in
use Library with
            General,
            Graphical,
            GIS: env  in
use General with
            stringToInt: proc(string -> int);
            errorAbort: proc(string);
            waitSymbol: proc(int);
            newline: proc(int);
            space: proc(int);
            intToBits: proc(int,int -> *int);
            bitsToInt: proc(*int -> int);
            power_2_k: proc(int -> int);
            vector_isu_sort: proc(*real);
            eq_int, lt_int: proc(int,int -> bool);
            eq_str, lt_str: proc(string,string -> bool);
            eq_peano, lt_peano: proc(Peano,Peano -> bool);
            eq_peanor, lt_peanor: proc(Peanor,Peanor -> bool);
            xyToPK: proc(XY -> int);
            pkToXY: proc(int -> XY);
            xyToPKR: proc(XY -> real);
            pkrToXY: proc(real -> XY);
            getQuadExtent: proc(Peanor -> Extent)   in
use Graphical with
              drawPoint: proc(XY,pixel,image,Extent);
              drawLineString: proc(List[XY],pixel,image,Extent);
              drawText: proc(string,real,real,pixel,XY,image,Extent);
```

```
            drawRectangle: proc(MBR,pixel,image,Extent);
            pointInWindow: proc(XY,MBR -> bool);
            lineVisibleInWindow: proc(XY,XY,MBR -> bool);
            lineStrThroughWindow: proc(List[XY],MBR,XY,real -> bool);
            getPoint: proc(file,image,Win_size,Extent,int -> XY);
            xHairGetPoint: proc(file,image,Win_size,Extent,int -> XY);
            dynaGetWinCornersA: proc(file,image,Win_size,Extent,
                                     int -> *XY);
            dynaGetWinCornersB: proc(file,image,Win_size,Extent,
                                     int -> *XY);
            getDragDxy: proc(file,image,Win_size,int -> XY);
            getZoomExtent: proc(string,file,image,Win_size,Extent,
                                Extent,int -> Extent);

            getLineStrMBR: proc(List[XY] -> MBR);
            getLineStrKeyPts: proc(List[XY] -> List[XY]);
            defaultPixel: proc(pixel,int -> pixel);
            colourToPixel: proc(int,int -> pixel);
            pixelToColour: proc(pixel,int -> int);
            rgb: proc(int -> **int);
            grayLevel: proc(int -> **int);
            invGrayLevel: proc(int -> **int);
            remap16: proc(int,int,int,*pixel -> pixel)  in
use GIS with
        getOSmapInfo: proc(string -> OS_map_info);
        getOSmapName: proc(XY -> OS_map_name);
        ntf625kToBasemap: proc(string -> Basemap);
        ntf250kToBasemap: proc(string -> Basemap);
        ntfcontourToBasemap: proc(string -> Basemap);
        ntfblToBasemap: proc(string -> Basemap);
        ntfllToBasemap: proc(string -> Basemap);
        ntfoscarToBasemap: proc(string -> Basemap);
        storeBasemap: proc(Map_id,Basemap,Extent);
        removeBasemap: proc();
        getPolyMBR: proc(Poly_id,PB_cid_chain,PB_gid_geometry -> MBR);
        pointInPolygon: proc(XY,PB_polygon,Map[Chain_id,PB_chain],
                             Map[Geom_id,PB_geometry],
                             Map[Geom_id,MBR] -> bool);
        gridNdxPoly: proc(Poly_id,MBR,Map[Peanor,List[Poly_id]],real);
        lqtNdxPoint: proc(Peanor,List[Point_id],Map[Point_id,XY],
                          Map[Peanor,List[Point_id]]);
        lqtNdxLine: proc(Peanor,List[Line_id],Map[Line_id,List[XY]],
                         Map[Line_id,MBR],Map[Line_id,List[XY]],
                         Map[Peanor,List[Line_id]]);
        lqtNdxPoly: proc(Peanor,List[Poly_id],Map[Poly_id,MBR],
                         Map[Peanor,List[Poly_id]]);
        fbffToRaw: proc(string,int,int,int -> Rawimage);
        tiffToRaw: proc(string -> Rawimage);
        rawToInterim: proc(Rawimage -> Interim_image);
        hsiToInterim: proc(string -> Interim_image);
        sunrasToInterim: proc(string -> Interim_image);
        tiffToInterim: proc(string -> Interim_image);
        interimToSunras: proc(Interim_image,string);
        previewRaw: proc(Rawimage,int,int -> image);
        previewStretchedRaw: proc(Rawimage,int,int -> image);
        freqCount: proc(Rawimage -> Frequency);
        linearStretch: proc(Rawimage,Frequency,int -> image);
        freqCount2: proc(image -> Frequency);
        linearStretch2: proc(image,Frequency,int -> image);
        getFreqChart: proc(Frequency,int -> Freq_chart);
        plotFreqChart: proc(image,Frequency,Freq_chart,pixel,pixel);
        rawToInterimImage: proc();
        interimToBaseimage: proc();
        storeRawimage: proc(Image_id,Rawimage);
        storeInterimImage: proc(Image_id,Interim_image);
        removeRawimage: proc();
```

```
        removeInterimImage: proc();
        removeBaseimage: proc();
        loadBaseimage: proc(Image_id,file -> image)  in

use Database with Raw, Interim, Processed, Derived, Index: env in
use Raw with raw_images: Map[Image_id,Rawimage] in
use Interim with interim_images: Map[Image_id,Interim_image] in
use Processed with base_maps: Map[Map_id,Basemap];
                   base_images: Map[Image_id,Baseimage] in
use Index with basemap_indices: Map[Peano,Map_id];
               baseimage_indices: Map[Peano,List[Image_id]] in
begin

!-------------------------------------------------------------------------!
!                                                                         !
!   Initialize required variables                                         !
!                                                                         !
!-------------------------------------------------------------------------!

  let draw = makeDrawFunction("image")'imageDraw

!-------------------------------------------------------------------------!
!                                                                         !
!  Determine the coordinates of the south-west corner of a map and the map !
!  extent by entering a given OS map name                                 !
!                                                                         !
!-------------------------------------------------------------------------!
  getOSmapInfo := proc(map_name: string -> OS_map_info)
  begin
     let x_swc := 0. ; let y_swc := 0.
     let series := ""
     let mapscale := 0.
     let side_length := 0.
     if letter(map_name(1|1)) then
     begin
        case map_name(1|1) of
           "T" : { x_swc := x_swc +  500000. }
           "N" : { y_swc := y_swc +  500000. }
           "H" : { y_swc := y_swc + 1000000. }
           "S" : { }
        default : { writeString("The given map name is invalid.") }
        let k := stringToAscii(map_name(2|1)) - 64
        if k >= 10 do k := k - 1
        let i = (k - 1) rem 5
        let j = 4 - (k - 1) div 5
        x_swc := x_swc + float(i) * 100000.
        y_swc := y_swc + float(j) * 100000.
        case length(map_name) of
           2 : begin
                  series := "s_625k"
                  mapscale := 625000.
                  side_length := 100000.
  !               writeString("'nMap scale = 1 : 625000") ; newline(1)
               end
           4 : if letter(map_name(3|1)) then
               begin
                  if map_name(3|1) = "N" do  y_swc := y_swc + 50000.
                  if map_name(4|1) = "E" do  x_swc := x_swc + 50000.
                  series := "s_250k"
                  mapscale := 250000.
                  side_length := 50000.
  !               writeString("'nMap scale = 1 : 250000") ; newline(1)
               end
               else
               begin
                  let num = stringToInt(map_name(3|2))
```

```
                x_swc := x_swc + float(num div 10) * 10000.
                y_swc := y_swc + float(num rem 10) * 10000.
                series := "s_50k"
                mapscale := 50000.
                side_length := 20000.
!               writeString("'nMap scale = 1 : 50000") ; newline(1)
             end
          6 : if letter(map_name(5|1)) then
             begin
                let num = stringToInt(map_name(3|2))
                x_swc := x_swc + float(num div 10) * 10000.
                y_swc := y_swc + float(num rem 10) * 10000.
                if map_name(5|1) = "N" do  y_swc := y_swc + 5000.
                if map_name(6|1) = "E" do  x_swc := x_swc + 5000.
                series := "s_10k"
                mapscale := 10000.
                side_length := 5000.
!               writeString("'nMap scale = 1 : 10000") ; newline(1)
             end
             else
             begin
                let num = stringToInt(map_name(3|4))
                x_swc := x_swc + float(num div 100) * 1000.
                y_swc := y_swc + float(num rem 100) * 1000.
                series := "s_2500"
                mapscale := 2500.
                side_length := 1000.
!               writeString("'nMap scale = 1 : 2500") ; newline(1)
             end
          8 : begin
                let num = stringToInt(map_name(3|4))
                x_swc := x_swc + float(num div 100) * 1000.
                y_swc := y_swc + float(num rem 100) * 1000.
                if map_name(7|1) = "N" do  y_swc := y_swc + 500.
                if map_name(8|1) = "E" do  x_swc := x_swc + 500.
                series := "s_1250"
                mapscale := 1250.
                side_length := 500.
!               writeString("'nMap scale = 1 : 1250") ; newline(1)
             end
       default : { writeString("The given map name is invalid.") }
    end
    else
    begin
       for i = 1 to length(map_name) do
       begin
          if ~digit(map_name(i|1)) do
                      { writeString("The given map name is invalid.") }
       end
       case length(map_name) of
          4 : begin
                x_swc := float(stringToInt(map_name(1|1))) * 100000. +
                         float(stringToInt(map_name(3|1)) - 1) * 25000.
                y_swc := float(stringToInt(map_name(2|1))) * 100000. +
                         float(stringToInt(map_name(4|1)) - 1) * 25000.
                series := "boundary_line"
                mapscale := 10000.
                side_length := 25000.
                newline(1)
!               writeString(map_name ++ " is a boundary_line map.");
!               newline(1)
             end
          6 : begin
                let num = stringToInt(map_name)
                x_swc := float(num div 1000) * 1000.
                y_swc := float(num rem 1000) * 1000.
```

```
                series := "oscar"
                mapscale := 10000.
                side_length := 5000.
  !             writeString("'nMap scale = 1 : 10000") ; newline(1)
  !             writeString(map_name ++ " is an OSCAR map."); newline(1)
              end
        default : { writeString("The given map name is invalid.") }
    end
    let extent = Extent(x_swc,y_swc,side_length,side_length)
    let os_map_info = OS_map_info(map_name,series,mapscale,extent)
    os_map_info
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Determine a set of map names that represent different OS map series   !
!   encompassing a given point                                            !
!                                                                         !
!-------------------------------------------------------------------------!
  getOSmapName := proc(point: XY -> OS_map_name)
  begin
    let x0 = truncate(point(x))
    let y0 = truncate(point(y))
    let i = x0 div 100000
    let j = y0 div 100000
    let k = (4 - j rem 5) * 5 + (i rem 5) + 1
    let s_625k = (if 0 <= i and i <= 4 then
                      case j of
                       0,1,2,3,4 : "S"
                       5,6,7,8,9 : "N"
                         default : "H"
                  else  "T" )  ++  (if k <= 8 then asciiToString(k+64)
                                    else asciiToString(k+65))
    let x1 = x0 rem 100000
    let y1 = y0 rem 100000
    let s_250k = s_625k ++ (if y1 div 50000 = 0 then "S" else "N") ++
                 (if x1 div 50000 = 0 then "W" else "E")
    let s_50k = s_625k ++ iformat(x1 div 20000 * 20 + y1 div 20000 * 2)
    let s_10k = s_625k ++ iformat(x1 div 10000 * 10 + y1 div 10000) ++
                (if (y1 rem 10000) div 5000 = 0 then "S" else "N")  ++
                (if (x1 rem 10000) div 5000 = 0 then "W" else "E")
    let s_2500 = s_625k ++ iformat(x1 div 1000 * 100 + y1 div 1000)
    let s_1250 = s_2500 ++
                 (if (y1 rem 1000) div 500 = 0 then "S" else "N") ++
                 (if (x1 rem 1000) div 500 = 0 then "W" else "E")
    let oscar = iformat((x0 div 5000) * 5000 + (y0 div 5000) * 5)
    let boundary_line = iformat(i * 10000 + j * 1000 +
                        (x1 div 25000 + 1) * 10 + (y1 div 25000 + 1))
    let os_map_name = OS_map_name(s_625k,s_250k,s_50k,s_10k,s_2500,s_1250,
                                  oscar,boundary_line)
    os_map_name
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Read an NTF 1:625000 map file and construct a basemap                 !
!                                                                         !
!-------------------------------------------------------------------------!
  ntf625kToBasemap := proc(fn: string -> Basemap)
  begin
    ! initialise data structures
    let ln_pid_point := m_empty[Point_id,LN_point](eq_int,lt_int)
    let ln_lid_line := m_empty[Line_id,LN_line](eq_int,lt_int)
    let ln_gid_geometry:= m_empty[Geom_id,LN_geometry](eq_int,lt_int)
    let ln_aid_attribute := m_empty[Attr_id,LN_attribute](eq_int,lt_int)
    let ln_kid_link := m_empty[Link_id,LN_link](eq_int,lt_int)
```

```
    let ln_nid_node := m_empty[Node_id,LN_node](eq_int,lt_int)
    let ln_tid_text := m_empty[Text_id,LN_text](eq_int,lt_int)
    let fcd := m_empty[FC,FD](eq_str,lt_str)
    ! read an NTF file
    let inputfile := open(fn,0)
    if inputfile = nilfile do
       { writeString("The file " ++ fn ++ " cannot be opened'n") }
    let fileEnv := makeReadEnv(inputfile)
    let fileString := use fileEnv with
                                    readLine: proc( -> string) in readLine
    let eoi := use fileEnv with endOfInput:proc( -> bool) in endOfInput
    writeString("'nReading the file and constructing a basemap, waiting
...");
    space(2)
    let wait_count := 0
    ! Volume Header Record (01)
    let VHR = fileString()
    ! Database Header Record (02)
    let DHR1 = fileString();  let DHR2 = fileString()
    ! Attribute Description Record (40)
    let ADR := fileString()
    while ADR(1|2)  = "40"   do
    begin
       ADR := fileString()
       if ADR(length(ADR) - 1 | 1) = "1" do { let ADR1 = fileString() }
    end
    ! Feature Classification Record (05)
    let FCR := ADR
    while FCR(1|2)  = "05"  do
    begin
       let fc := FCR(3|4)
       let fd := FCR(37| length(FCR) - 38)
       m_isu_insert[FC,FD](fcd,fc,fd)
       FCR := fileString()
    end
    !
    ! Section Header Record (07)
    !
    let SHR1 = FCR
    ! section of data ordered
    let sect_reference = SHR1(3|10)
    ! length of xy coord fields
    let xylen = stringToInt(SHR1(15|5))
    ! multiplies coords by xy_mult/1000
    let xy_mult =  float(stringToInt(SHR1(21|10)))/1000.
    ! multiplies height by z_mult/1000
    let z_mult = float(stringToInt(SHR1(37|10)))/1000.
    ! Eastings and Northings of map origin
    let x_orig = float(stringToInt(SHR1(47|10)))
    let y_orig = float(stringToInt(SHR1(57|10)))
    ! Continuation record
    ! Map coverage (Local coordinates)
    let SHR2 = fileString()
    let x_local_min = float(stringToInt(SHR2(3|10)))
    let y_local_min = float(stringToInt(SHR2(13|10)))
    let x_local_max = float(stringToInt(SHR2(23|10)))
    let y_local_max = float(stringToInt(SHR2(33|10)))
    ! Map coverage (National Grid coordinates)
    let x_min = x_orig + x_local_min
    let y_min = y_orig + y_local_min
    let x_max = x_orig + x_local_max
    let y_max = y_orig + y_local_max
    !
    ! Section Body Data
    !
    while ~eoi() do
```

```
begin
   !-------------------------------------------------------------!
   !                                                             !
   !  Feature Records                                            !
   !                                                             !
   !     Point Feature                                           !
   !                                                             !
   !        Point Record (15)                                    !
   !        Geometry Record (21)                                 !
   !        Attribute Record (14)                                !
   !                                                             !
   !-------------------------------------------------------------!
   let record := fileString()
   if record(1|2) = "15" do                        ! POINTREC
   begin
      let point_id = stringToInt(record(3|6))      ! sequential number of
                                                   ! point record
      let geom_id := stringToInt(record(9|6))      ! sequential number of
                                                   ! [GEOMETRY1] record
      let attr_id := stringToInt(record(17|6))     ! sequential number of
      let ln_point = LN_point(geom_id,attr_id)
      m_isu_insert[Point_id,LN_point](ln_pid_point,point_id,ln_point)
      record := fileString()                       ! GEOMETRY1 (21)
      let x = float(stringToInt(record(14|6))) * xy_mult + x_orig
      let y = float(stringToInt(record(20|6))) * xy_mult + y_orig
      let gtype = 1
      let num_coord = 1
      let xy = XY(x,y)
      let xy_list := l_make[XY]()
      xy_list := l_append[XY](xy_list,xy)
      let ln_geometry = LN_geometry(gtype, num_coord, xy_list)
      m_isu_insert[Geom_id,LN_geometry](ln_gid_geometry, geom_id,
                                        ln_geometry)
      record := fileString()                       ! ATTREC
      attr_id := stringToInt(record(3|6))
      let fc := record(11|4)                       ! Feature Code
      let RB := false
      let RU := false
      let OR := 0.
      let PN := ""
      let NU := ""
      let i := 15
      let att_len := length(record)
      while i < att_len - 2 do
      begin
         let val_type := record(1|2)
         i := i + 2
         case val_type of
         "RB"  :  { RB := true; i := i + 1 }
         "RU"  :  { RU := true; i := i + 1 }
         "OR"  :  { OR := float(stringToInt(record(i|4)))/10.;
                                                    i := i + 4 }
         "PN"  :  { while record(i|1) ~= "\" do
                    begin
                       PN := PN ++ record(i|1)
                       i := i + 1
                    end }
         default :  { }
      end
      let ln_attr_ssm := LN_attr_ssm(RB,RU,OR,PN,NU)
      let ln_attr := LN_attr(small_scale_map: ln_attr_ssm)
      let ln_attribute := LN_attribute(fc,ln_attr)
      m_isu_insert[Attr_id,LN_attribute](ln_aid_attribute,attr_id,
                                         ln_attribute)
   end
   !-------------------------------------------------------------!
```

```
!                                                                    !
!  Line Feature                                                      !
!                                                                    !
!     Line Record (23                                                !
!     Geometry Record (21)                                           !
!     Geometry Continuation Record (21)                              !
!              .                                                     !
!              .                                                     !
!     Attribute Record (14)                                          !
!                                                                    !
!--------------------------------------------------------------------!
if record(1|2) = "23" do                          ! LINEREC
begin
   wait_count := wait_count + 1
   waitSymbol(wait_count)
   let line_id = stringToInt(record(3|6))         ! line identity (I6)
   let geom_id := stringToInt(record(9|6))
   let attr_id := stringToInt(record(17|6))
   let gtype = 2                                  ! line type
   let ln_line = LN_line(geom_id,attr_id)
   m_isu_insert[Line_id,LN_line](ln_lid_line,line_id,ln_line)
   record := fileString()                         ! GEOMETRY1 (21)
   let num_coord = stringToInt(record(10|4))!  number of coordinate
                                                  ! pairs; in the range
                                                  ! 0002 to 9999
   let line_len := length(record)
   let line_string := record(14|(line_len - 15))
   while record(line_len - 1 | 1) = "1" do
   begin
      record := fileString()                      ! GEOMETRY1 (21)
      line_len := length(record)
      line_string := line_string ++ record(3| (line_len - 4))
   end
   let xy_list := l_make[XY]()
   let n := 0
   while  n < num_coord do
   begin
      let xy_string = line_string(1+13*n|12)
      let x = float(stringToInt(xy_string(1|6))) * xy_mult + x_orig
      let y = float(stringToInt(xy_string(7|6))) * xy_mult + y_orig
      let xy = XY(x,y)
      xy_list := l_prepend[XY](xy,xy_list)
      n := n + 1
   end
   xy_list := l_reverse[XY](xy_list)
   let ln_geometry = LN_geometry(gtype,num_coord,xy_list)
   m_isu_insert[Geom_id,LN_geometry](ln_gid_geometry,geom_id,
                                     ln_geometry)
   record := fileString()                         ! ATTREC
   attr_id := stringToInt(record(3|6))
   let fc := record(11|4)                         ! Feature Code
   let RB := false
   let RU := false
   let OR := 0.0
   let PN := ""
   let NU := ""
   let i := 15
   let att_len := length(record)
   while i < att_len - 2 do
   begin
      let val_type := record(i|2)
      i := i + 2
      case val_type of
      "PN"  : { while record(i|1) ~= "\" do
                begin
                   PN := PN ++ record(i|1)
```

```
                    i := i + 1
                end }
        "NU"   : { while record(i|1) ~= "\" do
                begin
                    NU := NU ++ record(i|1)
                    i := i + 1
                end }
        default: { }
    end
    let ln_attr_ssm := LN_attr_ssm(RB,RU,OR,PN,NU)
    let ln_attr := LN_attr(small_scale_map: ln_attr_ssm)
    let ln_attribute := LN_attribute(fc,ln_attr)
    m_isu_insert[Attr_id,LN_attribute](ln_aid_attribute,attr_id,
                                       ln_attribute)
end
!-------------------------------------------------------------!
!                                                             !
!  TEXT (NAME)                                                !
!                                                             !
!     Text Record (43)                                        !
!     Text Position Record (44)                               !
!     Text Representation Record (45)                         !
!                                                             !
!-------------------------------------------------------------!
if record(1|2) = "43" do                    ! TEXTREC (43)
begin
    wait_count := wait_count + 1
    waitSymbol(wait_count)
    let text_id = stringToInt(record(3|6))
    let attr_id = stringToInt(record(25|6))
    record := fileString()                  ! TEXTPOS (44)
    let geom_id = stringToInt(record(17|6))
    record := fileString()                  ! TEXTREP (45)
    let font =  stringToInt(record(9|4))
    let text_ht = float(stringToInt(record(13|3)))/10.! in mm
    let dig_postn = stringToInt(record(16|1))
    let orient = float(stringToInt(record(17|4)))/10. ! 0.1 of degree
    let ln_text := LN_text(geom_id,attr_id,text_ht,orient,font,
                           dig_postn)
    m_isu_insert[Text_id,LN_text](ln_tid_text,text_id,ln_text)
end
!-------------------------------------------------------------!
!                                                             !
!  Node Detail                                                !
!                                                             !
!     Node Record (16)                                        !
!     Node Continuation Record (16)                           !
!                                                             !
!-------------------------------------------------------------!
if record(1|2) = "16" do                     ! NODREC
begin
    wait_count := wait_count + 1
    waitSymbol(wait_count)
    let node_id = stringToInt(record(3|6))   ! sequential number of
                                             ! node record
    let geom_id_of_node = stringToInt(record(9|6)) ! identity of a
          ! [GEOMETRY1] record containing the position of the node
    let num_links = stringToInt(record(15|4)) ! the maximum number
                              ! of links that meet at the node is 9
    let link_list := l_make[Link]()
    let k := 19
    for i = 1 to num_links do
    begin
       ! node record, with num_links more than 5, requires
       ! a continuation record.
       if i = 6 do { record := fileString(); k := 3 }
```

```
            let direction = stringToInt(record(k|1))
            let geom_id_of_link = stringToInt(record(k+1|6))
            let orient = float(stringToInt(record(k+7|4)))/10.0
            let level = stringToInt(record(k+11|1))
            k := k + 12
            let link = Link(direction,geom_id_of_link,orient,level)
            link_list := l_prepend[Link](link,link_list)
         end
         link_list := l_reverse[Link](link_list)
         let ln_node := LN_node(geom_id_of_node,num_links,link_list)
         m_isu_insert[Node_id,LN_node](ln_nid_node,node_id,ln_node)
      end
   end
   writeString("'b'b |")
   let close_index = close(inputfile)
   let ln_tid_txt := LN_tid_txt(ln_tid_text: ln_tid_text)
   let ln_dm := LN_DM(ln_pid_point,ln_lid_line,ln_gid_geometry,
                      ln_aid_attribute,ln_kid_link,ln_nid_node,
                      ln_tid_txt,fcd)
   let basemap_dm = Basemap_DM(link_node: ln_dm)
   let basemap = Basemap(basemap_dm)
   basemap
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Read an NTF 1:250000 map file and construct a basemap                 !
!                                                                         !
!-------------------------------------------------------------------------!
ntf250kToBasemap := proc(fn: string -> Basemap)
begin
   ! initialise data structures
   let ln_pid_point := m_empty[Point_id,LN_point](eq_int,lt_int)
   let ln_lid_line := m_empty[Line_id,LN_line](eq_int,lt_int)
   let ln_gid_geometry:= m_empty[Geom_id,LN_geometry](eq_int,lt_int)
   let ln_aid_attribute := m_empty[Attr_id,LN_attribute](eq_int,lt_int)
   let ln_kid_link := m_empty[Link_id,LN_link](eq_int,lt_int)
   let ln_nid_node := m_empty[Node_id,LN_node](eq_int,lt_int)
   let fcd := m_empty[FC,FD](eq_str,lt_str)
   ! read a NTF file
   let inputfile := open(fn,0)
   if inputfile = nilfile do
       { errorAbort("The file " ++ fn ++" cannot be opened.") }
   let fileEnv := makeReadEnv(inputfile)
   let fileString := use fileEnv with
                                readLine:proc( -> string) in readLine
   let eoi := use fileEnv with endOfInput:proc( -> bool) in endOfInput
   writeString("'nReading the file and constructing a basemap, waiting
...");
   space(2)
   let wait_count := 0
   ! Volume Header Record (01)
   let VHR = fileString()
   ! Database Header Record (02)
   let DHR1 = fileString();  let DHR2 = fileString()
   ! Attribute Description Record (40)
   let ADR := fileString()
   while ADR(1|2)  = "40" do
   begin
      ADR := fileString()
      if ADR(length(ADR) - 1 | 1) = "1" do { let ADR1 = fileString() }
   end
   ! Feature Classification Record (05)
   let FCR := ADR
   while FCR(1|2)  = "05"  do
   begin
```

```
    let fc := FCR(3|4)
    let fd := FCR(37| length(FCR) - 38)
    m_isu_insert[FC,FD](fcd,fc,fd)
    FCR := fileString()
end
!
! Section Header Record (07)
!
let SHR1 = FCR
! section of data ordered
let sect_reference = SHR1(3|10)
! length of xy coord fields
let xylen = stringToInt(SHR1(15|5))
! multiplies coords by xy_mult/1000
let xy_mult =  float(stringToInt(SHR1(21|10)))/1000.
! multiplies height by z_mult/1000
let z_mult = float(stringToInt(SHR1(37|10)))/1000.
! Eastings and Northings of map origin
let x_orig = float(stringToInt(SHR1(47|10)))
let y_orig = float(stringToInt(SHR1(57|10)))
! Continuation record
! Map coverage (Local coordinates)
let SHR2 = fileString()
let x_local_min = float(stringToInt(SHR2(3|10)))
let y_local_min = float(stringToInt(SHR2(13|10)))
let x_local_max = float(stringToInt(SHR2(23|10)))
let y_local_max = float(stringToInt(SHR2(33|10)))
! Map coverage (National Grid coordinates)
let x_min = x_orig + x_local_min
let y_min = y_orig + y_local_min
let x_max = x_orig + x_local_max
let y_max = y_orig + y_local_max
!
! Section Body Data
!
while ~eoi() do
begin
    !-------------------------------------------------------------!
    !                                                             !
    !  Feature Records                                            !
    !                                                             !
    !     Point Feature                                           !
    !                                                             !
    !        Point Record (15)                                    !
    !        Geometry Record (21)                                 !
    !        Attribute Record (14)                                !
    !                                                             !
    !-------------------------------------------------------------!
    let record := fileString()
    if record(1|2) = "15" do                        ! POINTREC
    begin
        let point_id = stringToInt(record(3|6))     ! sequential number of
                                                    ! point record
        let geom_id := stringToInt(record(9|6))     ! sequential number of
                                                    ! [GEOMETRY1] record
        let attr_id := stringToInt(record(17|6))    ! sequential number of
        let ln_point = LN_point(geom_id,attr_id)
        m_isu_insert[Point_id,LN_point](ln_pid_point,point_id,ln_point)
        record := fileString()                       ! GEOMETRY1 (21)
        let x = float(stringToInt(record(14|5))) * xy_mult + x_orig
        let y = float(stringToInt(record(19|5))) * xy_mult + y_orig
        let gtype = 1
        let num_coord = 1
        let xy = XY(x,y)
        let xy_list := l_make[XY]()
        xy_list := l_append[XY](xy_list,xy)
```

```
    let ln_geometry = LN_geometry(gtype,num_coord,xy_list)
    m_isu_insert[Geom_id,LN_geometry](ln_gid_geometry,geom_id,
                                      ln_geometry)
    record := fileString()                          ! ATTREC
    attr_id := stringToInt(record(3|6))
    let fc := record(11|4)                          ! Feature Code
    let RB := false
    let RU := false
    let OR := 0.
    let PN := ""
    let NU := ""
    let i := 15
    let att_len := length(record)
    while i < att_len - 2 do
    begin
       let val_type := record(1|2)
       i := i + 2
       case val_type of
       "RB"  :  { RB := true; i := i + 1 }
       "RU"  :  { RU := true; i := i + 1 }
       "OR"  :  { OR := float(stringToInt(record(i|4)))/10.;
                                                   i := i + 4 }
       "PN"  :  { while record(i|1) ~= "\" do
                  begin
                     PN := PN ++ record(i|1)
                     i := i + 1
                  end }
       default :  { }
    end
    let ln_attr_ssm := LN_attr_ssm(RB,RU,OR,PN,NU)
    let ln_attr := LN_attr(small_scale_map: ln_attr_ssm)
    let ln_attribute := LN_attribute(fc,ln_attr)
    m_isu_insert[Attr_id,LN_attribute](ln_aid_attribute,attr_id,
                                       ln_attribute)
end
!-------------------------------------------------------------!
!                                                             !
!  Line Feature                                               !
!                                                             !
!      Line Record (23)                                       !
!      Geometry Record (21)                                   !
!      Geometry Continuation Record (21)                      !
!               .                                             !
!               .                                             !
!      Attribute Record (14)                                  !
!                                                             !
!-------------------------------------------------------------!
if record(1|2) = "23" do                        ! LINEREC
begin
    wait_count := wait_count + 1
    waitSymbol(wait_count)
    let line_id = stringToInt(record(3|6))      ! line identity (I6)
    let geom_id := stringToInt(record(9|6))
    let attr_id := stringToInt(record(17|6))
    let gtype = 2                               ! line type
    let ln_line = LN_line(geom_id,attr_id)
    m_isu_insert[Line_id,LN_line](ln_lid_line,line_id,ln_line)
    record := fileString()                      ! GEOMETRY1 (21)
    let num_coord = stringToInt(record(10|4))! number of coordinate
                                                ! pairs; in the range
                                                ! 0002 to 9999
    let line_len := length(record)
    let line_string := record(14|(line_len - 15))
    while record(line_len - 1 | 1) = "1" do
    begin
       record := fileString()                   ! GEOMETRY1 (21)
```

```
      line_len := length(record)
      line_string := line_string ++ record(3| (line_len - 4))
   end
   let xy_list := l_make[XY]()
   let n := 0
   while  n < num_coord do
   begin
      let xy_string = line_string(1+11*n|10)
      let x = float(stringToInt(xy_string(1|5))) * xy_mult + x_orig
      let y = float(stringToInt(xy_string(6|5))) * xy_mult + y_orig
      let xy = XY(x,y)
      xy_list := l_prepend[XY](xy,xy_list)
      n := n + 1
   end
   xy_list := l_reverse[XY](xy_list)
   let ln_geometry = LN_geometry(gtype,num_coord,xy_list)
   m_isu_insert[Geom_id,LN_geometry](ln_gid_geometry,geom_id,
                                     ln_geometry)
   record := fileString()                          ! ATTREC
   attr_id := stringToInt(record(3|6))
   let fc := record(11|4)                          ! Feature Code
   let RB := false
   let RU := false
   let OR := 0.0
   let PN := ""
   let NU := ""
   let i := 15
   let att_len := length(record)
   while i < att_len - 2 do
   begin
      let val_type := record(i|2)
      i := i + 2
      case val_type of
      "PN"  : { while record(i|1) ~= "\" do
                begin
                   PN := PN ++ record(i|1)
                   i := i + 1
                end }
      "NU"  : { while record(i|1) ~= "\" do
                begin
                   NU := NU ++ record(i|1)
                   i := i + 1
                end }
      default: { }
   end
   let ln_attr_ssm := LN_attr_ssm(RB,RU,OR,PN,NU)
   let ln_attr := LN_attr(small_scale_map: ln_attr_ssm)
   let ln_attribute := LN_attribute(fc,ln_attr)
   m_isu_insert[Attr_id,LN_attribute](ln_aid_attribute,attr_id,
                                      ln_attribute)
end
!-------------------------------------------------------------!
!                                                             !
!  Node Detail                                                !
!                                                             !
!     Node Record (16)                                        !
!     Node Continuation Record (16)                           !
!                                                             !
!-------------------------------------------------------------!
if record(1|2) = "16" do                        ! NODREC
begin
   wait_count := wait_count + 1
   waitSymbol(wait_count)
   let node_id = stringToInt(record(3|6))    ! sequential number of
                                             ! node record
   let geom_id_of_node = stringToInt(record(9|6)) ! identity of a
```

```
            ! [GEOMETRY1] record containing the position of the node
         let num_links = stringToInt(record(15|4)) ! number of links that
            ! meet at the node on OS 1:250000 data; the maximum is 9
         let link_list := l_make[Link]()
         let k := 19
         for i = 1 to num_links do
         begin
            if i = 6 do { record := fileString(); k := 3 }
                        ! node record with num_links more than 5,
                        ! requires a continuation record.
            let direction = stringToInt(record(k|1))
            let geom_id_of_link = stringToInt(record(k+1|6))
            let orient = float(stringToInt(record(k+7|4)))/10.0
            let level = stringToInt(record(k+11|1))
            k := k + 12
            let link = Link(direction,geom_id_of_link,orient,level)
            link_list := l_prepend[Link](link,link_list)
         end
         link_list := l_reverse[Link](link_list)
         let ln_node := LN_node(geom_id_of_node,num_links,link_list)
         m_isu_insert[Node_id,LN_node](ln_nid_node,node_id,ln_node)
      end
   end
   writeString("'b'b |")
   let close_index = close(inputfile)
   let ln_tid_txt := LN_tid_txt(none: nil)
   let ln_dm := LN_DM(ln_pid_point,ln_lid_line,ln_gid_geometry,
                      ln_aid_attribute,ln_kid_link,ln_nid_node,
                      ln_tid_txt,fcd)
   let basemap_dm = Basemap_DM(link_node: ln_dm)
   let basemap = Basemap(basemap_dm)
   basemap
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Read an NTF contour map file and construct a basemap                  !
!                                                                         !
!-------------------------------------------------------------------------!
ntfcontourToBasemap := proc(fn: string -> Basemap)
begin
   ! initialise data structures
   let sp_pid_point := m_empty[Point_id,SP_point](eq_int,lt_int)
   let sp_lid_line := m_empty[Line_id,SP_line](eq_int,lt_int)
   let fcd := m_empty[FC,FD](eq_str,lt_str)
   ! read an NTF file
   let inputfile := open(fn,0)
   if inputfile = nilfile do
       { errorAbort("The file " ++ fn ++" cannot be opened.") }
   let fileEnv := makeReadEnv(inputfile)
   let fileString := use fileEnv with
                                 readLine:proc( -> string) in readLine
   let eoi := use fileEnv with endOfInput:proc( -> bool) in endOfInput
   writeString("'nReading the file and constructing a basemap, waiting
...");
   space(2)
   let wait_count := 0
   ! Volume Header Record (01)
   let VHR = fileString()
   ! Database Header Record (02)
   let DHR1 = fileString();  let DHR2 = fileString()
   ! Attribute Description Record (40)
   let ADR := fileString()
   while ADR(1|2)  = "40"   do
   begin
      ADR := fileString()
```

```
   if ADR(length(ADR) - 1 | 1) = "1" do { let ADR1 = fileString() }
end
! Feature Classification Record (05)
let FCR := ADR
while FCR(1|2)  = "05"  do
begin
   let fc := FCR(3|4)
   let fd := FCR(37| length(FCR) - 38)
   m_isu_insert[FC,FD](fcd,fc,fd)
   FCR := fileString()
end
!
! Section Header Record (07)
!
let SHR1 = FCR
! section of data ordered
let sect_reference = SHR1(3|10)
! length of xy coord fields
let xyln = stringToInt(SHR1(15|5))
! multiplies coords by xy_mult/1000
let xy_mult =  float(stringToInt(SHR1(21|10)))/1000.
! multiplies height by z_mult/1000
let z_mult = float(stringToInt(SHR1(37|10)))/1000.
! Eastings and Northings of map origin
let x_orig = float(stringToInt(SHR1(47|10)))
let y_orig = float(stringToInt(SHR1(57|10)))
! continuation record
let SHR2 = fileString()
! Map coverage (Local coordinates)
let x_local_min = float(stringToInt(SHR2(3|10)))
let y_local_min = float(stringToInt(SHR2(13|10)))
let x_local_max = float(stringToInt(SHR2(23|10)))
let y_local_max = float(stringToInt(SHR2(33|10)))
! Map coverage (National Grid coordinates)
let x_min = x_orig + x_local_min
let y_min = y_orig + y_local_min
let x_max = x_orig + x_local_max
let y_max = y_orig + y_local_max
!
! Section Body Data
!
while ~eoi() do
begin
   !-------------------------------------------------------------!
   !                                                             !
   !  Feature Records                                            !
   !                                                             !
   !     Point Feature                                           !
   !                                                             !
   !        Point Record (15)                                    !
   !        Geometry Record (21)                                 !
   !                                                             !
   !-------------------------------------------------------------!
   let record := fileString()
   if record(1|2) = "15" do  ! POINTREC
   begin
      wait_count := wait_count + 1
      waitSymbol(wait_count)
      let point_id = stringToInt(record(3|6))
      let ht = float(stringToInt(record(11|6)))
      let fc = record(17|4)
      record := fileString()   ! GEOMETRY1 (21)
      let x = float(stringToInt(record(14|10))) * xy_mult + x_orig
      let y = float(stringToInt(record(24|10))) * xy_mult + y_orig
      let xy = XY(x,y)
      let attribute = SP_point_attr(contour: ht)
```

```
            let sp_point = SP_point(xy,fc,attribute)
            m_isu_insert[Point_id,SP_point](sp_pid_point,point_id,sp_point)
         end
         !-------------------------------------------------------------!
         !                                                             !
         !   Line Feature                                              !
         !                                                             !
         !      Line Record (23)                                       !
         !      Geometry Record (21)                                   !
         !      Geometry Continuation Record (21)                      !
         !              .                                              !
         !              .                                              !
         !                                                             !
         !-------------------------------------------------------------!
         if record(1|2) = "23" do                          ! LINEREC
         begin
            wait_count := wait_count + 1
            waitSymbol(wait_count)
            let line_id = stringToInt(record(3|6))         ! line identity (I6)
            let ht = float(stringToInt(record(11|6)))
            let fc = record(17|4)
            record := fileString()                         ! GEOMETRY1 (21)
            let num_coord = stringToInt(record(10|4))
            let line_len := length(record)
            let line_string := record(14|(line_len - 15))
            while record(line_len - 1 | 1) = "1" do
            begin
               record := fileString()                      ! GEOMETRY1 (21)
               line_len := length(record)
               line_string := line_string ++ record(3| (line_len - 4))
            end
            let xy_list := l_make[XY]()
            let n := 0
            while  n < num_coord do
            begin
               let xy_string = line_string(1+21*n|20)
               let x = float(stringToInt(xy_string(1|10))) *
                                                    xy_mult + x_orig
               let y = float(stringToInt(xy_string(11|10))) *
                                                    xy_mult + y_orig
               let xy = XY(x,y)
               xy_list := l_prepend[XY](xy,xy_list)
               n := n + 1
            end
            xy_list := l_reverse[XY](xy_list)
            let attribute = SP_line_attr(contour: ht)
            let sp_line = SP_line(xy_list,fc,attribute)
            m_isu_insert[Line_id,SP_line](sp_lid_line,line_id,sp_line)
         end
      end
      writeString("'b'b |")
      let close_index = close(inputfile)
      let sp_pid_txt = SP_tid_txt(none: nil)
      let sp_dm = SP_DM(sp_pid_point,sp_lid_line,sp_pid_txt,fcd)
      let basemap_dm = Basemap_DM(spaghetti: sp_dm)
      let basemap = Basemap(basemap_dm)
      basemap
   end

!-----------------------------------------------------------------------------!
!                                                                             !
!   Read an NTF boundaryline file and construct a basemap                     !
!                                                                             !
!-----------------------------------------------------------------------------!
   ntfblToBasemap := proc(fn: string -> Basemap)
   begin
```

```
    ! initialise data structures
    let pb_gid_geometry := m_empty[Geom_id,PB_geometry](eq_int,lt_int)
    let pb_aid_attribute := m_empty[Attr_id,PB_attribute](eq_int,lt_int)
    let pb_polyid_polygon := m_empty[Poly_id,PB_polygon](eq_int,lt_int)
    let pb_cid_chain := m_empty[Chain_id,PB_chain](eq_int,lt_int)
    let pb_cpolyid_cpolygon := m_empty[Cpoly_id,PB_cpolygon](eq_int,lt_int)
    let pb_collid_collection :=
                        m_empty[Coll_id,PB_collection](eq_int,lt_int)
    let fcd := m_empty[FC,FD](eq_str,lt_str)
    ! read an NTF file
    let inputfile := open(fn,0)
    if inputfile = nilfile do
        { errorAbort("The file " ++ fn ++" cannot be opened.") }
    let fileEnv := makeReadEnv(inputfile)
    let fileString := use fileEnv with
                                readLine: proc( -> string) in readLine
    let eoi := use fileEnv with endOfInput:proc( -> bool) in endOfInput
    writeString("'nReading the file and constructing a basemap, waiting
..."");
    space(2)
    let wait_count := 0
    ! Volume Header Record (01)
    let VHR = fileString()
    ! Database Header Record (02)
    let DHR1 = fileString();  let DHR2 = fileString()
    ! Attribute Description Record (40)
    let ADR := fileString()
    while ADR(1|2)  = "40"   do
    begin
       ADR := fileString()
       if ADR(length(ADR) - 1 | 1) = "1" do { let ADR1 = fileString() }
    end
    ! Feature Classification Record (05)
    let FCR := ADR
    while FCR(1|2)  = "05"  do
    begin
       let fc := FCR(3|4)
       let fd := FCR(37| length(FCR) - 38)
       m_isu_insert[FC,FC](fcd,fc,fd)
       FCR := fileString()
    end
    !
    ! Section Header Record (07)
    !
    let SHR1 = FCR
    ! section of data ordered
    let sect_reference = SHR1(3|10)
    ! length of xy coord fields
    let xylen = stringToInt(SHR1(15|5))
    ! multiplies coords by xy_mult/1000
    let xy_mult =  float(stringToInt(SHR1(21|10)))/1000.
    ! multiplies height by z_mult/1000
    let z_mult = float(stringToInt(SHR1(37|10)))/1000.
    ! Eastings and Northings of map origin
    let x_orig = float(stringToInt(SHR1(47|10)))
    let y_orig = float(stringToInt(SHR1(57|10)))
    ! Continuation record
    ! Map coverage (Local coordinates)
    let SHR2 = fileString()
    let x_local_min = float(stringToInt(SHR2(3|10)))
    let y_local_min = float(stringToInt(SHR2(13|10)))
    let x_local_max = float(stringToInt(SHR2(23|10)))
    let y_local_max = float(stringToInt(SHR2(33|10)))
    ! Map coverage (National Grid coordinates)
    let x_min = x_orig + x_local_min
    let y_min = y_orig + y_local_min
```

```
let x_max = x_orig + x_local_max
let y_max = y_orig + y_local_max
!
! Section Body Data
!
while ~eoi() do
begin
   ! initialise values for the variables used in Attribute Record
   let AI := 0                ! Admin_area_id
   let FC := ""               ! Feature Code
   let LK := 0                ! Link_id
   let NM := ""               ! Name
   let OP := ""               ! OPCS_Code (City,Dist & Ward only)
   let PI := 0                ! Polygon_id
   let HA := 0.0              ! Hectares (Area of polygon in tile)
   let HW := 0                ! MHW flag (0 or 1)
   let LV := 0                ! Level (Administrative area type)
   let SD := ""               ! Superseded_Date
   let CT := ""               ! Change_type
   let record := fileString()
   !-------------------------------------------------------------!
   !                                                             !
   !  Geometry (Link) Data                                       !
   !                                                             !
   !     Geometry Record (21)                                    !
   !     Geometry Continuation Record (21)                       !
   !              .                                              !
   !              .                                              !
   !     Attribute Record (14) for Geometry                      !
   !                                                             !
   !-------------------------------------------------------------!
   if record(1|2) = "21" do                    ! GEOMETRY1
   begin
      wait_count := wait_count + 1
      waitSymbol(wait_count)
      let geom_id = stringToInt(record(3|6))   ! X-ref form [POLYGON]/
                                               ! [CPOLY] (seed) and
                                               ! [CHAIN] (link)
      let gtype =  stringToInt(record(9|1))    ! "1" for seed,
                                               ! "2" for link.
      let num_coord = stringToInt(record(10|4))! number of coordinates
                                               ! pairs
      let line_len := length(record)
      let line_string := record(14|(line_len - 15))
      while record(line_len - 1 | 1) = "1" do
      begin
         record := fileString()  ! GEOMETRY1 (21) Continuation Record
         line_len := length(record)
         line_string := line_string ++ record(3| (line_len - 4))
      end
      ! Create a xy_list and add xy-coordinate pairs to the xy_list
      let xy_list := l_make[XY]()
      let n := 0
      while  n < num_coord do
      begin
         let xy_string = line_string(1+13*n|12)
         let x = float(stringToInt(xy_string(1|6))) *
                                                  xy_mult + x_orig
         let y = float(stringToInt(xy_string(7|6))) *
                                                  xy_mult + y_orig
         ! Adding a xy_coordinate pair to xy_list
         let xy = XY(x,y)
         xy_list := l_prepend[XY](xy,xy_list)
         n := n + 1
      end
      xy_list := l_reverse[XY](xy_list)
```

```
    let attr_id := stringToInt(line_string(1+13*num_coord|6))
    ! Adding a geometry record to the pb_gid_geometry
    let pb_geometry = PB_geometry(gtype,num_coord,xy_list,attr_id)
    m_isu_insert[Geom_id, PB_geometry](pb_gid_geometry, geom_id,
                                       pb_geometry)
    ! following an ATTREC Record (14)
    record := fileString()                    ! ATTREC Record for
                                              ! Geometry (Link)
    attr_id := stringToInt(record(3|6))
    let i := 9
    let att_len := length(record)
    while i < att_len - 2 do
    begin
       let val_type := record(i|2)
       i := i + 2
       case val_type of
       "FC" : { FC := record(i|4); i := i + 4 }
                                  ! See Feature Class Record
       "LK" : { LK := stringToInt(record(i|10));i := i + 10 }
                                  ! link_id
       "HW" : { HW := stringToInt(record(i|1));i := i + 1 }
                                  ! MHW flag (0 or 1)
       "CT" : { CT := record(i|2); i := i + 2 }
                                  ! Change Type
       "LV" : { LV := stringToInt(record(i|2));i := i + 2 }
                                  ! Level (Administrative Area Type)
       "SD" : { SD := record(i|8); i := i + 8 }
                                  ! YYYYMMDD ("99999999" if current)
       default :  { }
    end
    ! Adding an attribute to the attribute_table
    let pb_attribute := PB_attribute(AI,LK,PI,HW,LV,HA,
                                     FC,NM,OP,SD,CT)
    m_isu_insert[Attr_id,PB_attribute](pb_aid_attribute,attr_id,
                                       pb_attribute)
end
!-------------------------------------------------------------!
!                                                             !
!  Polygon Data                    Cpolygon Data              !
!                                                             !
!     Polygon Record (31)             Cpolygon Record (33)    !
!   * Attribute Record (14)           Attribute Record (14)   !
!     Chain Record (24)               Geometry Record (21)    !
!   * Geometry Record (21)                                    !
!                                                             !
!              * optional                                     !
!                                                             !
!-------------------------------------------------------------!
if record(1|2) = "31" or record(1|2) = "33" do
                                     ! POLYGON or CPOLY Record
begin
    let poly_type := 0
    case record(1|2) of
    "31" :   ! POLYGON Record
       begin
          wait_count := wait_count + 1
          waitSymbol(wait_count)
          poly_type := 1
          let poly_id = stringToInt(record(3|6))  ! Seed_id
          let chain_id = stringToInt(record(9|6))
          let rec_len = length(record)
          let geom_id := 0
          let attr_id := 0
          if rec_len > 16 do
          begin
             ! X-ref to (part-) polygon seed point.
```

```
            geom_id := stringToInt(record(15|6))
            ! X-ref to history and area.
            attr_id := stringToInt(record(23|6))
         end
         ! adding a polygon record to the polygon_table
         let pb_polygon = PB_polygon(chain_id,geom_id,attr_id)
         m_isu_insert[Poly_id,PB_polygon](pb_polyid_polygon,
                                          poly_id,pb_polygon)
      end
   "33" :   ! CPOLY Record
      begin
         wait_count := wait_count + 1
         waitSymbol(wait_count)
         poly_type := 3
         let cpoly_id = stringToInt(record(3|6))
         let num_parts =  stringToInt(record(9|4))
             ! (inc. containing polygon).
         let rec_len := length(record)
         let rec_string := record(13|(rec_len - 14))
         while record(rec_len - 1 | 1) = "1" do
         begin
            record := fileString()      !  Continuation Record
            rec_len := length(record)
            rec_string := rec_string ++ record(3|(rec_len - 4))
         end
         let polyid_sign_list := l_make[PB_polyid_sign]()
         let n := 0
         while n < num_parts do
         begin
            let id_string = rec_string(1+7*n|7)
            let poly_id = stringToInt(id_string(1|6))
            let sign = id_string(7|1)    ! '+' for containing poly;
                                         ! '-' for contained polys.
            let pb_polyid_sign = PB_polyid_sign(poly_id,sign)
            polyid_sign_list :=
                        l_prepend[PB_polyid_sign](pb_polyid_sign,
                                                  polyid_sign_list)
            n := n + 1
         end
         polyid_sign_list :=
                       l_reverse[PB_polyid_sign](polyid_sign_list)
         let geom_id := stringToInt(rec_string(1+7*num_parts|6))
         let attr_id := stringToInt(rec_string(1+7*num_parts+8|6))
         ! adding a cpoly record to the cpolygon_table
         let pb_cpolygon := PB_cpolygon(num_parts,polyid_sign_list,
                                        geom_id,attr_id)
         m_isu_insert[Cpoly_id,PB_cpolygon](pb_cpolyid_cpolygon,
                                            cpoly_id,pb_cpolygon)
      end
   default : {}
   ! read next record
   record := fileString()
   let rec_desc := record(1|2)
   let num = if poly_type = 3 then 2 else      ! the number of
             if rec_desc ="24" then 1 else 3   ! records following
   let count := 0                              ! the POLYGON Record
   while count < num do                        ! and CPOLY Record
   begin
      case rec_desc of
      "14" :        ! ATTREC Record for (Part-) POLYGON and CPOLY
                    ! including Foreshore and Sea [GEOMETRY1]
         begin
            let attr_id = stringToInt(record(3|6))
            let i := 9
            let att_len := length(record)
            while i < att_len - 2 do
```

```
        begin
           let val_type := record(i|2)
           i := i + 2
           case val_type of
           "FC" : { FC := record(i|4); i := i + 4 }
                          ! See Feature Class Record
                          ! Record (3901-3912)
           "PI" : { PI := stringToInt(record(i|6));i := i + 6 }
                          ! Polygon_id
           "HA" : { HA := float(stringToInt(record(i|8)))/1000.;
                                                  i := i + 8 }
                          ! Hectares of(part) polygon and
                          ! complex polygon Foreshore or Sea.
           default :  { }
        end
        ! Adding an attribute record to the attribute_table
        let pb_attribute = PB_attribute(AI,LK,PI,HW,LV,HA,
                                        FC,NM,OP,SD,CT)
        m_isu_insert[Attr_id,PB_attribute](pb_aid_attribute,
                                           attr_id,
                                           pb_attribute)
     end
  "24" :       ! CHAIN Record
     begin
        let chain_id = stringToInt(record(3|6))
        let num_parts = stringToInt(record(9|4))
            ! Number of links for (part) polygon within tile.
        let rec_len := length(record)
        let rec_string := record(13|(rec_len - 14))
        while record(rec_len - 1 | 1) = "1" do
        begin
           record := fileString()      !  Continuation Record
           rec_len := length(record)
           rec_string := rec_string ++ record(3|(rec_len - 4))
        end
        let link_list := l_make[PB_link]()
        let n := 0
        while n < num_parts do
        begin
           let id_string = rec_string(1+7*n|7)
           let geom_id = stringToInt(id_string(1|6))
                                 ! X-ref to link geometry
           let direction = stringToInt(id_string(7|1))
           let pb_link = PB_link(direction, geom_id)
           link_list := l_prepend[PB_link](pb_link,link_list)
           n := n + 1
        end
        link_list := l_reverse[PB_link](link_list)
        ! adding a chain record to the chain_table
        let pb_chain = PB_chain(num_parts,link_list)
        m_isu_insert[Chain_id,PB_chain](pb_cid_chain,chain_id,
                                        pb_chain)
     end
  "21" :       ! GEOMETRY Record (Seed)
     begin
        let geom_id = stringToInt(record(3|6))
        let gtype = 1                        ! "1" for seed
        let num_coord = 1
        let attr_id = 0                      ! unused for seeds
        let x = float(stringToInt(record(14|6))) *
                                             xy_mult + x_orig
        let y = float(stringToInt(record(20|6))) *
                                             xy_mult + y_orig
        ! Adding a geometry record to the pb_gid_geometry
        let xy = XY(x,y)
        let xy_list := l_make[XY]()
```

```
            xy_list := l_append[XY](xy_list,xy)
            let pb_geometry = PB_geometry(gtype,num_coord,xy_list,
                                          attr_id)
            m_isu_insert[Geom_id,PB_geometry](pb_gid_geometry,
                                              geom_id,pb_geometry)
         end
      default : { }
      count := count + 1
      if num > 1 and count < num do
      begin
         record := fileString()
         rec_desc := record(1|2)
      end
   end
end
!-------------------------------------------------------------!
!                                                             !
!  Collection Data                                            !
!                                                             !
!     COLLECTION Record (34)                                  !
!     ATTRIBUTE Record (14)                                   !
!                                                             !
!-------------------------------------------------------------!
if record(1|2) = "34" do   ! COLLECT Record
begin
   wait_count := wait_count + 1
   waitSymbol(wait_count)
   let coll_id = stringToInt(record(3|6))    ! Administrative Area
                                             ! ID 101001 to 800000.
   let num_parts = stringToInt(record(9|4)) ! Number of [POLYGON]
                                             ! and [CPOLY] records
   let rec_len := length(record)
   let rec_string := record(13|(rec_len - 14))
   while record(rec_len - 1 | 1) = "1" do
   begin
      record := fileString()                 ! Continuation Record
      rec_len := length(record)
      rec_string := rec_string ++ record(3|(rec_len - 4))
   end
   let polyid_list := l_make[PB_polyid]()
   let n := 0
   while n < num_parts do
   begin
      let id_string = rec_string(1+8*n|8)
      let poly_id = stringToInt(id_string(3|6))
      let poly_type = if id_string(1|2) = "31" then 1
                      else if id_string(1|2) = "33" then 3 else 0
        ! 1 : simple polygon; 3 : complex polygon; 0 : wrong type
      let pb_polyid = PB_polyid(poly_id, poly_type)
      polyid_list := l_prepend[PB_polyid](pb_polyid,polyid_list)
      n := n + 1
   end
   polyid_list := l_reverse[PB_polyid](polyid_list)
   let attr_id := stringToInt(rec_string(1+8*num_parts+2|6))
   ! adding a collection record to the collection_table
   let pb_collection = PB_collection(num_parts,polyid_list,
                                     attr_id)
   m_isu_insert[Coll_id,PB_collection](pb_collid_collection,
                                       coll_id,pb_collection)
   ! following an ATTREC Record (14)
   record := fileString()                    ! ATTREC Record for
                                             ! Name [COLLECTION]
   attr_id := stringToInt(record(3|6))
   rec_len := length(record)
   rec_string := record(9|(rec_len - 10))
   while record(rec_len - 1 | 1) = "1" do
```

```
            begin
               record := fileString()                    ! Continuation Record
               rec_len := length(record)
               rec_string := rec_string ++ record(3|(rec_len - 4))
            end
            let att_len = length(rec_string)
            let i := 1
            while i < att_len - 2 do
            begin
               let val_type := rec_string(i|2)
               i := i + 2
               case val_type of
               "AI" : { AI := stringToInt(rec_string(i|6)); i := i + 6 }
                                                  ! Admin Area Identifier
               "OP" : { OP := rec_string(i|6); i := i + 6 }
                                                  ! OPCS code
               "NM" : { while rec_string(i|1) ~= "\" do
                        begin
                           NM := NM ++ rec_string(i|1)
                           i := i + 1
                        end }
                                                  ! Admin Area Name or 'sea'
               "CT" : { CT := rec_string(i|2); i := i + 2 }
                                                  ! Change Type
               "SD" : { SD := rec_string(i|8); i := i + 8 }
                                                  ! YYYYMMDD
               default : { }
            end
            ! Adding an attribute to the attribute_table
            let pb_attribute := PB_attribute(AI,LK,PI,HW,LV,HA,
                                             FC,NM,OP,SD,CT)
            m_isu_insert[Attr_id,PB_attribute](pb_aid_attribute,attr_id,
                                               pb_attribute)
         end
      end
      writeString("'b'b |")
      let close_index = close(inputfile)
      let pb_dm := PB_DM(pb_collid_collection,pb_cpolyid_cpolygon,
                         pb_polyid_polygon,pb_cid_chain,pb_gid_geometry,
                         pb_aid_attribute,fcd)
      let basemap_dm = Basemap_DM(polygon_based: pb_dm)
      let basemap = Basemap(basemap_dm)
      basemap
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Read an NTF landline file and construct a basemap                     !
!                                                                         !
!-------------------------------------------------------------------------!
   ntfllToBasemap := proc(fn: string -> Basemap)
   begin
      ! initialise data structures
      let sp_pid_point := m_empty[Point_id,SP_point](eq_int,lt_int)
      let sp_lid_line := m_empty[Line_id,SP_line](eq_int,lt_int)
      let sp_tid_text := m_empty[Text_id,SP_text](eq_int,lt_int)
      let fcd := m_empty[FC,FD](eq_str,lt_str)
      ! read an NTF file
      let inputfile := open(fn,0)
      if inputfile = nilfile do
           { errorAbort("The file " ++ fn ++" cannot be opened.") }
      let fileEnv := makeReadEnv(inputfile)
      let fileString := use fileEnv with
                                  readLine:proc( -> string) in readLine
      let eoi := use fileEnv with endOfInput:proc( -> bool) in endOfInput
      let peekFileChar := use fileEnv with
```

```
                                        peekChar:proc( -> string) in peekChar
    writeString("'nReading the file and constructing a basemap, waiting
...");
    space(2)
    let wait_count := 0
    ! Volume Header Record (01)
    let VHR = fileString()
    ! Database Header Record (02)
    let DHR1 = fileString();  let DHR2 = fileString()
    ! Attribute Description Record (40)
    let ADR := fileString()
    while ADR(1|2)  = "40"   do
    begin
       ADR := fileString()
       if ADR(length(ADR) - 1 | 1) = "1" do { let ADR1 = fileString() }
    end
    ! Feature Classification Record (05)
    let FCR := ADR
    while FCR(1|2)  = "05"  do
    begin
       let fc := FCR(3|4)
       let fd := FCR(37| length(FCR) - 38)
       m_isu_insert[FC,FD](fcd,fc,fd)
       FCR := fileString()
    end
    !
    ! Section Header Record (07)
    !
    let SHR1 = FCR
    ! section of data ordered
    let sect_reference = SHR1(3|10)
    ! length of xy coord fields
    let xyln = stringToInt(SHR1(15|5))
    ! multiplies coords by xy_mult/1000
    let xy_mult =  float(stringToInt(SHR1(21|10)))/1000.
    ! multiplies height by z_mult/1000
    let z_mult = float(stringToInt(SHR1(37|10)))/1000.
    ! Eastings and Northings of map origin
    let x_orig = float(stringToInt(SHR1(47|10)))
    let y_orig = float(stringToInt(SHR1(57|10)))
    ! continuation record
    let SHR2 = fileString()
    let SHR3 = fileString()
    let map_scale = stringToInt(SHR3(31|9))
    let SHR4 = fileString()
    let SHR5 = fileString()
    ! Map coverage (Local coordinates)
    let side_length = case map_scale of
                      1250    :   500.0
                      2500    :  1000.0
                      10000   : 50000.0
                      default :     0.0
    ! Map coverage (National Grid coordinates)
    let x_min = x_orig
    let y_min = y_orig
    let x_max = x_orig + side_length
    let y_max = y_orig + side_length
    let distance := 0
    let point_id := 0
    let orient := 0.0
    let fc := ""
    let xy := XY(0.,0.)
    !
    ! Section Body Data
    !
    while ~eoi() do
```

```
begin
   !-------------------------------------------------------------!
   !                                                             !
   !  Feature Records                                            !
   !                                                             !
   !     Point Feature                                           !
   !                                                             !
   !        Point Record (15)                                    !
   !        Geometry Record (21)                                 !
   !        Attribute Record (14)  -- Optional                   !
   !                                                             !
   !-------------------------------------------------------------!
   let record := fileString()
   if record(1|2) = "15" do  ! POINTREC
   begin
      wait_count := wait_count + 1
      waitSymbol(wait_count)
      point_id := stringToInt(record(3|6))
      if record(9|2) = "OR" do
           { orient := float(stringToInt(record(11|6)))/10.0 }
      fc := record(17|4)
      record := fileString()   ! GEOMETRY1 (21)
      let x = float(stringToInt(record(14|6))) * xy_mult + x_orig
      let y = float(stringToInt(record(20|6))) * xy_mult + y_orig
      xy := XY(x,y)
      let landline_point_attr := Landline_point_attr(orient,distance)
      let attribute := SP_point_attr(landline: landline_point_attr)
      let sp_point := SP_point(xy,fc,attribute)
      m_isu_insert[Point_id,SP_point](sp_pid_point,point_id,sp_point)
   end
   if record(1|2) = "14" do  ! ATTREC (maybe associated with POINTREC)
   begin
      let distance := stringToInt(record(11|5))
      let landline_point_attr := Landline_point_attr(orient,distance)
      let attribute := SP_point_attr(landline: landline_point_attr)
      let sp_point := SP_point(xy,fc,attribute)
      m_isu_assign[Point_id,SP_point](sp_pid_point,point_id,sp_point)
   end
   !-------------------------------------------------------------!
   !                                                             !
   !  Line Feature                                               !
   !                                                             !
   !     Line Record (23)                                        !
   !     Geometry Record (21)                                    !
   !     Geometry Continuation Record (21)                       !
   !              .                                              !
   !              .                                              !
   !                                                             !
   !-------------------------------------------------------------!
   if record(1|2) = "23" do                    ! LINEREC
   begin
      wait_count := wait_count + 1
      waitSymbol(wait_count)
      let line_id = stringToInt(record(3|6))     ! line identity (I6)
      let fc = record(17|4)
      record := fileString()                     ! GEOMETRY1 (21)
      let num_coord = stringToInt(record(10|4))
      let line_len := length(record)
      let line_string := record(14|(line_len - 15))
      while record(line_len - 1 | 1) = "1" do
      begin
         record := fileString()                  ! GEOMETRY1 (21)
         line_len := length(record)
         line_string := line_string ++ record(3| (line_len - 4))
      end
      let xy_list := l_make[XY]()
```

```
            let n := 0
            while  n < num_coord do
            begin
               let xy_string = line_string(1+13*n|12)
               let x = float(stringToInt(xy_string(1|6))) *
                                                   xy_mult + x_orig
               let y = float(stringToInt(xy_string(7|6))) *
                                                   xy_mult + y_orig
               let xy = XY(x,y)
               xy_list := l_prepend[XY](xy,xy_list)
               n := n + 1
            end
            xy_list := l_reverse[XY](xy_list)
            let attribute = SP_line_attr(landline: nil)
            let sp_line = SP_line(xy_list,fc,attribute)
            m_isu_insert[Line_id,SP_line](sp_lid_line,line_id,sp_line)
         end
         !-------------------------------------------------------------!
         !                                                             !
         !  Text Feature                                               !
         !                                                             !
         !     Name Record (11)                                        !
         !     Name Position Record (12)                               !
         !     Geometry Record (21)                                    !
         !                                                             !
         !-------------------------------------------------------------!
         if record(1|2) = "11" do      ! NAMEREC (11)
         begin
            wait_count := wait_count + 1
            waitSymbol(wait_count)
            let text_id = stringToInt(record(3|6))
            let text_code = record(9|4)
            let text_len = stringToInt(record(13|2))
            let rec_len := length(record)
            let text_body := record(15|(rec_len - 16))
            while record(rec_len - 1 | 1) = "1" do
            begin
               record := fileString()
               rec_len := length(record)
               text_body := text_body ++ record(3| (rec_len - 4))
            end
            text_body := text_body(1 | length(text_body) - 15)
            record := fileString()   ! NAMPOSTN (12)
            let font =  stringToInt(record(3|4))
            let text_ht = float(stringToInt(record(7|3)))/10. ! in mm
            let dig_postn = stringToInt(record(10|1))
            let orient = float(stringToInt(record(11|4)))/10. ! 0.1 deg.
            record := fileString()   ! GEOMETRY1 (21)
            let x = float(stringToInt(record(14|6))) * xy_mult + x_orig
            let y = float(stringToInt(record(20|6))) * xy_mult + y_orig
            let xy = XY(x,y)
            let sp_text = SP_text(xy,text_code,text_body,text_ht,orient,
                                  font,dig_postn)
            m_isu_insert[Text_id,SP_text](sp_tid_text,text_id,sp_text)
         end
      end
      writeString("'b'b |")
      let close_index = close(inputfile)
      let sp_tid_txt := SP_tid_txt(sp_tid_text: sp_tid_text)
      let sp_dm := SP_DM(sp_pid_point,sp_lid_line,sp_tid_txt,fcd)
      let basemap_dm = Basemap_DM(spaghetti: sp_dm)
      let basemap = Basemap(basemap_dm)
      basemap
   end

!-----------------------------------------------------------------------!
```

```
!                                                                              !
!   Read an NTF OSCAR file and construct a basemap                             !
!                                                                              !
!------------------------------------------------------------------------------!
  ntfoscarToBasemap := proc(fn: string -> Basemap)
  begin
     ! initialise data structures
     let ln_pid_point := m_empty[Point_id,LN_point](eq_int,lt_int)
     let ln_lid_line := m_empty[Line_id,LN_line](eq_int,lt_int)
     let ln_gid_geometry:= m_empty[Geom_id,LN_geometry](eq_int,lt_int)
     let ln_aid_attribute := m_empty[Attr_id,LN_attribute](eq_int,lt_int)
     let ln_kid_link := m_empty[Link_id,LN_link](eq_int,lt_int)
     let ln_nid_node := m_empty[Node_id,LN_node](eq_int,lt_int)
     let ln_tid_text := m_empty[Text_id,LN_text](eq_int,lt_int)
     let fcd := m_empty[FC,FD](eq_str,lt_str)
     ! read an NTF file
     let inputfile := open(fn,0)
     if inputfile = nilfile do
         { errorAbort("The file " ++ fn ++" cannot be opened.") }
     let fileEnv := makeReadEnv(inputfile)
     let fileString := use fileEnv with
                                   readLine:proc( -> string) in readLine
     let eoi := use fileEnv with endOfInput:proc( -> bool) in endOfInput
     writeString("'nReading the file and constructing a basemap, waiting
..."");
     space(2)
     let wait_count := 0
     ! Volume Header Record (01)
     let VHR = fileString()
     ! Database Header Record (02)
     let DHR1 = fileString();  let DHR2 = fileString()
     ! Attribute Description Record (40)
     let ADR := fileString()
     while ADR(1|2)  = "40"   do
     begin
        ADR := fileString()
        if ADR(length(ADR) - 1 | 1) = "1" do { let ADR1 = fileString() }
     end
     ! Feature Classification Record (05)
     let FCR := ADR
     while FCR(1|2)  = "05"  do
     begin
        let fc := FCR(3|4)
        let fd := FCR(37| length(FCR) - 38)
        m_isu_insert[FC,FD](fcd,fc,fd)
        FCR := fileString()
     end
     !
     ! Section Header Record (07)
     !
     let SHR1 = FCR
     ! section of data ordered
     let sect_reference = SHR1(3|10)
     ! length of xy coord fields
     let xylen = stringToInt(SHR1(15|5))
     ! multiplies coords by xy_mult/1000
     let xy_mult =  float(stringToInt(SHR1(21|10)))/1000.
     ! multiplies height by z_mult/1000
     let z_mult = float(stringToInt(SHR1(37|10)))/1000.
     ! Eastings and Northings of map origin
     let x_orig = float(stringToInt(SHR1(47|10)))
     let y_orig = float(stringToInt(SHR1(57|10)))
     ! Continuation record
     ! Map coverage (Local coordinates)
     let SHR2 = fileString()
     let x_local_min = float(stringToInt(SHR2(3|10)))
```

```
let y_local_min = float(stringToInt(SHR2(13|10)))
let x_local_max = float(stringToInt(SHR2(23|10)))
let y_local_max = float(stringToInt(SHR2(33|10)))
! Map coverage (National Grid coordinates)
let x_min = x_orig + x_local_min
let y_min = y_orig + y_local_min
let x_max = x_orig + x_local_max
let y_max = y_orig + y_local_max
!
! Section Body Data
!
while ~eoi() do
begin
   !-------------------------------------------------------------!
   !                                                             !
   !  Feature Records                                            !
   !                                                             !
   !     Point Feature                                           !
   !                                                             !
   !        Point Record (15)                                    !
   !        Geometry Record (21)                                 !
   !        Attribute Record (14)                                !
   !                                                             !
   !-------------------------------------------------------------!
   let record := fileString()
   if record(1|2) = "15" do                          ! POINTREC
   begin
      wait_count := wait_count + 1
      waitSymbol(wait_count)
      let point_id = stringToInt(record(3|6))    ! sequential number of
                                                 ! point record
      let geom_id := stringToInt(record(9|6))    ! sequential number of
                                                 ! [GEOMETRY1] record
      let attr_id := stringToInt(record(17|6))   ! sequential number of
      let ln_point = LN_point(geom_id,attr_id)
      m_isu_insert[Point_id,LN_point](ln_pid_point,point_id,ln_point)
      record := fileString()                      ! GEOMETRY1 (21)
      let x = float(stringToInt(record(14|4))) * xy_mult + x_orig
      let y = float(stringToInt(record(18|4))) * xy_mult + y_orig
      let gtype = 1
      let num_coord = 1
      let xy = XY(x,y)
      let xy_list := l_make[XY]()
      xy_list := l_append[XY](xy_list,xy)
      let ln_geometry = LN_geometry(gtype,num_coord,xy_list)
      m_isu_insert[Geom_id,LN_geometry](ln_gid_geometry,geom_id,
                                        ln_geometry)
      record := fileString()                          ! ATTREC
      attr_id := stringToInt(record(3|6))
      let fc := record(11|4)                          ! Feature Code
      let SY := 0
      let LL := 0.
      let SC := ""
      let PN := ""
      let RN := ""
      let FW := ""
      let i := 15
      let att_len := length(record)
      while i < att_len - 2 do
      begin
         let val_type := record(1|2)
         i := i + 2
         case val_type of
         "SY"  :  { SY := stringToInt(record(i|6)); i := i + 6 }
         "LL"  :  { LL := float(stringToInt(record(i|5)))/10.;
                                                  i := i + 5 }
```

```
      "SC"  :  { SC := record(i|1); i := i + 1 }
      "PN"  :  { while record(i|1) ~= "\" do
                 begin
                    PN := PN ++ record(i|1)
                    i := i + 1
                 end }
      "RN"  :  { RN := record(i|8); i := i + 8 }
      "FW"  :  { FW := record(i|1); i := i + 1 }
      default :  { }
   end
   let ln_attr_oscar := LN_attr_oscar(SY,LL,SC,PN,RN,FW)
   let ln_attr := LN_attr(oscar: ln_attr_oscar)
   let ln_attribute := LN_attribute(fc,ln_attr)
   m_isu_insert[Attr_id,LN_attribute](ln_aid_attribute,attr_id,
                                      ln_attribute)
end
!-------------------------------------------------------------!
!                                                             !
!  Line Feature                                               !
!                                                             !
!     Line Record (23)                                        !
!     Geometry Record (21)                                    !
!     Geometry Continuation Record (21)                       !
!              .                                              !
!              .                                              !
!     Attribute Record (14)                                   !
!                                                             !
!-------------------------------------------------------------!
if record(1|2) = "23" do                        ! LINEREC
begin
   wait_count := wait_count + 1
   waitSymbol(wait_count)
   let line_id = stringToInt(record(3|6))       ! line identity (I6)
   let geom_id := stringToInt(record(9|6))
   let attr_id := stringToInt(record(17|6))
   let gtype = 2                                ! line type
   let ln_line = LN_line(geom_id,attr_id)
   m_isu_insert[Line_id,LN_line](ln_lid_line,line_id,ln_line)
   record := fileString()                       ! GEOMETRY1 (21)
   ! number of coordinates pairs in the range 0002 to 9999
   let num_coord = stringToInt(record(10|4))
   let line_len := length(record)
   let line_string := record(14|(line_len - 15))
   while record(line_len - 1 | 1) = "1" do
   begin
      record := fileString()                    ! GEOMETRY1 (21)
      line_len := length(record)
      line_string := line_string ++ record(3| (line_len - 4))
   end
   let xy_list := l_make[XY]()
   let n := 0
   while  n < num_coord do
   begin
      let xy_string = line_string(1+9*n|8)
      let x = float(stringToInt(xy_string(1|4))) *
                                                xy_mult + x_orig
      let y = float(stringToInt(xy_string(5|4))) *
                                                xy_mult + y_orig
      let xy = XY(x,y)
      xy_list := l_prepend[XY](xy,xy_list)
      n := n + 1
   end
   xy_list := l_reverse[XY](xy_list)
   let ln_geometry = LN_geometry(gtype,num_coord,xy_list)
   m_isu_insert[Geom_id,LN_geometry](ln_gid_geometry,geom_id,
                                     ln_geometry)
```

```
    record := fileString()                          ! ATTREC
    attr_id := stringToInt(record(3|6))
    let fc := record(11|4)                          ! Feature Code
    let SY := 0
    let LL := 0.
    let SC := ""
    let PN := ""
    let RN := ""
    let FW := ""
    let i := 15
    let att_len := length(record)
    while i < att_len - 2 do
    begin
       let val_type := record(1|2)
       i := i + 2
       case val_type of
       "SY"  :  { SY := stringToInt(record(i|6)); i := i + 6 }
       "LL"  :  { LL := float(stringToInt(record(i|5)))/10.;
                                                  i := i + 5 }
       "SC"  :  { SC := record(i|1); i := i + 1 }
       "PN"  :  { while record(i|1) ~= "\" do
                  begin
                     PN := PN ++ record(i|1)
                     i := i + 1
                  end }
       "RN"  :  { RN := record(i|8); i := i + 8 }
       "FW"  :  { FW := record(i|1); i := i + 1 }
       default :  { }
    end
    let ln_attr_oscar := LN_attr_oscar(SY,LL,SC,PN,RN,FW)
    let ln_attr := LN_attr(oscar: ln_attr_oscar)
    let ln_attribute := LN_attribute(fc,ln_attr)
    m_isu_insert[Attr_id,LN_attribute](ln_aid_attribute,attr_id,
                                       ln_attribute)
end
!-------------------------------------------------------------!
!                                                             !
!  Node Detail                                                !
!                                                             !
!     Node Record (16)                                        !
!     Node Continuation Record (16)                           !
!                                                             !
!-------------------------------------------------------------!
if record(1|2) = "16" do                        ! NODREC
begin
    wait_count := wait_count + 1
    waitSymbol(wait_count)
    ! sequential number of node record
    let node_id = stringToInt(record(3|6))
    ! [GEOMETRY1] record containing the position of the node
    let geom_id_of_node = stringToInt(record(9|6))
    ! identity of a number of links that meet at the node; the
    ! maximum is 9
    let num_links = stringToInt(record(15|4))
    let link_list := l_make[Link]()
    let k := 19
    for i = 1 to num_links do
    begin
       ! node reocrd, with num_links more than 5,
       ! requires  a continuation record.
       if i = 6 do { record := fileString(); k := 3 }
       let direction = stringToInt(record(k|1))
       let geom_id_of_link = stringToInt(record(k+1|6))
       let orient = float(stringToInt(record(k+7|4)))/10.0
       let level = stringToInt(record(k+11|1))
       k := k + 12
```

```
            let link = Link(direction,geom_id_of_link,orient,level)
            link_list := l_prepend[Link](link,link_list)
         end
         link_list := l_reverse[Link](link_list)
         let ln_node := LN_node(geom_id_of_node,num_links,link_list)
         m_isu_insert[Node_id,LN_node](ln_nid_node,node_id,ln_node)
      end
   end
   writeString("'b'b |")
   let close_index = close(inputfile)
   let ln_tid_txt := LN_tid_txt(none: nil)
   let ln_dm := LN_DM(ln_pid_point,ln_lid_line,ln_gid_geometry,
                      ln_aid_attribute,ln_kid_link,ln_nid_node,
                      ln_tid_txt,fcd)
   let basemap_dm = Basemap_DM(link_node: ln_dm)
   let basemap = Basemap(basemap_dm)
   basemap
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Store a basemap                                                       !
!                                                                         !
!-------------------------------------------------------------------------!
storeBasemap := proc(map_id: Map_id; basemap: Basemap; map_extent: Extent)
begin
   if ~m_contains[Map_id,Basemap](base_maps,map_id) then
   begin
      m_isu_insert[Map_id,Basemap](base_maps,map_id,basemap)
      let x_min = map_extent(x_min)
      let y_min = map_extent(y_min)
      let side_length = map_extent(x_range)
      let xy = XY(x_min / 100., y_min / 100.)
      let peano_key = xyToPK(xy)
      let peano = Peano(peano_key,side_length)
      m_isu_insert[Peano,Map_id](basemap_indices,peano,map_id)
   end
   else  { m_isu_assign[Map_id,Basemap](base_maps,map_id,basemap) }
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Remove a basemap                                                      !
!                                                                         !
!-------------------------------------------------------------------------!
removeBasemap := proc()
begin
   if ~m_isEmpty[Image_id,Basemap](base_maps) then
   begin
      let prtMapID = proc(map_id: Map_id; basemap: Basemap)
          { writeString(map_id);space(2); }
      writeString("'nThe map database contains following basemaps: 'n")
      m_app[Map_id,Basemap](base_maps,prtMapID)
      newline(1)
      writeString("'nEnter a map name: "); let map_id = readLine()
      if ~m_contains[Map_id,Basemap](base_maps,map_id) then
      begin
         if map_id ~= "" do
           { writeString("The database does not contain the query
basemap.'n") }
      end
      else
      begin
         writeString("Are you sure that the basemap is to be destroyed " ++
map_id  ++ "?'nPlease confirm with '"YES'" to proceed, otherwise no action
taken.'n"); let confirm = readLine()
```

```
            if confirm = "YES" do
            begin
               m_isu_remove[Map_id,Basemap](base_maps,map_id)
               writeString("Done!'n")
            end
         end
      end
      else
         { writeString("'nNo basemap available.'n") }
   end


!-------------------------------------------------------------------------!
!                                                                         !
!   Determine the MBR of a polygon                                        !
!                                                                         !
!-------------------------------------------------------------------------!
   getPolyMBR := proc(poly_id: Poly_id; pb_cid_chain: PB_cid_chain;
                      pb_gid_geometry: PB_gid_geometry -> MBR)
   begin
      let pb_chain = m_find[Chain_id,PB_chain](pb_cid_chain,poly_id)
      let link_list := pb_chain(link_list)
      let geom_id_of_link := l_first[PB_link](link_list)(geom_id_of_link)
      let xy_list := m_find[Geom_id,PB_geometry](pb_gid_geometry,
                                                 geom_id_of_link)(xy_list)
      let xy := l_first[XY](xy_list)
      let x_min := xy(x); let y_min := xy(y)
      let x_max := x_min; let y_max := y_min
      while link_list isnt empty do
      begin
         geom_id_of_link := hd[PB_link](link_list)(geom_id_of_link)
         xy_list := m_find[Geom_id,PB_geometry](pb_gid_geometry,
                                                geom_id_of_link)(xy_list)
         while xy_list isnt empty do
         begin
            xy := hd[XY](xy_list)
            let x = xy(x); let y = xy(y)
            if x < x_min do x_min := x
            if y < y_min do y_min := y
            if x > x_max do x_max := x
            if y > y_max do y_max := y
            xy_list := tl[XY](xy_list)
         end
         link_list := tl[PB_link](link_list)
      end
      let mbr = MBR(x_min,y_min,x_max,y_max)
      mbr
   end

!-------------------------------------------------------------------------!
!                                                                         !
!  Point-in-polygon test                                                  !
!                                                                         !
!  * If a point is on the boundary of a polygon, then this point is       !
!    counted as inside.                                                   !
!                                                                         !
!-------------------------------------------------------------------------!
   pointInPolygon := proc(test_pt: XY; pb_polygon: PB_polygon;
                          pb_cid_chain: Map[Chain_id,PB_chain];
                          pb_gid_geometry: Map[Geom_id,PB_geometry];
                          chain_mbr: Map[Geom_id,MBR]  -> bool)
   begin
      let in_poly := false
      let xt = test_pt(x); let yt= test_pt(y)
      let chain_id = pb_polygon(chain_id)
      let pb_chain = m_find[Chain_id,PB_chain](pb_cid_chain,chain_id)
```

```
let link_list := pb_chain(link_list)
let num_parts = pb_chain(num_parts)
let i := 1
let x_intersec_list := l_make[real]()
while link_list isnt empty do
begin
   let geom_id_of_link = hd[PB_link](link_list)(geom_id_of_link)
   ! determine whether the test point is within the mbr of a chain
   let chainmbr = m_find[Geom_id,MBR](chain_mbr,geom_id_of_link)
   if chainmbr(y_min) <= yt and yt <= chainmbr(y_max) do
   begin
      !  the joint link of the current link
      let direction = hd[PB_link](link_list)(direction)
      let k := if direction = 1 then i - 1 else i + 1
      if k < 1 or k > num_parts do k := num_parts
      let joint_link := l_nth[PB_link](pb_chain(link_list),k)
      let joint_link_xy_list =
                  m_find[Geom_id,PB_geometry](pb_gid_geometry,
                                    geom_id_of_link)(xy_list)
      let joint_link_xy_num =
                  m_find[Geom_id,PB_geometry](pb_gid_geometry,
                                    geom_id_of_link)(num_coord)
      let y0 := case joint_link(direction) of
            1 : { l_nth[XY](joint_link_xy_list,2)(y) }
            2 : { l_nth[XY](joint_link_xy_list,
                           joint_link_xy_num - 1)(y) }
      default : { l_nth[XY](joint_link_xy_list,2)(y) }
      ! the current link
      let xy_list := m_find[Geom_id,PB_geometry](pb_gid_geometry,
                                          geom_id_of_link)(xy_list)
      let num_coord = m_find[Geom_id,PB_geometry](pb_gid_geometry,
                                          geom_id_of_link)(num_coord)
      let j := 1
      let x1 := l_first[XY](xy_list)(x)
      let y1 := l_first[XY](xy_list)(y)
      xy_list := tl[XY](xy_list)
      while xy_list isnt empty do
      begin
         j := j + 1
         let xy = hd[XY](xy_list)
         let x2 = xy(x)
         let y2 = xy(y)
         if (y1 <= yt and yt <= y2) or (y2 <= yt and yt <= y1) do
         begin
            if y1 = y2
            then if (x1 <= xt and xt <= x2) or
                                          (x2 <= xt and xt <= x1) do
               { x_intersec_list := l_prepend[real](xt,x_intersec_list)
                 xy_list := l_make[XY]()
                 link_list := l_make[PB_link]() }
            else
               case yt of
               y1 : begin
                       if ~l_contains[real](x_intersec_list,x1) do
                       begin
                          if (y0 < y1 and y1 <= y2) or
                                           (y2 < y1 and y1 <= y0)
                          then { x_intersec_list :=
                                    l_prepend[real](x1,x_intersec_list) }
                          else
                          begin
                             x_intersec_list :=
                                 l_prepend[real](x1,x_intersec_list)
                             x_intersec_list :=
                                 l_prepend[real](x1,x_intersec_list)
                          end
```

```
                        end
                     end
                  y2 : { }
               default : begin
                        let x_intersec = (x2-x1)/(y2-y1)*(yt-y1)+x1
                        x_intersec_list :=
                              l_prepend[real](x_intersec,x_intersec_list)
                     end
            end
            y0 := y1 ; y1 := y2; x1 := x2
            xy_list := tl[XY](xy_list)
         end
      end
      link_list := tl[PB_link](link_list)
      i := i + 1
   end
   let num_of_intersections = l_length[real](x_intersec_list)
   case num_of_intersections of
         0 : { in_poly := false }
         1 : { in_poly := true  }
   default : begin
            let x_intersec_vec := vector 1 to
                                          num_of_intersections of 0.0
            for i = 1 to num_of_intersections do
               { x_intersec_vec(i) := l_nth[real](x_intersec_list,i) }
            vector_isu_sort(x_intersec_vec)
            for i = 1 to num_of_intersections by 2 do
            begin
               if x_intersec_vec(i) <= xt and
                                       xt <= x_intersec_vec(i+1) do
                  { in_poly := true }
            end
         end
   in_poly
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Spatial indexing polygons using a grid-cell coded structure           !
!                                                                         !
!-------------------------------------------------------------------------!
gridNdxPoly := proc(poly_id: Poly_id; mbr: MBR;
                    polygon_index: Map[Peanor,List[Poly_id]]; sl: real)
begin
   ! determine limits of the cell covered by a polygon mbr
   let x1 = float(truncate(mbr(x_min) / sl)) * sl
   let y1 = float(truncate(mbr(y_min) / sl)) * sl
   let x2 := float(truncate(mbr(x_max) / sl)) * sl
   let y2 := float(truncate(mbr(y_max) / sl)) * sl
   ! exclude the case that the mbr is on the top or right border of the map
   if x2 = mbr(x_max) do x2 := x2 - sl
   if y2 = mbr(y_max) do y2 := y2 - sl
   ! construct polygon index
   for i = truncate(x1 / sl) to truncate(x2 / sl)  do
   for j = truncate(y1 / sl) to truncate(y2 / sl)  do
   begin
      let xy = XY(float(i) * sl, float(j) * sl)
      let peano_key = xyToPKR(xy)
      let peano = Peanor(peano_key,sl)
      if m_contains[Peanor,List[Poly_id]](polygon_index,peano) then
      begin
         let polyid_list := m_find[Peanor,List[Poly_id]](polygon_index,
                                                         peano)
         if ~l_contains[Poly_id](polyid_list,poly_id) do
         begin
            polyid_list := l_prepend[Poly_id](poly_id,polyid_list)
```

```
            m_isu_assign[Peanor,List[Poly_id]](polygon_index,peano,
                                                polyid_list)
         end
      end
      else
      begin
         let polyid_list := l_make[Poly_id]()
         polyid_list := l_prepend[Poly_id](poly_id,polyid_list)
         m_isu_insert[Peanor,List[Poly_id]](polygon_index,peano,
                                             polyid_list)
      end
   end
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Spatial indexing points using a linear quadtree structure             !
!                                                                         !
!-------------------------------------------------------------------------!
lqtNdxPoint := proc(peano: Peanor; pointid_list: List[Point_id];
                    pid_point: Map[Point_id,XY];
                    point_index: Map[Peanor,List[Point_id]])
begin
   let quad_extent = getQuadExtent(peano)
   let xmin = quad_extent(x_min)
   let ymin = quad_extent(y_min)
   let range = quad_extent(x_range)
   let sl = range / 2.
   let quad_pid_list := vector 0 to 3 of l_make[Point_id]()
   while pointid_list isnt empty do
   begin
      let point_id = hd[Point_id](pointid_list)
      let pt = m_find[Point_id,XY](pid_point,point_id)
      let x = pt(x); let y = pt(y)
      let m = truncate((x - xmin) / sl)
      let n = truncate((y - ymin) / sl)
      if m >= 0 and m <= 1 and n >= 0 and n <= 1 do
      begin
         let i = 2 * m + n
         quad_pid_list(i) := l_prepend[Point_id](point_id,
                                                 quad_pid_list(i))
      end
      pointid_list := tl[Point_id](pointid_list)
   end
   for i = 0 to 3 do
   begin
      ! construct a point index
      let xy = XY(xmin + float(i div 2) * sl, ymin +
                                              float(i rem 2) * sl)
      let peano_key = xyToPKR(xy)
      let peano = Peanor(peano_key,sl)
      m_isu_insert[Peanor,List[Point_id]](point_index,peano,
                                          quad_pid_list(i))
   end
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Spatial indexing lines using a linear quadtree structure              !
!                                                                         !
!-------------------------------------------------------------------------!
lqtNdxLine := proc(peano: Peanor; lid_list: List[Line_id];
                   lid_line: Map[Line_id,List[XY]];
                   line_mbr: Map[Line_id,MBR];
                   line_key_pts: Map[Line_id,List[XY]];
                   line_index: Map[Peanor,List[Line_id]])
```

```
begin
   let quad_extent = getQuadExtent(peano)
   let xmin = quad_extent(x_min)
   let ymin = quad_extent(y_min)
   let range = quad_extent(x_range)
   let xmax = xmin + range
   let ymax = ymin + range
   let sl = range / 2.
   let quad_lid_list := vector 0 to 3 of l_make[Line_id]()
   while lid_list isnt empty do
   begin
      let line_id = hd[Line_id](lid_list)
      let l_mbr = m_find[Line_id,MBR](line_mbr,line_id)
      let ls_xmin = l_mbr(x_min)
      let ls_ymin = l_mbr(y_min)
      let ls_xmax = l_mbr(x_max)
      let ls_ymax = l_mbr(y_max)
      let quad_mbr := vector 0 to 3 of MBR(0.,0.,0.,0.)
      let quad_visit := vector 0 to 3 of false
      for i = 0 to 3 do
      begin
         let quad_xmin = xmin + float(i div 2) * sl
         let quad_ymin = ymin + float(i rem 2) * sl
         let quad_xmax = quad_xmin + sl
         let quad_ymax = quad_ymin + sl
         quad_mbr(i) := MBR(quad_xmin,quad_ymin,quad_xmax,quad_ymax)
         if ls_xmax < quad_xmin or ls_xmin > quad_xmax or
            ls_ymax < quad_ymin or ls_ymin > quad_ymax then { }
         else
         begin
            let key_pts_list := m_find[Line_id,List[XY]](line_key_pts,
                                                         line_id)
            let p1 :=  hd[XY](key_pts_list)
            key_pts_list := tl[XY](key_pts_list)
            while key_pts_list isnt empty and ~quad_visit(i) do
            begin
               let p2 = hd[XY](key_pts_list)
               let l_xmin = if p1(x) < p2(x) then p1(x) else p2(x)
               let l_ymin = if p1(y) < p2(y) then p1(y) else p2(y)
               let l_xmax = if p1(x) > p2(x) then p1(x) else p2(x)
               let l_ymax = if p1(y) > p2(y) then p1(y) else p2(y)
               if l_xmax < quad_xmin or l_xmin > quad_xmax or
                  l_ymax < quad_ymin or l_ymin > quad_ymax then { }
               else
               begin
                  let xy_list := m_find[Line_id,List[XY]](lid_line,
                                                          line_id)
                  while hd[XY](xy_list)(x) ~= p1(x) or
                                        hd[XY](xy_list)(y) ~= p1(y) do
                     { xy_list := tl[XY](xy_list) }
                  let pt1 := hd[XY](xy_list)
                  xy_list := tl[XY](xy_list)
                  let finished := false
                  while xy_list isnt empty and
                                        ~quad_visit(i) and ~finished do
                  begin
                     let pt2 = hd[XY](xy_list)
                     if lineVisibleInWindow(pt1,pt2,quad_mbr(i)) do
                     begin
                        quad_lid_list(i) := l_prepend[Line_id](line_id,
                                                   quad_lid_list(i))
                        quad_visit(i) := true
                     end
                     if pt2(x) = p2(x) and pt2(y) = p2(y) do
                        { finished := true }
                     pt1 := pt2
```

```
                        xy_list := tl[XY](xy_list)
                     end
                  end
                  p1 := p2
                  key_pts_list := tl[XY](key_pts_list)
               end
            end
         end
         lid_list := tl[Line_id](lid_list)
      end
      for i = 0 to 3 do
      begin
         ! construct a line index
         let xy = XY(xmin + float(i div 2) * sl, ymin +
                                    float(i rem 2) * sl)
         let peano_key = xyToPKR(xy)
         let peano = Peanor(peano_key,sl)
         m_isu_insert[Peanor,List[Line_id]](line_index,peano,
                                          quad_lid_list(i))
      end
   end


!-------------------------------------------------------------------------!
!                                                                         !
!   Spatial indexing polygons using a linear quadtree structure           !
!                                                                         !
!-------------------------------------------------------------------------!
   lqtNdxPoly := proc(peano: Peanor; pid_list: List[Poly_id];
                      poly_mbr: Map[Poly_id,MBR];
                      polygon_index: Map[Peanor,List[Poly_id]])
   begin
      let quad_extent = getQuadExtent(peano)
      let xmin = quad_extent(x_min)
      let ymin = quad_extent(y_min)
      let range = quad_extent(x_range)
      let xmax = xmin + range
      let ymax = ymin + range
      let sl = range / 2.
      while pid_list isnt empty do
      begin
         let poly_id = hd[Poly_id](pid_list)
         let mbr = m_find[Poly_id,MBR](poly_mbr,poly_id)
         let x1 := mbr(x_min)
         let y1 := mbr(y_min)
         let x2 := mbr(x_max)
         let y2 := mbr(y_max)
         ! exclude the case that the boundary of a polygon mbr is beyond
         ! the border of the working quadrant and a special case that
         ! the top or right boundary is on the top or right border of the
         ! first subquadrant.
         if x1 < xmin do  x1 := xmin
         if y1 < ymin do  y1 := ymin
         if x2 >= xmax then x2 := xmax - sl
                       else if x2 = xmin + sl do x2 := xmin
         if y2 >= ymax then y2 := ymax - sl
                       else if y2 = ymin + sl do y2 := ymin
         for i = truncate(x1 / sl) to truncate(x2 / sl) do
         for j = truncate(y1 / sl) to truncate(y2 / sl) do
         begin
            let xy = XY(float(i) * sl, float(j) * sl)
            let peano_key = xyToPKR(xy)
            let peano = Peanor(peano_key,sl)
            if m_contains[Peanor,List[Poly_id]](polygon_index,peano) then
            begin
               let polyid_list := m_find[Peanor,List[Poly_id]](polygon_index,
```

```
                                                                    peano)
            if ~l_contains[Poly_id](polyid_list,poly_id) do
            begin
               polyid_list := l_prepend[Poly_id](poly_id,polyid_list)
               m_isu_assign[Peanor,List[Poly_id]](polygon_index,peano,
                                                  polyid_list)
            end
         end
         else
         begin
            let polyid_list := l_make[Poly_id]()
            polyid_list := l_prepend[Poly_id](poly_id,polyid_list)
            m_isu_insert[Peanor,List[Poly_id]](polygon_index,peano,
                                               polyid_list)
         end
      end
      pid_list := tl[Poly_id](pid_list)
   end
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Read a flat binary format file (FBFF) and store it as a rawimage      !
!                                                                         !
!-------------------------------------------------------------------------!
fbffToRaw := proc(fn: string; width, height, depth: int -> Rawimage)
begin
   let num = if (width * height) rem 4 = 0 then (width * height) div 4
             else (width * height) div 4 + 1
   let size = num * 4
   let data := vector 1 to num of 0
   !
   !  create an environment for importing a flat binary format file
   !
   let inputfile = open(fn,0)
   if inputfile = nilfile do
   begin writeString("The file "); writeString(fn);
         writeString(" cannot be opened'n"); abort()
   end
   let position := seek(inputfile,0,0)
   writeString("'nReading ...."); writeInt(size); newline(1)
   let nbytes := readBytes(inputfile,data,0,size)
   writeInt(nbytes); writeString(" bytes.");
   ! default colourmap
   let nc = power_2_k(depth)
   let ct = grayLevel(nc)
   let rawimage := Rawimage(data,width,height,depth,ct)
   let void = close(inputfile)
   rawimage
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Read a TIFF image file and store it as a rawimage                     !
!                                                                         !
!-------------------------------------------------------------------------!
tiffToRaw := proc(fn: string -> Rawimage)
begin
   ! create an environment for importing a tiff image file
   let inputfile = open(fn,0)
   if inputfile = nilfile do
   begin writeString("The file "); writeString(fn);
         writeString(" cannot be opened'n"); abort()
   end
   let fileEnv = makeReadEnv(inputfile)
   let readfile = use fileEnv with readByte: proc(-> int) in readByte
```

```
      !  use a vector of integers as a buffer for reading more data from
      !  the file at one time.
      !  the buffer size = 8k bytes ( 2048 * 4-byte integers)
      let bufsize := 8192
      let intbuf := bufsize div 4
      let buf := vector 1 to intbuf of 0
      !
      !  read header    (length = 8 bytes)
      !
      !  bytes 0-1 define the byte order
      !  hex 49 49 :  from least significant to most significant
      !  hex 4D 4D :  from most significant to least significant
      !
      let byte0 = readfile(); let byte1 = readfile(); let LtoM := true
      if ~(byte0 = 73 and byte1 = 73) and ~(byte0 = 77 and byte1 = 77)
      then { errorAbort("This is not a TIFF image file.   Aborting....") }
      else if (byte0 = 73 and byte1 = 73)
           then  LtoM := true
           else  LtoM := false
      !
      !  calculate the value of a 4-byte word depending on byte-order
      !
      let word_value =
          if LtoM then { proc(word: int -> int);
                              getByte(word,0) +
                              getByte(word,1) * 256 +
                              getByte(word,2) * 256 * 256 +
                              getByte(word,3) }
                  else proc(word: int -> int); word
      !
      !  bytes 2-3 denote the TIFF version number.  (hex 2A)
      !
      let byte2 = readfile(); let byte3 = readfile()
      if ~(byte2 = 42 and byte3 = 0) and ~(byte2 = 0 and byte3 = 42) do
         { errorAbort("This is not a standard TIFF image file.
Aborting....") }
      !
      !  bytes 4-7 contain the offset (in bytes) of the first Image File
      !  Directory
      !
      let offset :=  if LtoM then readfile() + readfile() * 256 +
                                  readfile() * 256 * 256 +
                                  readfile() * 256 * 256 * 256
                             else readfile() * 256 * 256 * 256 +
                                  readfile() * 256 * 256 +
                                  readfile() * 256 + readfile()
      let position := seek(inputfile,offset,0)
      !  read IFD information into buffer
      let count =  if LtoM then readfile() + readfile() * 256
                           else readfile() * 256 + readfile()
      !
      !  Each IFD entry has 12 bytes.
      !  tag: 2 bytes; field type: 2 bytes;
      !  length: 4 bytes; value offset: 4 bytes.
      !  the length of IFD = the number of tag * 12 bytes + 4 bytes
      !
      let ifd_len := count * 12 + 4
      offset := offset + 2
      position := seek(inputfile,offset,0)
      let nb := readBytes(inputfile,buf,0,ifd_len)
      !
      !  Image file directory
      !
      !  consists of a 2-byte count of the number of entries
      !  (= the number of fields) followed by a sequence of 12-byte
      !  filed entries, ended by a 4-byte offset of the next Image File
```

```
!  Directory (or 0 if none).
!
!  get IFD information from the buffer
!
type ifd is structure(ftype,length,value: int)
let default_ifd = ifd(0,0,0)
let IFD := vector 254 to 320 of default_ifd
let tag := 0; let ftype := 0; let length := 0;
let value := 0; let m := 1
for i = 1 to  count  do
begin
   tag := if LtoM then getByte(buf(m),0) + getByte(buf(m),1) * 256
                  else getByte(buf(m),0) * 256 + getByte(buf(m),1)
   ftype := if LtoM then getByte(buf(m),2) + getByte(buf(m),3) * 256
                    else getByte(buf(m),2) * 256 + getByte(buf(m),3)
   m := m + 1 ; length := word_value(buf(m))
   m := m + 1 ; value := if ftype = 3 and getByte(buf(m),2) = 0 and
                                          getByte(buf(m),3) = 0
                         then if LtoM then getByte(buf(m),0) +
                                           getByte(buf(m),1) * 256
                                      else getByte(buf(m),0) * 256 +
                                           getByte(buf(m),1)
                         else word_value(buf(m))
   IFD(tag) := ifd(ftype,length,value)
   m := m + 1
end
let nextifd := word_value(buf(m))       ! offset of next IFD
                                        ! (or 0 if none)
let ras_width = IFD(256)(value)         ! image width
let ras_height = IFD(257)(value)        ! image height
let ras_depth = IFD(258)(value)         ! image depth
let compress = IFD(259)(value)          ! compression method
let photointp = IFD(262)(value)         ! photometric interpretation
let strips_number = IFD(273)(length)    ! the number of strips
let strips_offset = IFD(273)(value)     ! strip offset
let samples_per_pixel = IFD(277)(value)! samples per pixel
let strips_count = IFD(279)(length)     ! strip byte counts
let strips_count_offset = IFD(279)(value)
let x_resolution = IFD(282)(value)      ! x resolution
let y_resolution = IFD(283)(value)      ! y resolution
let planar_config = IFD(284)(value)     ! planar configuration
let resolution_unit = IFD(296)(value)   ! resolution unit
let colourmap_length = IFD(320)(length) ! clormap length and offset
let colourmap_offset = IFD(320)(value)
!  strip offsets
m := 1 ; let n := 0
let strip_offset := vector 1 to strips_number of 0
if strips_number > 1 then
begin
   position := seek(inputfile,strips_offset,0)
   nb := readBytes(inputfile,buf,0,strips_number * 4)
   for i = 1 to  strips_number  do
   if IFD(273)(ftype) = 3 then
      begin
         strip_offset(i):= if LtoM then getByte(buf(m),n) +
                                        getByte(buf(m),n+1) * 256
                                   else getByte(buf(m),n) * 256 +
                                        getByte(buf(m),n+1)
         n := n + 2
         if n = 4 do { m := m + 1 ; n := 0 }
      end
   else
      strip_offset(i) := word_value(buf(i))
end
else
   strip_offset(1) := strips_offset
```

```
!  strips count offsets
m := 1 ;  n := 0
let strip_count := vector 1 to strips_count of 0
if strips_count > 1 then
begin
   position := seek(inputfile,strips_count_offset,0)
   nb := readBytes(inputfile,buf,0, strips_count * 4)
   for i = 1 to strips_count do
   if IFD(279)(ftype) = 3 then
   begin
      strip_count(i):= if LtoM then getByte(buf(m),n) +
                                    getByte(buf(m),n+1) * 256
                               else getByte(buf(m),n) * 256 +
                                    getByte(buf(m),n+1)
      n := n + 2
      if n = 4 do { m := m + 1 ; n := 0 }
   end
   else
      strip_count(i) := word_value(buf(i))
end
else
   strip_count(1) := strips_count_offset

let nc = power_2_k(ras_depth)
let ct := vector 0 to nc-1 using proc(i:int -> *int);
                                                  vector 1 to 3 of 0
m := 1 ; n := 0
!  create or read a colourmap
case photointp of
0 : { ct := invGrayLevel(nc) }
    !  bilevel and grayscale image: create a colourmap,
    !  0 is imaged as white, 2 ** Bitspersample-1 is imaged as black.
1 : { ct := grayLevel(nc) }
    !  bilevel and grayscale image: create a colourmap,
    !  0 is imaged as balck, 2 ** Bitspersample-1 is imaged as white.
3 : begin !  paletted colour image: read the colourmap.
       position := seek(inputfile,colourmap_offset,0)
       nb := readBytes(inputfile,buf,0,colourmap_length * 2)
       for j = 1 to 3 do
          for i = 0 to nc-1 do
          begin
             ct(i,j) := getByte(buf(m),n+1)
             n := n + 2
             if n = 4 do { m := m + 1 ; n := 0 }
          end
    end
default : {  }
!
!  read image data
!  samples_per_pixel = 1  --> bilevel (Class B), grayscale (Class G),
!                             and paletted colour images (Class P).
!                             raster_depth =  1, 4 , or 8 bits
!                     = 3  --> RGB images (Class R, true colour)
!                             raster_depth =  24 bits
!  redefine the buffer size for reading the strips of the image
intbuf := if strip_count(1) rem 4 = 0
          then strip_count(1) div 4
          else  strip_count(1) div 4 + 1
let stripbuf := vector 1 to  intbuf  of 0
! define an 1-D arrary to store rawimage data
let num = case ras_depth of
          1:  { intbuf * strips_number * 8}
          4:  { intbuf * strips_number * 2}
          8:  { intbuf * strips_number }
    default:  { intbuf * strips_number }
```

```
let data := vector 1 to num of 0
writeString("'nstrips : ")
for i = 1 to strips_number do { writeString("."); }
let k := 1   !  positions and reads the first strip image data
position := seek(inputfile,strip_offset(k),0)
nb := readBytes(inputfile,stripbuf,0,strip_count(k))
writeString("'nreading: ")
writeString(".");
let first = 0; let last = 4; let step = 1
let p := 1 ; let q := 0
m := 1 ; n := first
case ras_depth of
1 : {   }
4 : begin
       let half_width = if ras_width rem 2 = 0
                        then ras_width div 2
                        else  (ras_width div 2) + 1
       let x1 = proc(x:int -> int); x div 16
       let x2 = proc(x:int -> int); x rem 16
       for j = ras_height - 1 to 0 by -1 do
       begin
          for i = 0 to half_width - 1 do
          begin
             let x = getByte(stripbuf(m),n)
             data(p) := setByte(data(p),q,x1(x))
             q := q + 1
             if q = 4 do { p := p + 1; q := 0 }
             data(p) := setByte(data(p),q,x2(x))
             q := q + 1
             if q = 4 do { p := p + 1; q := 0 }
             n := n + step
             if m = intbuf and n = last and strips_number > k  do
             begin
                m := 1 ; n := first
                k := k + 1
                writeString(".");
                position := seek(inputfile,strip_offset(k),0)
                nb := readBytes(inputfile,stripbuf,0,strip_count(k))
             end
             if n = last do { m := m + 1; n := first }
          end
       end
    end
8 : begin
       for j = ras_height - 1 to 0 by -1 do
          for i = 0 to ras_width - 1 do
          begin
             let x = getByte(stripbuf(m),n)
             data(p) := setByte(data(p),q,x)
             q := q + 1
             if q = 4 do { p := p + 1; q := 0 }
             n := n + step
             if m = intbuf and n = last and strips_number > k  do
             begin
                m := 1 ; n := first
                k := k + 1
                writeString(".");
                position := seek(inputfile,strip_offset(k),0)
                nb := readBytes(inputfile,stripbuf,0,strip_count(k))
             end
             if n = last do { m := m + 1; n := first }
          end
    end
default:  begin
             writeString("'nThe depth of the TIFF image is ");
             writeInt(ras_depth) ; writeString(" bits'n")
```

```
                writeString("The depth should be 1, 4 or 8 bits.'n")
            end
        if samples_per_pixel = 3 do
        begin
        ! reserved for 24 bits
        end
        let rawimage := Rawimage(data,ras_width,ras_height,ras_depth,ct)
        rawimage
    end

!-------------------------------------------------------------------------!
!                                                                         !
!    Convert a raw image data to an interim image                         !
!                                                                         !
!-------------------------------------------------------------------------!
    rawToInterim := proc(rawimage: Rawimage -> Interim_image)
    begin
        let data = rawimage(data)
        let width = rawimage(width)
        let height = rawimage(height)
        let depth = rawimage(depth)
        let ct = rawimage(colourmap)
        let nc =  power_2_k(depth)
        let default_pixel = defaultPixel(off,depth)
        let pixel_table := vector 0 to nc-1 of default_pixel
        for i = 0 to nc-1 do { pixel_table(i) := colourToPixel(i,depth) }
        let newimage := image width by height of default_pixel
        writeString("'nReading the raw image and constructing an interim image,
waiting ...");
        space(2)
        let wait_count := 0
        let m := 1; let n := 0
        for j = height-1 to 0 by -1 do
        begin
            wait_count := wait_count + 1
            waitSymbol(wait_count)
            for i = 0 to width-1  do
            begin
                let x = getByte(data(m),n)
                setPixel(newimage,i,j,pixel_table(x))
                n := n + 1
                if n = 4 do { m := m + 1; n := 0 }
            end
        end
        writeString("'b'b |")
        let interim_image = Interim_image(newimage,ct)
        interim_image
    end

!-------------------------------------------------------------------------!
!                                                                         !
!    Read an HSI-format image file and store it as an interim image       !
!                                                                         !
!-------------------------------------------------------------------------!
    hsiToInterim := proc(fn: string -> Interim_image)
    begin
        !  create an environment for importing an image file
        let inputfile = open(fn,0)
        if inputfile = nilfile do
        begin
            writeString("The file "); writeString(fn);
            writeString(" cannot be opened'n"); abort()
        end
        let fileEnv = makeReadEnv(inputfile)
        let readfile = use fileEnv with readByte: proc(-> int) in readByte
        !  uses a vector of integers as a buffer in order to read more data
```

```
!  from the file at one time (buffer size = 8k bytes)
let bufsize = 8192
let intbuf = bufsize div 4
let buf := vector 1 to  intbuf  of 0
!  read the header of an HSI image
!  size of header:
!  32 bytes, if the image is true colour
!  32 bytes + 3bytes * the number of colours, if the image is
!  paletted colour
!  first six bytes are magic numbers
let offset := 0
let position := seek(inputfile,offset,0)
let nbytes := readBytes(inputfile,buf,0,bufsize)
let mg0 = getByte(buf(1),0); let mg1 = getByte(buf(1),1)
let mg2 = getByte(buf(1),2); let mg3 = getByte(buf(1),3)
let mg4 = getByte(buf(2),0); let mg5 = getByte(buf(2),1)
if mg0 ~= 109 or mg1 ~= 104 or mg2 ~= 119 or
   mg3 ~= 97  or mg4 ~= 110 or mg5 ~= 104 do
   begin
      writeString("The file "); writeString(fn);
      writeString(" is not a HSI image format.'n"); abort()
   end
!  two bytes represent the version number of an HSI image
let vn = 256 * getByte(buf(2),2) + getByte(buf(2),3)
!  two bytes denote horizontal and vertical pixels respectively
let ras_width = 256 * getByte(buf(3),0) + getByte(buf(3),1)
let ras_height = 256 * getByte(buf(3),2) + getByte(buf(3),3)
!  two bytes represent paletted colours
let nc = 256 * getByte(buf(4),0) + getByte(buf(4),1)
!  two bytes denote horizonal and vertical resolution respectively
let hres = 256 * getByte(buf(4),2) + getByte(buf(4),3)
let vres = 256 * getByte(buf(5),0) + getByte(buf(5),1)
!  two bytes for gamma value
let gamma = 256 * getByte(buf(5),2) + getByte(buf(5),3)
!  skip 12 reserved bytes buf(6),buf(7) and buf(8)
let m := 9; let n := 0
!  read colourmap information
let ct := vector 0 to nc-1 using proc(i:int -> *int);
                                          vector 1 to 3 of 0
for i = 0 to nc-1 do  ! R-G-B-R-G-B-R-G-B .....
for j = 1 to 3 do
begin
   ct(i,j) := getByte(buf(m),n)
   n := n + 1
   if n = 4 do { m := m + 1 ; n := 0 }
end
let ras_depth = case nc of
          2 : 1
          4 : 2
         16 : 4
         64 : 6
        256 : 8
    default : 1
let default_pixel = defaultPixel(off,ras_depth)
let colour_index := vector 0 to nc-1 of default_pixel
for i = 0 to nc-1 do  colour_index(i) := colourToPixel(i,ras_depth)
let raster := image ras_width by ras_height of default_pixel
!
!  read image data
!
let image_size = ras_width * ras_height + 32 + nc * 3
let nb = if image_size rem bufsize = 0 then image_size div bufsize
         else image_size div bufsize + 1
writeString("Blocks  : ")
for i = 1 to nb do { writeString("."); }
writeString("'nReading : .");
```

```
      for j = ras_height - 1 to 0 by -1 do
      for i = 0 to ras_width - 1 do
      begin
         let x = getByte(buf(m),n)
         setPixel(raster,i,j,colour_index(x))
         n := n + 1
         if m = intbuf and  n = 4  do
         begin
            m := 1; n := 0
            offset := offset + bufsize
            position := seek(inputfile,offset,0)
            nbytes := readBytes(inputfile,buf,0,bufsize)
            writeString(".");
         end
         if n = 4 do  { m := m + 1; n := 0 }
      end
      let infndx = close(inputfile)
      let interim_image = Interim_image(raster,ct)
      interim_image
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Read a Sunras image file and store it as an interim image             !
!                                                                         !
!-------------------------------------------------------------------------!
   sunrasToInterim := proc(fn: string -> Interim_image)
   begin
      let inputfile = open(fn,0)
      if inputfile = nilfile do
      begin
         writeString("The file ");writeString(fn);
         writeString(" cannot be opened.'n"); abort()
      end
      let fileEnv = makeReadEnv(inputfile)
      let readfile = use fileEnv with readByte: proc(-> int) in readByte
      !  use a vector of integers as a buffer in order to read more data
      !  from the file at one time (buffer size = 8k bytes)
      let bufsize = 8192
      let intbuf = bufsize div 4
      let buf := vector 1 to  intbuf  of 0
      !
      !  The file structure of the SUN standard raster image file.
      !  It consists of three parts:
      !  1.  header: contains 8 integers (32 bytes)
      !  2.  colourmap information: 3bytes * the number of colours.
      !  3.  image data: 1 byte per pixel.
      !
      !  Part 1: read image header
      !
      let offset := 0
      let position := seek(inputfile,offset,0)
      let nbytes := readBytes(inputfile,buf,0,bufsize)
      !  first integer (4 bytes) is ras_magic (magic number)
      let mg0 = getByte(buf(1),0); let mg1 = getByte(buf(1),1)
      let mg2 = getByte(buf(1),2); let mg3 = getByte(buf(1),3)
      if mg0 ~= 89 or mg1 ~= 166 or mg2 ~= 106 or mg3 ~= 149 do
      begin
         writeString("The file "); writeString(fn);
         writeString(" is not a SUN raster image.'n")
         abort()
      end
      !  2nd integer is ras_width (image width in pixels)
      let ras_width =  buf(2)
      !  3rd integer is ras_height (image height in pixels)
      let ras_height = buf(3)
```

```
!  4th integer is ras_depth (the depth is either 1 or 8)
let ras_depth =  buf(4)
!  5th integer is ras_length (the length in bytes of the image data)
let ras_length = buf(5)
!  6th integer is ras_type
let ras_type = buf(6)
!  7th integer is ras_maptypes
let ras_maptypes = buf(7)
!  8th integer is ras_maplength
let ras_maplength =  buf(8)

let default_pixel = defaultPixel(off,ras_depth)
let raster := image ras_width by ras_height of default_pixel
let nc = ras_maplength div 3
let ct := vector 0 to nc-1 using proc(i:int -> *int);
                                              vector 1 to 3 of 0
if ras_maplength > 0 then
begin
   !
   !  Part 2: read the colourmap information
   !
   !  read colourmap, if the image is binary or paletted colour.
   !  each colour has 3 bytes holding R-G-B intensity information
   !  (value from 0 - 255).
   let m := 9; let n := 0
   for j = 1 to 3 do  ! R-R-R-R...G-G-G-G...B-B-B-B....
   for i = 0 to nc-1 do
   begin
      ct(i,j) := getByte(buf(m),n)
      n := n + 1
      if n = 4 do  { m := m + 1 ; n := 0 }
   end
   let colour_index := vector 0 to nc-1 of default_pixel
   for i = 0 to nc-1 do colour_index(i) := colourToPixel(i,ras_depth)
   !
   !  Part 3: read image data
   !
   let image_size = ras_length + 32 + ras_maplength
   let nb = if image_size rem bufsize = 0
            then image_size div bufsize
            else image_size div bufsize + 1
   writeString("Blocks  : ")
   for i = 1 to nb do { writeString("."); }
   writeString("'nReading : .");
   for j = ras_height - 1 to 0 by -1 do
   for i = 0 to ras_width - 1 do
   begin
      let x = getByte(buf(m),n)
      setPixel(raster,i,j,colour_index(x))
      n := n + 1
      if m = intbuf and  n = 4  do
      begin
         m := 1; n := 0
         offset := offset + bufsize
         position := seek(inputfile,offset,0)
         nbytes := readBytes(inputfile,buf,0,bufsize)
         writeString(".");
      end
      if n = 4 do  { m := m + 1; n := 0 }
   end
end
else
   { writeString("The 24-bit ture colour is not implemented yet.'n") }
let infndx = close(inputfile)
let interim_image = Interim_image(raster,ct)
interim_image
```

```
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Read a TIFF image file and store it as an interim image               !
!                                                                         !
!-------------------------------------------------------------------------!
   tiffToInterim := proc(fn: string -> Interim_image)
   begin
      let inputfile = open(fn,0)
      if inputfile = nilfile do
      begin
         writeString("The file "); writeString(fn);
         writeString(" cannot be opened'n"); abort()
      end
      let fileEnv = makeReadEnv(inputfile)
      let readfile = use fileEnv with readByte: proc(-> int) in readByte
      ! use a vector of integers as a buffer for reading more data from
      ! file at one time.  buffer size = 8k bytes ( 2048 * 4-byte integers)
      let bufsize := 8192
      let intbuf := bufsize div 4
      let buf := vector 1 to intbuf of 0
      !
      !  read header    (length = 8 bytes)
      !
      !  bytes 0-1 define the byte order
      !  hex 49 49 : from least significant to most significant
      !  hex 4D 4D : from most significant to least significant
      !
      let byte0 = readfile(); let byte1 = readfile(); let LtoM := true
      if ~(byte0 = 73 and byte1 = 73) and ~(byte0 = 77 and byte1 = 77)
      then
         begin
            writeString("This is not a TIFF image file.  Aborting....'n")
            abort()
         end
      else
         if (byte0 = 73 and byte1 = 73)
         then  LtoM := true
         else  LtoM := false
      !  calculate the value of a 4-byte word depending on byte-order
      let word_value =
         if byte0 = 73 and byte1 = 73
         then { proc(word:int -> int); getByte(word,0) +
                                       getByte(word,1) * 256 +
                                       getByte(word,2) * 256 * 256 +
                                       getByte(word,3) }
         else proc(word:int -> int); word
      !  bytes 2-3 denote the TIFF version number.  (hex 2A)
      let byte2 = readfile(); let byte3 = readfile()
      if ~(byte2 = 42 and byte3 = 0) and ~(byte2 = 0 and byte3 = 42) do
         { errorAbort("This is not a standard TIFF image file.") }
      !  bytes 4-7 contain the offset (in bytes) of the first Image File
      !  Directory
      let offset := if LtoM then readfile() +
                                 readfile()*256 +
                                 readfile()*256*256 +
                                 readfile()*256*256*256
                            else readfile()*256*256*256 +
                                 readfile()*256*256 +
                                 readfile()*256 +
                                 readfile()
      let position := seek(inputfile,offset,0)
      !  read the IFD information into the buffer
      let count = if LtoM
                  then readfile()+readfile()*256
```

```
            else readfile()*256+readfile()
!
!  Each IFD entry has 12 bytes.
!  tag: 2 bytes; field type: 2 bytes;
!  length of the field: 4 bytes; value offset: 4 bytes.
!  the length of IFD = the number of tag * 12 bytes + 4 bytes
!
let ifd_len := count * 12 + 4
offset := offset + 2
position := seek(inputfile,offset,0)
let nb := readBytes(inputfile,buf,0,ifd_len)
!
!  Image file directory
!
!  consists of a 2-byte count of the number of entries
!  (= the number of fields) followed by a sequence of 12-byte
!  filed entries, ended by a 4-byte offset of the next Image
!  File Directory (or 0 if none).
!
!  get the IFD information from the buffer
!
type ifd is structure(ftype,length,value: int)
let default_ifd = ifd(0,0,0)
let IFD := vector 254 to 320 of default_ifd
let tag := 0; let ftype := 0; let length := 0;
let value := 0; let m := 1
for i = 1 to  count  do
begin
   tag := if LtoM
          then getByte(buf(m),0) + getByte(buf(m),1) * 256
          else getByte(buf(m),0) * 256 + getByte(buf(m),1)
   ftype := if LtoM
            then getByte(buf(m),2) + getByte(buf(m),3) * 256
            else getByte(buf(m),2) * 256 + getByte(buf(m),3)
   m := m + 1 ; length := word_value(buf(m))
   m := m + 1 ; value := if ftype = 3 and getByte(buf(m),2) = 0 and
                              getByte(buf(m),3) = 0
                         then if LtoM then getByte(buf(m),0) +
                                           getByte(buf(m),1) * 256
                                      else getByte(buf(m),0) * 256 +
                                           getByte(buf(m),1)
                         else word_value(buf(m))
   IFD(tag) := ifd(ftype,length,value)
   m := m + 1
end
!  the offset of the next IFD (or 0 if none)
let nextifd := word_value(buf(m))
!  image width
let ras_width = IFD(256)(value)
!  image height
let ras_height = IFD(257)(value)
!  image depth
let ras_depth = IFD(258)(value)
!  compression method
let compress = IFD(259)(value)
!  photometric interpretation
let photointp = IFD(262)(value)
!  the number of strips and strip offset
let strips_number = IFD(273)(length)
let strips_offset = IFD(273)(value)
!  samples per pixel
let samples_per_pixel = IFD(277)(value)
!  strip byte counts
let strips_count = IFD(279)(length)
let strips_count_offset = IFD(279)(value)
!  x resolution
```

```
let x_resolution = IFD(282)(value)
!  y resolution
let y_resolution = IFD(283)(value)
!  planar configuration
let planar_config = IFD(284)(value)
!  resolution unit
let resolution_unit = IFD(296)(value)
!  colourmap length and offset
let colourmap_length = IFD(320)(length)
let colourmap_offset = IFD(320)(value)
!
!  strip offsets
!
m := 1 ; let n := 0
let strip_offset := vector 1 to strips_number of 0
if strips_number > 1 then
begin
   position := seek(inputfile,strips_offset,0)
   nb := readBytes(inputfile,buf,0,strips_number * 4)
   for i = 1 to  strips_number  do
   if IFD(273)(ftype) = 3
   then
      begin
         strip_offset(i):= if LtoM then getByte(buf(m),n) +
                                        getByte(buf(m),n+1) * 256
                                   else getByte(buf(m),n) * 256 +
                                        getByte(buf(m),n+1)
         n := n + 2
         if n = 4 do { m := m + 1 ; n := 0 }
      end
   else
      strip_offset(i) := word_value(buf(i))
end
else strip_offset(1) := strips_offset
!
!  strip count offsets
!
m := 1 ;  n := 0
let strip_count := vector 1 to strips_count of 0
if strips_count > 1 then
begin
   position := seek(inputfile,strips_count_offset,0)
   nb := readBytes(inputfile,buf,0, strips_count * 4)
   for i = 1 to strips_count do
   if IFD(279)(ftype) = 3 then
   begin
      strip_count(i):= if LtoM then getByte(buf(m),n) +
                                    getByte(buf(m),n+1) * 256
                               else getByte(buf(m),n) * 256 +
                                    getByte(buf(m),n+1)
      n := n + 2
      if n = 4 do { m := m + 1 ; n := 0 }
   end
   else
      strip_count(i) := word_value(buf(i))
end
else
   strip_count(1) := strips_count_offset
let nc = power_2_k(ras_depth)
let ct := vector 0 to nc-1 using proc(i:int -> *int);
                                          vector 1 to 3 of 0
m := 1 ; n := 0
!
!  create or read a colourmap
!
case photointp of
```

```
0 : { ct := invGrayLevel(nc) }
    !  bilevel and grayscale image: create a colourmap,
    !  0 is imaged as white, 2 ** Bitspersample-1 is imaged as black.
1 : { ct := grayLevel(nc) }
    !  bilevel and grayscale image: create a colourmap,
    !  0 is imaged as balck, 2 ** Bitspersample-1 is imaged as white.
3 : begin !  paletted colour image: read the colourmap.
       position := seek(inputfile,colourmap_offset,0)
       nb := readBytes(inputfile,buf,0,colourmap_length * 2)
       for j = 1 to 3 do
          for i = 0 to nc-1 do
          begin
             ct(i,j) := getByte(buf(m),n+1)
             n := n + 2
             if n = 4 do { m := m + 1 ; n := 0 }
          end
    end
default : {  }
!  convert colour_index table to pixel bitplanes
let default_pixel = defaultPixel(off,ras_depth)
let colour_index := vector 0 to nc-1 of default_pixel
for i = 0 to nc-1 do colour_index(i) := colourToPixel(i,ras_depth)
let raster := image ras_width by ras_height of default_pixel
!
!  read image data
!  samples_per_pixel = 1  --> bilevel (Class B), grayscale (Class G),
!                             and paletted colour images (Class P).
!                             raster_depth =  1, 4 , or 8 bits
!                     = 3  --> RGB images (Class R, true colour)
!                             raster_depth =  24 bits
!
!  redefine the buffer size for reading strips of the image
!
intbuf := if strip_count(1) rem 4 = 0  then strip_count(1) div 4
                                       else strip_count(1) div 4 + 1
let stripbuf := vector 1 to  intbuf  of 0
!  use dots to represent required processing time,
!  each dot means a strip.
writeString("Strips : ")
for i = 1 to strips_number do { writeString("."); }
!  position and read the first strip image data
let k := 1
position := seek(inputfile,strip_offset(k),0)
nb := readBytes(inputfile,stripbuf,0,strip_count(k))
writeString("'nReading: ")
writeString(".");
let first = 0; let last = 4; let step = 1
m := 1 ; n := first
case ras_depth of
1 : begin
    ! reserved for 1-bit
    end
4 : begin
       let half_width = if ras_width rem 2 = 0
                        then ras_width div 2
                        else (ras_width div 2)+1
       let row_pxl := vector 0 to 2 * half_width of 0
       let x1 =  proc(x:int -> int); x div 16
       let x2 =  proc(x:int -> int); x rem 16
       for j = ras_height - 1 to 0 by -1 do
       begin
          for i = 0 to half_width - 1 do
          begin
             let x = getByte(stripbuf(m),n)
             row_pxl(2*i) := x1(x)
             row_pxl(2*i+1) := x2(x)
```

```
                    n := n + step
                    if m = intbuf and n = last and strips_number > k  do
                    begin
                       m := 1 ; n := first
                       k := k + 1
                       writeString(".");
                       position := seek(inputfile,strip_offset(k),0)
                       nb := readBytes(inputfile,stripbuf,0,strip_count(k))
                    end
                    if n = last do { m := m + 1; n := first }
                 end
                 for i = 0 to ras_width - 1 do
                 begin
                    setPixel(raster,i,j,colour_index(row_pxl(i)))
                 end
              end
           end
       8 : begin
              for j = ras_height - 1 to 0 by -1 do
              for i = 0 to ras_width - 1 do
              begin
                 let x = getByte(stripbuf(m),n)
                 setPixel(raster,i,j,colour_index(x))
                 n := n + step
                 if m = intbuf and n = last and strips_number > k  do
                 begin
                    m := 1 ; n := first
                    k := k + 1
                    writeString(".");
                    position := seek(inputfile,strip_offset(k),0)
                    nb := readBytes(inputfile,stripbuf,0,strip_count(k))
                 end
                 if n = last do { m := m + 1; n := first }
              end
           end
       default: begin
                   writeString("'nThe depth of the TIFF image is ");
                   writeInt(ras_depth) ; writeString(" bits'n")
                   writeString("The depth should be 1, 4 or 8 bits.'n")
                end
       if samples_per_pixel = 3 do
       begin
          ! reserved for 24 bits
       end
       let infndx = close(inputfile)
       let interim_image = Interim_image(raster,ct)
       interim_image
   end

!-------------------------------------------------------------------------!
!                                                                         !
!    Convert an interim image to a Sunras format file                     !
!                                                                         !
!-------------------------------------------------------------------------!
   interimToSunras := proc(interim_image: Interim_image; fn: string)
   begin
       let raster = interim_image(raster)
       let ct = interim_image(colourmap)
       let nc = upb[*int](ct)-lwb[*int](ct) + 1
       !  create an environment for exporting an image file
       let sunras = fn ++ ".ras"
       let newfile = create(sunras,384)
       let outputfile = open(sunras,1)
       if outputfile = nilfile do begin
                                    writeString("The file ");
                                    writeString(sunras);
```

```
                                writeString(" cannot be opened'n");
                                abort()
                            end
let bufsize = 8192
let intbuf = bufsize div 4
let buf := vector 1 to intbuf of 0
!
!  The file structure of the SUN standard raster image file.
!  It consists of three parts:
!  1.  header: contains 8 integers (32 bytes)
!  2.  colourmap information: 3bytes * the number of colours.
!  3.  image data: 1 byte per pixel.
!
!  Part 1: write the header
!
!  first integer (4 bytes) is ras_magic (magic number)
buf(1) := setByte(buf(1),0,89);  buf(1) := setByte(buf(1),1,166)
buf(1) := setByte(buf(1),2,106); buf(1) := setByte(buf(1),3,149)
!  2nd integer is ras_width (image width in pixels)
let ras_width = xDim(raster)
buf(2) := ras_width
!  3rd integer is ras_height (image height in pixels)
let ras_height = yDim(raster)
buf(3) := ras_height
!  4th integer is ras_depth (the depth is either 1 or 8)
let ras_depth := zDim(raster)
buf(4) := ras_depth
!  5th integer is ras_length (the length in bytes of the imge data)
let ras_length = ras_width * ras_height
buf(5) := ras_length
!  6th integer is ras_type
buf(6) := 1
!  7th integer is ras_maptypes
buf(7) := 1
!  8th integer is ras_maplength
buf(8) := nc * 3
!
!  Part 2: write colourmap information
!
let m := 9; let n := 0
for j = 1 to 3 do
for i = 0 to nc-1 do
begin
   buf(m) := setByte(buf(m),n,ct(i,j))
   n := n + 1
   if n = 4  do  { m := m + 1; n := 0 }
end
!
!  Part 3:  write image data
!
let ns = ras_length div 8192 + 1
writeString("Blocks  : ");
for i = 1 to ns do { writeString("."); }
writeString("'nWriting : ");
let last_pxl := defaultPixel(off,ras_depth)
let byte := pixelToColour(last_pxl,ras_depth)
for j = ras_height - 1 to 0 by -1 do
begin
   for i = 0 to ras_width - 1 do
   begin
      let pxl = getPixel(raster,i,j)
      if pxl ~= last_pxl do { byte := pixelToColour(pxl,ras_depth) }
      buf(m) := setByte(buf(m),n,byte)
      n := n + 1
      if m = intbuf and n = 4  do
      begin
```

```
            let nb = writeBytes(outputfile,buf,0,bufsize)
            writeString(".");
            m := 1 ; n := 0
         end
         if n = 4 do  { m := m + 1 ; n := 0 }
         last_pxl := pxl
      end
      !
      !  store the image one line at a time, each line of the image
      !  is rounded out to a multiple of 16 bits.
      !
      if ras_width rem 2 = 1 do
      begin
         buf(m) := setByte(buf(m),n,0)
         n := n + 1
         if n = 4 do  { m := m + 1 ; n := 0 }
      end
   end
   !
   !  count the size and writes the last-time buffer
   !  note that the last time is often not fully filled
   !
   let last_bufsize = 4 * (m - 1) + n
   let nb = writeBytes(outputfile,buf,0,last_bufsize)
   writeString(".");
   let outfndx = close(outputfile)
   writeString("'nDone!");newline(1)
end


!-------------------------------------------------------------------------!
!                                                                         !
!    Preview a raw image                                                  !
!                                                                         !
!-------------------------------------------------------------------------!
previewRaw := proc(rawimage: Rawimage; win_x,win_y: int -> image)
begin
   let data = rawimage(data)
   let width = rawimage(width)
   let height = rawimage(height)
   let depth = rawimage(depth)
   let nc =  power_2_k(depth)
   let default_pixel = defaultPixel(off,depth)
   let pixel_table := vector 0 to nc-1 of default_pixel
   for i = 0 to nc-1 do { pixel_table(i) := colourToPixel(i,depth) }
   let x_step = if width rem win_x = 0 then width div win_x
                                       else width div win_x + 1
   let y_step = if height rem win_y = 0 then height div win_y
                                        else height div win_y + 1
   let step = if x_step > y_step then x_step else y_step
   let disp_xsize = if width rem step = 0 then width div step
                                          else width div step + 1
   let disp_ysize = if height rem step = 0 then height div step
                                           else height div step + 1
   let newimage := image disp_xsize by disp_ysize of default_pixel
   let k := 0
   for j = height-1 to 0 by -step  do
   begin
       for i = 0 to width-1 by step do
       begin
          k := i  + (height - 1 - j) * width
          let x = getByte(data(k div 4 + 1), k rem 4)
          let r = i div step
          let s = j div step
          setPixel(newimage,r,s,pixel_table(x))
       end
```

```
      end
      newimage
   end

!-------------------------------------------------------------------------!
!                                                                         !
!    Preview a linear-stretched raw image                                 !
!                                                                         !
!-------------------------------------------------------------------------!
   previewStretchedRaw := proc(rawimage: Rawimage; win_x,win_y: int -> image)
   begin
      let data = rawimage(data)
      let width = rawimage(width)
      let height = rawimage(height)
      let depth = rawimage(depth)
      let nc =  power_2_k(depth)
      let default_pixel = defaultPixel(off,depth)
      let pixel_table := vector 0 to nc-1 of default_pixel
      for i = 0 to nc-1 do { pixel_table(i) := colourToPixel(i,depth) }
      let x_step = if width rem win_x = 0 then width div win_x
                                          else width div win_x + 1
      let y_step = if height rem win_y = 0 then height div win_y
                                           else height div win_y + 1
      let step = if x_step > y_step then x_step else y_step
      let disp_xsize = if width rem step = 0 then width div step
                                             else width div step + 1
      let disp_ysize = if height rem step = 0 then height div step
                                              else height div step + 1
      let newimage := image disp_xsize by disp_ysize of default_pixel

      let frequency = freqCount(rawimage)
      let count = frequency(count)
      let lower := frequency(lower)
      let upper := frequency(upper)
      let range = nc - 1
      let domain = upper - lower

      let k := 0
      for j = height-1 to 0 by -step  do
      begin
          for i = 0 to width-1 by step do
          begin
             k := i  + (height - 1 - j) * width
             let x = getByte(data(k div 4 + 1), k rem 4)
             let r = i div step
             let s = j div step
             ! linear stretch
             if x >= lower and x <= upper do
             begin
                let y = ((x - lower) * range) div domain
                setPixel(newimage,r,s,pixel_table(y))
             end
          end
      end
      newimage
   end

!-------------------------------------------------------------------------!
!                                                                         !
!    Determine the frequency of the brightness values of a raw image      !
!                                                                         !
!-------------------------------------------------------------------------!
   freqCount := proc(rawimage: Rawimage -> Frequency)
   begin
      let width = rawimage(width)
      let height = rawimage(height)
```

```
      let depth = rawimage(depth)
      let data = rawimage(data)
      let first = lwb[int](data)
      let last =  upb[int](data)
      let nc =  power_2_k(depth) - 1
      let count := vector 0 to nc of 0
      for i = first to last do
         for j = 0 to 3 do
             begin
                let x = getByte(data(i),j)
                count(x) := count(x) + 1
             end
      !
      ! determine the lower bound, upper bound and median
      !
      let sum := 0
      for i = 0 to nc do { sum := sum + count(i) }
      let lower := 0
      let median := 0
      let upper := nc
      while float(count(lower))/float(sum) < 0.001  do lower := lower + 1
      while float(count(upper))/float(sum) < 0.001  do upper := upper - 1
      !
      ! determine the miminum and maximum values
      !
      let vmin := count(0); let vmax := count(0)
      for i = lower to upper do
         begin
            if count(i) < vmin do vmin := count(i)
            if count(i) > vmax do vmax := count(i)
         end
      let cum_prob := 0.0
      while cum_prob < 0.5 do      ! 50 %
         begin
            cum_prob := cum_prob + float(count(median))/float(sum)
            median := median + 1
         end
      writeString("'nThe lower bound of the frequency is : ");
                                                    writeInt(lower)
      writeString("'nThe median of the frequency is :      ");
                                                    writeInt(median)
      writeString("'nThe upper bound of the frequency is : ");
                                                    writeInt(upper)
      let frequency = Frequency(count,lower,median,upper,vmin,vmax)
      frequency
   end

!-------------------------------------------------------------------------!
!                                                                         !
!    Perform a linear contrast stretch on a raw image                     !
!                                                                         !
!-------------------------------------------------------------------------!
   linearStretch := proc(rawimage: Rawimage; frequency: Frequency;
                                                  new_depth: int -> image)
   begin
      let count = frequency(count)
      let lower := frequency(lower)
      let upper := frequency(upper)
      let data = rawimage(data)
      let width = rawimage(width)
      let height = rawimage(height)
      let nc =  power_2_k(new_depth)
      let default_pixel = defaultPixel(off,new_depth)
      let pixel_table := vector 0 to nc-1 of default_pixel
      for i = 0 to nc-1 do { pixel_table(i) := colourToPixel(i,new_depth) }
      let newimage := image width by height of default_pixel
```

```
      let range = nc - 1
      let domain = upper - lower
      let m := 1; let n := 0
      for j = height-1 to 0 by - 1  do
            for i = 0 to width-1 do
                begin
                   let x = getByte(data(m),n)
                   n := n + 1
                   ! linear stretch
                   if x >= lower and x <= upper do
                      begin
                         let y = ((x - lower) * range) div domain
                         setPixel(newimage,i,j,pixel_table(y))
                      end
                    if n = 4 do { m := m + 1 ; n := 0 }
                end
     newimage
    end
!-------------------------------------------------------------------------!
!                                                                         !
!    Determine the frequency of the brightness values of an interim image !
!                                                                         !
!-------------------------------------------------------------------------!
   freqCount2 := proc(interim: image -> Frequency)
   begin
      let width = xDim(interim)
      let height = yDim(interim)
      let depth = zDim(interim)
      let nc =  power_2_k(depth) - 1
      let default_pixel = defaultPixel(off,depth)
      let pixel_table := vector 0 to nc-1 of default_pixel
      for i = 0 to nc-1 do { pixel_table(i) := colourToPixel(i,depth) }
      let count := vector 0 to nc of 0
      for j = 0 to height-1 do
          for i = 0 to width-1 do
             begin
                let p = getPixel(interim,i,j)
                let x = pixelToColour(p,depth)
                count(x) := count(x) + 1
             end
      !
      ! determine the lower bound, upper bound and median
      !
      let sum := 0
      for i = 0 to nc do { sum := sum + count(i) }
      let lower := 0
      let median := 0
      let upper := nc
      while float(count(lower))/float(sum) < 0.001  do lower := lower + 1
      while float(count(upper))/float(sum) < 0.001  do upper := upper - 1
      !
      ! determine the miminum and maximum values
      !
      let vmin := count(0); let vmax := count(0)
      for i = lower to upper do
         begin
            if count(i) < vmin do vmin := count(i)
            if count(i) > vmax do vmax := count(i)
         end
      let cum_prob := 0.0
      while cum_prob < 0.5 do      ! 50 %
         begin
            cum_prob := cum_prob + float(count(median))/float(sum)
            median := median + 1
         end
      writeString("'nThe lower bound of the frequency is : ");
```

```
                                                              writeInt(lower)
      writeString("'nThe median of the frequency is :      ");
                                                              writeInt(median)
      writeString("'nThe upper bound of the frequency is : ");
                                                              writeInt(upper)
      let frequency = Frequency(count,lower,median,upper,vmin,vmax)
      frequency
   end

!-------------------------------------------------------------------------!
!                                                                         !
!    Perform a linear contrast stretch on an interim image                !
!                                                                         !
!-------------------------------------------------------------------------!
   linearStretch2 := proc(interim: image; frequency: Frequency;
                                                  new_depth: int -> image)
   begin
      let count = frequency(count)
      let lower := frequency(lower)
      let upper := frequency(upper)
      let width = xDim(interim)
      let height = yDim(interim)
      let depth = zDim(interim)
      let default_pixel = defaultPixel(off,new_depth)
      let nc =  power_2_k(new_depth)
      let pixel_table := vector 0 to nc-1 of default_pixel
      for i = 0 to nc-1 do { pixel_table(i) := colourToPixel(i,new_depth) }
      let newimage := image width by height of default_pixel
      let range = nc - 1
         begin
            let domain = upper - lower
            for j = 0 to height-1  do
               for i = 0 to width-1 do
                   begin
                      let p := getPixel(interim,i,j)
                      let x = pixelToColour(p,depth)
                      ! linear stretch
                      if x >= lower and x <= upper do
                         begin
                            let y = ((x - lower) * range) div domain
                            setPixel(newimage,i,j,pixel_table(y))
                         end
                   end
         end
     newimage
    end

!-------------------------------------------------------------------------!
!                                                                         !
!    Generate the frequency chart of an image                             !
!                                                                         !
!-------------------------------------------------------------------------!
   getFreqChart := proc(frequency: Frequency; ras_depth: int -> Freq_chart)
   begin
      let count := frequency(count)
      let lower = frequency(lower); let upper := frequency(upper)
      let vmin := frequency(vmin); let vmax :=  frequency(vmax)
      let original_chart := [0., 0.]
      let stretch_chart := [0., 0.]
      let level = power_2_k(ras_depth)
      for i = 0 to level-1 do
      begin
         original_chart := original_chart ++
                              [float(i), 0.] ^ [ float(i), float(count(i))]
      end
      for i = lower to upper do
```

```
      begin
         stretch_chart := stretch_chart ++
                              [float(i), 0.] ^ [ float(i), float(count(i))]
      end
      let freq_chart = Freq_chart(original_chart,stretch_chart,level)
      freq_chart
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Draw the frequency chart of an image before and after a linear contrast !
!   stretch                                                               !
!                                                                         !
!-------------------------------------------------------------------------!
   plotFreqChart := proc(window: image; frequency: Frequency;
                            freq_chart: Freq_chart; bg_col, fg_col: pixel)
   begin
     let win_xsize = 150; let win_ysize = 150
     let origin_win := image win_xsize by win_ysize of bg_col
     let stretch_win := image win_xsize by win_ysize of bg_col
     let chart1 := colour freq_chart(original_chart) in fg_col
     let chart2 := colour freq_chart(stretch_chart) in fg_col
     let vmin = frequency(vmin); let vmax = frequency(vmax)
     let lower = frequency(lower); let upper = frequency(upper)
     draw(origin_win,chart1,float(0),float(freq_chart(level)-1),
                                              float(vmin),float(vmax))
     copy origin_win onto limit window at 100,350
     draw(stretch_win,chart2,float(lower),float(upper),
                                              float(vmin),float(vmax))
     copy stretch_win onto limit window at 100,150
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Convert a raw image to an interim image                               !
!                                                                         !
!-------------------------------------------------------------------------!
  rawToInterimImage := proc()
  begin
     let contained := false
     let overwrite := false
     if ~m_isEmpty[Image_id,Rawimage](raw_images) then
     begin
        let writeImgId = proc(image_id: Image_id; raw_image: Rawimage)
           { writeString(image_id); space(2) }
        writeString("'nImage database contains the following raw images: ")
        newline(1)
        m_app[Image_id,Rawimage](raw_images, writeImgId)
        writeString("'nEnter a rawimage name : ");
                                                 let image_id = readLine()
        if m_contains[Image_id,Rawimage](raw_images,image_id) then
        begin
           if m_contains[Image_id,Interim_image](interim_images,image_id) do
           begin
              contained := true
              writeString("'nInterim image database already contains " ++
image_id ++ ", do you want to overwrite it ?!'nPlease confirm with '"YES'" to
proceed, otherwise no action taken.'n"); let confirm = readLine()
              if confirm = "YES" do { overwrite := true }
           end
           if ~contained or (contained and overwrite) do
           begin
              let rawimage = m_find[Image_id,Rawimage](raw_images,image_id)
              let interim_image =  rawToInterim(rawimage)
              if ~contained then
                 { m_isu_insert[Image_id,Interim_image](interim_images,
```

```
                                                        image_id,
                                                        interim_image) }
               else
                  { m_isu_assign[Image_id,Interim_image](interim_images,
                                                        image_id,
                                                        interim_image) }
               writeString("'nDone.'n")
            end
         end
         else
            { if image_id ~= "" do writeString("Raw image database does not
contain " ++ image_id ++ ".'n") }
      end
      else
         { writeString("'nNo rawimage available.") }
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Convert an interim image to a baseimage                               !
!                                                                         !
!-------------------------------------------------------------------------!
   interimToBaseimage := proc()
   begin
      let contained := false
      let overwrite := false
      if ~m_isEmpty[Image_id,Interim_image](interim_images) then
      begin
         let writeImgId = proc(image_id: Image_id;
                               interim_image: Interim_image)
         { writeString(image_id); space(2) }
         writeString("'nImage database contains the following interim images:
'n")
         m_app[Image_id,Interim_image](interim_images, writeImgId)
         writeString("'nEnter an interim image name: ");
                                                  let image_id = readLine()
         if m_contains[Image_id,Interim_image](interim_images,image_id) then
         begin
            if m_contains[Image_id,Baseimage](base_images,image_id) do
            begin
               contained := true
               writeString("'nBaseimage database already contains " ++
image_id ++ ", do you want to overwrite it ?!'nPlease confirm with '"YES'" to
proceed, otherwise no action taken.'n"); let confirm = readLine()
               if confirm = "YES" do { overwrite := true }
            end
            if ~contained or (contained and overwrite) do
            begin
               let interim_image =
                     m_find[Image_id,Interim_image](interim_images,image_id)
               let raster = interim_image(raster)
               let ras_width = xDim(raster)
               let ras_height = yDim(raster)
               let ct = interim_image(colourmap)
               writeString("'nSupply positional information for the image.")
               writeString("'nEnter the coordinates of SW corner of the
image.")
               writeString("'n  x = "); let x_min = readReal()
               writeString("  y = "); let y_min = readReal()
               writeString("The resolution (m) of a pixel = ");
                                                     let resol = readReal()
               let trash = readLine()
               let x_range = resol * float(ras_width)
               let y_range = resol * float(ras_height)
               let x_max = x_min + x_range
               let y_max = y_min + y_range
```

```
    let range = if x_range >= y_range then x_range else y_range
    let av_km =  range / 1000.
    let side_length := if av_km >= 100. then 100000. else
       { if av_km < 100. and av_km >= 50. then 50000. else
       { if av_km <  50. and av_km >= 20. then 20000. else
       { if av_km <  20. and av_km >=  5. then  5000. else
       { if av_km <   5. and av_km >= 0.5 then  1000. else 500.}}}}
    let side_length := if resol <= 5. then 1000. else 5000.
    let extent = Extent(x_min,y_min,x_range,y_range)
    let gc_aid_attribute := m_empty[Attr_id,GC_attribute](eq_int,
                                                          lt_int)
    let raster_dm := GC_DM(raster,extent,ct,gc_aid_attribute)
    let baseimage_dm = Baseimage_DM(grid_cell: raster_dm)
    let baseimage = Baseimage(baseimage_dm)
    if ~contained then
       { m_isu_insert[Image_id,Baseimage](base_images,image_id,
                                          baseimage) }
    else
    begin
       m_isu_assign[Image_id,Baseimage](base_images,image_id,
                                        baseimage)
       ! remove image_id from baseimage_indices
       let rmImageId = proc(peano: Peano; l: List[Image_id])
       if l_contains[Image_id](l,image_id) do
          { l := l_isu_remove[Image_id](image_id,l) }
       m_app[Peano,List[Image_id]](baseimage_indices,rmImageId)
    end
    ! construct image index
    writeString("Constructing baseimage_indices ...'n")
    let sl = truncate(side_length)
    let x1 = truncate(x_min / side_length) * sl
    let y1 = truncate(y_min / side_length) * sl
    let x2 = truncate(x_max / side_length) * sl
    let y2 = truncate(y_max / side_length) * sl
    for i = x1 to x2 by sl do
    for j = y1 to y2 by sl do
    begin
       let xy = XY(float(i)/100., float(j)/100.)
       let peano_key = xyToPK(xy)
       let peano = Peano(peano_key,side_length)
       if m_contains[Peano,List[Image_id]](baseimage_indices,
                                           peano) then
       begin
          let image_id_list :=
                  m_find[Peano,List[Image_id]](baseimage_indices,
                                               peano)
          if ~l_contains[Image_id](image_id_list,image_id) do
          begin
             image_id_list := l_append[Image_id](image_id_list,
                                                 image_id)
             m_isu_assign[Peano,List[Image_id]](baseimage_indices,
                                                peano,
                                                image_id_list)
          end
       end
       else
       begin
          let image_id_list := l_make[Image_id]()
          image_id_list := l_append[Image_id](image_id_list,
                                              image_id)
          m_isu_insert[Peano,List[Image_id]](baseimage_indices,
                                             peano,image_id_list)
       end
    end
    writeString("Done !'n")
end
```

```
        end
        else
           { if image_id ~= "" do writeString("Interim image database does
not contain " ++ image_id ++ ".'n") }
     end
     else
        { writeString("'nNo interim image available.") }
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Store a raw image                                                     !
!                                                                         !
!-------------------------------------------------------------------------!
  storeRawimage := proc(image_id: Image_id; rawimage: Rawimage)
  begin
     writeString("'nDo you want to store the data in the raw image database?
");
     let ans = readLine()
     if ans = "Y" or ans = "y" do
     begin
        if ~m_contains[Image_id,Rawimage](raw_images,image_id) then
        begin
           m_isu_insert[Image_id,Rawimage](raw_images,image_id,rawimage)
           writeString("Done!'n")
        end
        else
        begin
           writeString("'nRaw image database already contains " ++ image_id
++ ", do you want to overwrite it ?!  'nPlease confirm with '"YES'" to
proceed, otherwise no action taken.'n"); let confirm = readLine()
           if confirm = "YES" do
           begin
              m_isu_assign[Image_id,Rawimage](raw_images,image_id,
                                              rawimage)
              writeString("Done!'n")
           end
        end
     end
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Store an interim image                                                !
!                                                                         !
!-------------------------------------------------------------------------!
  storeInterimImage := proc(image_id: Image_id; interim_image: Interim_image)
  begin
     writeString("'nDo you want to store the data in the interim image
database? ");
     let ans = readLine()
     if ans = "Y" or ans = "y" do
     begin
        if ~m_contains[Image_id,Interim_image](interim_images,image_id) then
        begin
           m_isu_insert[Image_id,Interim_image](interim_images,image_id,
                                                interim_image)
           writeString("Done!'n")
        end
        else
        begin
           writeString("'nInterim image database already contains " ++
image_id ++ ", do you want to overwrite it ?!  'nPlease confirm with '"YES'"
to proceed, otherwise no action taken.'n"); let confirm = readLine()
           if confirm = "YES" do
           begin
```

```
            m_isu_assign[Image_id,Interim_image](interim_images,image_id,
                                                  interim_image)
            writeString("Done!'n")
         end
      end
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Remove a rawimage                                                     !
!                                                                         !
!-------------------------------------------------------------------------!
  removeRawimage := proc()
  begin
     if ~m_isEmpty[Image_id,Rawimage](raw_images) then
     begin
        let writeImgId = proc(image_id: Image_id; raw_image: Rawimage)
           { writeString(image_id); space(2) }
        writeString("'nImage database contains the following raw images: ")
        newline(1)
        m_app[Image_id,Rawimage](raw_images, writeImgId)
        newline(1)
        writeString("'nEnter a rawimage name : ");
                                                  let image_id := readLine()
        if ~m_contains[Image_id,Rawimage](raw_images,image_id) then
        begin
           if image_id ~= "" do
           {  writeString("The database does not contain the queried raw
image.'n") }
        end
        else
        begin
           writeString("Are you sure you wish to destroy the raw image " ++
image_id  ++ " ?'nPlease confirm with '"YES'" to proceed, otherwise no action
taken.'n"); let confirm = readLine()
           if confirm = "YES" do
           begin
              m_isu_remove[Image_id,Rawimage](raw_images,image_id)
              writeString("Done!'n")
           end
        end
     end
     else
        { writeString("'nNo raw image available.'n") }
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Remove an interim image                                               !
!                                                                         !
!-------------------------------------------------------------------------!
  removeInterimImage := proc()
  begin
     if ~m_isEmpty[Image_id,Interim_image](interim_images) then
     begin
        let writeImgId = proc(image_id: Image_id;
                              interim_image: Interim_image)
           { writeString(image_id); space(2) }
        writeString("'nImage database contains the following interim images:
")
        newline(1)
        m_app[Image_id,Interim_image](interim_images, writeImgId)
        newline(1)
        writeString("'nEnter an interim image name : ");
                                                  let image_id := readLine()
```

```
        if ~m_contains[Image_id,Interim_image](interim_images,image_id) then
        begin
           if image_id ~= "" do
              { writeString("The database does not contain the query interim
image.'n") }
        end
        else
        begin
           writeString("Are you sure you wish to destroy the interim image "
++ image_id  ++ " ?'nPlease confirm with '"YES'" to proceed, otherwise no
action taken.'n"); let confirm = readLine()
           if confirm = "YES" do
           begin
              m_isu_remove[Image_id,Interim_image](interim_images,image_id)
              writeString("Done!'n")
           end
        end
     end
     else
        { writeString("'nNo interim image available.'n") }
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Remove a baseimage                                                    !
!                                                                         !
!-------------------------------------------------------------------------!
  removeBaseimage := proc()
  begin
     if ~m_isEmpty[Image_id,Baseimage](base_images) then
     begin
        let writeImgId = proc(image_id: Image_id; base_image: Baseimage)
           { writeString(image_id); space(2) }
        writeString("'nImage database contains the following base images: ")
        newline(1)
        m_app[Image_id,Baseimage](base_images, writeImgId)
        newline(1)
        writeString("'nEnter a base image name : ");
                                                  let image_id = readLine()
        if ~m_contains[Image_id,Baseimage](base_images,image_id) then
        begin
           if image_id ~= "" do
              { writeString("The database does not contain the queried base
image.'n") }
        end
        else
        begin
           writeString("Are you sure you wish to destroy the base image " ++
image_id  ++ " ?'nPlease confirm with '"YES'" to proceed, otherwise no action
taken.'n"); let confirm = readLine()
           if confirm = "YES" do
           begin
              m_isu_remove[Image_id,Baseimage](base_images,image_id)
              ! remove image_id from baseimage_indices
              let rmImageId = proc(peano: Peano; l: List[Image_id])
              if l_contains[Image_id](l,image_id) do
                 { l := l_isu_remove[Image_id](image_id,l) }
              m_app[Peano,List[Image_id]](baseimage_indices,rmImageId)
              writeString("Done!'n")
           end
        end
     end
     else
        { writeString("'nNo base image available.'n") }
  end
```

```
!-------------------------------------------------------------------------!
!                                                                         !
!   Load a baseimage                                                      !
!                                                                         !
!-------------------------------------------------------------------------!
  loadBaseimage := proc(image_id: Image_id; window_file: file -> image)
  begin
     let baseimage = m_find[Image_id,Baseimage](base_images,image_id)
     let baseimage_dm = baseimage(data_model)'grid_cell
     let raster = baseimage_dm(raster)
     let ct = baseimage_dm(colourmap)
     let ras_width = xDim(raster); let ras_height = yDim(raster)
     let ras_depth = zDim(raster)
     writeString("Width = ");writeInt(ras_width);
     writeString("Height = ");writeInt(ras_height);
     writeString("Depth = ");writeInt(ras_depth);newline(1)
     let nc = upb[*int](ct)-lwb[*int](ct) + 1
     let default_pixel = defaultPixel(off,ras_depth)
     let colour_index := vector 0 to nc -1 of default_pixel
     for i = 0 to nc -1 do colour_index(i) := colourToPixel(i,ras_depth)
     for i = 0 to nc-1 do
         {colourMap(window_file,colour_index(i),
                                    ct(i,3)*256*256+ct(i,2)*256+ct(i,1))}
     raster
   end

end
```

APPENDIX F: THE PROTOTYPE IGIS PROGRAM

```
!-------------------------------------------------------------------------!
!                                                                         !
!  A Prototype IGIS based on the Persistent Programming Language Napier88 !
!                                                                         !
!    Main Functions:                                                      !
!                                                                         !
!      View & Query: The display of vector maps or raster images or       !
!                    a superimposition of them and the search for         !
!                    geographical entities                                !
!                                                                         !
!      Spatial Indexing: The construction of entity index tables          !
!                                                                         !
!      Management: The data management of vector maps and raster images   !
!                                                                         !
!      Preprocessing: The pre-processing of raw and interim images        !
!                                                                         !
!      Import & Export: The import and export of vector map data and      !
!                       raster image data                                 !
!                                                                         !
!-------------------------------------------------------------------------!

type Font is structure(constant characters: *image; constant fontHeight:int;
                       constant descender: int; constant info: string)
type FontPack is structure(font: Font; stringToTile,
                                       charToTile: proc(string -> image))
type drawFunction is variant(imageDraw: proc(image,pic,real,real,real,real);
                             fileDraw: proc(file,pic,real,real,real,real);
                             fail: null)
let PS = PS ()
use PS with System, Vector, IO, String, Graphical, Device, Format, Font, X,
            User, Arithmetical, Event, GlasgowLibraries: env in
use X with XOpenWindow : proc (string,int,int,int,int -> file);
           makeReadEnv : proc(file -> env) in
use Arithmetical with pi: real;
                      sqrt: proc(real -> real);
                      rabs: proc(real -> real);
                      float: proc(int -> real);
                      truncate: proc(real -> int)   in
use IO with writeString: proc ( string );
            PrimitiveIO:env;
            readLine: proc(-> string);
            endOfInput: proc(-> bool);
            readString: proc(-> string);
            writeInt: proc(int);
            writeBool: proc(bool);
            writeReal: proc(real);
            readInt: proc(-> int);
            readReal: proc(-> real);
            readChar: proc(-> string);
            integerWidth: int;
            realWidth: int;
            spaceWidth: int;
            makeReadEnv: proc(file -> env) in
use PrimitiveIO with open: proc(string,int -> file);
                     close: proc(file -> int) in
use Format with iformat: proc(int -> string);
                fformat: proc(real,int,int -> string) in
use Font with screenR12, screenR14, screenB14, serifR16,
              gallantR19: FontPack in
use String with length: proc(string -> int);
                digit: proc(string -> bool);
                asciiToString: proc(int -> string);
                stringToAscii: proc(string -> int);
                letter: proc(string -> bool) in
use Vector with lwb, upb: proc[ W ]( *W -> int ) in
use System with abort: proc() in
```

```
use Graphical with Outline,Raster:env in
use Outline with Odraw: proc(proc(int,int,int), proc(int,int,int,int,int),
                        proc(pixel -> int), int,int,int,int,int,pic,
                        real,real,real,real);
                    makeDrawFunction: proc(string -> drawFunction) in
use Raster with getPixel: proc(image,int,int -> pixel);
                xDim, yDim, zDim: proc(image -> int);
                line: proc(image,int,int,int,int,pixel,int) in
use Device with getScreen: proc(file -> image);
                colourMap: proc(file,pixel,int);
                locator:  proc(file,*int);
                getCursor: proc(file -> image);
                getCursorInfo: proc(file,*int);
                setCursorInfo: proc(file,*int);
                setCursor: proc(file,image);
                colourOf: proc(file,pixel -> int) in
use Event with hangup, interrupt, quit, timer: proc() in
use GlasgowLibraries with BulkTypes: env in
use BulkTypes with Maps, Lists : env in
use Maps with
        m_empty:  proc[A, Z](proc(A,A -> bool), proc(A,A -> bool)
                                                        -> Map[A,Z]);
        m_isEmpty:  proc[A,Z](Map[A,Z] -> bool);
        m_isu_insert:  proc[A,Z](Map[A,Z], A, Z);
        m_isu_remove:  proc[A,Z](Map[A,Z], A);
        m_isu_assign:  proc[A,Z](Map[A,Z], A, Z);
        m_find:  proc[A,Z](Map[A,Z], A -> Z);
        m_length:  proc[A,Z](Map[A,Z] -> int);
        m_contains:  proc[A,Z](Map[A,Z], A -> bool);
        m_copy:  proc[A,Z](Map[A,Z] -> Map[A,Z]);
        m_isu_union:  proc[A,Z](Map[A,Z], Map[A,Z]);
        m_isu_clear:  proc[A,Z](Map[A,Z]);
        m_filter:  proc[A,Z](Map[A,Z], proc(A,Z -> bool) -> Map[A,Z]);
        m_diff:  proc[A,Z](Map[A,Z],Map[A,Z] -> Map[A,Z]);
        m_app:  proc[A,Z](Map[A,Z], proc(A,Z)) in
use Lists with hd:  proc[T](List[T] -> T);
               tl:  proc[T](List[T] -> List[T]);
               l_isEmpty:  proc[T](List[T] -> bool);
               l_make:  proc[T](->List[T]);
               l_length:  proc[T](List[T] -> int);
               l_append:  proc[T](List[T], T -> List[T]);
               l_app:  proc[T](List[T], proc(T));
               l_map:  proc[T,X](List[T], proc(T->X) -> List[X]);
               l_join:  proc[T](List[T],List[T] -> List[T]);
               l_isu_join:  proc[T](List[T],List[T] -> List[T]);
               l_prepend: proc[T](T,List[T]->List[T]);
               l_reverse: proc[T](List[T] -> List[T]);
               l_contains:  proc[T](List[T], T -> bool);
               l_first:  proc[T](List[T] -> T);
               l_last:   proc[T](List[T] -> T);
               l_nth:  proc[T](List[T], int -> T);
               l_rest:  proc[T](List[T] -> List[T] );
               l_isu_remove:  proc[T](T, List[T] -> List[T])  in

use User with Library, Database: env in
use Library with General,Graphical,GIS: env  in
use General with stringToInt: proc(string -> int);
                errorAbort:  proc(string);
                waitSymbol:  proc(int);
                newline:  proc(int);
                space:  proc(int);
                intToBits:  proc(int,int -> *int);
                bitsToInt:  proc(*int -> int);
                power_2_k:  proc(int -> int);
                vector_isu_sort:  proc(*real);
                eq_int, lt_int:  proc(int,int -> bool);
```

```
                eq_str, lt_str:  proc(string,string -> bool);
                eq_peano, lt_peano:  proc(Peano,Peano -> bool);
                eq_peanor, lt_peanor:  proc(Peanor,Peanor -> bool);
                xyToPK:  proc(XY -> int);
                pkToXY:  proc(int -> XY);
                xyToPKR:  proc(XY -> real);
                pkrToXY:  proc(real -> XY);
                getQuadExtent:  proc(Peanor -> Extent)   in

use Graphical with
            drawPoint:  proc(XY,pixel,image,Extent);
            drawLineString:  proc(List[XY],pixel,image,Extent);
            drawText: proc(string,real,real,pixel,XY,image,Extent);
            drawRectangle:  proc(MBR,pixel,image,Extent);
            makeCircle:  proc(XY,real -> List[XY]);
            pointInWindow: proc(XY,MBR -> bool);
            lineVisibleInWindow: proc(XY,XY,MBR -> bool);
            lineStrThroughWindow: proc(List[XY],MBR,XY,real -> bool);
            getPoint:  proc(file,image,Win_size,Extent,int -> XY);
            xHairGetPoint:  proc(file,image,Win_size,Extent,int -> XY);
            dynaGetWinCornersA:  proc(file,image,Win_size,Extent,
                                      int -> *XY);
            dynaGetWinCornersB:  proc(file,image,Win_size,Extent,
                                      int -> *XY);
            dynaGetCircle:  proc(file,image,Win_size,Extent,
                                 int -> Circle);
            getDragDxy:  proc(file,image,Win_size,int -> XY);
            getZoomExtent:  proc(string,file,image,Win_size,Extent,
                                 Extent,int -> Extent);
            getLineStrMBR:  proc(List[XY] -> MBR);
            getLineStrKeyPts:  proc(List[XY] -> List[XY]);
            defaultPixel:  proc(pixel,int -> pixel);
            colourToPixel:  proc(int,int -> pixel);
            pixelToColour:  proc(pixel,int -> int);
            rgb:  proc(int -> **int);
            grayLevel:  proc(int -> **int);
            invGrayLevel:  proc(int -> **int);
            remap16:  proc(int,int,int,*pixel -> pixel);
            viewImage:  proc(image,XY,image,Win_size);
            popupMenu:  proc(*string,*proc(),bool,file,image,
                             Win_size,int);
            dialogueBox:  proc(string,string,file,image,Win_size,int
                               -> string);
            writeMessage:  proc(string,file,image,Win_size,int
                                -> Transient_image);
            eraseMessage:  proc(Transient_image,image)   in

use GIS with
        getOSmapInfo: proc(string -> OS_map_info);
        getOSmapName: proc(XY -> OS_map_name);
        ntf625kToBasemap:  proc(string -> Basemap);
        ntf250kToBasemap:  proc(string -> Basemap);
        ntfcontourToBasemap:  proc(string -> Basemap);
        ntfblToBasemap:  proc(string -> Basemap);
        ntfllToBasemap:  proc(string -> Basemap);
        ntfoscarToBasemap:  proc(string -> Basemap);
        storeBasemap:  proc(Map_id,Basemap,Extent);
        removeBasemap: proc();
        getPolyMBR:  proc(Poly_id,PB_cid_chain,PB_gid_geometry -> MBR);
        pointInPolygon:  proc(XY,PB_polygon,Map[Chain_id,PB_chain],
                              Map[Geom_id,PB_geometry],
                              Map[Geom_id,MBR] -> bool);
        gridNdxPoly:  proc(Poly_id,MBR,Map[Peanor,List[Poly_id]],real);
        lqtNdxPoint:  proc(Peanor,List[Point_id],Map[Point_id,XY],
                           Map[Peanor,List[Point_id]]);
        lqtNdxLine:  proc(Peanor,List[Line_id],Map[Line_id,List[XY]],
```

```
                        Map[Line_id,MBR],Map[Line_id,List[XY]],
                        Map[Peanor,List[Line_id]]);
        lqtNdxPoly:  proc(Peanor,List[Poly_id],Map[Poly_id,MBR],
                        Map[Peanor,List[Poly_id]]);
        fbffToRaw:  proc(string,int,int,int -> Rawimage);
        tiffToRaw:  proc(string -> Rawimage);
        previewRaw:  proc(Rawimage,int,int -> image);
        previewStretchedRaw: proc(Rawimage,int,int -> image);
        freqCount:  proc(Rawimage -> Frequency);
        linearStretch:  proc(Rawimage,Frequency,int -> image);
        rawToInterim: proc(Rawimage -> Interim_image);
        hsiToInterim:  proc(string -> Interim_image);
        sunrasToInterim:  proc(string -> Interim_image);
        tiffToInterim:  proc(string -> Interim_image);
        interimToSunras: proc(Interim_image,string);
        freqCount2:  proc(image -> Frequency);
        linearStretch2:  proc(image,Frequency,int -> image);
        getFreqChart:  proc(Frequency,int -> Freq_chart);
        plotFreqChart:  proc(image,Frequency,Freq_chart,pixel,pixel);
        rawToInterimImage:  proc();
        interimToBaseimage:  proc();
        storeRawimage:  proc(Image_id,Rawimage);
        storeInterimImage:  proc(Image_id,Interim_image);
        removeRawimage:  proc();
        removeInterimImage:  proc();
        removeBaseimage:  proc();
        loadBaseimage:  proc(Image_id,file -> image)  in
use Database with Raw,Interim,Processed,Derived,Index: env in
use Raw with raw_images:  Map[Image_id, Rawimage] in
use Interim with interim_images:  Map[Image_id, Interim_image] in
use Processed with base_maps:  Map[Map_id,Basemap];
                   base_images:  Map[Image_id, Baseimage] in
use Index with basemap_indices:  Map[Peano,Map_id];
               baseimage_indices:  Map[Peano,List[Image_id]];
               entity_mbrs:  Map[Map_id,Entity_mbr];
               entity_indices:  Map[Map_id,Entity_index] in

begin
!-------------------------------------------------------------------------!
!                                                                         !
!   Initialize default values                                             !
!                                                                         !
!-------------------------------------------------------------------------!
!
! Link and Node data model
!
  let ln_pid_point := m_empty[Point_id,LN_point](eq_int,lt_int)
  let ln_lid_line := m_empty[Line_id,LN_line](eq_int,lt_int)
  let ln_gid_geometry:= m_empty[Geom_id,LN_geometry](eq_int,lt_int)
  let ln_aid_attribute := m_empty[Attr_id,LN_attribute](eq_int,lt_int)
  let ln_kid_link := m_empty[Link_id,LN_link](eq_int,lt_int)
  let ln_nid_node := m_empty[Node_id,LN_node](eq_int,lt_int)
  let ln_tid_text := m_empty[Text_id,LN_text](eq_int,lt_int)
!
! Polygon_based data model
!
  let pb_gid_geometry := m_empty[Geom_id,PB_geometry](eq_int,lt_int)
  let pb_aid_attribute := m_empty[Attr_id,PB_attribute](eq_int,lt_int)
  let pb_polyid_polygon := m_empty[Poly_id,PB_polygon](eq_int,lt_int)
  let pb_cid_chain := m_empty[Chain_id,PB_chain](eq_int,lt_int)
  let pb_cpolyid_cpolygon := m_empty[Cpoly_id,PB_cpolygon](eq_int,lt_int)
  let pb_collid_collection := m_empty[Coll_id,PB_collection](eq_int,lt_int)
!
! Spaghetti data model
!
  let sp_pid_point := m_empty[Point_id,SP_point](eq_int,lt_int)
```

```
  let sp_lid_line := m_empty[Line_id,SP_line](eq_int,lt_int)
  let sp_tid_text := m_empty[Text_id,SP_text](eq_int,lt_int)
!
! Feature code table
!
  let fcd := m_empty[FC,FD](eq_str,lt_str)
!
! Drawing status table
!
  let drawn_table := m_empty[string,string](eq_str,lt_str)
!
! Map coverage
!
  let map_extent :=  Extent(0.,0.,0.,0.)
  let image_extent := Extent(0.,0.,0.,0.)
  let draw_extent := Extent(0.,0.,0.,0.)
  let x_min := 0.
  let y_min := 0.
  let x_max := 0.
  let y_max := 0.
  let map_mbr := MBR(x_min,y_min,x_max,y_max)
  let x_range := 0.
  let y_range := 0.
  let x_cent := 0.
  let y_cent := 0.
!
! Root window size
!
  let win_width = 800
  let win_height = 600
  let win_depth = 8

  let hw_ratio = float(win_height) / float(win_width)
                      ! the aspect ratio of the display window
  let vec_depth = 4   ! the number of bit-planes used for vector data
  let ras_depth = 4   ! the number of bit-planes used for raster data
  let pntr_bp := 7    ! the pointer bit-plane
  let start_bp = 4    ! the start bit-plane of the popup menu and the
                      ! dialogue box
  let pxl_resol := 0.
  let tile_count := 1
  let fc := ""
  let map_id := ""

! the count of lines and polygons
  let no_lines := 0
  let no_polys := 0

! index threshold

  let pt_ndx_threshold := 1000
  let ln_ndx_threshold := 100
  let poly_ndx_threshold := 200

  let txt_size = 8.0
  let snap_target_size = 10.0     ! the size of the target window

  let data := vector 1 to 7 of 0  ! event control

  let map_loaded := false

!-------------------------------------------------------------------------!
!                                                                         !
!   Initialize an X window display device                                 !
!                                                                         !
!-------------------------------------------------------------------------!
```

```
  let win_size = Win_size(win_width,win_height)
  let window_file = open("WINDOW: zdim:8, xdim:800, ydim:600", 0)
  if window_file = nilfile do errorAbort("Cannot open an X window");
  let screen = getScreen (window_file)
  let draw = makeDrawFunction("image")'imageDraw
!
! setup colours: bits 0 ~ 3 for raster images; bits 4 ~ 7 for vector maps
!
  let nc_ras = power_2_k(ras_depth)
  let ras_cndx := vector 0 to nc_ras - 1 of defaultPixel(off,ras_depth)
  for i = 0 to nc_ras - 1 do ras_cndx(i) := colourToPixel(i,ras_depth)

  let nc_vec = power_2_k(vec_depth)
  let vec_cndx := vector 0 to nc_vec - 1 of defaultPixel(off,vec_depth)
  for i = 0 to nc_vec - 1 do vec_cndx(i) := colourToPixel(i,vec_depth)

! define colours
  let black     =  vec_cndx(0)
  let olive     =  vec_cndx(1)
  let purple    =  vec_cndx(2)
  let red       =  vec_cndx(3)
  let aqua      =  vec_cndx(4)
  let green     =  vec_cndx(5)
  let blue      =  vec_cndx(6)
  let dkgray    =  vec_cndx(7)    ! dk gray
  let gray      =  vec_cndx(8)    ! lt gray
  let yellow    =  vec_cndx(9)
  let magenta   =  vec_cndx(10)
  let pink      =  vec_cndx(11)
  let cyan      =  vec_cndx(12)
  let lime      =  vec_cndx(13)
  let navy      =  vec_cndx(14)   ! sky
  let white     =  vec_cndx(15)

!-------------------------------------------------------------------------!
!                                                                         !
!   Define a default colourmap for displaying maps                        !
!                                                                         !
!-------------------------------------------------------------------------!
  let map_colourmap := proc()
  begin
     ! 16 default colours for the display of vector data
     let ct = rgb(nc_vec)
     for i = 0 to nc_vec - 1 do
        for j = 0 to nc_ras - 1 do
           { colourMap(window_file,ras_cndx(j)++vec_cndx(i),
                                   ct(i,3)*256*256+ct(i,2)*256+ct(i,1)) }
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Define a default colourmap for displaying images                      !
!                                                                         !
!-------------------------------------------------------------------------!
  let image_colourmap := proc()
  begin
     ! 16 gray levels for the display of raster data
     let ct = grayLevel(nc_ras)
     for i = 0 to nc_ras - 1 do
        { colourMap(window_file,ras_cndx(i)++off++off++off++off,
                                   ct(i,3)*256*256+ct(i,2)*256+ct(i,1)) }
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   A set of procedures for retrieving a basemap                          !
```

```
!                                                                              !
!------------------------------------------------------------------------------!

  let basemapDM_name := proc(dm: Basemap_DM -> string)
  begin
     project dm as X onto
        link_node  : "link_node"
        polygon_based: "polygon_based"
        spaghetti  : "spaghetti"
        default    : "none"
  end

  let load_link_node := proc(basemap: Basemap)
  begin
     let basemap_dm = basemap(data_model)'link_node
     ln_pid_point := m_copy[Point_id,LN_point](basemap_dm(point))
     ln_lid_line := m_copy[Line_id,LN_line](basemap_dm(line))
     ln_gid_geometry := m_copy[Geom_id,LN_geometry](basemap_dm(geometry))
     ln_aid_attribute := m_copy[Attr_id,LN_attribute](basemap_dm(attribute))
     ln_kid_link := m_copy[Link_id,LN_link](basemap_dm(link))
     ln_nid_node := m_copy[Node_id,LN_node](basemap_dm(node))
     if basemap_dm(txt) is ln_tid_text do
       {ln_tid_text := m_copy[Text_id,LN_text](basemap_dm(txt)'ln_tid_text)}
     fcd:= m_copy[FC,FD](basemap_dm(fcd))
  end

  let load_polygon_based := proc(basemap: Basemap)
  begin
     let basemap_dm = basemap(data_model)'polygon_based
     pb_gid_geometry := m_copy[Geom_id,PB_geometry](basemap_dm(geometry))
     pb_aid_attribute := m_copy[Attr_id,PB_attribute](basemap_dm(attribute))
     pb_polyid_polygon := m_copy[Poly_id,PB_polygon](basemap_dm(polygon))
     pb_cid_chain := m_copy[Chain_id,PB_chain](basemap_dm(chain))
     pb_cpolyid_cpolygon :=
                        m_copy[Cpoly_id,PB_cpolygon](basemap_dm(cpolygon))
     pb_collid_collection :=
                        m_copy[Coll_id,PB_collection](basemap_dm(collection))
     fcd:= m_copy[FC,FD](basemap_dm(fcd))
  end

  let load_spaghetti := proc(basemap: Basemap)
  begin
     let basemap_dm = basemap(data_model)'spaghetti
     sp_pid_point := m_copy[Point_id,SP_point](basemap_dm(point))
     sp_lid_line := m_copy[Line_id,SP_line](basemap_dm(line))
     if basemap_dm(txt) is sp_tid_text do
        {sp_tid_text := m_copy[Text_id,SP_text](basemap_dm(txt)'sp_tid_text)}
     fcd := m_copy[FC,FD](basemap_dm(fcd))
  end

!------------------------------------------------------------------------------!
!                                                                              !
!   Draw a circle                                                              !
!                                                                              !
!------------------------------------------------------------------------------!
   let drawCircle := proc(circle: List[XY]; cir_col: pixel; window: image;
                          draw_extent: Extent)
     { drawLineString(circle,cir_col,window,draw_extent) }

!------------------------------------------------------------------------------!
!                                                                              !
!   A set of procedures for drawing a basemap                                  !
!                                                                              !
!------------------------------------------------------------------------------!
!
! Link and node data model
```

```
!
! define feature colours
  let LN_fc_col = proc(FC: string -> pixel)
  begin
     let fc_col = case FC(1|3) of
                  "511" :  blue    ! Coast
                  "512" :  blue    ! Foreshore
                  "514" :  black   ! Lighthouse
                  "521" :  blue    ! Main River
                  "522" :  blue    ! Secondary River
                  "523" :  blue    ! Minor River
                  "524" :  blue    ! Canal
                  "525" :  blue    ! Lake
                  "531" :  cyan    ! Motorway
                  "532" :  red     ! Primary Route
                  "533" :  red     ! A Road
                  "534" :  magenta ! B Road
                  "535" :  black
                  "536" :  black   ! Services
                  "537" :  black   !
                  "538" :  black   ! Toll
                  "539" :  black   ! Ferry
                  "541" :  black   ! Settlement
                  "542" :  black   ! Urban Area
                  "551" :  black   ! Railway
                  "552" :  red     ! Railway Station
                  "553" :  black   ! Level Crossing
                  "561" :  green   ! Woodland
                  "562" :  green   ! Geographical Area
                  "571" :  black   ! National Boundary
                  "572" :  black   ! County Boundary
                  "581" :  black   !
                  "582" :  green   ! National Park
                  "583" :  black   ! Radio Mast
                  "584" :  black   !
                  default : green
     fc_col
  end

! draw a point
  let drawLNpoint = proc(point_id: Point_id; ln_point: LN_point)
  begin
     let geom_id = ln_point(geom_id)
     let attr_id = ln_point(attr_id)
     let point = l_first[XY](m_find[Geom_id,LN_geometry](ln_gid_geometry,
                                                         geom_id)(xy_list))
     let fc = m_find[Attr_id,LN_attribute](ln_aid_attribute,attr_id)(fc)
     let point_col =  LN_fc_col(fc)
     drawPoint(point,point_col,screen(ras_depth|vec_depth),draw_extent)
  end

! draw a line
  let drawLNline = proc(line_id: Line_id; ln_line: LN_line)
  begin
      let geom_id = ln_line(geom_id)
      let attr_id = ln_line(attr_id)
      let xy_list = m_find[Geom_id,LN_geometry](ln_gid_geometry,
                                                geom_id)(xy_list)
      let fc = m_find[Attr_id,LN_attribute](ln_aid_attribute,attr_id)(fc)
      let line_col = LN_fc_col(fc)
      drawLineString(xy_list,line_col,screen(ras_depth|vec_depth),
                                                          draw_extent)
  end

! draw a text
  let drawLNtext = proc(text_id: Text_id; ln_text: LN_text)
```

```
  begin
      !  reserved !
  end

! draw a feature type
  let drawLNfeature = proc(attr_id: Attr_id; ln_attribute: LN_attribute)
  begin
     if ln_attribute(fc) = fc do
     begin
        let gtype = m_find[Geom_id,LN_geometry](ln_gid_geometry,
                                                          attr_id)(gtype)
        case gtype of
        1: begin
              let ln_point = m_find[Point_id,LN_point](ln_pid_point,attr_id)
              drawLNpoint(attr_id,ln_point)
           end
        2: begin
              let ln_line = m_find[Line_id,LN_line](ln_lid_line,attr_id)
              drawLNline(attr_id,ln_line)
           end
        default: writeString("'nIllegal value!")
     end
  end

! draw a node
  let drawLNnode = proc(node_id: Node_id; ln_node: LN_node)
  begin
     let geom_id_of_node = ln_node(geom_id_of_node)
     let node = l_first[XY](m_find[Geom_id,LN_geometry](ln_gid_geometry,
                                                  geom_id_of_node)(xy_list))
     let node_col = black
     drawPoint(node,node_col,screen(ras_depth|vec_depth),draw_extent)
  end

! draw a link
  let drawLNlink = proc(node_id: Node_id; ln_node: LN_node)
  begin
     let link_list := ln_node(link_list)
     while link_list isnt empty do
     begin
        let geom_id_of_link = hd[Link](link_list)(geom_id_of_link)
        let xy_list = m_find[Geom_id,LN_geometry](ln_gid_geometry,
                                                   geom_id_of_link)(xy_list)
        let link_col = red
        drawLineString(xy_list,link_col,screen(ras_depth|vec_depth),
                                                              draw_extent)
        link_list := tl[Link](link_list)
     end
  end

!
! Polygon_based data model
!
! draw a point
  let point_col = black
  let drawPBpoint = proc(geom_id: Geom_id; pb_geometry: PB_geometry)
  begin
     let gtype = pb_geometry(gtype)
     if gtype = 1 do
     begin
        let pt = l_first[XY](pb_geometry(xy_list))
        drawPoint(pt,point_col,screen(ras_depth|vec_depth),draw_extent)
     end
  end

! draw a link
```

```
  let link_col = blue
  let drawPBlink = proc(geom_id: Geom_id; pb_geometry: PB_geometry)
  begin
     let gtype = pb_geometry(gtype)
     if gtype = 2 do
     begin
        let xy_list = pb_geometry(xy_list)
        drawLineString(xy_list,link_col,screen(ras_depth|vec_depth),
                                                                draw_extent)
     end
  end

!
! Spaghetti data model
!
! define feature colours
!
  let SP_fc_col = proc(FC:string -> pixel)
  begin
     let fc_col :=  case FC of
                    "0001" : red
                    "0004" : green
                    "0007" : magenta
                    "0010" : cyan
                    "0011" : blue
                    "0025" : red
                    "0026" : yellow
                    "0057" : magenta
                    default : green
      fc_col
   end

! draw a point
  let drawSPpoint = proc(point_id: Point_id; sp_point: SP_point)
  begin
     let point_col = SP_fc_col(sp_point(fc))
     drawPoint(sp_point(xy),point_col,screen(ras_depth|vec_depth),
                                                                draw_extent)
  end

! draw a line
  let drawSPline = proc(line_id: Line_id; sp_line: SP_line)
  begin
     let line_col = SP_fc_col(sp_line(fc))
     drawLineString(sp_line(xy_list),line_col,screen(ras_depth|vec_depth),
                                                                draw_extent)
  end

! draw a feature type
  let drawSPpointFeature = proc(point_id: Point_id; sp_point: SP_point)
  begin
     if sp_point(fc) = fc do drawSPpoint(point_id,sp_point)
  end

  let drawSPlineFeature = proc(line_id: Line_id; sp_line: SP_line)
  begin
     if sp_line(fc) = fc do  drawSPline(line_id,sp_line)
  end

! draw a text
  let txt_col = black
  let drawSPtext = proc(text_id: Text_id; sp_text: SP_text)
  begin
     let txt = sp_text(text_body)
     let txt_ht = sp_text(text_ht) * x_range / 1000.
     let txt_orient = sp_text(orient)
```

```
    let insert_pt = sp_text(xy)
    drawText(txt,txt_ht,txt_orient,txt_col,insert_pt,
                                    screen(ras_depth|vec_depth),draw_extent)
  end

!-------------------------------------------------------------------------------!
!                                                                               !
!   Load a basemap                                                              !
!                                                                               !
!-------------------------------------------------------------------------------!
  let load_basemap := proc(map_id: Map_id)
  begin
    let os_map_info = getOSmapInfo(map_id)
    writeString("'nThe map series of " ++ map_id ++ " is " ++
                os_map_info(series) ++ ", map scale = 1 : " ++
                iformat(truncate(os_map_info(mapscale))) ++ ".")
    let basemap = m_find[Map_id,Basemap](base_maps,map_id)
    let dm_name = basemapDM_name(basemap(data_model))
    writeString("'nThe data model is " ++ dm_name ++ ".")
    case dm_name of
    "link_node" : { load_link_node(basemap) }
    "polygon_based" : { load_polygon_based(basemap) }
    "spaghetti" : { load_spaghetti(basemap) }
    default : { }
  end

!-------------------------------------------------------------------------------!
!                                                                               !
!   Determine the drawing extent of a basemap                                   !
!                                                                               !
!-------------------------------------------------------------------------------!
  let get_draw_extent := proc(map_id: Map_id)
  begin
    ! Map coverage
    let os_map_info = getOSmapInfo(map_id)
    map_extent := os_map_info(extent)
    let map_xmin := map_extent(x_min)
    let map_ymin := map_extent(y_min)
    let map_xrange := map_extent(x_range)
    let map_yrange := map_extent(y_range)
    let map_xmax := map_xmin + map_xrange
    let map_ymax := map_ymin + map_yrange
    map_mbr := MBR(map_xmin, map_ymin, map_xmax, map_ymax)
    if map_xrange * hw_ratio <= map_yrange then
    begin
       x_range := map_yrange / hw_ratio
       y_range := map_yrange
    end
    else
    begin
       x_range := map_xrange
       y_range := map_xrange * hw_ratio
    end
    ! the map is left justified
    x_min := map_xmin
    y_min := map_ymin
    x_max := x_min + x_range
    y_max := y_min + y_range
    x_cent := (x_min + x_max) / 2.
    y_cent := (y_min + y_max) / 2.

!   ! centre the map in the window
!   x_cent := (map_xmin + map_xmax) / 2.
!   y_cent := (map_ymin + map_ymax) / 2.
!   x_min := x_cent - x_range / 2.
!   y_min := y_cent - y_range / 2.
```

```
!     x_max := x_min + x_range
!     y_max := y_min + y_range

      draw_extent := Extent(x_min, y_min, x_range, y_range)
   end

!-------------------------------------------------------------------------!
!                                                                         !
!    Draw a basemap                                                       !
!                                                                         !
!-------------------------------------------------------------------------!
   let draw_basemap := proc(map_id: Map_id)
   begin
      let basemap = m_find[Map_id,Basemap](base_maps,map_id)
      let dm_name = basemapDM_name(basemap(data_model))
      case dm_name of
      "link_node" : m_app[Line_id,LN_line](ln_lid_line,drawLNline)
      "polygon_based" : m_app[Geom_id,PB_geometry](pb_gid_geometry,
                                                   drawPBlink)
      "spaghetti" : begin
                       m_app[Line_id,SP_line](sp_lid_line,drawSPline)
                       case getOSmapInfo(map_id)(series) of
                       "s_50k" : { }
                       default : m_app[Text_id,SP_text](sp_tid_text,
                                                        drawSPtext)
                    end
      default : { }
   end

!-------------------------------------------------------------------------!
!                                                                         !
!    Create the mbrs of lines and polygons for a basemap                  !
!                                                                         !
!-------------------------------------------------------------------------!
   let get_entity_mbr := proc(map_id: Map_id)
   begin
      load_basemap(map_id)
      let basemap = m_find[Map_id,Basemap](base_maps,map_id)
      let dm_name = basemapDM_name(basemap(data_model))
      let line_mbr := m_empty[Line_id,MBR](eq_int,lt_int)
      let poly_mbr := m_empty[Poly_id,MBR](eq_int,lt_int)
      let line_key_pts := m_empty[Line_id,List[XY]](eq_int,lt_int)
      no_lines := 0
      no_polys := 0
      space(5)
      case dm_name of
      "link_node" :
      begin
         let buildLNlineMBR = proc(line_id: Line_id; ln_line: LN_line)
         begin
            no_lines := no_lines + 1
            waitSymbol(no_lines)
            let geom_id = ln_line(geom_id)
            let xy_list = m_find[Geom_id,LN_geometry](ln_gid_geometry,
                                                      geom_id)(xy_list)
            let key_pts_list = getLineStrKeyPts(xy_list)
            let mbr = getLineStrMBR(key_pts_list)
            m_isu_insert[Line_id,MBR](line_mbr,line_id,mbr)
            m_isu_insert[Line_id,List[XY]](line_key_pts,line_id,
                                           key_pts_list)
         end
         m_app[Line_id,LN_line](ln_lid_line,buildLNlineMBR)
      end
      "polygon_based" :
      begin
         let buildPBlinkMBR = proc(geom_id: Geom_id; pb_geometry: PB_geometry)
```

```
        begin
           let gtype = pb_geometry(gtype)
           if gtype = 2 do
           begin
              no_lines := no_lines + 1
              waitSymbol(no_lines)
              let key_pts_list = getLineStrKeyPts(pb_geometry(xy_list))
              let mbr = getLineStrMBR(key_pts_list)
              m_isu_insert[Geom_id,MBR](line_mbr,geom_id,mbr)
              m_isu_insert[Geom_id,List[XY]](line_key_pts,geom_id,
                                                          key_pts_list)
           end
        end
        let buildPBpolyMBR = proc(poly_id: Poly_id; pb_polygon: PB_polygon)
        begin
           if m_contains[Attr_id,PB_attribute](pb_aid_attribute,poly_id) do
           begin
              let FC = m_find[Attr_id,PB_attribute](pb_aid_attribute,
                                                          poly_id)(fc)
              if FC = "3901" do    !!!!  Application specific
              begin
                 no_polys := no_polys + 1
                 waitSymbol(no_polys)
                 let mbr = getPolyMBR(poly_id,pb_cid_chain,pb_gid_geometry)
                 m_isu_insert[Poly_id,MBR](poly_mbr,poly_id,mbr)
              end
           end
        end
        m_app[Geom_id,PB_geometry](pb_gid_geometry,buildPBlinkMBR)
        m_app[Poly_id,PB_polygon](pb_polyid_polygon,buildPBpolyMBR)
     end
     "spaghetti" :
     begin
        let buildSPlineMBR = proc(line_id: Line_id; sp_line: SP_line)
        begin
           no_lines := no_lines + 1
           waitSymbol(no_lines)
           let key_pts_list = getLineStrKeyPts(sp_line(xy_list))
           let mbr = getLineStrMBR(key_pts_list)
           m_isu_insert[Line_id,MBR](line_mbr,line_id,mbr)
           m_isu_insert[Line_id,List[XY]](line_key_pts,line_id,
                                                       key_pts_list)
        end
        m_app[Line_id,SP_line](sp_lid_line,buildSPlineMBR)
     end
     default : { }
     let entity_mbr := Entity_mbr(line_mbr,poly_mbr,line_key_pts)
     m_isu_insert[Map_id,Entity_mbr](entity_mbrs,map_id,entity_mbr)
     writeString("'b'b |")
  end

  let tmp_pid_point := m_empty[Point_id,XY](eq_int,lt_int)
  let tmp_lid_line := m_empty[Line_id,List[XY]](eq_int,lt_int)

!-------------------------------------------------------------------------!
!                                                                         !
!   Construct coordinate tables for point and line entities               !
!                                                                         !
!-------------------------------------------------------------------------!
  let build_tmp_entity := proc(map_id: Map_id)
  begin
     load_basemap(map_id)
     let basemap = m_find[Map_id,Basemap](base_maps,map_id)
     let dm_name = basemapDM_name(basemap(data_model))
     case dm_name of
     "link_node" :
```

```
    begin
       let build_pid_point = proc(point_id: Point_id; ln_point: LN_point)
       begin
          let geom_id = ln_point(geom_id)
          let point =
              l_first[XY](m_find[Geom_id,LN_geometry](ln_gid_geometry,
                                                     geom_id)(xy_list))
          m_isu_insert[Point_id,XY](tmp_pid_point,point_id,point)
       end
       m_app[Point_id,LN_point](ln_pid_point,build_pid_point)

       let build_lid_line = proc(line_id: Line_id; ln_line: LN_line)
       begin
          let geom_id = ln_line(geom_id)
          let xy_list = m_find[Geom_id,LN_geometry](ln_gid_geometry,
                                                    geom_id)(xy_list)
          m_isu_insert[Line_id,List[XY]](tmp_lid_line,line_id,xy_list)
       end
       m_app[Line_id,LN_line](ln_lid_line,build_lid_line)
    end

    "polygon_based" :
    begin
       let build_pid_point :=
           proc(geom_id: Geom_id; pb_geometry: PB_geometry)
       begin
          let gtype = pb_geometry(gtype)
          if gtype = 1 do
             { let point = l_first[XY]((pb_geometry)(xy_list))
               m_isu_insert[Point_id,XY](tmp_pid_point,geom_id,point) }
       end
       m_app[Geom_id,PB_geometry](pb_gid_geometry,build_pid_point)

       let build_lid_line :=
           proc(geom_id: Geom_id; pb_geometry: PB_geometry)
       begin
          let gtype = pb_geometry(gtype)
          if gtype = 2 do
             {let xy_list = pb_geometry(xy_list)
              m_isu_insert[Line_id,List[XY]](tmp_lid_line,geom_id,xy_list)}
       end
       m_app[Geom_id,PB_geometry](pb_gid_geometry,build_lid_line)
    end

    "spaghetti" :
    begin
       let build_pid_point = proc(point_id: Point_id; sp_point: SP_point)
          {m_isu_insert[Point_id,XY](tmp_pid_point,point_id,sp_point(xy))}
       m_app[Point_id,SP_point](sp_pid_point,build_pid_point)

       let build_lid_line = proc(line_id: Line_id; sp_line: SP_line)
          {m_isu_insert[Line_id,List[XY]](tmp_lid_line,line_id,
                                          sp_line(xy_list))}
       m_app[Line_id,SP_line](sp_lid_line,build_lid_line)
    end
    default : { }
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Initialize a point index table                                        !
!                                                                         !
!-------------------------------------------------------------------------!
  let initialize_point_index := proc(peano: Peanor;
                                     pid_point: Map[Point_id,XY];
                                     point_index: Map[Peanor,List[Point_id]])
```

```
  begin
     let pointid_list := l_make[Point_id]()
     if ~m_isEmpty[Point_id,XY](pid_point) do
     begin
        let appendPointID = proc(point_id: Point_id; point: XY)
           { pointid_list := l_prepend[Point_id](point_id,pointid_list) }
        m_app[Point_id,XY](pid_point,appendPointID)
        pointid_list := l_reverse[Point_id](pointid_list)
     end
     m_isu_insert[Peanor,List[Point_id]](point_index,peano,pointid_list)
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Initialize a line index table                                         !
!                                                                         !
!-------------------------------------------------------------------------!
  let initialize_line_index := proc(peano: Peanor;
                                    line_mbr: Map[Line_id,MBR];
                                    line_index: Map[Peanor,List[Line_id]])
  begin
     let lineid_list := l_make[Line_id]()
     if ~m_isEmpty[Line_id,MBR](line_mbr) do
     begin
        let appendLineID := proc(line_id: Line_id; mbr: MBR)
           { lineid_list := l_prepend[Line_id](line_id,lineid_list) }
        m_app[Line_id,MBR](line_mbr,appendLineID)
        lineid_list := l_reverse[Line_id](lineid_list)
     end
     m_isu_insert[Peanor,List[Line_id]](line_index,peano,lineid_list)
  end
!-------------------------------------------------------------------------!
!                                                                         !
!   Initialize a polygon index table                                      !
!                                                                         !
!-------------------------------------------------------------------------!
  let initialize_polygon_index :=
      proc(peano: Peanor;
           poly_mbr: Map[Poly_id,MBR];
           polygon_index: Map[Peanor,List[Poly_id]])
  begin
     let polyid_list := l_make[Poly_id]()
     if ~m_isEmpty[Poly_id,MBR](poly_mbr) do
     begin
        let appendPolyID := proc(poly_id: Poly_id; mbr: MBR)
           { polyid_list := l_prepend[Poly_id](poly_id,polyid_list) }
        m_app[Poly_id,MBR](poly_mbr,appendPolyID)
        polyid_list := l_reverse[Poly_id](polyid_list)
     end
     m_isu_insert[Peanor,List[Poly_id]](polygon_index,peano,polyid_list)
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Initialize an entity index table for a basemap                        !
!                                                                         !
!-------------------------------------------------------------------------!
  let initialize_entity_index :=
      proc(map_id: Map_id;
           line_mbr: Map[Line_id,MBR];
           poly_mbr: Map[Poly_id,MBR];
           point_index: Map[Peanor,List[Point_id]];
           line_index: Map[Peanor,List[Line_id]];
           polygon_index: Map[Peanor,List[Poly_id]])
  begin
     let map_extent = getOSmapInfo(map_id)(extent)
```

```
      let sl = map_extent(x_range)
      let x = float(truncate(map_extent(x_min) / sl)) * sl
      let y = float(truncate(map_extent(y_min) / sl)) * sl
      let xy = XY(x,y)
      let peano_key = xyToPKR(xy)
      let peano = Peanor(peano_key,sl)
      initialize_point_index(peano,tmp_pid_point,point_index)
      initialize_line_index(peano,line_mbr,line_index)
      initialize_polygon_index(peano,poly_mbr,polygon_index)
   end

   let gt_than_point_threshold := proc(peano: Peanor;
                                       pointid_list: List[Point_id] -> bool)
   begin
      let num = l_length[Point_id](pointid_list)
      let tf = if num > pt_ndx_threshold then true else false
      tf
   end

   let gt_than_line_threshold := proc(peano: Peanor;
                                      lineid_list: List[Line_id] -> bool)
   begin
      let num = l_length[Line_id](lineid_list)
      let tf = if num > ln_ndx_threshold then true else false
      tf
   end

   let gt_than_poly_threshold := proc(peano: Peanor;
                                      polyid_list: List[Poly_id] -> bool)
   begin
      let num = l_length[Poly_id](polyid_list)
      let tf = if num > poly_ndx_threshold then true else false
      tf
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Construct point,line and polygon index tables for a basemap           !
!                                                                         !
!-------------------------------------------------------------------------!
   let construct_entity_index := proc(map_id: Map_id)
   begin
      m_isu_clear[Point_id,XY](tmp_pid_point)
      m_isu_clear[Line_id,List[XY]](tmp_lid_line)
      build_tmp_entity(map_id)
      let point_index := m_empty[Peanor,List[Point_id]](eq_peanor,lt_peanor)
      let line_index := m_empty[Peanor,List[Line_id]](eq_peanor,lt_peanor)
      let polygon_index := m_empty[Peanor,List[Poly_id]](eq_peanor,lt_peanor)
      let line_mbr = m_find[Map_id,Entity_mbr](entity_mbrs,map_id)(line)
      let poly_mbr = m_find[Map_id,Entity_mbr](entity_mbrs,map_id)(polygon)
      let line_key_pts = m_find[Map_id,Entity_mbr](entity_mbrs,
                                                   map_id)(key_pts)
      initialize_entity_index(map_id,line_mbr,poly_mbr,point_index,
                                              line_index,polygon_index)

      let map_length = getOSmapInfo(map_id)(extent)(x_range)
      let pt_min_qsl := 0.
      let ln_min_qsl := 0.
      let poly_min_qsl := 0.
      ! build a LQT-coded point index table
      if ~m_isEmpty[Point_id,XY](tmp_pid_point) do
      begin
         let side_length := map_length
         let buildPointIndex := proc(peano: Peanor;
                                     pointid_list: List[Point_id])
            { lqtNdxPoint(peano,pointid_list,tmp_pid_point,point_index) }
```

```
    writeString("'nIndexing points ...")
    writeString("'nSide_length   Total_quadrants   Gt_than_threshold")
    let finished := false
    while ~finished  do
    begin
       let ndx_tmp1 = point_index
       let ndx_tmp2 = m_filter[Peanor,List[Point_id]](point_index,
                                                  gt_than_point_threshold)
       newline(1);writeString(fformat(side_length,6,3));space(8);
       writeInt(m_length[Peanor,List[Point_id]](ndx_tmp1));space(10);
       writeInt(m_length[Peanor,List[Point_id]](ndx_tmp2))
       if m_length[Peanor,List[Point_id]](ndx_tmp2) > 0 then
       begin
          side_length := side_length / 2.
          point_index := m_diff[Peanor,List[Point_id]](ndx_tmp1,ndx_tmp2)
          m_app[Peanor,List[Point_id]](ndx_tmp2,buildPointIndex)
       end
       else { finished := true }
    end
    writeString("'nMin_side_length = ");
    writeString(fformat(side_length,9,3));space(5);
    writeString("The number of quadrants = ");
    writeInt(m_length[Peanor,List[Point_id]](point_index));newline(1)
    pt_min_qsl := side_length
end

! build a LQT-coded line index table
if ~m_isEmpty[Line_id,List[XY]](tmp_lid_line) do
begin
    let side_length := map_length
    let buildLineIndex := proc(peano: Peanor; lid_list: List[Line_id])
       {lqtNdxLine(peano,lid_list,tmp_lid_line,line_mbr,line_key_pts,
                                                        line_index)}
    writeString("'nIndexing lines ...")
    writeString("'nSide_length   Total_quadrants   Gt_than_threshold")
    let finished := false
    while ~finished  do
    begin
       let ndx_tmp1 = line_index
       let ndx_tmp2 = m_filter[Peanor,List[Line_id]](line_index,
                                                  gt_than_line_threshold)
       newline(1);writeString(fformat(side_length,6,3));space(8);
       writeInt(m_length[Peanor,List[Line_id]](ndx_tmp1));space(10);
       writeInt(m_length[Peanor,List[Line_id]](ndx_tmp2))
       if m_length[Peanor,List[Line_id]](ndx_tmp2) > 0 then
       begin
          side_length := side_length / 2.
          line_index := m_diff[Peanor,List[Line_id]](ndx_tmp1,ndx_tmp2)
          m_app[Peanor,List[Line_id]](ndx_tmp2,buildLineIndex)
       end
       else { finished := true }
    end
    writeString("'nMin_side_length = ");
    writeString(fformat(side_length,9,3));space(5);
    writeString("The number of quadrants = ");
    writeInt(m_length[Peanor,List[Line_id]](line_index));newline(1)
    ln_min_qsl := side_length
end

! build a LQT-coded polygon index table
if ~m_isEmpty[Poly_id,MBR](poly_mbr) do
begin
    let side_length := map_length
    let buildPolyIndex := proc(peano: Peanor;
                               polyid_list: List[Poly_id])
       { lqtNdxPoly(peano,polyid_list,poly_mbr,polygon_index) }
```

```
        writeString("'nIndexing polygons ...")
        writeString("'nSide_length   Total_quadrants   Gt_than_threshold")
        let finished := false
        while ~finished  do
        begin
           let ndx_tmp1 = polygon_index
           let ndx_tmp2 = m_filter[Peanor,List[Poly_id]](polygon_index,
                                                    gt_than_poly_threshold)
           newline(1);writeString(fformat(side_length,6,3));space(8);
           writeInt(m_length[Peanor,List[Poly_id]](ndx_tmp1));space(10);
           writeInt(m_length[Peanor,List[Poly_id]](ndx_tmp2))
           if m_length[Peanor,List[Poly_id]](ndx_tmp2) > 0 then
           begin
              side_length := side_length / 2.
              polygon_index := m_diff[Peanor,List[Poly_id]](ndx_tmp1,
                                                            ndx_tmp2)
              m_app[Peanor,List[Poly_id]](ndx_tmp2,buildPolyIndex)
           end
           else { finished := true }
        end
        writeString("'nMin_side_length = ");
        writeString(fformat(side_length,9,3));space(5);
        writeString("The number of quadrants = ");
        writeInt(m_length[Peanor,List[Poly_id]](polygon_index));newline(1)
        poly_min_qsl := side_length
     end
     let min_quad_sl := Min_quad_sl(pt_min_qsl,ln_min_qsl,poly_min_qsl)
     let entity_index := Entity_index(point_index,line_index,polygon_index,
                                                                min_quad_sl)
     m_isu_insert[Map_id,Entity_index](entity_indices,map_id,entity_index)
  end

!-------------------------------------------------------------------------!
!                                                                         !
!   Search for a point                                                    !
!                                                                         !
!-------------------------------------------------------------------------!
  let searchPoint := proc(point_index: Map[Peanor,List[Point_id]];
                          pid_point: Map[Point_id,XY]; ht_col: pixel;
                          min_side_length, map_length, aperture: real;
                          fd: file; window: image; win_size: Win_size;
                          draw_extent: Extent; start: int -> Point_id)
  begin
     let result := -1    !   > 0 : found;  = 0 : not found;   < 0 : quit
     let side_length := min_side_length
     let pt = getPoint(fd,window,win_size,draw_extent,start)
     let x = pt(x); let y = pt(y)
     let finished := if x = 0. and y = 0. then true else false
     while ~finished do
     begin
        let xy = XY((float(truncate(x / side_length))) * side_length,
                    (float(truncate(y / side_length))) * side_length)
        let peano_key = xyToPKR(xy)
        let peano = Peanor(peano_key,side_length)
        if m_contains[Peanor,List[Point_id]](point_index,peano) do
        begin
           let pointid_list := m_find[Peanor,List[Point_id]](point_index,
                                                             peano)
           while ~finished and pointid_list isnt empty do
           begin
              let point_id = hd[Point_id](pointid_list)
              let point = m_find[Point_id,XY](pid_point,point_id)
              let target_win = MBR(x - aperture / 2., y - aperture / 2.,
                                   x + aperture / 2., y + aperture / 2.)
              let found = pointInWindow(point,target_win)
              if found do
```

```
            begin
               drawPoint(point,ht_col,window,draw_extent)
               result := point_id
               finished := true
            end
            pointid_list := tl[Point_id](pointid_list)
         end
      end
      side_length := side_length * 2.0
      if side_length > map_length do
      begin
         result := 0
         finished := true
         writeString("'nThe map database does not contain points " ++
                     "in this area. 'n")
      end
   end
   result
end

!-------------------------------------------------------------------------!
!                                                                         !
!   Search for a line                                                     !
!                                                                         !
!-------------------------------------------------------------------------!
let searchLine := proc(line_index: Map[Peanor,List[Line_id]];
                       lid_line: Map[Line_id,List[XY]];
                       line_mbr: Map[Line_id,MBR];
                       ht_col: pixel; min_side_length,
                       map_length, aperture: real;
                       fd: file; window: image; win_size: Win_size;
                       draw_extent: Extent; start: int -> Line_id)
begin
   let result := -1    !   > 0 : found;  = 0 : not found;   < 0 : quit
   let side_length := min_side_length
   let pt = getPoint(fd,window,win_size,draw_extent,start)
   let x = pt(x); let y = pt(y)
   let finished := if x = 0. and y = 0. then true else false
   while ~finished do
   begin
      let xy = XY((float(truncate(x / side_length))) * side_length,
                  (float(truncate(y / side_length))) * side_length)
      let peano_key = xyToPKR(xy)
      let peano = Peanor(peano_key,side_length)
      if m_contains[Peanor,List[Line_id]](line_index,peano) do
      begin
         let lineid_list := m_find[Peanor,List[Line_id]](line_index,peano)
         while ~finished and lineid_list isnt empty do
         begin
            let line_id = hd[Line_id](lineid_list)
            let xy_list = m_find[Line_id,List[XY]](lid_line,line_id)
            let mbr = m_find[Line_id,MBR](line_mbr,line_id)
            let found = lineStrThroughWindow(xy_list,mbr,pt,aperture)
            if found do
            begin
               drawLineString(xy_list,ht_col,window,draw_extent)
               result := line_id
               finished := true
            end
            lineid_list := tl[Line_id](lineid_list)
         end
      end
      side_length := side_length * 2.0
      if side_length > map_length do
      begin
         result := 0
```

```
            finished := true
            writeString("'nThe map database does not contain lines" ++
                        " in this area. 'n")
         end
      end
      result
   end

!
! highlight a chain
!
   let htChain = proc(chain_id: Chain_id; pb_chain: PB_chain)
   begin
      let link_list := pb_chain(link_list)
      while link_list isnt empty do
      begin
         let geom_id_of_link = hd[PB_link](link_list)(geom_id_of_link)
         let xy_list = m_find[Geom_id,PB_geometry](pb_gid_geometry,
                                                   geom_id_of_link)(xy_list)
         let link_col := red
         drawLineString(xy_list,link_col,screen(ras_depth|vec_depth),
                                                              draw_extent)
         link_list := tl[PB_link](link_list)
      end
   end

!
! highlight a simple polygon
!
   let htPolygon = proc(poly_id: Poly_id; pb_polygon: PB_polygon)
   begin
      let pb_chain = m_find[Chain_id,PB_chain](pb_cid_chain,poly_id)
      htChain(poly_id,pb_chain)
      if m_contains[Attr_id,PB_attribute](pb_aid_attribute,poly_id) do
      begin
         let HA = m_find[Attr_id,PB_attribute](pb_aid_attribute,poly_id)(HA)
         let PI = m_find[Attr_id,PB_attribute](pb_aid_attribute,poly_id)(PI)
         let PI_position =
                   l_first[XY](m_find[Geom_id,PB_geometry](pb_gid_geometry,
                                                          poly_id)(xy_list))
         let id_str = iformat(PI)
         writeString(" Label = ");space(6 - length(id_str));
         writeString(id_str); writeString(";  Area = ");
         writeString(fformat(HA,10,2));writeString(" Hectares")
         let txt = iformat(PI)
         let map_length = getOSmapInfo(map_id)(extent)(x_range)
         let txt_ht = txt_size * map_length / 1000.
         drawText(txt,txt_ht,0.0,red,PI_position,
                  screen(ras_depth|vec_depth),draw_extent)
      end
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Search for a polygon                                                  !
!                                                                         !
!-------------------------------------------------------------------------!
   let searchPolygon := proc(polygon_index: Map[Peanor,List[Poly_id]];
                             poly_mbr: Map[Poly_id, MBR];
                             line_mbr: Map[Line_id, MBR];
                             ht_col: pixel; min_side_length, map_length: real;
                             fd: file; window: image; win_size: Win_size;
                             draw_extent: Extent; start: int -> Poly_id)
   begin
      let result := -1    !   > 0 : found;  = 0 : not found;   < 0 : quit
      let side_length := min_side_length
```

```
      let pt = getPoint(fd,window,win_size,draw_extent,start)
      let x = pt(x); let y = pt(y)
      let finished := if x = 0. and y = 0. then true else false
      while ~finished do
      begin
         let xy = XY((float(truncate(x / side_length))) * side_length,
                     (float(truncate(y / side_length))) * side_length)
         let peano_key = xyToPKR(xy)
         let peano = Peanor(peano_key,side_length)
         if m_contains[Peanor,List[Poly_id]](polygon_index,peano) do
         begin
            let polyid_list := m_find[Peanor,List[Poly_id]](polygon_index,
                                                                peano)
            while ~finished and polyid_list isnt empty do
            begin
               let poly_id = hd[Poly_id](polyid_list)
               let mbr = m_find[Poly_id,MBR](poly_mbr,poly_id)
               let pb_polygon =
                       m_find[Poly_id,PB_polygon](pb_polyid_polygon,poly_id)
               if pointInWindow(pt,mbr) then
               begin
                  let in_poly = pointInPolygon(pt,pb_polygon,pb_cid_chain,
                                               pb_gid_geometry,line_mbr)
                  case in_poly of
                  true :
                  begin
                     writeString("'nThe point is inside the polygon ID = ");
                     writeInt(poly_id);newline(1)
                     let pb_polygon =
                        m_find[Poly_id,PB_polygon](pb_polyid_polygon,poly_id)
                     htPolygon(poly_id,pb_polygon)
                     result := poly_id
                     finished := true
                  end
                  false : { polyid_list := tl[Poly_id](polyid_list) }
                  default: {}
               end
               else { polyid_list := tl[Poly_id](polyid_list) }
            end
         end
         side_length := side_length * 2.0
         if side_length > map_length do
         begin
            result := 0
            finished := true
            writeString("'nThe map database does not contain polygons " ++
                        "in this area.'n")
         end
      end
      result
   end

!-------------------------------------------------------------------------!
!                                                                         !
!   Check whether a file is readable                                      !
!                                                                         !
!-------------------------------------------------------------------------!
   let fileReadable := proc(fn: string -> bool)
   begin
      let suceed := true
      let inputfile = open(fn,0)
      if inputfile = nilfile then
      begin
         let message =
               "The file " ++
               fn ++
```

```
                "cannot be opened!'nCheck the file whether is existent " ++
                "and readable."
            let prompt = ""
            let trash = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
            suceed := false
        end
        else
            { let ndx = close(inputfile) }
        suceed
    end

!-------------------------------------------------------------------------!
!                                                                         !
!  Waiting for an input from either a click of mouse button 2 or          !
!  a keystroke of the ESC or Enter key                                    !
!                                                                         !
!-------------------------------------------------------------------------!
    let inputWaiting := proc(fd: file)
    begin
        use makeReadEnv (fd) with
            inputPending : proc (-> bool);
            readChar : proc (-> string)   in
        begin
            let data := vector 1 to 7 of 0
            let finished := false
            while ~finished do
            begin
                while ~inputPending() and data(6) = 0 do
                    { locator(fd,data) }
                if data(6) = 1 then { finished := true }
                else
                begin
                    let char = readChar()
                    let asc = stringToAscii(char)
                    if asc = 10 or asc = 27 do { finished := true }
                end
            end
        end
    end


! define a default colourmap
    map_colourmap()
    image_colourmap()


!#########################################################################
!#                                                                       #
!#  Main  program                                                        #
!#                                                                       #
!#########################################################################

!*************************************************************************
!*                                                                       *
!*  VIEW & QUERY                                                         *
!*                                                                       *
!*************************************************************************
    let view_query := proc()       ! VIEW & QUERY
    begin
        !=================================================================!
        let view_maps := proc()     ! View maps
        begin
            ! colourmap for vector data (16 colours)
            map_colourmap()
            ! setup background colour for displaying maps
```

```
let bg_col =  olive
! redefine colour
for i = 0 to nc_ras - 1 do    ! light gray
 { colourMap(window_file, ras_cndx(i)++bg_col,
                                        192*256*256+192*256+192) }
let bg = image win_width by win_height of bg_col
let dm_name := ""
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let load_map := proc()      ! Load a map
begin
   copy bg onto screen(ras_depth|vec_depth)
   map_loaded := false
   if ~m_isEmpty[Map_id,Basemap](base_maps) then
   begin
      let id_str := ""
      let no_lines := 1
      let prtMapId = proc(map_id: Map_id; basemap: Basemap)
      begin
         id_str := id_str ++ map_id ++ " "
         if length(id_str) >= 50 * no_lines do
         begin
            id_str := id_str ++ "'n"
            no_lines := no_lines + 1
         end
      end
      m_app[Map_id,Basemap](base_maps,prtMapId)
      let message =
          "Map database contains the following basemaps:'n" ++
          id_str
      let prompt = "Enter a map name: "
      map_id := dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
      if ~m_contains[Map_id,Basemap](base_maps,map_id) then
      begin
         if map_id ~= "" do
         begin
            let message = "Map database does not contain " ++ map_id
            let prompt = ""
            let trash = dialogueBox(message,prompt,window_file,
                                    screen,win_size,start_bp)
       end
      end
      else
      begin
         load_basemap(map_id)
         get_draw_extent(map_id)
         drawRectangle(map_mbr,black,screen(ras_depth|vec_depth),
                                                      draw_extent)
         writeString("'n'nDrawing the map ...'n")
         draw_basemap(map_id)
         let message = "Loading entity indices of the map ..."
         let msg_img = writeMessage(message,window_file,screen,
                                                win_size,start_bp)
         m_isu_clear[Point_id,XY](tmp_pid_point)
         m_isu_clear[Line_id,List[XY]](tmp_lid_line)
         build_tmp_entity(map_id)
         eraseMessage(msg_img,screen)
         writeString("'nDone!'n")
         map_loaded := true
      end
   end
   else
   begin
      let message = "No basemap available."
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
```

```
                                                    win_size,start_bp)
   end
end
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let zoom_map := proc()     ! Zoom a map
begin
   if map_loaded then
   begin
      let zoom := proc(opt: string)
      begin
         draw_extent := Extent(x_min,y_min,x_max - x_min,
                                                      y_max - y_min)
         let new_extent = getZoomExtent(opt,window_file,screen,
                           win_size,draw_extent,map_extent,pntr_bp)
         if (new_extent(x_min) = draw_extent(x_min) and
             new_extent(y_min) = draw_extent(y_min) and
             new_extent(x_range) = draw_extent(x_range) and
             new_extent(y_range) = draw_extent(y_range))  or
             (new_extent(x_range) = 0. or new_extent(y_range) = 0.)
         then { }
         else
         begin
            x_min := new_extent(x_min)
            y_min := new_extent(y_min)
            x_max := x_min + new_extent(x_range)
            y_max := y_min + new_extent(y_range)
            draw_extent := Extent(x_min, y_min,
                                  x_max - x_min, y_max - y_min)
            copy bg onto screen(ras_depth|vec_depth)
            drawRectangle(map_mbr,black,screen(ras_depth|vec_depth),
                                                      draw_extent)
            writeString("Redrawing features ...'n")
            draw_basemap(map_id)
            writeString("Done!'n")
         end
      end
      let a = proc(); zoom("a")
      let c = proc(); { for i = 1 to 50000 do { }; zoom("c") }
      let p = proc(); { for i = 1 to 50000 do { }; zoom("p") }
      let x = proc(); zoom("x")
      let w = proc(); { for i = 1 to 50000 do { }; zoom("w") }
      let items = vector @ 0 of ["Zoom", "All", "Center", "Pan",
                                          "X","Window","Exit"]
      let actions = vector @ 0 of [a,c,p,x,w]
      popupMenu(items,actions,true,window_file,screen,win_size,
                                                      start_bp)
   end
   else
   begin
      let message = "A map should be loaded first."
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
                                          win_size,start_bp)
   end
end
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let query_map := proc()     ! Query
begin
   if map_loaded then
   begin
      if ~m_contains[Map_id,Entity_index](entity_indices,map_id) then
      begin
         let message = "The map " ++ map_id ++
                       " has not been indexed yet."
         let prompt = ""
         let trash = dialogueBox(message,prompt,window_file,screen,
```

```
                                                    win_size,start_bp)
        end
        else
        begin
           let entity_index =
                    m_find[Map_id,Entity_index](entity_indices,map_id)
           let map_extent = getOSmapInfo(map_id)(extent)
           let map_length = map_extent(x_range)
           draw_extent := Extent(x_min,y_min,
                                 x_max - x_min,y_max - y_min)
           let aperture = snap_target_size * (x_max - x_min) /
                                              float(win_size(width))
           let ht_col = red
           !-----------------------------------------------------------!
           let query_point := proc()    ! Query a point
           begin
              let point_index = entity_index(point)
              let min_side_length = entity_index(min_quad_sl)(point)
              writeString("'nMouse button 1:  digitise a point.")
              writeString("'nMouse button 2:  exit.'n")
              let finished := false
              while ~finished do
              begin
                 let pointid = searchPoint(point_index,tmp_pid_point,
                       ht_col,min_side_length,map_length,aperture,
                       window_file,screen,win_size,draw_extent,pntr_bp)
                 if pointid > 0 then
                 begin
                    writeString("'nSearched Point ID = ");
                    writeString(iformat(pointid));space(3);
                 end
                 else if pointid < 0 do { finished := true }
              end
           end
           !-----------------------------------------------------------!
           let query_line := proc()     ! Query a line
           begin
              let line_index = entity_index(line)
              let min_side_length = entity_index(min_quad_sl)(line)
              let line_mbr = m_find[Map_id,Entity_mbr](entity_mbrs,
                                                       map_id)(line)
              writeString("'nMouse button 1:  digitise a line.")
              writeString("'nMouse button 2:  exit.'n")
              let finished := false
              while ~finished do
              begin
                 let lineid = searchLine(line_index,tmp_lid_line,
                                line_mbr,ht_col,min_side_length,
                                map_length,aperture,window_file,
                                screen,win_size,draw_extent,pntr_bp)
                 if lineid > 0 then
                 begin
                    writeString("'nSearched ID = ");
                    writeString(iformat(lineid))
                 end
                 else if lineid < 0 do { finished := true }
              end
           end
           !-----------------------------------------------------------!
           let query_polygon := proc()      ! Query a polygon
           begin
              let polygon_index = entity_index(polygon)
              let map_origin_peano =
                  Peanor(xyToPKR(XY(map_extent(x_min),
                                    map_extent(y_min))),map_length)
              if m_length[Peanor,List[Poly_id]](polygon_index) = 1 and
```

```
      l_length[Poly_id](m_find[Peanor,List[Poly_id]](
      polygon_index,map_origin_peano)) = 0 then
      { writeString("'nThe basemap does not contain " ++
                    "polygon entity.'n") }
   else
   begin
      let min_side_length =
          entity_index(min_quad_sl)(polygon)
      let line_mbr = m_find[Map_id,Entity_mbr](entity_mbrs,
                                               map_id)(line)
      let poly_mbr = m_find[Map_id,Entity_mbr](entity_mbrs,
                                             map_id)(polygon)
      writeString("'nMouse button 1:  digitise a polygon.")
      writeString("'nMouse button 2:  exit.'n")
      let finished := false
      while ~finished do
      begin
         let polyid = searchPolygon(polygon_index,poly_mbr,
                          line_mbr,ht_col,min_side_length,
                          map_length,window_file,screen,
                          win_size,draw_extent,pntr_bp)
         if polyid > 0 then
         begin
            writeString("'nSearched ID = ");
            writeString(iformat(polyid))
         end
         else if polyid < 0 do { finished := true }
      end
   end
end
!-------------------------------------------------------------!
let rectangle_region := proc()
begin
   let rectangle = dynaGetWinCornersA(window_file,screen,
                                win_size,draw_extent,pntr_bp)
   let x := vector 0 to 1 of 0.
   let y := vector 0 to 1 of 0.
   x(0) := rectangle(1)(x)
   y(0) := rectangle(1)(y)
   x(1) := rectangle(2)(x)
   y(1) := rectangle(2)(y)
   let target_win = MBR(x(0),y(0),x(1),y(1))
   let map_extent = getOSmapInfo(map_id)(extent)
   let map_length = map_extent(x_range)
   let point_index = entity_index(point)
   let line_index = entity_index(line)
   let polygon_index = entity_index(polygon)
   let line_mbr = m_find[Map_id,Entity_mbr](entity_mbrs,
                                            map_id)(line)
   let poly_mbr = m_find[Map_id,Entity_mbr](entity_mbrs,
                                           map_id)(polygon)
   let point_msl = entity_index(min_quad_sl)(point)
   let line_msl = entity_index(min_quad_sl)(line)
   let poly_msl = entity_index(min_quad_sl)(polygon)
   !
   ! search points
   !
   writeString("'nThe following points have been found:'n")
   let sl := point_msl
   let cell_xmin := float(truncate(x(0) / sl)) * sl
   let cell_ymin := float(truncate(y(0) / sl)) * sl
   let cell_xmax := float(truncate(x(1) / sl)) * sl
   let cell_ymax := float(truncate(y(1) / sl)) * sl
   let m := truncate((cell_xmax - cell_xmin) / sl)
   let n := truncate((cell_ymax - cell_ymin) / sl)
   for i = 0 to m do
```

```
for j = 0 to n do
begin
   let x0 = cell_xmin + float(i) * sl
   let y0 = cell_ymin + float(j) * sl
   let x_sw := x0
   let y_sw := y0
   let finished := false
   while ~finished do
   begin
      let xy = XY(x_sw,y_sw)
      let peano_key = xyToPKR(xy)
      let peano = Peanor(peano_key,sl)
      if m_contains[Peanor,List[Point_id]](point_index,
                                           peano) do
      begin
         let pointid_list :=
             m_find[Peanor,List[Point_id]](point_index,
                                           peano)
         while pointid_list isnt empty do
         begin
            let point_id = hd[Point_id](pointid_list)
            let point =
                m_find[Point_id,XY](tmp_pid_point,
                                    point_id)
            let found = pointInWindow(point,target_win)
            if found do
            begin
               let id_str = iformat(point_id)
               space(10 - length(id_str));
               writeString(id_str);
            end
            pointid_list := tl[Point_id](pointid_list)
         end
         finished := true
      end
      sl := sl * 2.0
      x_sw := float(truncate(x0 / sl)) * sl
      y_sw := float(truncate(y0 / sl)) * sl
      if sl > map_length do { finished := true }
   end
   sl := point_msl
end
newline(1)
!
! search lines
!
let id_table := m_empty[int,int](eq_int,lt_int)
writeString("'nThe following lines have been found:'n")
sl := line_msl
cell_xmin := float(truncate(x(0) / sl)) * sl
cell_ymin := float(truncate(y(0) / sl)) * sl
cell_xmax := float(truncate(x(1) / sl)) * sl
cell_ymax := float(truncate(y(1) / sl)) * sl
m := truncate((cell_xmax - cell_xmin) / sl)
n := truncate((cell_ymax - cell_ymin) / sl)
for i = 0 to m do
for j = 0 to n do
begin
   let x0 = cell_xmin + float(i) * sl
   let y0 = cell_ymin + float(j) * sl
   let x_sw := x0
   let y_sw := y0
   let finished := false
   while ~finished do
   begin
      let xy = XY(x_sw,y_sw)
```

```
            let peano_key = xyToPKR(xy)
            let peano = Peanor(peano_key,sl)
            if m_contains[Peanor,List[Line_id]](line_index,
                                                peano) do
            begin
               let lineid_list :=
                   m_find[Peanor,List[Line_id]](line_index,
                                                peano)
               while lineid_list isnt empty do
               begin
                  let line_id = hd[Line_id](lineid_list)
                  let xy_list :=
                      m_find[Line_id,List[XY]](tmp_lid_line,
                                               line_id)
                  let mbr = m_find[Line_id,MBR](line_mbr,
                                               line_id)
                  let found := false
                  let pt1 := hd[XY](xy_list)
                  xy_list := tl[XY](xy_list)
                  while xy_list isnt empty and ~found do
                  begin
                     let pt2 = hd[XY](xy_list)
                     found := lineVisibleInWindow(pt1,pt2,
                                                target_win)
                     pt1 := pt2
                     xy_list := tl[XY](xy_list)
                  end
                  if found and ~m_contains[int,int](id_table,
                                                line_id) do
                  begin
                     let id_str = iformat(line_id)
                     space(10 - length(id_str));
                     writeString(id_str);
                     let xy_list =
                        m_find[Line_id,List[XY]](tmp_lid_line,
                                                line_id)
                     drawLineString(xy_list,ht_col,screen,
                                               draw_extent)
                     m_isu_insert[int,int](id_table,line_id,
                                                line_id)
                  end
                  lineid_list := tl[Line_id](lineid_list)
               end
               finished := true
            end
            sl := sl * 2.0
            x_sw := float(truncate(x0 / sl)) * sl
            y_sw := float(truncate(y0 / sl)) * sl
            if sl > map_length do { finished := true }
         end
         sl := line_msl
      end
      newline(1)
      !
      ! search polygons
      !
      let map_origin_peano =
          Peanor(xyToPKR(XY(map_extent(x_min),
                            map_extent(y_min))),map_length)
      if m_length[Peanor,List[Poly_id]](polygon_index) = 1 and
         l_length[Poly_id](m_find[Peanor,List[Poly_id]](
         polygon_index,map_origin_peano)) = 0 then
         { writeString("'nThe basemap does not contain " ++
                       "polygon entity.'n") }
      else
      begin
```

```
let id_table := m_empty[int,int](eq_int,lt_int)
writeString("'nThe following polygons " ++
                                    "have been found:'n")
sl := poly_msl
cell_xmin := float(truncate(x(0) / sl)) * sl
cell_ymin := float(truncate(y(0) / sl)) * sl
cell_xmax := float(truncate(x(1) / sl)) * sl
cell_ymax := float(truncate(y(1) / sl)) * sl
m := truncate((cell_xmax - cell_xmin) / sl)
n := truncate((cell_ymax - cell_ymin) / sl)
for i = 0 to m do
for j = 0 to n do
begin
   let x0 = cell_xmin + float(i) * sl
   let y0 = cell_ymin + float(j) * sl
   let x_sw := x0
   let y_sw := y0
   let finished := false
   while ~finished do
   begin
      let xy = XY(x_sw,y_sw)
      let peano_key = xyToPKR(xy)
      let peano = Peanor(peano_key,sl)
      if m_contains[Peanor,
             List[Poly_id]](polygon_index,peano) do
      begin
         let polyid_list :=
         m_find[Peanor,List[Poly_id]](polygon_index,
                                                peano)
         while polyid_list isnt empty do
         begin
            let found := false
            let poly_id = hd[Poly_id](polyid_list)
            let mbr = m_find[Poly_id,MBR](poly_mbr,
                                             poly_id)
            let pb_polygon :=
                m_find[Poly_id,
                       PB_polygon](pb_polyid_polygon,
                                             poly_id)
            if mbr(x_max) < target_win(x_min) or
               mbr(x_min) > target_win(x_max) or
               mbr(y_max) < target_win(y_min) or
               mbr(y_min) > target_win(y_max) then {}
            else
            begin
               let found := false
               if mbr(x_min) < target_win(x_min) and
                  mbr(x_max) > target_win(x_max) and
                  mbr(y_min) < target_win(y_min) and
                  mbr(y_max) > target_win(y_max) do
               begin
                  let k := 0
                  let inside := true
                  while inside and k < 4 do
                  begin
                     let p = k div 2
                     let q = k rem 2
                     let pt = XY(x(p),y(q))
                     inside := pointInPolygon(pt,
                            pb_polygon,pb_cid_chain,
                            pb_gid_geometry,line_mbr)
                     k := k + 1
                  end
                  if k = 4 do { found := true }
               end
               if ~found do
```

```
                        begin
                        let pb_chain =
                              m_find[Chain_id,
                              PB_chain](pb_cid_chain,poly_id)
                        let link_list := pb_chain(link_list)
                        while ~found and
                                    link_list isnt empty do
                        begin
                           let geom_id_of_link =
                                 hd[PB_link](link_list)
                                          (geom_id_of_link)
                           let xy_list := m_find[Geom_id,
                                  PB_geometry](pb_gid_geometry,
                                     geom_id_of_link)(xy_list)
                           let pt1 := hd[XY](xy_list)
                           xy_list := tl[XY](xy_list)
                           while ~found and
                                       xy_list isnt empty do
                           begin
                              let pt2 = hd[XY](xy_list)
                              found := lineVisibleInWindow(pt1,
                                           pt2,target_win)
                              pt1 := pt2
                              xy_list := tl[XY](xy_list)
                           end
                           link_list := tl[PB_link](link_list)
                        end
                        end
                        if found and ~m_contains[int,
                                    int](id_table,poly_id) do
                        begin
                           let id_str = iformat(poly_id)
                           writeString("id = ");
                           space(6 - length(id_str));
                           writeString(id_str ++ ";  ");
                           let pb_polygon =
                           m_find[Poly_id,
                                PB_polygon](pb_polyid_polygon,
                                                  poly_id)
                           htPolygon(poly_id,pb_polygon)
                               m_isu_insert[int,int](id_table,
                                           poly_id,poly_id)
                        end
                     end
                     polyid_list := tl[Poly_id](polyid_list)
                  end
                  finished := true
               end
               sl := sl * 2.0
               x_sw := float(truncate(x0 / sl)) * sl
               y_sw := float(truncate(y0 / sl)) * sl
               if sl > map_length do { finished := true }
            end
            sl := poly_msl
         end
      end
   end
end
!-----------------------------------------------------------!
let circle_region := proc()
begin
   let circle = dynaGetCircle(window_file,screen,win_size,
                        draw_extent, pntr_bp)
   writeString("'nThe coordinates of centre are ");
   writeReal(circle(center)(x));writeReal(circle(center)(y))
   writeString("'nThe radius of the circle = ");
   writeReal(circle(radius))
```

```
            end
            !-------------------------------------------------------------!
            let items = vector @ 0 of ["Query Entities","Point","Line",
                                       "Polygon","Rectangle Region",
                                       "Exit"]
            let actions = vector @ 0 of [query_point,query_line,
                                         query_polygon,rectangle_region]
            popupMenu(items,actions,true,window_file,screen,win_size,
                                                              start_bp)
         end
      end
      else
      begin
         let message = "A map should be loaded first."
         let prompt = ""
         let trash = dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
      end
   end
   !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
   let items = vector @ 0 of ["Maps", "Load", "Zoom", "Query", "Exit"]
   let actions = vector @ 0 of [load_map, zoom_map, query_map]
   popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
end
!=====================================================================!
let view_images := proc()     ! View images
begin
   let raster := nilimage
   let refresh = image win_width by win_height of
                                    defaultPixel(off,win_depth)
   let map_overlay := image win_width by win_height of
                                          defaultPixel(off,vec_depth)
   let bg = image win_width by win_height of black
   let r_sxy := XY(0.,0.)
   let v_sxy := XY(0.,0.)
   !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
   let get_image := proc()       ! Get an image
   begin
      if ~m_isEmpty[Image_id,Baseimage](base_images) then
      begin
         let id_str := ""
         let no_lines := 1
         let prtImgId = proc(image_id: Image_id; base_image: Baseimage)
         begin
            id_str := id_str ++ image_id ++ " "
            if length(id_str) >= 50 * no_lines do
            begin
               id_str := id_str ++ "'n"
               no_lines := no_lines + 1
            end
         end
         m_app[Image_id,Baseimage](base_images, prtImgId)
         let message :=
           "The image database contains the following baseimages:'n" ++
           id_str
         let prompt = "Enter an image name: "
         let image_id := dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)

         if ~m_contains[Image_id,Baseimage](base_images,image_id) then
         begin
            if image_id ~= "" do
            begin
               let message = "The database does not contain " ++
                             "the queried image."
               let prompt = ""
```

```
            let trash = dialogueBox(message,prompt,window_file,
                                    screen,win_size,start_bp)
         end
      end
      else
      begin
         ! set background colour to black
         copy bg onto screen(ras_depth|vec_depth)
         raster := loadBaseimage(image_id,window_file)

         let baseimage =
             m_find[Image_id,Baseimage](base_images,image_id)  !!!
         let baseimage_dm = baseimage(data_model)'grid_cell    !!!
         image_extent := baseimage_dm(extent)
         x_min := image_extent(x_min)
         y_min := image_extent(y_min)
         pxl_resol := image_extent(x_range) / float(xDim(raster))
         x_range := pxl_resol * float(win_width)
         y_range := pxl_resol * float(win_height)

         let shift_x = float((win_size(width) - xDim(raster)) div 2)
         let shift_y = float((win_size(height) - yDim(raster)) div 2)
         let shift_pt = XY(shift_x,shift_y)
         x_min := x_min - shift_x * pxl_resol
         y_min := y_min - shift_y * pxl_resol
         x_max := x_min + x_range
         y_max := y_min + y_range
         draw_extent := Extent(x_min,y_min,
                               x_max - x_min,y_max - y_min)
         r_sxy := shift_pt
         viewImage(raster,shift_pt,screen,win_size)
      end
   end
   else
   begin
      let message = "No base image available."
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
   end
end
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let pan_image := proc()      ! Pan an image
begin
   if raster ~= nilimage then
   begin
      let pntr_bp = 4
      let ras_depth = zDim(raster)
      let erase_pxl = defaultPixel(off,ras_depth)
      if ras_depth < win_depth do
      begin
         let img1 = image 1 by 1 of on
         let img2 := image 1 by 1 of
                          defaultPixel(off,win_depth - ras_depth)
         copy img1 onto img2((pntr_bp - ras_depth) | 1)
         let pxl = getPixel(img2,0,0)
         let nc = power_2_k(ras_depth)
         let cndx := vector 0 to nc- 1 of erase_pxl
         for i = 0 to nc - 1 do cndx(i) := colourToPixel(i,ras_depth)
         ! set up the colour of the pointer
         for i = 0 to nc - 1 do
            { colourMap(window_file, cndx(i) ++ pxl,
                           255*256*256 + 255 * 256) } ! cyan
      end
      let done := false
      while ~done do
```

```
      begin
         let dxy = getDragDxy(window_file,screen,win_size,pntr_bp)
         if dxy(x) = 0. and dxy(y) = 0. then { done := true }
         else
         begin
            x_min := x_min - dxy(x) * pxl_resol
            y_min := y_min - dxy(y) * pxl_resol
            x_max := x_min + x_range
            y_max := y_min + y_range
            draw_extent := Extent(x_min,y_min,
                                  x_max - x_min, y_max - y_min)
            r_sxy := XY(r_sxy(x) + dxy(x), r_sxy(y) + dxy(y))
            v_sxy := XY(v_sxy(x) + dxy(x), v_sxy(y) + dxy(y))
            let result = image win_width by win_height of
                                                ras_cndx(0) ++ black
            viewImage(map_overlay,v_sxy,result(ras_depth|vec_depth),
                                                          win_size)
            viewImage(raster,r_sxy,result(0|ras_depth),win_size)
            copy result onto screen
         end
      end
      let new_cursor := image 16 by 16 of off
      line(new_cursor, 0,15,8,0, on, 12)
      line(new_cursor, 0,15,0,9, on, 12)
      line(new_cursor, 0,15,5,12, on, 12)
      setCursor(window_file,new_cursor)
   end
   else
   begin
      let message = "An image should be loaded first."
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
   end
end
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let overlay_map := proc()     ! Overlay
begin
   let message = ""
   let prompt = "Enter a map name: "
   map_id := dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
   if ~m_contains[Map_id,Basemap](base_maps,map_id) then
   begin
      if map_id ~= "" do
      begin
         let message =
             "The database does not contain the queried basemap."
         let prompt = ""
         let trash = dialogueBox(message,prompt,window_file,screen,
                                                   win_size,start_bp)
      end
   end
   else
   begin
      v_sxy := XY(0.,0.)
      load_basemap(map_id)
      writeString("'n'nDrawing the map, please wait ...'n")
      copy screen(0 | ras_depth) onto refresh
      copy refresh onto screen
      draw_basemap(map_id)
      copy screen(ras_depth|vec_depth) onto map_overlay
      writeString("Done!'n")
   end
end
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
```

```
        let items = vector @ 0 of ["Images","Load","Pan",
                                   "Overlay Map","Exit"]
        let actions = vector @ 0 of [get_image, pan_image, overlay_map]
        map_colourmap()
        image_colourmap()
        let bg_col = olive
        for i = 0 to nc_ras - 1 do    ! light gray
         { colourMap(window_file, ras_cndx(i)++bg_col,
                                              192*256*256+192*256+192) }
        popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
     end
     !=========================================================================!
     let items = vector @ 0 of ["View & Query", "Maps", "Images", "Exit"]
     let actions = vector @ 0 of [view_maps, view_images]
     popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
  end

!*****************************************************************************
!*                                                                           *
!*  SPATIAL INDEXING                                                         *
!*                                                                           *
!*****************************************************************************
  let spatial_index := proc()    ! SPATIAL INDEXING
  begin
     !=========================================================================!
     let build_mbrs := proc()     ! Build MBRs of basemaps
     begin
        if ~m_isEmpty[Map_id,Basemap](base_maps) then
        begin
           let createEntityMBR = proc(map_id: Map_id; basemap: Basemap)
           begin
              if ~m_contains[Map_id,Entity_mbr](entity_mbrs,map_id) do
                 { get_entity_mbr(map_id) }
           end
           let message = "Creating MBR tables ... "
           let msg_img = writeMessage(message,window_file,screen,
                                                    win_size,start_bp)
           m_app[Map_id,Basemap](base_maps,createEntityMBR)
           eraseMessage(msg_img,screen)
        end
        else
        begin
           let message = "Basemap database is empty."
           let prompt = ""
           let trash = dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
        end
     end
     !=========================================================================!
     let index_basemaps := proc()    ! Indexing basemaps
     begin
        if ~m_isEmpty[Map_id,Basemap](base_maps) then
        begin
           let constructEntityIndex = proc(map_id: Map_id; basemap: Basemap)
           begin
              if ~m_contains[Map_id,Entity_index](entity_indices,map_id) do
                 { construct_entity_index(map_id) }
           end
           let message = "Constructing index tables ... "
           let msg_img = writeMessage(message,window_file,screen,
                                                    win_size,start_bp)
           m_app[Map_id,Basemap](base_maps,constructEntityIndex)
           eraseMessage(msg_img,screen)
        end
        else
        begin
```

```
      let message = "Basemap database is empty."
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
   end
end
!=============================================================================!
let draw_lqtndx := proc()     ! Draw LQT-indexed diagrams
begin
   if ~m_isEmpty[Map_id,Basemap](base_maps) then
   begin
      let id_str := ""
      let prtMapId = proc(map_id: Map_id; basemap: Basemap)
      begin
         id_str := id_str ++ map_id ++ " "
         let no_lines := 1
         if length(id_str) >= 50 * no_lines do
         begin
            id_str := id_str ++ "'n"
            no_lines := no_lines + 1
         end
      end
      m_app[Map_id,Basemap](base_maps,prtMapId)
      let message =
          "Map database contains the following basemaps:'n" ++ id_str
      let prompt = "Enter a map name: "
      map_id := dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
      if ~m_contains[Map_id,Entity_index](entity_indices,map_id) then
      begin
         if map_id ~= "" do
         begin
            let message = "The map " ++ map_id ++
                          " has not been indexed yet."
            let prompt = ""
            let trash = dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
         end
      end
      else
      begin
         let map_extent = getOSmapInfo(map_id)(extent)
         if x_min = 0. and y_min = 0. and x_max = 0. and y_max = 0. do
            { get_draw_extent(map_id) }
         draw_extent := Extent(x_min,y_min,
                               x_max - x_min, y_max - y_min)
         let drawLQTndx := proc(peano: Peanor; id_list: List[int])
         begin
            let quad_extent = getQuadExtent(peano)
            let xmin = quad_extent(x_min)
            let ymin = quad_extent(y_min)
            let range = quad_extent(x_range)
            let quadrant_mbr = MBR(xmin,ymin,
                                   xmin + range,ymin + range)
            drawRectangle(quadrant_mbr,yellow,
                          screen(ras_depth|vec_depth),draw_extent)
            let num = l_length[int](id_list)
            let position = XY(xmin + range/2., ymin + range/2.)
            ! display the number of entries in each quadrant
            ! from level 0 to 5
            if range >= map_extent(x_range) / 64. do
            begin
               let txt = iformat(num)
               let txt_ht = txt_size * map_extent(x_range) / 1000.
               drawText(txt,txt_ht,0.0,black,position,
                        screen(ras_depth|vec_depth),draw_extent)
```

```
            end
         end

         let entity_index =
                  m_find[Map_id,Entity_index](entity_indices,map_id)
         let draw_point_lqtndx = proc()
         begin
            let point_index = entity_index(point)
            m_app[Peanor,List[Point_id]](point_index,drawLQTndx)
         end
         let draw_line_lqtndx = proc()
         begin
            let line_index = entity_index(line)
            m_app[Peanor,List[Line_id]](line_index,drawLQTndx)
         end
         let draw_polygon_lqtndx = proc()
         begin
            let polygon_index = entity_index(polygon)
            m_app[Peanor,List[Point_id]](polygon_index,drawLQTndx)
         end
         let bg_col = olive
         for i = 0 to nc_ras - 1 do    ! light gray
            { colourMap(window_file,ras_cndx(i)++bg_col,
                                    192*256*256+192*256+192) }
         let bg = image win_width by win_height of bg_col
         let erase_screen = proc()
             { copy bg onto screen(ras_depth|vec_depth) }
         let items = vector @ 0 of ["Draw LQT Diagrams","Point","Line",
                                    "Polygon","Erase Screen","Exit"]
         let actions =
             vector @ 0 of [draw_point_lqtndx,draw_line_lqtndx,
                            draw_polygon_lqtndx,erase_screen]
         popupMenu(items,actions,true,window_file,screen,win_size,
                                                          start_bp)
      end
   end
   else
   begin
      let message = "No basemap available."
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
   end
end
!=========================================================================!
let prt_lqtndx := proc()    ! Display LQT-indexed information
begin
   if ~m_isEmpty[Map_id,Basemap](base_maps) then
   begin
      let id_str := ""
      let prtMapId = proc(map_id: Map_id; basemap: Basemap)
      begin
         id_str := id_str ++ map_id ++ " "
         let no_lines := 1
         if length(id_str) >= 50 * no_lines do
         begin
            id_str := id_str ++ "'n"
            no_lines := no_lines + 1
         end
      end
      m_app[Map_id,Basemap](base_maps,prtMapId)
      let message =
          "Map database contains the following basemaps:'n" ++ id_str
      let prompt = "Enter a map name: "
      map_id := dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
```

```
if ~m_contains[Map_id,Entity_index](entity_indices,map_id) then
begin
   if map_id ~= "" do
   begin
      let message =
          "The map " ++ map_id ++ " has not been indexed yet."
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
   end
end
else
begin
   let entity_index =
       m_find[Map_id,Entity_index](entity_indices,map_id)
   let lqt_index := m_empty[Peanor,List[int]](eq_peanor,lt_peanor)
   let min_side_length := 0.
   let map_extent = getOSmapInfo(map_id)(extent)

   let linearPK = proc(peano: Peanor; id_list: List[int])
   begin
      ! linearize peano key
      let quad_extent = getQuadExtent(peano)
      let quad_xmin = quad_extent(x_min)
      let quad_ymin = quad_extent(y_min)
      let quad_range = quad_extent(x_range)
      let xr = (quad_xmin - map_extent(x_min)) / min_side_length
      let yr = (quad_ymin - map_extent(y_min)) / min_side_length
      let linear_pk_key = xyToPKR(XY(xr,yr))
      let linear_peano = Peanor(linear_pk_key, quad_range)
      m_isu_insert[Peanor,List[int]](lqt_index,linear_peano,
                                                    id_list)
   end

   let printLinearPK = proc(peano: Peanor; id_list: List[int])
   begin
      writeString(fformat(peano(peano_key),10,0)); space(4);
      writeString(fformat(peano(side_length),6,3));space(8);
      writeInt(l_length[int](id_list))
      newline(1)
   end

   let opt := 0
   writeString("Display Point(P), Line(L) or polyGon(G) " ++
               "linearized peano information? ");
   let ans = readLine()
   case ans of
   "p", "P":
   begin
      opt := 1
      let point_index = entity_index(point)
      min_side_length := entity_index(min_quad_sl)(point)
      m_app[Peanor,List[Point_id]](point_index,linearPK)
   end
   "l", "L":
   begin
      opt := 2
      let line_index = entity_index(line)
      min_side_length := entity_index(min_quad_sl)(line)
      m_app[Peanor,List[Line_id]](line_index,linearPK)
   end
   "g", "G":
   begin
      opt := 3
      let polygon_index = entity_index(polygon)
      min_side_length := entity_index(min_quad_sl)(polygon)
```

```
                m_app[Peanor,List[Point_id]](polygon_index,linearPK)
             end
             default: { }
             if opt >= 1 and opt <= 3 do
             begin
                writeString("'nLinearized PK   Side_length    " ++
                            "No of entries 'n'n")
                m_app[Peanor,List[int]](lqt_index,printLinearPK)
                m_isu_clear[Peanor,List[int]](lqt_index)
             end
          end
       end
       else
       begin
          let message = "No basemap available."
          let prompt = ""
          let trash = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
       end
    end
    !=========================================================================!
    let items = vector @ 0 of ["Spatial Indexing", "Build MBRs",
                               "Indexing Maps", "Draw LQT Diagrams",
                               "Print LQT index", "Exit"]
    let actions = vector @ 0 of [build_mbrs, index_basemaps, draw_lqtndx,
                                                          prt_lqtndx]
    popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
  end

!*****************************************************************************
!*                                                                           *
!*  MANAGEMENT                                                               *
!*                                                                           *
!*****************************************************************************
  let management := proc()      ! MANAGEMENT
  begin
    !=========================================================================!
    let remove_image := proc()      ! Remove an image from the database
    begin
       let items = vector @ 0 of ["Remove Images", "Raw Image",
                                  "Interim Image", "Baseimage", "Exit"]
       let actions = vector @ 0 of [removeRawimage, removeInterimImage,
                                                        removeBaseimage]
       popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
    end
    !=========================================================================!
    let clear_mbr_tables := proc()  ! Clear MBR tables
    begin
       let clear_mbr = proc(map_id: Map_id; entity_mbr: Entity_mbr)
       begin
          let line_mbr = entity_mbr(line)
          let poly_mbr = entity_mbr(polygon)
          m_isu_clear[Line_id,MBR](line_mbr)
          m_isu_clear[Poly_id,MBR](poly_mbr)
       end
       let message = "Are you sure you wish to clear all mbr tables? "
       let prompt = "Confirm with YES or NO? "
       let confirm = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
       if confirm = "YES" do
       begin
          m_app[Map_id,Entity_mbr](entity_mbrs,clear_mbr)
          m_isu_clear[Map_id,Entity_mbr](entity_mbrs)
          let message = "All mbr tables have been erased!"
          let prompt = ""
          let trash = dialogueBox(message,prompt,window_file,screen,
```

```
                                                        win_size,start_bp)
   end
end
!=============================================================================!
let clear_ndx_tables := proc()  ! Clear indexing tables
begin
   let clear_ndx = proc(map_id: Map_id; entity_index: Entity_index)
   begin
      let point_index = entity_index(point)
      let line_index = entity_index(line)
      let polygon_index = entity_index(polygon)
      m_isu_clear[Peanor,List[Point_id]](point_index)
      m_isu_clear[Peanor,List[Line_id]](line_index)
      m_isu_clear[Peanor,List[Poly_id]](polygon_index)
   end
   let message = "Are you sure you wish to clear all index tables? "
   let prompt = "Confirm with YES or NO? "
   let confirm = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
   if confirm = "YES" do
   begin
      m_app[Map_id,Entity_index](entity_indices,clear_ndx)
      m_isu_clear[Map_id,Entity_index](entity_indices)
      let message = "All index tables have been erased!"
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
   end
end
!=============================================================================!
let map_statistics := proc()  ! Map statistics
begin
   integerWidth := 6
   spaceWidth := 5
   let message = ""
   let prompt = "Enter a map name: "
   map_id := dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
   if ~m_contains[Map_id,Basemap](base_maps,map_id) then
   begin
      if map_id ~= "" do
      begin
         let message =
             "The database does not contain the queried basemap."
         let prompt = ""
         let trash = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
      end
   end
   else
   begin
      load_basemap(map_id)
      let basemap = m_find[Map_id,Basemap](base_maps,map_id)
      let dm_name = basemapDM_name(basemap(data_model))
      case dm_name of
      "link_node" :
      begin
         writeString("'nThe number of elements in the tables: 'n " ++
                     " Point       Line    Geometry   Attribute  " ++
                     " Link    Node      Text       FCD 'n" ++
                     " ----------------------------------------" ++
                     "----------------------------'n"); space(3)
         writeInt(m_length[Point_id,LN_point](ln_pid_point));
         writeInt(m_length[Line_id,LN_line](ln_lid_line));
         writeInt(m_length[Geom_id,LN_geometry](ln_gid_geometry));
         writeInt(m_length[Attr_id,LN_attribute](ln_aid_attribute));
```

```
            writeInt(m_length[Link_id,LN_link](ln_kid_link));
            writeInt(m_length[Node_id,LN_node](ln_nid_node));
            let basemap_dm = basemap(data_model)'link_node
            if basemap_dm(txt) is ln_tid_text then
               { writeInt(m_length[Text_id,LN_text](ln_tid_text));}
            else { writeInt(0); }
            writeInt(m_length[FC,FD](basemap_dm(fcd))); newline(1)
         end
         "polygon_based" :
         begin
            writeString("'nThe number of elements in the tables: 'n" ++
                        " Geometry   Attribute   Polygon   Chain " ++
                        " Cpolygon   Collection     FCD 'n" ++
                        " ------------------------------------" ++
                        "---------------------------------'n"); space(3)
            let basemap_dm = basemap(data_model)'polygon_based
            writeInt(m_length[Geom_id,PB_geometry](pb_gid_geometry));
            writeInt(m_length[Attr_id,PB_attribute](pb_aid_attribute));
            writeInt(m_length[Poly_id,PB_polygon](pb_polyid_polygon));
            writeInt(m_length[Chain_id,PB_chain](pb_cid_chain));
            writeInt(m_length[Cpoly_id,PB_cpolygon](pb_cpolyid_cpolygon));
            writeInt(m_length[Coll_id,
                              PB_collection](pb_collid_collection));
            writeInt(m_length[FC,FD](basemap_dm(fcd))); newline(1)
         end
         "spaghetti" :
         begin
            writeString("'nThe number of elements in the tables: 'n" ++
                        " Point      Line       Text       FCD 'n" ++
                        "-------------------------------------'n");
            space(3)
            writeInt(m_length[Point_id,SP_point](sp_pid_point));
            writeInt(m_length[Line_id,SP_line](sp_lid_line));
            let basemap_dm = basemap(data_model)'spaghetti
            if basemap_dm(txt) is sp_tid_text then
               { writeInt(m_length[Text_id,SP_text](sp_tid_text));}
            else { writeInt(0); }
            writeInt(m_length[FC,FD](basemap_dm(fcd))); newline(1)
         end
         default : { }
      end
   end
   let prt_MI_ndx := proc()    ! print map and image indices
   begin
      let prtMapNdx = proc(peano: Peano; map_id: Map_id)
      begin
         writeString(iformat(peano(peano_key))); space(4);
         writeString(fformat(peano(side_length),6,3));space(8);
         writeString(map_id)
         newline(1)
      end
      let prtImgNdx = proc(peano: Peano; img_id_list: List[Image_id])
      begin
         writeString(iformat(peano(peano_key))); space(4);
         writeString(fformat(peano(side_length),6,3));space(8);
         while img_id_list isnt empty do
         begin
            let img_id = hd[Image_id](img_id_list)
            writeString(img_id); space(2);
            img_id_list := tl[Image_id](img_id_list)
         end
         newline(1)
      end
      writeString("Display Map(M), Image(I) or Quit(Q)? ");
      let ans = readLine()
      case ans of
```

```
        "m", "M": {m_app[Peano,Map_id](basemap_indices,prtMapNdx)}
        "i", "I": {m_app[Peano,List[Image_id]](baseimage_indices,prtImgNdx)}
        default : { }
    end
    !=========================================================================!
    let items = vector @ 0 of ["Management", "Remove Maps", "Remove Images",
                               "Raw to Interim Conversion",
                               "Interim to Baseimage Conversion",
                               "Clear MBR tables", "Clear Index tables",
                               "Map Statistics",
                               "Print Map and Image Indices", "Exit"]
    let actions = vector @ 0 of [removeBasemap, remove_image,
                                 rawToInterimImage, interimToBaseimage,
                                 clear_mbr_tables, clear_ndx_tables,
                                 map_statistics, prt_MI_ndx]
    popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
  end

!*****************************************************************************
!*                                                                           *
!*  PREPROCESSING                                                            *
!*                                                                           *
!*****************************************************************************
  let preprocess := proc()    ! PREPROCESSING
  begin
    !=========================================================================!
    let process_raw := proc()   ! Raw image
    begin
      if ~m_isEmpty[Image_id,Rawimage](raw_images) then
      begin
        let id_str := ""
        let no_lines := 1
        let prtImgId = proc(image_id: Image_id; raw_image: Rawimage)
        begin
          id_str := id_str ++ image_id ++ " "
          if length(id_str) >= 50 * no_lines do
          begin
            id_str := id_str ++ "'n"
            no_lines := no_lines + 1
          end
        end
        m_app[Image_id,Rawimage](raw_images, prtImgId)
        let message :=
            "Image database contains the following raw images:'n" ++
            id_str
        let prompt = "Enter a raw image name: "
        let image_id := dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
        if ~m_contains[Image_id,Rawimage](raw_images,image_id) then
        begin
          if image_id ~= "" do
          begin
            let message = "Image database does not contain " ++
                          image_id
            let prompt = ""
            let trash = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
          end
        end
        else
        begin
          let rawimage = m_find[Image_id,Rawimage](raw_images,image_id)
          let img_width = rawimage(width)
          let img_height = rawimage(height)
          let img_depth = rawimage(depth)
          writeString("Width = ");writeInt(img_width);
```

```
writeString("Height = ");writeInt(img_height);
writeString("Depth = ");writeInt(img_depth); newline(1)
let nc = power_2_k(img_depth)
let ct = rawimage(colourmap)
let default_pixel = defaultPixel(off,img_depth)
let colour_index := vector 0 to nc-1 of default_pixel
for i = 0 to nc-1 do
    { colour_index(i) := colourToPixel(i,img_depth) }

let bg = image win_width by win_height of black
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let preview_raw := proc()    ! Preview a raw image
begin
   let message = "Loading the raw image ..."
   let msg_img = writeMessage(message,window_file,screen,
                                                win_size,start_bp)
   let tmp_img := image win_width by win_height of
                                    defaultPixel(off,win_depth)
   copy previewRaw(rawimage,win_width,win_height) onto tmp_img
   eraseMessage(msg_img,screen)
   for i = 0 to nc-1 do
      { colourMap(window_file,colour_index(i),
                          ct(i,3)*256*256+ct(i,2)*256+ct(i,1)) }
   copy tmp_img onto screen
   inputWaiting(window_file)
   map_colourmap()
   image_colourmap()
end
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let preview_LCS_raw  := proc()     ! Preview and linear-contrast
                                   ! stretch a raw image
begin
   let message = "Performing a linear-contrast stretch ..."
   let msg_img = writeMessage(message,window_file,screen,
                                                win_size,start_bp)
   let tmp_img := image win_width by win_height of
                                    defaultPixel(off,win_depth)
   copy previewStretchedRaw(rawimage,win_width,
                                      win_height) onto tmp_img
   eraseMessage(msg_img,screen)
   for i = 0 to nc-1 do
      { colourMap(window_file,colour_index(i),
                          ct(i,3)*256*256+ct(i,2)*256+ct(i,1)) }
   copy tmp_img onto screen
   inputWaiting(window_file)
   map_colourmap()
   image_colourmap()
end
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let LCS_reduce_raw := proc()      ! Linear-contrast stretch and
                                  ! reduce the depth of a raw
                                  ! image
begin
   let msg1 = ""
   let prmpt1 = "Enter a new depth: "
   let new_depth = stringToInt(dialogueBox(msg1,prmpt1,
                        window_file,screen, win_size,start_bp))
   let nc = power_2_k(new_depth)
   let ct = grayLevel(nc)
   let default_pixel = defaultPixel(off,new_depth)
   let colour_index := vector 0 to nc-1 of default_pixel
   for i = 0 to nc-1 do colour_index(i) :=
                                    colourToPixel(i,new_depth)
   for i = 0 to nc-1 do
      { colourMap(window_file,colour_index(i),
                          ct(i,3)*256*256+ct(i,2)*256+ct(i,1)) }
```

```
            let msg2 = "Determining the frequency of luminance " ++
                       "itensity value ..."
            let msg2_img = writeMessage(msg2,window_file,screen,
                                                win_size,start_bp)
            let frequency = freqCount(rawimage)
            eraseMessage(msg2_img,screen)
            let msg3 = "Performing a linear-contrast stretch and 'n" ++
                       "remapping to a new depth ..."
            let msg3_img = writeMessage(msg3,window_file,screen,
                                                win_size,start_bp)
            let result = linearStretch(rawimage,frequency,new_depth)
            !  project the central part of the image on the screen
            let xr := 0; let yr :=0; let xs := 0; let ys := 0
            if img_width < win_width then
               { xs := (win_width - img_width) div 2 }
            else { xr := (img_width - win_width) div 2 }
            if img_height < win_height then
                { ys := (win_height - img_height) div 2 }
            else { yr := (img_height - win_height) div 2 }
            copy bg onto screen(ras_depth|vec_depth)
            copy limit result at xr,yr onto limit screen at xs,ys
            inputWaiting(window_file)
            let msg4 ="Do you want to store the result? "
            let prmpt4 = "Confirm with (Y) or (N): "
            let ans = dialogueBox(msg4,prmpt4,
                                  window_file,screen,win_size,start_bp)
            if ans = "Y" or ans = "y" do
            begin
               let interim_image = Interim_image(result,ct)
               if ~m_contains[Image_id,Interim_image](interim_images,
                                                        image_id)
               then
                  { m_isu_insert[Image_id,Interim_image](interim_images,
                                              image_id,interim_image) }
               else
               begin
                  let msg5 ="Image database already contains " ++
                            image_id ++ ", do you want to update it? "
                  let prmpt5 = "Confirm with (Y) or (N) :"
                  let ans = dialogueBox(msg5,prmpt5,window_file,screen,
                                                win_size,start_bp)
                  if ans = "Y" or ans = "y" do
                   {m_isu_assign[Image_id,Interim_image](interim_images,
                                              image_id,interim_image)}
               end
            end
            map_colourmap()
            image_colourmap()
         end
         !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
         let items = vector @ 0 of ["Raw Image", "Original",
                                    "Linear-Contrast Stretch",
                                    "Linear-Contrast Stretch and Reduce",
                                    "Exit"]
         let actions = vector @ 0 of [preview_raw, preview_LCS_raw,
                                      LCS_reduce_raw]
         popupMenu(items,actions,true,window_file,screen,
                                                win_size,start_bp)
      end
   end
   else
   begin
      let message ="No raw image available."
      let prompt = ""
      let trash = dialogueBox(message,prompt,window_file,screen,
                                                win_size,start_bp)
```

```
   end
end
!=============================================================================!
let process_interim := proc()     ! Interim image
begin
   if ~m_isEmpty[Image_id,Interim_image](interim_images) then
   begin
      let id_str := ""
      let no_lines := 1
      let prtImgId =
          proc(image_id: Image_id; interim_image: Interim_image)
      begin
         id_str := id_str ++ image_id ++ " "
         if length(id_str) >= 50 * no_lines do
         begin
            id_str := id_str ++ "'n"
            no_lines := no_lines + 1
         end
      end
      m_app[Image_id,Interim_image](interim_images, prtImgId)
      let message :=
          "Image database contains the following interim images:'n" ++
          id_str
      let prompt = "Enter an interim image name: "
      let image_id := dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
      if ~m_contains[Image_id,Interim_image](interim_images,image_id)
      then
         if image_id ~= "" do
         begin
            let message = "Image database does not contain " ++
                          image_id
            let prompt = ""
            let trash = dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
         end
      else
      begin
         let interim_image =
             m_find[Image_id,Interim_image](interim_images,image_id)
         let raster = interim_image(raster)
         let img_width = xDim(raster)
         let img_height = yDim(raster)
         let img_depth = zDim(raster)
         writeString("Width = ");writeInt(img_width);
         writeString("Height = ");writeInt(img_height);
         writeString("Depth = ");writeInt(img_depth); newline(1)
         let nc = power_2_k(img_depth)
         let ct = interim_image(colourmap)
         let default_pixel = defaultPixel(off,img_depth)
         let colour_index := vector 0 to nc-1 of default_pixel
         for i = 0 to nc-1 do
            { colour_index(i) := colourToPixel(i,img_depth) }
         for i = 0 to nc-1 do
            { colourMap(window_file,colour_index(i),
                              ct(i,3)*256*256+ct(i,2)*256+ct(i,1)) }
         !  project the central part of the image on the screen
         let xr := 0; let yr :=0; let xs := 0; let ys := 0
         if img_width < win_width then
            { xs := (win_width - img_width) div 2 }
         else { xr := (img_width - win_width) div 2 }
         if img_height < win_height then
            { ys := (win_height - img_height) div 2 }
         else { yr := (img_height - win_height) div 2 }
         let bg = image win_width by win_height of ras_cndx(0) ++
                  black
```

```
copy bg onto screen
copy limit raster at xr,yr onto limit screen at xs,ys
inputWaiting(window_file)
let overwrite := false
let result := image img_width by img_height of default_pixel
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let LCS_reduce_interim := proc()  ! Linear-contrast stretch
                                  ! and reduce an interim image
begin
   let msg1 = ""
   let prmpt1 = "Enter a new depth: "
   let new_depth = stringToInt(dialogueBox(msg1,prmpt1,
                     window_file,screen, win_size,start_bp))

   let nc = power_2_k(new_depth)
   let ct = grayLevel(nc)
   let default_pixel = defaultPixel(off,new_depth)
   let colour_index := vector 0 to nc-1 of default_pixel
   for i = 0 to nc-1 do colour_index(i) :=
                                    colourToPixel(i,new_depth)
   for i = 0 to nc-1 do
      { colourMap(window_file,colour_index(i),
                     ct(i,3)*256*256+ct(i,2)*256+ct(i,1)) }

   let msg2 = "Determining the frequency of luminance " ++
              "itensity value ..."
   let msg2_img = writeMessage(msg2,window_file,screen,
                                          win_size,start_bp)
   let frequency = freqCount2(raster)
   eraseMessage(msg2_img,screen)
   let msg3 = "Performing a linear-contrast stretch and 'n" ++
              "remapping to a new depth ..."
   let msg3_img = writeMessage(msg3,window_file,screen,
                                          win_size,start_bp)
   result := linearStretch2(raster,frequency,new_depth)
   copy bg onto screen(ras_depth|vec_depth)
   copy limit result at xr,yr onto limit screen at xs,ys
   let msg4 ="Do you want to save the result? "
   let prmpt4 = "Confirm with (Y) or (N) :"
   let ans = dialogueBox(msg4,prmpt4, window_file,screen,
                                         win_size,start_bp)
   if ans = "Y" or ans = "y" do { overwrite := true }
end
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
let trim_interim := proc()    ! Trim an interim image
begin
   let msg1 = "Enter the number of pixels to be trimmed off "++
              "from the margins: "
   let erase = image win_width by win_height of colour_index(0)
   let finish := false
   while ~finish do
   begin
      let prmpt1 = "Left margin: "
      let left = stringToInt(dialogueBox(msg1,prmpt1,
                     window_file,screen, win_size,start_bp))
      let prmpt2 = "Right margin: "
      let right = stringToInt(dialogueBox(msg1,prmpt2,
                     window_file,screen, win_size,start_bp))
      let prmpt3 = "Top margin: "
      let top = stringToInt(dialogueBox(msg1,prmpt3,
                     window_file,screen, win_size,start_bp))
      let prmpt4 = "Bottom margin: "
      let bottom = stringToInt(dialogueBox(msg1,prmpt4,
                     window_file,screen, win_size,start_bp))
      copy erase onto screen
      let new_width := img_width - left - right
```

```
            let new_height := img_height - top - bottom
            copy bg onto screen(ras_depth|vec_depth)
            result := image new_width by new_height of default_pixel
            copy limit raster to new_width by new_height at
                                              left,bottom onto result
            copy result onto screen
            let xs = if new_width > win_width
                     then new_width - win_width else 0
            let ys = if new_height > win_height
                     then new_height - win_height else 0
            if xs > 0 or ys > 0 do
            begin
               copy limit result at 0,ys onto screen
               copy limit result at xs,ys onto screen
               copy limit result at xs,0 onto screen
            end
            let msg5 = ""
            let prmpt5 = "Trim it again? "
            let ans = dialogueBox(msg5,prmpt5,window_file,screen,
                                                 win_size,start_bp)
            if ans ="N" or ans = "n" do
            begin
               finish := true
               let message = ""
               let prompt = "Save the result? "
               let ans = dialogueBox(message,prompt,window_file,
                                    screen, win_size,start_bp)
               if ans = "Y" or ans = "y" do
               begin
                  overwrite := true
                  let interim_image = Interim_image(result,ct)
                  m_isu_assign[Image_id,
                               Interim_image](interim_images,
                                              image_id,interim_image)
                  let message =
                      "The original image has been overwritten."
                  let prompt = ""
                  let trash = dialogueBox(message,prompt,window_file,
                                         screen,win_size,start_bp)
               end
            end
         end
         map_colourmap()
         image_colourmap()
      end
      !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
      let items = vector @ 0 of ["Interim Image",
                                 "Linear-Contrast Stretch and Reduce",
                                 "Trim", "Exit"]
      let actions = vector @ 0 of [LCS_reduce_interim, trim_interim]
      popupMenu(items,actions,true,window_file,screen,
                                              win_size,start_bp)
      if overwrite do
      begin
         let interim_image = Interim_image(result,ct)
         m_isu_assign[Image_id,Interim_image](interim_images,
                                              image_id,interim_image)
         let message = "The original image has been overwritten."
         let prompt = ""
         let trash = dialogueBox(message,prompt,window_file,screen,
                                              win_size,start_bp)
         overwrite := false
      end
   end
end
else
```

```
        begin
           let message = "No interim image available."
           let prompt = ""
           let trash = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
        end
     end
     !=========================================================================!
     let items = vector @ 0 of ["Preprocess", "Raw", "Interim", "Exit"]
     let actions = vector @ 0 of [process_raw, process_interim]
     popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
  end

!*****************************************************************************
!*                                                                           *
!*  IMPORT / EXPORT                                                          *
!*                                                                           *
!*****************************************************************************
  let conversion := proc()    ! IMPORT / EXPORT
  begin
     !=========================================================================!
     let import_map_data := proc()     ! Import map data
     begin
        !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
        let import_ntf := proc()    ! NTF V2.0 map data
        begin
           let message = ""
           let prompt ="Enter a file name (.ntf): "
           let  map_id = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
           let proceed := true
           let fn = map_id ++ ".ntf"
           if fileReadable(fn)  do
           begin
              if m_contains[Map_id,Basemap](base_maps,map_id) do
              begin
                 let message = "Map database already contains " ++
                               map_id ++ ", overwrite it?"
                 let prompt = "YES (= Proceed) or Else(= Quit)? "
                 let confirm = dialogueBox(message,prompt,window_file,
                                    screen, win_size,start_bp)
                 if confirm = "YES" then { proceed := true }
                 else  { proceed := false }
              end
              if proceed do
              begin
                 let os_map_info = getOSmapInfo(map_id)
                 map_extent := os_map_info(extent)
                 case  os_map_info(series)  of
                 "s_625k" :
                 begin
                    let message = "Importing 1:625,000 scale map data " ++
                                  "to map database ..."
                    let msg_img = writeMessage(message,window_file,screen,
                                                        win_size,start_bp)
                    let basemap = ntf625kToBasemap(fn)
                    storeBasemap(map_id,basemap,map_extent)
                    eraseMessage(msg_img,screen)
                 end
                 "s_250k" :
                 begin
                    let message = "Importing 1:250,000 scale map data " ++
                                  "to map database ..."
                    let msg_img = writeMessage(message,window_file,screen,
                                                        win_size,start_bp)
                    let basemap = ntf250kToBasemap(fn)
```

```
                storeBasemap(map_id,basemap,map_extent)
                eraseMessage(msg_img,screen)
             end
             "s_50k" :
             begin
                let message = "Importing 1:50,000 scale map data " ++
                              "to map database ..."
                let msg_img = writeMessage(message,window_file,screen,
                                           win_size,start_bp)
                let basemap = ntfcontourToBasemap(fn)
                storeBasemap(map_id,basemap,map_extent)
                eraseMessage(msg_img,screen)
             end
             "boundary_line" :
             begin
                let message = "Importing 1:10,000 scale map data " ++
                              "to map database ..."
                let msg_img = writeMessage(message,window_file,screen,
                                           win_size,start_bp)
                let basemap = ntfblToBasemap(fn)
                storeBasemap(map_id,basemap,map_extent)
                eraseMessage(msg_img,screen)
             end
             "s_10k", "s_2500", "s_1250" :
             begin
                let message =
                    "Importing 1:10,000, 1:2,500, or 1:1,250 scale'n" ++
                    "landline data to basemap database."
                let msg_img = writeMessage(message,window_file,screen,
                                           win_size,start_bp)
                let basemap = ntfllToBasemap(fn)
                storeBasemap(map_id,basemap,map_extent)
                eraseMessage(msg_img,screen)
             end
             "oscar" :
             begin
                let message = "Importing OSCAR data to basemap database."
                let msg_img = writeMessage(message,window_file,screen,
                                           win_size,start_bp)
                let basemap = ntfoscarToBasemap(fn)
                storeBasemap(map_id,basemap,map_extent)
                eraseMessage(msg_img,screen)
             end
             default :
             begin
                let message = map_id ++ " is not NTF map data!"
                let prompt = ""
                let trash = dialogueBox(message,prompt,window_file,
                                        screen,win_size,start_bp)
             end
          end
       end
    end
    !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
    let items = vector @ 0 of ["Map Data", "NTF 2.0", "Exit"]
    let actions = vector @ 0 of [import_ntf]
    popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
end
!=====================================================================!
let import_image_data := proc()     ! Import image data
begin
    !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
    let import_fbff := proc()    ! FBFF
    begin
       let message = "Importing an FBFF image to the Raw image database."
       let prmpt1 = "Enter a rawimage file name: "
```

```
      let image_id = dialogueBox(message,prmpt1,window_file,screen,
                                                    win_size,start_bp)
      let prmpt2 = "Enter the width of the image: "
      let img_width = stringToInt(dialogueBox(message,prmpt2,
                                 window_file,screen,win_size,start_bp))
      let prmpt3 = "Enter the height of the image: "
      let img_height = stringToInt(dialogueBox(message,prmpt3,
                                 window_file,screen,win_size,start_bp))
      let prmpt4 = "Enter the depth of the image: "
      let img_depth = stringToInt(dialogueBox(message,prmpt4,
                                 window_file,screen,win_size,start_bp))
      let fn = image_id ++ ".dat"
      if fileReadable(fn) do
      begin
         let rawimage = fbffToRaw(fn,img_width,img_height,img_depth)
         storeRawimage(image_id,rawimage)
      end
   end
   !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
   let import_tiff := proc()    ! TIFF
   begin
      let message = "Importing a TIFF image to the image database.'n" ++
                    "Input to Raw or Interim image database? "
      let prompt = "Enter (r = Raw) or (i = Interim): "
      let ans = dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
      case ans of
      "r","R" :
      begin
         let message = ""
         let prompt = "Enter a file name: "
         let image_id = dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
         let fn = image_id ++ ".tif"
         if fileReadable(fn) do
         begin
            let rawimage = tiffToRaw(fn)
            storeRawimage(image_id,rawimage)
         end
      end
      "i","I" :
      begin
         let message = ""
         let prompt = "Enter a file name: "
         let image_id = dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
         let fn = image_id ++ ".tif"
         if fileReadable(fn) do
         begin
            let interim_image = tiffToInterim(fn)
            storeInterimImage(image_id,interim_image)
         end
      end
      default : { }
   end
   !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
   let import_sunras := proc()    ! SunRas
   begin
      let message =
          "Importing a SunRas image to the Interim image database."
      let prompt = "Enter a file name: "
      let image_id = dialogueBox(message,prompt,window_file,screen,
                                                    win_size,start_bp)
      let fn = image_id ++ ".ras"
      if fileReadable(fn) do
      begin
```

```
            let interim_image = sunrasToInterim(fn)
            storeInterimImage(image_id,interim_image)
         end
      end
      !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
      let import_hsi := proc()    ! HSI
      begin
         let message = "Importing an HSI image to the Interim image " ++
                       "database."
         let prompt = "Enter a file name: "
         let image_id = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
         let fn = image_id ++ ".hsi"
         if fileReadable(fn) do
         begin
            let interim_image = hsiToInterim(fn)
            storeInterimImage(image_id,interim_image)
         end
      end
      !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
      let items = vector @ 0 of ["Image Data", "FBFF", "TIFF", "SunRas",
                                 "HSI", "Exit"]
      let actions = vector @ 0 of [import_fbff, import_tiff, import_sunras,
                                   import_hsi]
      popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
   end
   !====================================================================!
   let export_map_data := proc()     ! Export map data
   begin

   end
   !====================================================================!
   let export_image_data := proc()   ! Export image data
   begin
      !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
      let export_sunras := proc()    ! SunRas
      begin
         let message = "Export an interim image to a SunRas file."
         let prompt = "Enter a file name: "
         let image_id = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
         if m_contains[Image_id,Interim_image](interim_images,image_id)
         then
         begin
            let interim_image =
                m_find[Image_id,Interim_image](interim_images,image_id)
            interimToSunras(interim_image,image_id)
         end
         else
         begin
            let message = "Image database does not contain " ++ image_id
            let prompt = ""
            let trash = dialogueBox(message,prompt,window_file,screen,
                                                        win_size,start_bp)
         end
      end
      !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!
      let items = vector @ 0 of ["Image Data", "SunRas", "Exit"]
      let actions = vector @ 0 of [export_sunras]
      popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
   end
   !====================================================================!
   let items = vector @ 0 of ["Import / Export", "Import Map Data",
                              "Import Image Data", "Export Map Data",
                              "Export Image Data", "Exit"]
   let actions = vector @ 0 of [import_map_data, import_image_data,
```

```
                                    export_map_data, export_image_data]
      popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
  end
!*************************************************************************!
! root window
  let root = image win_width by win_height of navy
  let raster = m_find[Image_id,Interim_image](interim_images,"title")(raster)
  let img_width = xDim(raster)
  let img_height = yDim(raster)
  let title_tmp := image img_width by img_height of off
  let xr = (win_width - img_width) div 2
  let yr = (win_height - img_height) div 2
  copy root onto screen(ras_depth|vec_depth)
  copy limit screen(start_bp | 1) at xr,yr onto title_tmp
  copy raster(0 | 1) onto limit screen(start_bp | 1) at xr,yr
  inputWaiting(window_file)
  copy title_tmp onto limit screen(start_bp|1) at xr,yr

  let items = vector @ 0 of ["Main Menu","View & Query",
                             "Spatial Indexing",
                             "Management","Preprocessing",
                             "Import/Export","Exit"]
  let actions = vector @ 0 of [view_query, spatial_index, management,
                               preprocess, conversion]
  popupMenu(items,actions,true,window_file,screen,win_size,start_bp)
  let void = close(window_file)
end
```