



University
of Glasgow

<https://theses.gla.ac.uk/>

Theses Digitisation:

<https://www.gla.ac.uk/myglasgow/research/enlighten/theses/digitisation/>

This is a digitised version of the original print thesis.

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses

<https://theses.gla.ac.uk/>
research-enlighten@glasgow.ac.uk



UNIVERSITY OF GLASGOW

**THE DESIGN AND IMPLEMENTATION
OF
A PROTOTYPE GEOGRAPHIC INFORMATION SYSTEM
USING A NOVEL ARCHITECTURE
BASED ON PS-ALGOL**

BY

ABDULHAKIM A. ABDALLAH

B. Sc. (Civil Eng.),

P.G.Diploma (Photogrammetry & Remote Sensing)

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
March 1990**

ProQuest Number: 11003390

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 11003390

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

TO
MY MOTHER & FATHER
& TO MY FIANCEE LAI CHING WONG

ACKNOWLEDGMENTS

The author wishes to express his sincere gratitude to his supervisor, Professor G. Petrie, for suggesting this research topic, for his continuous advice and supervision and for supplying most of the materials to carry out the survey of the GIS systems. Without all of this help, this research work would have never come to its present form.

The perseverance and the continuous help of the author's other supervisor, Dr. R. Cooper who also supplied most of the documentation about the PS-algol language is gratefully acknowledged. That he gave so much of his time over the period when he was carrying out his own research and writing up his own Ph. D. made his assistance even more highly appreciated.

The author also wishes to extend his gratitude to Professor M. Atkinson, head of the Department of Computing Science, for granting the use of the equipment and facilities available in the Department without restriction and making them very accessible.

Thanks and gratitude are also due to Professor I. B. Thompson (Head of the Department of Geography & Topographic Science) and to the Topographic Science Staff Members, especially Mr. B.D.F. Methley, Mr. D.A. Tait and Mr. J.W. Shearer, for their help throughout the research period.

Sincere thanks are also due to my fellow research colleagues in the Department of Geography and Topographic Science, in particular Mr. Z. Li and Mr. A. Azizi, for their assistance, comradeship and support.

The author also wishes to acknowledge the help received from the staff and graduate students of the Department of Computing Science who were always ready to help; particularly Dr. Francis Wai, Dr. Paul Philbrow, Mr. Mark Dunlop and Mr. Djamel Abderrahmane.

Finally, the author wishes to thank and extend his sincere gratitude to his sponsor, the Hariri Foundation, over the last five years. It was the Foundation's generosity which enabled me to follow an English language course, then the post graduate Diploma course in

Photogrammetry and Remote Sensing and finally this research work carried out for the Ph.D. degree. Thanks are due especially to the Foundation's representative, Miss Mona Knio, for her continuous concern and encouragement.

ABSTRACT

This thesis is concerned with the design and implementation of a novel architecture for a geographic information system based on the use of a new database language called PS-algol, in conjunction with a hybrid database structure.

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

- i) the definition of a database;
- ii) the components and functions of a database management system;
- iii) the features of PS-algol;
- iv) the new system architecture;
- v) the use of operational management system;
- vi) data entry as carried out by the system;
- vii) the facility for the cartographic representation of features;
- viii) data retrieval and its potential use; and
- ix) the generation of hard-copy output.

The thesis also includes a review of existing geographical information systems against which the novelty of the new approach can be judged.

Table of Contents

VOLUME I

| | page |
|--|------------|
| ACKNOWLEDGEMENTS | i |
| ABSTRACT..... | iii |
| CONTENTS..... | iv |
| CHAPTER 1 : INTRODUCTION | |
| 1.1 Introduction..... | 1 |
| 1.2 GIS - User Requirements..... | 2 |
| 1.3 GIS Data Requirements..... | 3 |
| 1.4 Hardware for GIS..... | 4 |
| 1.5 Software for GIS..... | 6 |
| 1.6 GIS Data Structures..... | 9 |
| 1.7 GIS Software Production..... | 10 |
| 1.7.1 Language Requirements..... | 11 |
| 1.7.2 Data Base Requirements..... | 13 |
| 1.8 Summary..... | 14 |
| CHAPTER 2 : DATABASES AND DATABASE STRUCTURES | |
| 2.1 Introduction..... | 17 |
| 2.2 Data Base Definition..... | 17 |
| 2.2.1 Data Base Architecture..... | 17 |
| 2.3 Basic Data Structures..... | 20 |
| 2.3.1 Definitions of Terms..... | 21 |
| 2.3.2 Simple Lists..... | 21 |
| 2.3.3 Ordered Sequential Files..... | 22 |
| 2.3.4 Indexed Files..... | 23 |
| 2.4 The Classical Database Structures..... | 25 |
| 2.4.1 Hierarchical Data Structure..... | 25 |
| 2.4.2 Network Data Structure..... | 26 |
| 2.4.3 Relational Data Structure..... | 27 |
| 2.5 Database Management System..... | 29 |
| 2.5.1 DBMS Functions..... | 29 |
| 2.5.2 DBMS Software Components..... | 31 |
| 2.6 Query Languages..... | 33 |
| 2.7 Vector Data Bases..... | 35 |
| 2.7.1 Point Entity..... | 35 |

| | |
|------------------------|----|
| 2.7.2 Line Entity..... | 36 |
| 2.7.3 Area Entity..... | 36 |
| 2.8 Discussion..... | 37 |

CHAPTER 3 : STATE-OF-THE-ART IN GIS

| | |
|--|----|
| 3.1 Introduction..... | 38 |
| 3.2 Digital Mapping..... | 38 |
| 3.3 Automated Mapping/Facilities Management (AM/FM) Systems..... | 41 |
| 3.4 Land Information Systems (LIS)..... | 42 |
| 3.5 Geographic Information Systems (GIS)..... | 43 |
| 3.5.1 Summary..... | 44 |
| 3.6 Hierarchical Systems..... | 45 |
| 3.6.1 Intergraph IGDS/DMRS..... | 45 |
| 3.6.1.1 Overall IGDS/DMRS System Design..... | 45 |
| 3.6.1.2 DMRS Overview & Components..... | 46 |
| 3.6.1.3 DMRS Database Organization..... | 48 |
| 3.6.1.4 IGDS..... | 50 |
| 3.6.1.5 IGDS/DMRS System Overview & Interaction..... | 50 |
| 3.6.1.6 The Interface Concept Using the Disk Data Scanner..... | 53 |
| 3.6.1.7 Creation of a Land Database Using the IGDS/DMRS System..... | 54 |
| 3.6.1.8 FRAMME..... | 58 |
| 3.6.2 Synercom Informap..... | 59 |
| 3.6.2.1 Overview of Informap..... | 60 |
| 3.6.2.2 Informap Data Base Organization..... | 63 |
| 3.6.2.3 Odyssey..... | 65 |
| 3.6.2.4 Environmental Mapping Information System (EMIS)..... | 68 |
| 3.7 Network Based Systems..... | 70 |
| 3.7.1 Strings..... | 70 |
| 3.8 Relational Systems..... | 72 |
| 3.8.1 SysScan DNMS..... | 72 |
| 3.8.1.1 DNMS Input/Output..... | 73 |
| 3.8.1.2 DNMS Data Structure..... | 74 |
| 3.8.1.3 System Architecture..... | 74 |
| 3.8.2 ESRI ARC/INFO..... | 74 |
| 3.8.2.1 The Software Tools..... | 76 |
| 3.8.2.2 The GIS Model..... | 77 |
| 3.8.2.3 ARC/INFO Approach..... | 78 |
| 3.9 Object Oriented Systems..... | 79 |
| 3.9.1 SysScan GINIS..... | 80 |
| 3.9.1.1 GINIS- Overall System..... | 80 |
| 3.9.1.2 System Architecture..... | 81 |
| 3.9.1.3 GINIS Data Structure..... | 82 |

| | |
|--|----|
| 3.9.2 Wild System 9..... | 86 |
| 3.9.2.1 System 9 Components..... | 86 |
| 3.9.2.2 Data Base Management System..... | 89 |
| 3.9.2.3 Topology & Data Sharing..... | 89 |
| 3.9.2.4 System 9 Data Base..... | 90 |
| 3.9.2.5 Projects..... | 90 |
| 3.9.2.6 Partitions..... | 90 |
| 3.9.2.7 Features & Feature Classes..... | 91 |
| 3.9.2.8 Simple and Complex Features..... | 91 |
| 3.9.2.9 Attributes..... | 92 |
| 3.9.2.10 Geometric Primitives..... | 93 |
| 3.9.2.11 Themes..... | 94 |
| 3.9.2.12 Summary..... | 94 |
| 3.10 Discussion..... | 94 |
| 3.11 Conclusion..... | 98 |

CHAPTER 4 : PS-ALGOL, THE LANGUAGE

| | |
|---|-----|
| 4.1 Introduction..... | 99 |
| 4.2 Language Design and Aspects..... | 99 |
| 4.3 Applying the Language Design Principles to a Data Base Language..... | 102 |
| 4.3.1 The Idea of Persistence..... | 102 |
| 4.3.2 The Data Objects & the Principle of Completeness..... | 103 |
| 4.3.3 The Conceptual Store..... | 105 |
| 4.3.4 Binding..... | 106 |
| 4.3.5 The Persistent Store & Store Interface..... | 107 |
| 4.3.5.1 The Persistence Store..... | 108 |
| 4.3.5.2 The Persistence Store Interface..... | 108 |
| 4.4 Language Syntax..... | 109 |
| 4.4.1 Identifiers & Object Declarations..... | 109 |
| 4.4.2 Compound Data Objects..... | 110 |
| 4.4.2.1 Vectors..... | 110 |
| 4.4.2.2 Structures..... | 112 |
| 4.4.2.3 Images..... | 112 |
| 4.4.2.4 Assignment and Equality of Pointers..... | 113 |
| 4.4.3 Procedures..... | 113 |
| 4.4.4 Data Persistency..... | 114 |
| 4.4.4.1 Tables..... | 114 |
| 4.4.4.2 Data Base Procedures..... | 115 |
| 4.4.4.3 Data Base Conventions..... | 116 |
| 4.5 Graphics in PS-algol..... | 116 |
| 4.5.1 Pictures..... | 117 |

| | |
|---|-----|
| 4.5.2 Pixels..... | 119 |
| 4.5.3 Images..... | 119 |
| 4.6 Management of Data in PS-algol..... | 120 |
| 4.7 Conclusion..... | 121 |

CHAPTER 5 : SYSTEM ARCHITECTURE

| | |
|--|-----|
| 5.1 Introduction..... | 122 |
| 5.2 General Considerations in Data Structuring..... | 122 |
| 5.3 Data Structure for the Project..... | 123 |
| 5.3.1 Hierarchical Data Structure Used for Feature Coding..... | 124 |
| 5.3.1.1 Feature Coding System..... | 125 |
| 5.3.2 Relational Data Structure Used for Data Entities..... | 127 |
| 5.3.2.1 Data Structure for Point Entities..... | 128 |
| 5.3.2.2 Data Structure for Line Entities..... | 129 |
| 5.3.2.3 Data Structure for Area Entities..... | 129 |
| 5.3.2.4 Data Structure for Text Entities..... | 130 |
| 5.4 Overall System Configuration..... | 131 |
| 5.5 Individual Data Bases Within the Prototype GIS..... | 132 |
| 5.5.1 The Main Data Base (MDB)..... | 133 |
| 5.5.2 'Symbols' Data Base..... | 133 |
| 5.5.2.1 Areal Symbols..... | 135 |
| 5.5.2.2 Point Symbols..... | 138 |
| 5.5.3 'global' Data Base..... | 138 |
| 5.5.3.1 'typemenu'..... | 138 |
| 5.5.3.2 'angmenu'..... | 139 |
| 5.5.3.3 'scalmenu'..... | 139 |
| 5.5.3.4 'hatchspace'..... | 140 |
| 5.5.3.5 'default.menu'..... | 140 |
| 5.5.3.6 'linemenu'..... | 141 |
| 5.5.4 'Code 2' Data Base..... | 142 |
| 5.5.5 '%\$Modules'..... | 144 |
| 5.6 Summary..... | 145 |

CHAPTER 6 : OPERATIONAL MANAGEMENT SYSTEM

| | |
|--|-----|
| 6.1 Introduction..... | 146 |
| 6.2 Operational Management System (OMS)..... | 146 |
| 6.3 OMS Components..... | 146 |
| 6.4 OMS Description..... | 148 |
| 6.5 Global Procedures..... | 149 |
| 6.5.1 Global Procedures Description..... | 150 |
| 6.6 Summary..... | 155 |

CHAPTER 7 : DATA ENTRY AND CODE TRANSFER MODULE

| | | |
|---------|---|-----|
| 7.1 | Introduction..... | 156 |
| 7.2 | Data Formats..... | 156 |
| 7.3 | Digitization and Data Structuring..... | 157 |
| 7.4 | 'trans-code' Module..... | 160 |
| 7.4.1 | The Configuration & Data Entry Operation..... | 163 |
| 7.4.2 | The Code Transfer Operation..... | 166 |
| 7.4.2.1 | Option 'Zoom In'..... | 166 |
| 7.4.2.2 | Option 'Zoom Out'..... | 166 |
| 7.4.2.3 | Option 'Start'..... | 166 |
| 7.4.2.4 | Operational Aspects of Configuration & Data Entry..... | 167 |
| 7.5 | Procedures Used in this Module..... | 174 |
| 7.6 | Summary..... | 180 |

CHAPTER 8 : CARTOGRAPHIC REPRESENTATION MODULE

| | | |
|---------|--|-----|
| 8.1 | Introduction..... | 181 |
| 8.2 | Cartographic Representation of Features..... | 181 |
| 8.2.1 | General Procedures..... | 183 |
| 8.2.2 | Polygon Representation..... | 186 |
| 8.2.2.1 | Hatching Polygons..... | 193 |
| 8.2.2.2 | Filling Polygons..... | 198 |
| 8.2.3 | Line Representation..... | 201 |
| 8.2.4 | Point Representation..... | 207 |
| 8.3 | Scrolling..... | 210 |
| 8.4 | Quit & Storage of Cartographic Symbols..... | 212 |
| 8.5 | Summary..... | 213 |

CHAPTER 9 : DATA RETRIEVAL MODULE

| | | |
|-------|---|-----|
| 9.1 | Introduction..... | 214 |
| 9.2 | Module Architecture..... | 215 |
| 9.3 | Screen Layout..... | 216 |
| 9.4 | Database Opening & Map Selection..... | 220 |
| 9.5 | Zooming..... | 221 |
| 9.6 | Menu Handling..... | 221 |
| 9.7 | Query Processing & Information Retrieval..... | 222 |
| 9.7.1 | Retrieval by Type..... | 223 |
| 9.7.2 | Retrieval by Layer..... | 225 |
| 9.7.3 | Retrieval by Entity..... | 226 |
| 9.7.4 | Retrieval by a Combined Selection from | |

| | |
|--|-----|
| the Type and Layer Menus..... | 230 |
| 9.7.5 Retrieval by a Combined Selection from the Type and Entity Menus..... | 231 |
| 9.8 Displaying the Results..... | 232 |
| 9.9 Saving the Results..... | 232 |
| 9.10 Summary..... | 233 |

CHAPTER 10 : HARD-COPY DATA OUTPUT

| | |
|---|-----|
| 10.1 Introduction..... | 234 |
| 10.2 Module Organization..... | 234 |
| 10.3 Screen Layout..... | 235 |
| 10.4 Selection of Map Images..... | 235 |
| 10.5 Display to Screen..... | 235 |
| 10.6 Retrieval of Map Images..... | 236 |
| 10.7 Selection of Output Devices..... | 236 |
| 10.8 File Generation..... | 236 |
| 10.8.1 Output to a Raster-based Laser Printers..... | 237 |
| 10.8.2 Output to a Vector-based Plotter..... | 239 |
| 10.8.3 PS-algol Picture Data Structures..... | 239 |
| 10.8.4 Use of Ghost..... | 242 |
| 10.8.5 The Plotting Program..... | 243 |
| 10.9 Summary..... | 243 |

CHAPTER 11 : CONCLUSION & RECOMMENDATIONS

PART I: Conclusion

| | |
|--|-----|
| 11.1 Introduction..... | 245 |
| 11.2 Learning the Language..... | 245 |
| 11.3 Properties of the Language..... | 246 |
| 11.4 Modelling the Real World..... | 248 |
| 11.5 Data Storage & Retrieval in PS-algol..... | 248 |
| 11.6 Modular Programming..... | 249 |
| 11.7 Speed of Processing..... | 249 |
| 11.8 The Prototype GIS..... | 250 |

PART II: Recommendations

| | |
|--|-----|
| 11.9 Introduction..... | 251 |
| 11.10 General Recommendations..... | 251 |
| 11.11 Modules Recommendations..... | 251 |
| 11.11.1 Data Entry..... | 252 |
| 11.11.2 Cartographic Representation..... | 252 |

| | | |
|---------|--------------------------|-----|
| 11.11.3 | Data Retrieval..... | 252 |
| 11.11.4 | Data Output..... | 252 |
| 11.11.5 | Applications Module..... | 253 |
| 11.12 | Summary..... | 253 |
| 11.13 | Epilogue..... | 254 |

| | |
|---------------------------|------------|
| BIBLIOGRAPHY | 256 |
|---------------------------|------------|

VOLUME II

APPENDIX A: THE FEATURE CODING SYSTEM

| | |
|-------|--------------------------------------|
| A.1 | Introduction |
| A.2 | The Feature Coding System |
| A.3 | Special Codes for Feature Attributes |
| A.3.1 | Unspecified |
| A.3.2 | Other |
| A.3.3 | Abandoned |
| A.4 | The Feature Coding System Listing |

APPENDIX B: CREATION OF THE DIFFERENT DATABASES

APPENDIX C: GLOBAL PROCEDURES

APPENDIX D: DATA ENTRY MODULE

APPENDIX E: CARTOGRAPHIC REPRESENTATION MODULE

APPENDIX F: DATA RETRIEVAL MODULE

APPENDIX G: HARD-COPY DATA OUTPUT MODULE

APPENDIX H: OPERATIONAL MANAGEMENT SYSTEM PROGRAM

CHAPTER 1

CHAPTER 1 : INTRODUCTION

1.1 *Introduction*

The use of computer systems for the storage, analysis, and display of spatially related natural resources data is now widespread in industry, academia, and government. These systems, commonly called Geographic Information Systems (GIS), are valuable for addressing many planning or resource management concerns which are related to geographical locations and distribution.

However, automation in the form of geographic information systems is a relatively new and important concept of analyzing and graphically communicating knowledge about the world. They have introduced an important attempt to construct a more rational means of dealing with the real world and some of its problems.

The entire history of GIS covers only about twenty years. Changes in the field have been numerous and advancement has been very rapid during this period. One of the most obvious reasons for the development of GISs is the transition from the manual production and use of maps to the use of digitally produced maps instead, i.e. the developments in the science of digital mapping have been an important stimulant. These are the result of developments in computer hardware technology, which have been the subject of major technological advances. The machines themselves have progressed from the early batch-oriented configurations, to on-line time sharing systems, to systems in which minicomputers were intelligent peripherals connected to a large central computer, to the present day systems in which minicomputers, graphics workstations and even microcomputers are stand-alone central processors for complete GISs [Dangermond, 1986]. These advances in hardware have been paralleled by corresponding advances in software, including developments in operating systems, programming languages, database structures and management systems, etc. Software production has been enhanced by improved implementation techniques covering both better programming languages and improved software development environments.

These advances in computer technology (both hardware and software) have affected dramatically the way in which geographic data handling can be carried out. Furthermore,

the history of using computers for mapping and spatial analysis shows that there have been corresponding advances in automated data capture and in data analysis and presentation accompanying the developments in computer science. For example, in the field of surveying and mapping, field survey data is often acquired using electronic total stations which exhibit a high degree of automation in measurement and in recording data in digital form. Modern photogrammetric instruments also feature many devices such as correlators, integral computers, electronic scales, etc., all of which contribute to a speed-up in the rate at which data can be captured and input to digital mapping systems and geographic information systems. Thus the implementation, development and use of these systems is a response to innovations both in the field of computing science and in the area of surveying and mapping.

The introduction which follows describes in outline only the goals of a GIS, the hardware used, and the software required. The technology required to produce such software is described: first there will be a presentation of the minimum hardware required; then the different software modules which are necessary to carry out the different tasks required by a GIS, and the language requirements will be outlined and discussed. It should be noted here, however, that improved program development technologies will lead to the faster development of more easily and better maintained software for GISs.

1.2 *GIS - User Requirements*

Geographic Information Systems have been given different names according to each person's point of view or according to the application required of the system. Some of these names or terms are *geo-base information system*, *natural resource information system*, *geo-data system*, *land information system*, *spatial information system* and so on. A general definition given by Burrough [1986] states that a GIS is "a powerful set of tools for collecting, storing, retrieving at will, transforming, and displaying spatial data from the real world". On the other hand, Parker [1988] defines a GIS as "...an information technology which stores, analyzes, and displays both spatial and non-spatial data".

It is very clear from the above definitions that, the tasks required of the system (whatever the name is) are to capture spatial data in a specific form compatible with the computer environment; to manipulate these data in a particular way; and then to display the results. These are, in very general terms, the tasks that should be undertaken in what is called a

geographic information system.

These tasks can be further distinguished into more detailed and well defined requirements, the details of which will be discussed later in this Chapter.

1.3 GIS Data Requirements

Having discussed briefly the user requirements, the data requirements should be mentioned as well.

The data may come from many sources. Existing maps, photogrammetrically-derived coordinate data, field survey data and remotely-sensed image data may be used to provide the topographic or geographic base for the GIS. Other data sources are the population, agricultural and other censuses which are the basis of many GISs concerned principally with the analysis of socio-economic data. Alternatively or additionally, data concerned with geology, soil, vegetation, forestry, agriculture and land use, etc. may be included if the GIS is concerned with environmental management and planning. Yet again, the data sources may be land or property registers if the GIS is mainly concerned with land holdings or a cadastral system.

From these examples, it can be seen that a GIS may be required to handle either graphic data such as map or image data, or non-graphic data in the form of textual, tabular or numerical information or any combination of the two. Thus data structures have to be devised and implemented with this in mind.

A further concern must be with data manipulation, transformation and maintenance, which is a very demanding task in the case of business applications and is even more difficult when complex data has to be handled such as that used in topographically or geographically based information systems.

Data manipulation, transformation and maintenance in GISs involve a number of tools which are of great importance. Among these manipulation, transformation and maintainance capabilities are :

- the ability of the GIS to add and delete or update the data (which includes both the

different types of entities - points, lines, or areas - and the feature attributes, such as cartographic representation, text, text font, etc. attached to the data);

- the availability of procedures that are capable of carrying out the shifting, moving and rotating of data objects;
- data transformation so that the data fits into a given scale;
- the facility to enlarge and reduce areas of interest;
- the ability to allow polygon overlay and merging; and
- finally, the ability of the system to report the results of the trend during a session, either in the form of a number, a text-based prompt, a table, etc. or as graphical output in the form of a map, graph, diagram, etc.

These are, in brief, some of the capabilities that can be found in a GIS. Depending on the needs of a particular organization, other modules may be implemented and coupled into the system, such as projection transformation, clipping, edge matching, and so on.

It is obvious from the above description, that any particular geographic information system environment should be able to maintain its data to a high level of accuracy, authenticity and currency, in order to get the most out of the system capabilities. Furthermore the system capabilities listed above should be available and be capable of being handled by the user with a certain ease in a computer environment.

From the definitions of the term GIS given before, the components of a GIS would be as in Fig. 1.1 [after Burrough, 1986].

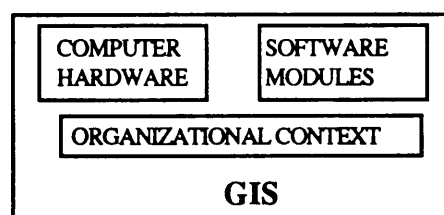


Fig. 1.1 GIS components

1.4 *Hardware for GIS*

Although the actual hardware is likely to vary from one GIS to another, the components will usually include:

- A powerful processor with a significant quantity of memory, to be able to carry out the considerable number of calculations needed by the system;
- Storage devices which will usually be required to hold large amounts of programs and data. In the case of large amounts of data, these units will be chosen according to their access speed;
- A digitizer, or some sort of data capture instrument, be it a manual point and line-following digitizer; a semi-, or a fully- automatic line-following digitizer; or on the other hand, a raster scanner;
- Output devices are at the end of the chain of computer hardware where they serve as the computer's means of delivering the results of the processing to the users. These devices range from line printers for report generation, to the more sophisticated colour plotters for graphical output, and from text-based monochrome screens to high-resolution graphics colour monitors.

All or a combination of these devices can be put in use in a GIS. Thus a typical hardware configuration for a GIS would look like in Fig 1.2.

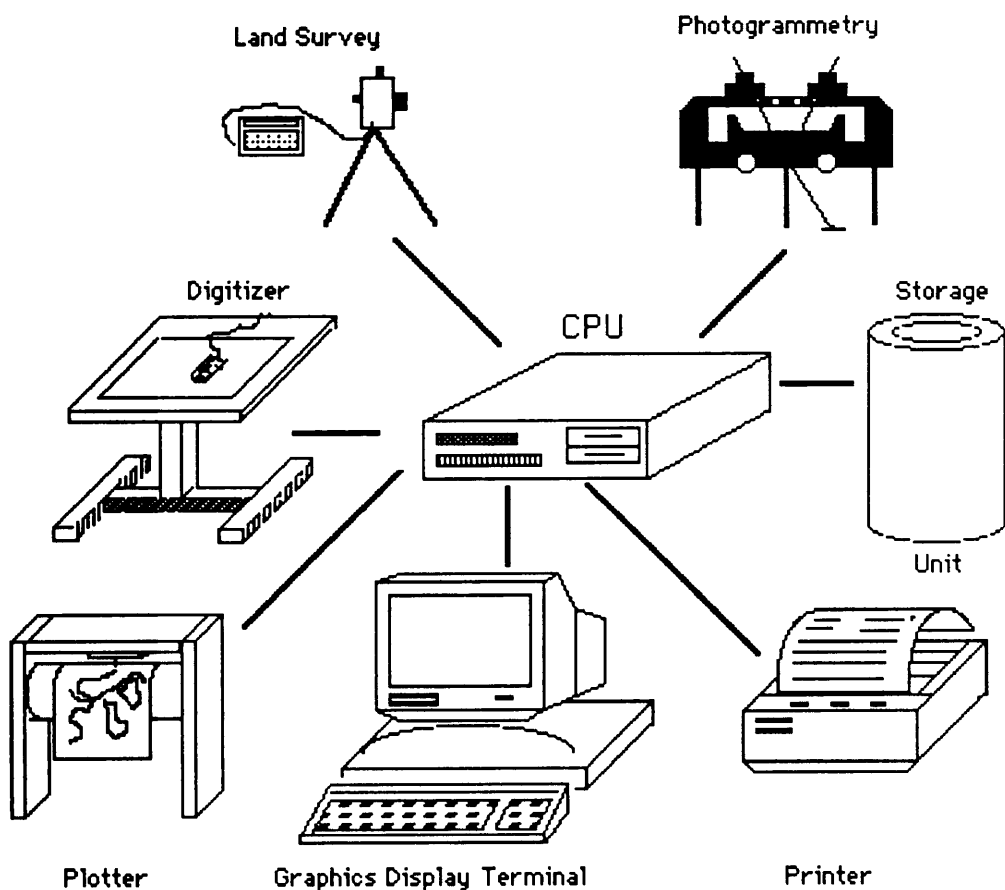


Fig. 1.2 Hardware for GIS

1.5 Software for GIS

With regard to the software involved with GISs, the trend has moved from isolated programs performing single functions, to suites of programs, at first working independently of the others, but later integrated and using shared databases. Present systems incorporate programs which are usable in an interactive, time-sharing mode, and are fully inter-working.

The software tools that should be available in a GIS consist of five basic technical modules. These modules are:

- a- Data input and verification;
- b- Data storage and database management;
- c- Data transformation;
- d- Query input, and
- e- Data output.

The relationship between these five software modules is shown below in Fig. 1.3.

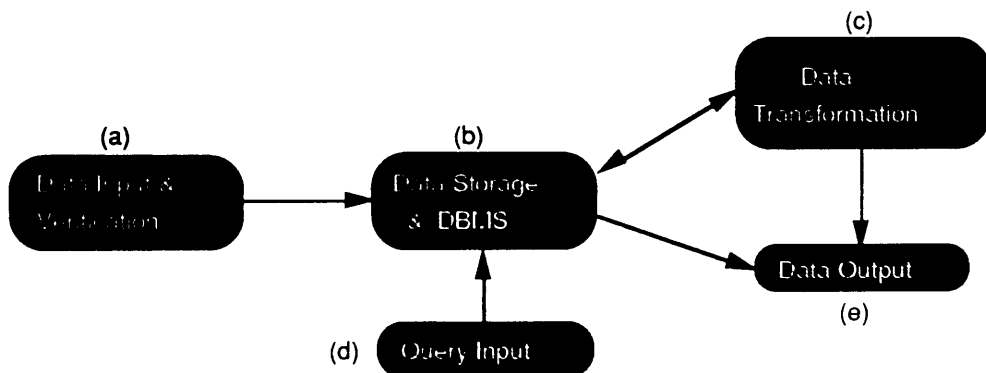


Fig. 1.3 Software for GIS

a) **Data input**, Fig. 1.4, covers all aspects of transforming data captured in the form of existing maps, field observations and digital data acquired via imaging sensors (including aerial photography, and satellite imagery) into a compatible digital form. A wide range of data is available for this purpose, including coordinate data from digitizers, lists of data in text files, raster image data from scanners, etc. The verification of the data is an important issue as is the maintenance of the quality and integrity of the input data.

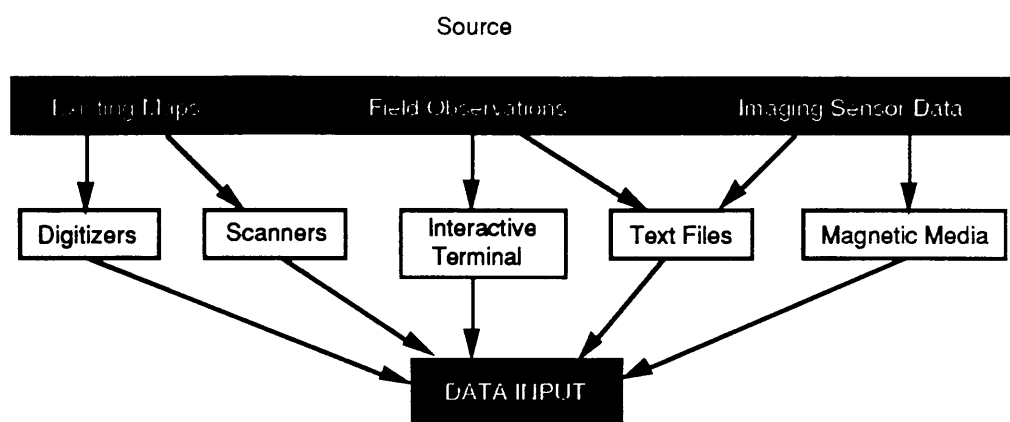


Fig. 1.4 Data input for GIS

b) The *Data storage* and database management system, shown diagrammatically in Fig. 1.5, reside in and form the core of any GIS, because they are concerned with the way in which the structuring and organizing of the data about the position and the attributes of the geographical elements is carried out. When linked, these form the topology of the real world model.

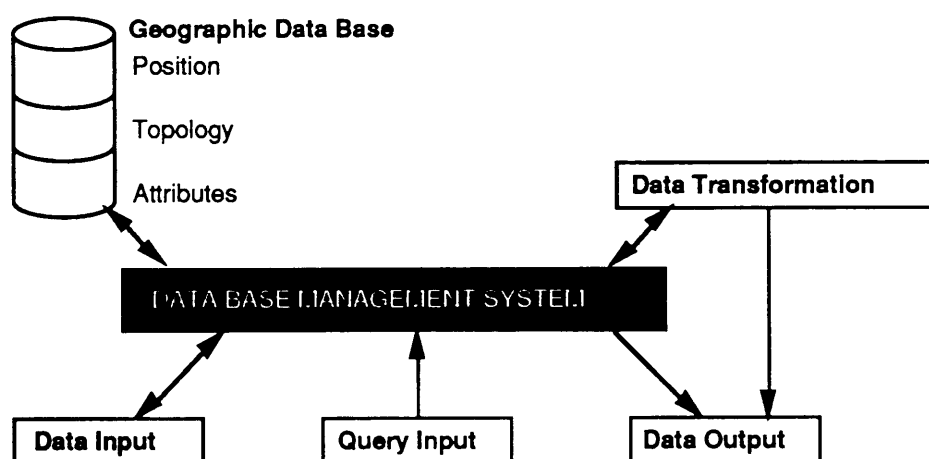


Fig. 1.5 Data storage and handling in GIS

The data storage module must be concerned both with the way in which data must be handled in the computer and how they are perceived by the users of the system. The matter of how quickly and efficiently the information may be retrieved from this module will also be of concern both to the system designer and to the user of the GIS.

The *Data Base Management System*, as the term suggests, organizes and manages the data stored in the database of the GIS.

c) *Data transformation* embraces two classes of operation, namely the *utilization and analysis* and the *maintenance* of the data. Utilization and analysis is concerned with three sets of activities. These are: i) removal of errors; ii) geometric transformation of the data; and iii) the analysis of data. Data maintenance, on the other hand, is concerned with the integrity of the sequence of activities which has taken place during the utilization and analysis stage, that is to define the logical beginning and end of these activities, whether they should be committed as permanent changes over the data, or these changes should be neglected. As illustrated in Fig. 1.6, data to be subject to the utilization and analysis operation come from the database. After they have been handled by the set of activities mentioned above, they can either be sent back to the store (shown as a thick continuous line) or they can be displayed on some type of output device (shown as a dashed line).

Some transformations are needed to remove errors from the data or to bring them up to date or to match them to other data sets. Other transformations must be able to operate on both the spatial and non-spatial types of the data, either separately or in combination. Many of these transformations, such as those associated with scale-changing, fitting data to a new position, logical retrieval of data and the calculation of areas and perimeters, are of such a general nature that one should expect to find them in a GIS in one form or another. Finally a large array of analysis methods that can be applied to the data in order to achieve answers to the questions asked of the GIS is also required.

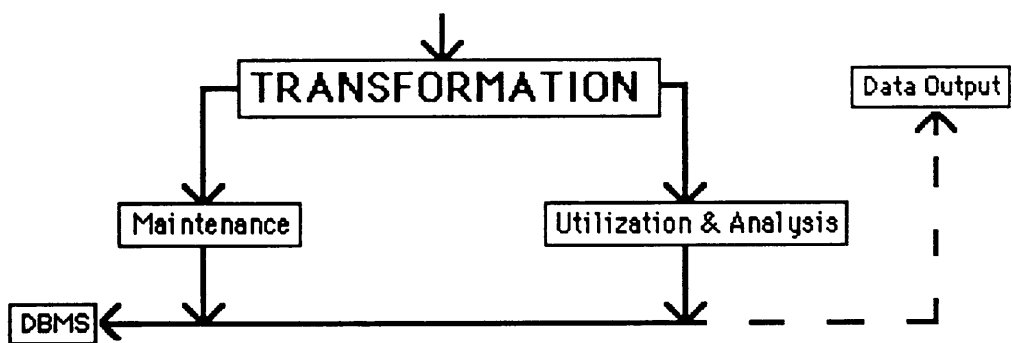


Fig. 1.6 Operations on GIS data

d) *Query input* represents the means with which a user can interface the database. Database access to the end user is provided (usually) by a variety of interfaces (called query languages), which vary from the handling point of view from the simple data maintenance screens; query-by-example screens; interactive English-like languages; through 'build it yourself' screens and reports; to programs written in some host languages such as FORTRAN and others.

e) The last module, *data output and presentation*, shown diagrammatically in Fig. 1.7, is concerned with the ways in which the data are displayed and the results of analyses are reported to users. Data may be presented as maps, tables and figures in a variety of ways, ranging from an ephemeral image on a Cathode Ray Tube (CRT) through hard-copy output drawn on a printer or plotter to information recorded on magnetic media in digital form.

DISPLAY & REPORTING

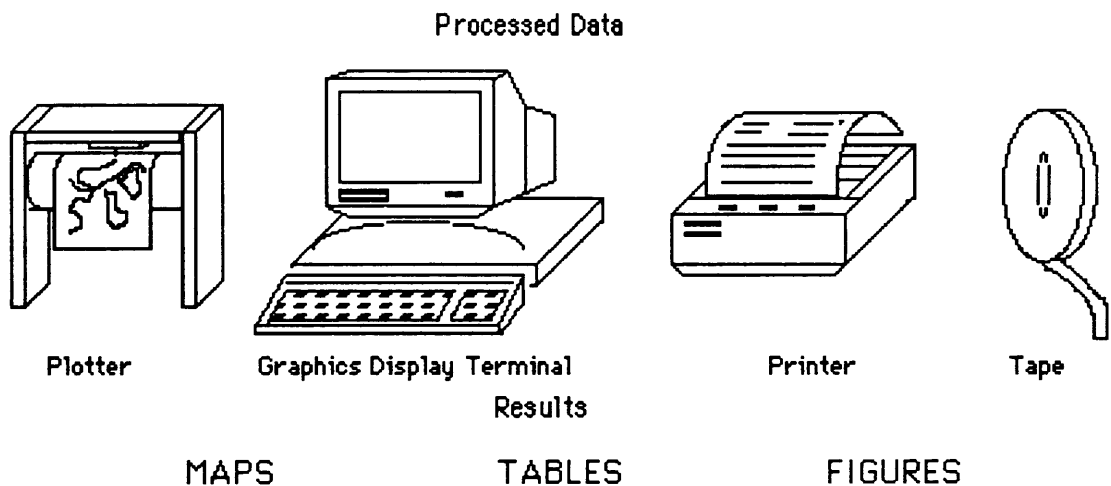


Fig. 1.7 Output means for GIS

From what has been discussed above, one can conclude that the software for GIS is necessarily complex and its production should be assisted by appropriate system development technologies.

1.6 GIS Data Structures

The data structures required for a GIS are unlike many other kinds of data structure handled routinely by modern information systems since they are complicated by the fact that they must include information about position, possible topological connections, and the attributes of the objects recorded or stored in the system. Moreover, maps were created to be viewed by human eyes and to be understood by human brains, and cannot be directly perceived by computers. Therefore, another approach has to be adopted to be able to use computers for the analysis and display of data in map form.

There are two natural organizations for spatial data representation in a computer environment, namely raster format and vector format. Both of these formats could serve the

purpose of handling spatial data. For example, using vector format, all geographic data are reduced to the three basic entities, namely points, lines and areas. On the other hand, the form of the objects could be represented by a set of points on a grid, in what is commonly known as a raster format, Fig. 1.8.

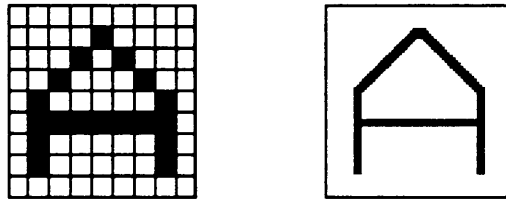


Fig. 1.8 Raster (left) and vector formats

The simplest raster data structure consists of an array of grid cells, also known as pixels (from the words picture elements). Each grid cell is referenced by a row and column number, together with a number representing the type or value of the attribute being mapped. In raster structures, points are represented by single grid cells; lines are represented by a chain of grid cells strung out in a given direction; and areas are represented by an agglomeration of neighbouring cells.

A raster representation assumes that the geographical space can be treated as though it were a flat Cartesian surface, whereupon each pixel or grid cell is then by implication associated with a square parcel of land. The resolution, or scale of the raster data is then the relation between the cell size in the database and the size of the cell on the ground.

On the other hand, the vector representation of an object is an attempt to represent the objects as exactly as possible using individual points and lines. The coordinate space is assumed to be continuous, not quantized as with the raster space, allowing all positions, lengths and dimensions to be defined very accurately. Furthermore, vector methods allow complex data to be stored in a minimum of space. Generally speaking, there is no one preferred method for the vector data structure employed in GISs.

1.7 *GIS Software Production*

The production of GIS software usually requires two software development aspects:

- i- a high level programming language sufficient to specify all of the facilities of the modules described in the previous sections; and

- ii- database facilities to handle the storage and retrieval of large amounts of complex data.

1.7.1 *Language Requirements*

In this section, the first aspect is dealt with, whereas the second aspect is dealt with in the next section (1.7.2).

The facilities which are specifically useful for GISs include:

- the ability to model complex objects;
- the ability to represent graphical objects as data values; and
- the ability to handle directly the hardware described in Section 1.4.

The first facility suggests a programming language which goes beyond languages such as BASIC or PASCAL and provides sophisticated data modelling constructs. Few languages provide direct constructs for representing graphical objects, so implementors are forced to build their own representations on top of more general mechanisms. This leads to different representations being used for each model and reduces the possibility of inter-working between models. The use of a language which provides specific graphical constructs would create a more coherent implementation system.

Direct access to special purpose hardware is another desirable feature. The discontinuities which arise when tackling each part of the implementation task in a different language are potentially costly, both in terms of development and maintenance time. Thus, for example, using the high-level components in PASCAL, and the low-level components in C, requires the additional tasks of keeping the individual components compatible and of becoming skilled in two different languages. One aspect of being able to use special purpose hardware in a GIS is the ability to drive a high resolution graphics screen directly. This would permit the user-interface to be programmed in the same way as the rest of the program.

Nowadays, hundreds of high level languages exist and they can be distinguished by the types of application for which they are designed which strongly influence the structure of a particular language. The basic design of a programming language, in which a programmer

would look like an end user, has been the objective of intensive research and development for many years. So, choosing a language for a particular application plays a major rôle in the success or failure of that application. However, there are some criteria that should be taken into consideration by language designers to end up with successful results. These criteria, [Atkinson et al. 1984], can be summarized as follows :

- a- programs should be easy to read and understand;
- b- the language should be easy to learn and remember; and
- c- the language should be succinct.

These criteria are of great importance in setting out a framework within which the language can be designed. However, for a user to choose a language to work with, there are some other criteria besides the design requirements which have to be satisfied, for example :

- a- the availability of support in the working environment in which the programming procedure is to take place;
- b- the suitability of the language to the subject in question;
- c- the generality of use of the language; and
- d- standardization of the language.

However, for a new language to be worthy to be included in the *de facto* list of available and proven languages such as BASIC, COBOL, FORTRAN, PASCAL, and C, it must supply both support software at a stable and suitable level, and of course, high performance in terms of the speed of execution of code [Macro & Buxton 1987].

On the other hand, it might be considered as a good practice as well, if the data and program structures available in a language be investigated in order to assess how well they fit with the structure of the problems likely to be encountered in the subject area in question. Nevertheless, in addition to what has been mentioned earlier, the specific facilities provided by a particular language may make the adoption and use of this language favourable as compared with others. The facilities which may be sought after can be considered to operate at two levels, the first being the availability of an adequate set of facilities for the specification of subroutines, procedures, functions into subprogram sections (a feature which is available in most modern languages but with considerable differences in their approach). The second is the support of a modular approach, which

gives advantages in terms of separate compilation and long term maintenance.

1.7.2 Data Base Requirements

The other main issue, which is of great concern to most agencies dealing with GISs, is the storage unit that should hold the maps which will inevitably be one of the main forms of output from the GIS. Different agencies make use of various types of databases. Nonetheless, it is essential to comprehend that, in digital mapping applications, the database is the most critical component of the total system. Thus its structure has to meet the internal demands of the data collecting organization. Furthermore, the varied requirements of users in terms of digital map display and production is an element of high concern. The database design depends in fact on the degree of sophistication and interaction needed with the stored digital data. However, the essence of the digital topographic database is to provide an information base from which a variety of other non-graphic information as well as graphic products may be produced on demand. Furthermore, the increased use of computer-based technology is paving the way for the sharing of common information, via a standardized mechanism which is acceptable to all users. As well, the design of a digital topographic database depends on the user's needs and the operations associated with the stored data. So, in order to be able to produce a graphic map from digital data, the following information must be available :

- a- a feature coding system to be able to identify individual elements to be included on the map;
- b- positional coordinates for all features;
- c- attributes associated with each feature, describing the feature's apparent properties; and
- d- graphic information as to how the feature should be represented in terms of symbol design and dimension, line width, colour, etc.

Often associated with the feature coding system is a structure of overlays. Each overlay will comprise a specific class of feature (e.g. Agriculture, Buildings, Roads, etc...). Nowadays, the issue is not only that of storing the necessary data but rather of providing the capability to manipulate these data and make best use of them, by obtaining as much information as possible from the stored data.

The data required is thus extremely complex, whereas the data organization provided by current database management systems is often extremely simple. Most current GISs make use of relational databases, in which all data are forced into the form of tables. Emergent database techniques such as Semantic Data Modelling, and Object Oriented Database Systems provide a more complex and flexible data structure mechanism, but these have not as yet been implemented with sufficient efficiency and reliability to become commonly available or popular.

The choice is therefore between the use of proven and optimized database systems, which have the drawback of forcing the implementor to maintain three widely differing views of the data. For example, using relations, one will end up with three views:

- the program's data structure;
- relations; and
- the real world.

which are very different [Atkinson et al. 1989]. Therefore, all the time, the software creator has to maintain complex mapping between these views. This greatly complicates the production of the GIS software, its maintenance or modification and the actual operational use of the system.

A further drawback of the present implementation of information systems using separate language and database management systems is that this requires the maintenance of an interface between these two conflicting views of the data [Atkinson et al. 1978]. The so-called "impedance-mismatch" can give rise to problems in making a consistent view of the software.

The answer to this problem is to use a language which incorporates both full programming facilities and database facilities as well. These languages are called Database Programming Languages [Atkinson and Buneman 1987] and are a continuing area of research. This thesis demonstrates the use of such a language, PS-algol, to produce a prototype GIS.

1.8 *Summary*

It is a common rule that the use of a better technology will lead to better results. GISs are

no exception. Better technologies in this field are concerned with the facilities provided by the programming languages to produce better software. If this can be achieved, the direct consequence is faster development, smoother programs and more easily maintained software.

The aim of the work reported in this thesis is to demonstrate that a language which combines the facilities appropriate for GIS products as described in Section 1.7 with the database facilities as described in Section 1.7.2 provides a suitable implementation technology for the development of easily built and well maintained geographic information systems. So the suitability of the language is tested against three main criteria:-

- i- the ability to structure the data in a simple, easy to understand, and comprehensive manner;
- ii- the graphics handling ability; and
- iii- the speed of the process.

The results of this work tested against these criteria are discussed in the conclusion of the thesis.

On the other hand, a direct comparison between already available systems and the work carried out and reported in this thesis could be somewhat unfair, since these existing production GIS systems tend to use industrial quality software whose implementation may have taken anything between a dozen and a thousand man-years. Whereas, in the case of the present project, the work has been carried out using a prototype language (which has involved only twenty man-years of effort in comparison with other software), while the actual GIS system itself is only a prototype (involving 3 man-years of effort) and is still in need of further development and refinement. Thus it can only be viewed as being research quality software.

This thesis is devised as follows; Chapter 2 is a review of database systems, their types and components. It covers the definition of the term DBMS and its rôle and functions and the suitability of each of them to a GIS application. Chapter 3 gives an overview of current GIS developments together with a review of some existing well-known GIS systems which are representative of these developments. Chapter 4, on the other hand, is an introduction to PS-algol, the database language used in the present project to develop a

prototype GIS. The actual data structure applied in the project is described in Chapter 5, together with a general view of the complete system layout. Chapter 6 describes the Operational Management System of the system and the global procedures used. Then Chapter 7 through to Chapter 10 present a detailed description of the different modules employed in the system, namely: Data Entry and Data Code Transformation, Cartographic Data Representation, Data Retrieval, and finally the Hard Copy Data Output module. In the end, Chapter 11 is the conclusion and gives recommendations for further work.

CHAPTER 2

CHAPTER 2: DATABASES AND DATABASE STRUCTURES

2.1 *Introduction*

Geographic information systems have considerably higher requirements in terms of data volume and complex spatial data handling than many of the normal applications encountered in other types of information systems. Because of this, it is particularly important for geographic information systems to have efficient data storage formats and associated optimal algorithms for data manipulation and retrieval.

As discussed previously in Section 1.7, spatial data has two natural organisations, namely vector format and raster format. In this thesis, only vector data is handled, and so what follows is an overview of related topics for the data handling and processing of vector data within a database environment. First a database definition is given, and then the different types of basic file types are presented. Next, data structures are discussed and there will be some discussion of Data Base Management Systems (DBMS) and Query Languages.

In this chapter, the use of databases in normal (non-graphic) applications will be presented first, then the differences in terms of the specific GIS requirements of databases will be presented later.

2.2 *Data Base Definition*

A Data Base (**DB**) is a shared collection of interrelated data designed to meet the varied information needs of an organization. A database has two major properties: it is integrated and it is shared. It is integrated so that previously distinct data files have been logically and coherently organised to eliminate (or reduce) redundancy and to facilitate data access. It is shared so that potentially all qualified users in an organization have access to the same data, for use in a variety of activities.

2.2.1 *Data Base Architecture*

Database architecture is divided into three levels [Date, 1981(a)]: internal, conceptual and external. The internal level is the one closest to physical storage, that means, it is the one

concerned with the way in which data are actually stored; the external level, on the other hand, is the one closest to the users, and is concerned with the way in which the data are viewed by individual users; and finally the conceptual level is a 'level of indirection' between the other two.

If the external level is concerned with individual user views, the conceptual level may be thought of as defining the views of the entire group of users. An external view can thus be thought of as the content of the database as it is seen by some particular user. So, different users running different applications on the database will have different external views. Each particular external view is defined by an external schema. This is also true for both conceptual view and conceptual schema. If the conceptual view can be thought of as being the representation of the entire information content of the database, the conceptual schema is therefore its definition. Finally, the internal view will be defined by a corresponding internal schema. The declaration of the internal schema is carried out using a Data Definition Language (DDL) [Olle, 1978]. Fig. 2.1 represents the database system architecture and its linkages between the different schemas.

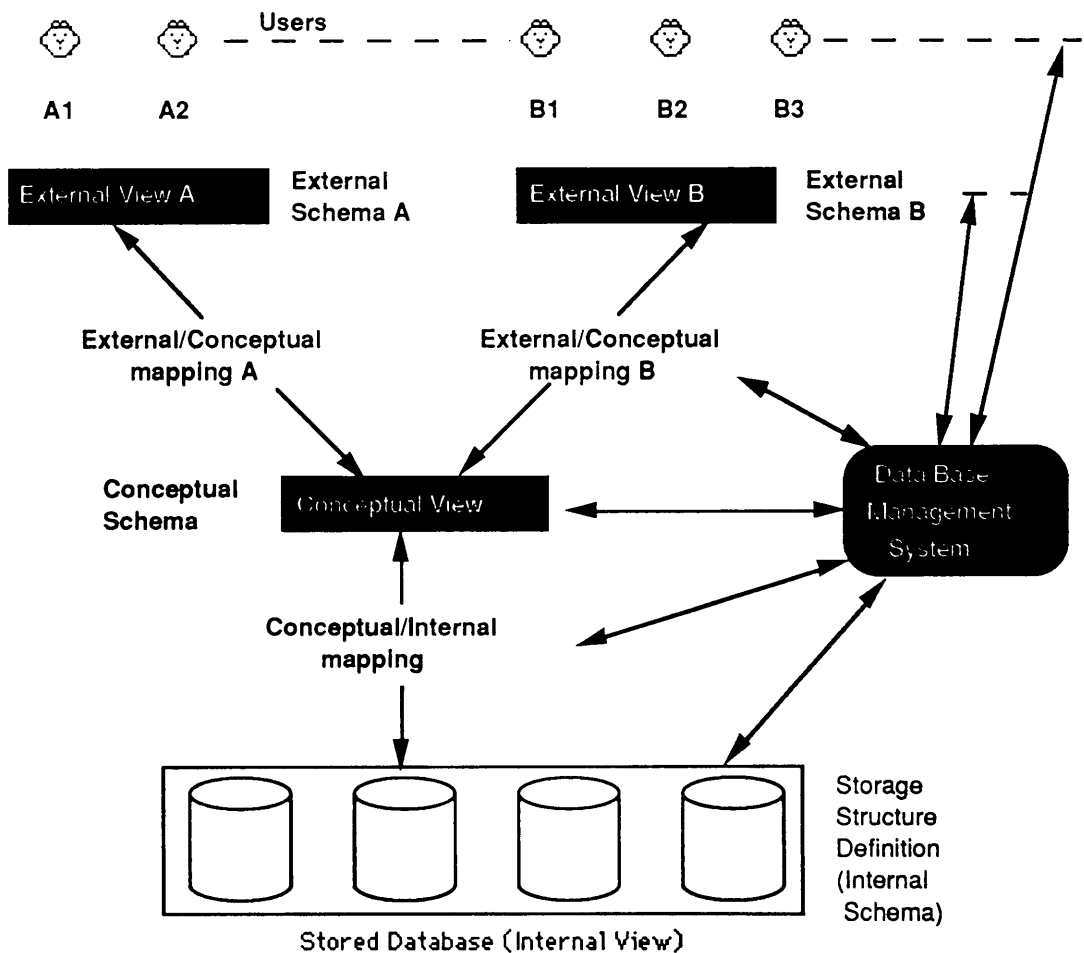


Fig. 2.1 Data Base System Architecture

a- Conceptual Schema

The conceptual schema defines a global model of the database. The underlying data model may be hierarchical, network, or relational. Ideally, the data model should make no reference as to how the data model is implemented (in terms of inverted lists, multiple linked lists, and so on) or how the data are physically stored or accessed.

b- External Schemas

External schemas are derived from the conceptual schema. They define subsets, or views, of the real data. Many external schemas are defined for a single conceptual schema, e.g. external schemas A and B in Fig. 2.1

Normally, the data model used to define external schemas is the same as that used in the conceptual schema (e.g. both might be relational). However, some DBMS products allow the conceptual schema to be defined with one data model and the external schemas defined using a different data model.

c- Internal Schema

The internal schema (or storage schema) for a database defines the storage files that contain the actual data records for the database. Normally, there is one storage file for each conceptual file or relation described in the conceptual schema. Also, the internal schema defines the details of the data structures and mechanisms that are used by the DBMS to locate records and to establish associations between records.

However, data representation in the database depends on the application in hand, and must be specified by writing the storage structure definition. In addition, the associated mapping between the internal schema and the conceptual schema should also be specified.

d- Conceptual/Internal Mapping

The conceptual/internal mapping defines the correspondence between the conceptual view and the stored database and it represents the activities taking place which are handled

routinely by what is known as the Data Base Storage System or DBSS which will be described later in Section 2.5.2.

e- External/Conceptual Mapping

The external/conceptual mapping defines the correspondence between a particular external view and the conceptual view. It also represents those activities taking place between the two views which in turn are handled by the so-called Data Base Control System or DBCS which will also be discussed later in Section 2.5.2.

2.3 *Basic Data Structures*

Before presenting in detail the ways in which data can be stored in a computer, the ways in which data can be prepared for storage and access must be explained in general. Speed of accessing and the linking and cross referencing of data represent the main aims for any data storage system. There are several ways of achieving this, some of which are more efficient than others.

It is common to all database structures that the data to be entered to the database are written in the form of records. Records are of the form of one-dimensional arrays of fixed length, divided into a number of equal partitions, known as fields. Records are said to be ideal if all the items in the database have equal number of attributes (fields). But when the attributes are of variable length (such as names), fixed length records are inconvenient and instead variable length records may be used.

The basic way of holding data in computers consists of a file. According to the way of accessing the data in these files, they can be divided into the following groups:

- a- Simple lists;
- b- Ordered sequential files; and
- c- Indexed files.

Fig. 2.2 shows an area made up of two contiguous polygons 'I' and 'II', each of which is bounded by a series of nodes 1, 2, ..., 6, and connecting segments a, b, ..., g, which represent, say, two different parcels of land. The aim is to represent these parcels within

the different groups (a) to (c) mentioned above.

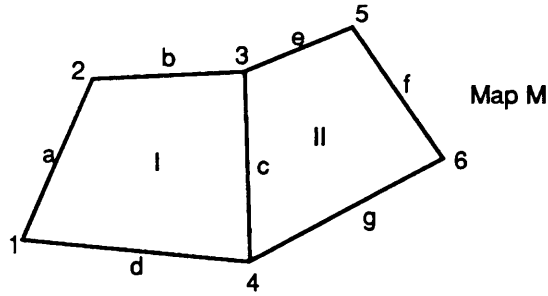


Fig. 2.2 Map representation of two polygons

2.3.1 Definitions of Terms

It is worthwhile here to define the notions of 'node' and 'segment'. A 'node' is of type 'point' but in addition, nodes represent the intersections between two lines or more. In Fig. 2.2 above, points '3' and '4' represent two nodes. Thus all nodes are point features, but not all points are nodes - since they may be defining an individual (point) on the ground, whereas the term node is invariably used in connection with areas and lines.

A 'segment', on the other hand, is formed by that part of a line joining two points. So, in Fig. 2.2, 'a' to 'g' represent segments joining the different points shown in the diagram. Each segment is defined by a connected series (or string) of points whose coordinates have been defined.

A 'line' therefore, can then be defined as a collection of segments (each consisting of a connected series of points) and nodes. Generally speaking, each line starts and ends with a node and there will be a node at each intersection or junction with another line.

If the end points of a series of connected segments join up or coincide, forming a closed perimeter, this produces a 'polygon' which encloses an area. In Fig. 2.2, the line formed by segments 'a'; 'b'; 'c' and 'd' would be regarded as defining such a perimeter, thus forming polygon 'I'. Similarly for polygon 'II' which is composed of segments 'c'; 'e'; 'f'; and 'g'.

2.3.2 Simple Lists

These represent the simplest way of storing items. As each new item is added to the list, it

is simply placed at the end of the file, which gets longer with the increase in the number of newly entered items (see Fig. 2.3). It is very easy to add data to this type of data holder, but retrieving these data is quite slow and inefficient due to the sequential search procedure involved with its use. It is thus very clear that a structured data model is needed to speed up the process of data retrieval.

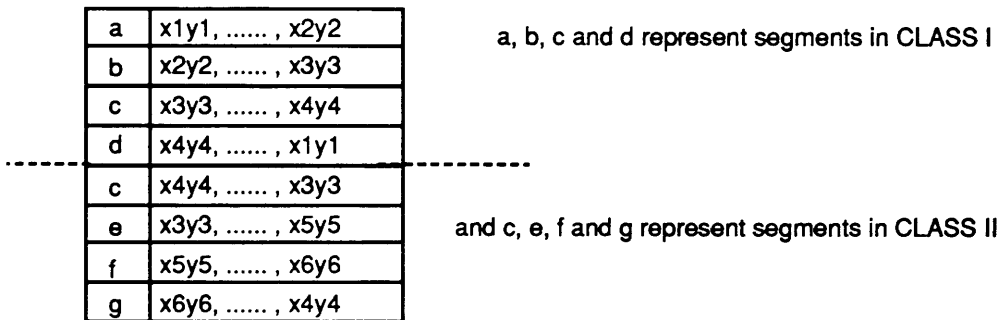


Fig. 2.3 Simple list

2.3.3 Ordered Sequential Files

This type of data holder is similar to a dictionary, in which data (in the case of a dictionary, words) are ordered/structured alphabetically. In this type of structure, the addition of a new item to a certain 'class' of data means that extra room must be reserved or created within that class for it to be inserted, but the advantage drawn from such structures is the speed with which any individual stored item can be reached. Inherent in the use of Ordered Sequential Files is the use of the Quick Search method, otherwise known as 'Binary Search' method [Horowitz & Sahni 1982], which enables a faster search for a particular item (see Fig. 2.4). In this method, the search begins by examining the record in the middle of the file rather than that at either end of the file. Assuming that the file being searched is ordered by an increasing order of keys (a key is a single attribute with values that are unique within a relation), thus, based on the results of the comparison with the middle key, one of the following cases will result. First, the record being searched for is the middle one. Second, this record is in the lower part of the file and third, it is in the upper part of the file. Consequently, after each comparison operation, either the search terminates successfully or the size of the file remaining to be searched diminishes by half of the original size.

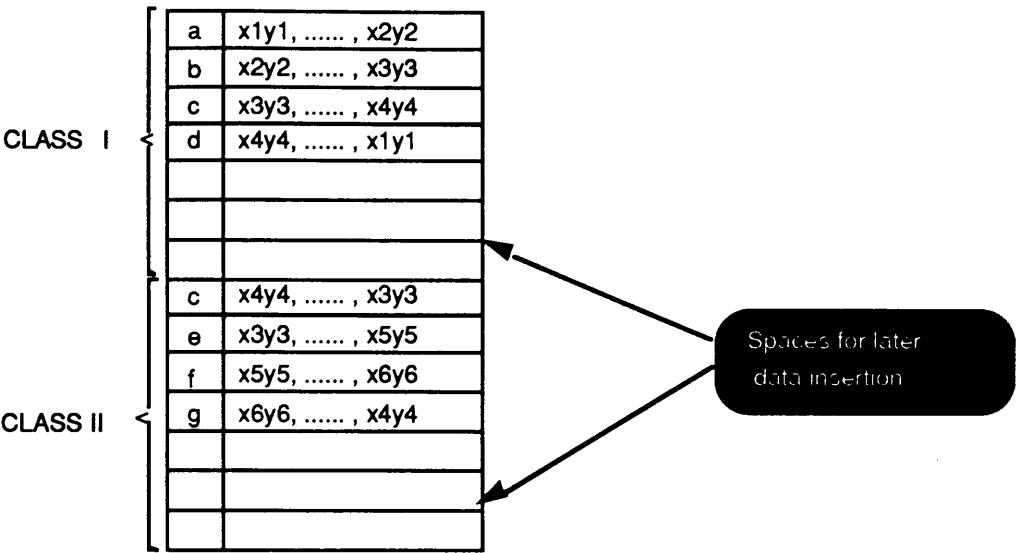


Fig 2.4 Ordered sequential file

2.3.4 Indexed Files

In an indexed file, the records are stored in physical sequence according to their keys. The file management system, or access method, builds an index, separate from the data records, which contains the key values together with pointers to the data records themselves. This index permits individual records to be accessed at random without accessing other records.

There are two ways in which the original data can be accessed with indexed files. The first is called direct files in which data items in the files themselves provide the main order of the file. Whereas in the second, called inverted files, the locations of items in the main file are specified according to topic, which is given in a second file.

In direct files (Fig. 2.5) the record for each item contains sufficient information for the search to jump over unnecessary items. Each item contains not only the series name and other information but also a number indicating the storage location of series names beginning with the key. Then the search for a particular record is made by constructing a simple index file that lists the correspondence between the first letter of the series name and its storage location. The search then proceeds by a sequential search of the index, followed by a sequential search of the appropriate data block.

| Direct Files | | |
|--------------|------------|-----------|
| Index | | |
| Item Key | Record No. | File Item |
| A | 1 | a |
| | 2 | b |
| | 3 | c |
| | 4 | d |
| B | (na+1) 5 | c |
| | 6 | e |
| | 7 | f |
| | 8 | g |

Fig 2.5 Direct files

The use of an inverted file index requires first that it be constructed by performing an initial sequential search on the data for each topic. The results are then assembled in the inverted file or index, which provides the key for further data access, see Fig. 2.6.

| Inverted Files | | | | | | |
|---|------------|----|---------|------|-------|-----|
| Soil Profile Number | Attributes | | | | | |
| | S | pH | De | Dr | T | E |
| 1 | I | 4 | deep | good | sandy | no |
| 2 | II | 5 | shallow | good | clay | yes |
| S = series, De = depth, Dr = Drainage, T = Textuxre, E = erosion. | | | | | | |

| Index (inverted file) | |
|-----------------------|--|
| Topic | Profile (sequential numbers in original file) |
| Deep | 1 |
| Shallow | 2 |
| Good drainage | 1 2 |
| Poor drainage | |
| Sandy | 1 |
| Clay | 2 |
| Eroded | 2 |

Fig. 2.6 An inverted file with its index file

Generally speaking, indexed files permit rapid access to the individual items held in a database, but on the other hand, more effort is required to keep them updated. When an alteration has been made to items in the original files, the addition or deletion of a record in a direct file means that both the file and its index must be modified.

When a new record is written to a file accessed by an inverted file index, the new record does not have to be placed at a special position within the file; it can be simply added at the end of the file, but the index must be updated. Nevertheless, another disadvantage of indexed files is that very often data can be only accessed through the key contained in the index files; so other information can be only retrieved using sequential search methods which makes it unpopular for geographic databases.

2.4 The Classical Database Structures

Knowing that a database consists of data held in many files, an organization (which is called a structure) should be available which allows extraction of these data from files. In what follows, three main types of database structures are described.

To simplify the presentation, the same map 'M' shown in Fig. 2.2 is used to explain these structures. Map 'M' consists of two polygons 'I' and 'II', made up of several segments, some of which are shared by both polygons.

2.4.1 Hierarchical Data Structure

The hierarchical approach represents a parent-child or a one-to-many relationship. It assumes that each part of the hierarchy can be reached using a key that fully describes the data structure, and it assumes that there is a good correlation between the key attributes and the associated attributes that the item may possess.

Hierarchies are usually implemented using a tree type data structure and extensive use of pointers. The tree data structure has the property that each element of the structure, except the root, has only one path coming in, but there may be zero or many paths coming out of it. Fig. 2.7, gives a diagrammatic representation of the tree structure and its terminology.

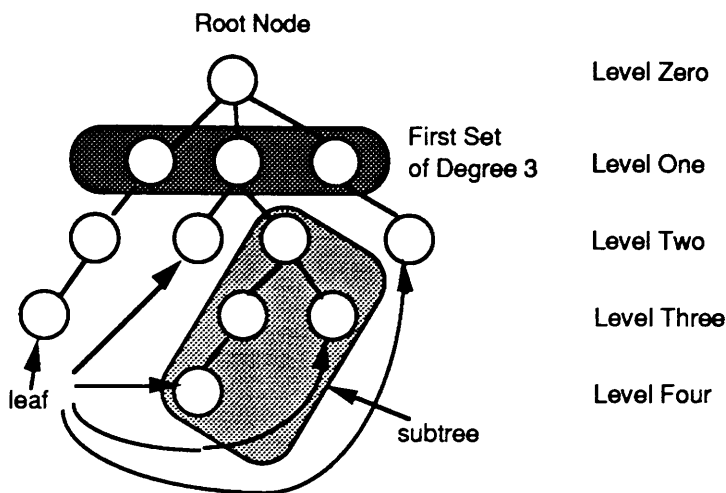


Fig. 2.7 The representation of the tree structure and its terminology

This approach has the advantage that it is easy to understand and it is easy to update and expand. However, data access via the keys is easy for the key attribute, but on the other hand, is very difficult for the associated attributes. This hierarchical data structure can be very limiting in the case of environmental data, where the flexibility of issuing queries based on the use of many associated attributes is a dominant requirement. Furthermore, index files existing with hierarchical systems have to be maintained all the way through, which so often leads to certain values being repeated many times, the result of which is a rise in data redundancy. The hierarchical approach to represent map M is illustrated as in Fig. 2.8. The redundancy can easily be seen with four records of nodes 3 and 4, and two records of each of the other nodes 1, 2, 5 and 6.

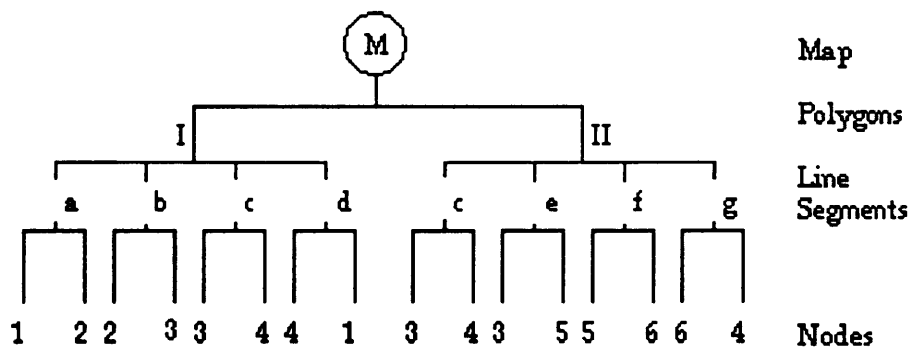


Fig. 2.8 Hierarchical representation of map M

2.4.2 Network Data Structure

The network approach, unlike the hierarchical one, is not restricted to the paths up and

down the taxonomic pathways, but also makes use of pointers to link any two items in the database irrespective of their actual position (physical adjacency on the disk). It has proved to be a very useful approach when the relations or linkages can be specified beforehand. Furthermore, network systems avoid data redundancy and make good use of all available data. But on the other hand, the database is enlarged by the overhead of the pointers, which in complex systems can become quite a substantial part of the database. This means that whenever the updating or maintenance of the database is being carried out, it will involve manipulating the pointer structures as well. A network data structure is illustrated by Fig. 2.9.

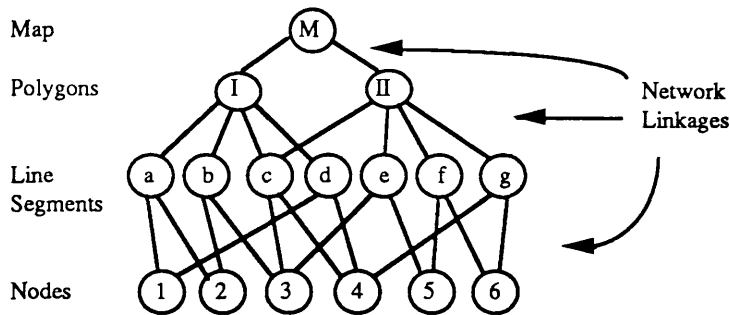


Fig. 2.9 Representation of map M by network model

From this it will be seen that the data redundancy in the line segments (e.g. line c) and nodes (1 to 6) which occurred in the hierarchical model of Map M (see Fig. 2.8) is eliminated.

2.4.3 Relational Data Structure

The relational approach does not store any pointers and makes no use of any hierarchy. It stores data in records (also known as tuples), in which a certain number of attribute values are grouped together in two-dimensional tables called relations. Each relation is usually a separate file. The pointer structures in the network approach and the keys in a hierarchical model are replaced by data redundancy in the form of identification codes that are used as unique keys to identify the records in each file. The great advantage of the relational data structure is that its structure is very flexible and can meet the demands of all queries that can be formulated using the rules of Boolean logic and the associated mathematical operations. These allow different kinds of data to be searched, combined and compared. Also, the addition or deletion of data is very easy, because it is just a matter of adding or deleting a tuple. Furthermore, the simplicity of the model has led many researchers to develop highly

efficient techniques for manipulating relations. On the other hand, some aspects of the relational data structures can be regarded as disadvantageous. For example, the searching procedures are often laborious, because many of the operations involve sequential searches through the files to find the right data to satisfy the specified relation. The relational approach is illustrated in Fig. 2.10.

relation Map

| | | |
|---|---|----|
| M | I | II |
|---|---|----|

relation Polygon

| | | | | |
|----|---|---|---|---|
| I | a | b | c | d |
| II | c | e | f | g |

relation Segments

| | | | |
|----|---|---|---|
| I | a | 1 | 2 |
| I | b | 2 | 3 |
| I | c | 3 | 4 |
| I | d | 4 | 1 |
| II | e | 3 | 5 |
| II | f | 5 | 6 |
| II | g | 6 | 4 |
| II | c | 4 | 3 |

Fig. 2.10 Relational representation of map M

The first relation in the relational representation of map M in Fig. 2.10 is the 'Map' relation. Besides the identifier of the map 'M', it contains two keys referring to the other relation 'Polygon' representing the two polygons present in the map.

The second relation 'Polygon' contains two tuples for the two polygons 'I' and 'II', each of which has four fields containing the segments forming that particular polygon. For example (see Fig. 2.7), polygon 'I' contains segments 'a', 'b', 'c' and 'd'.

The third relation 'Segments' is a relation accessed by the polygon identifier, in this case, either 'I' or 'II'. This relation contains the segment identifier, and the two endpoints of the

segment, e.g. segment 'a' lies between the two points '1' and '2'. Thus the tuple of the first segment 'a' would contain first the key which identifies the polygon 'T', then the key which identifies the segment 'a', and finally the two points forming the endpoints of the segment. This can go further and make another relation, say the 'Point' relation in which each point is represented by a tuple containing its 'X' and 'Y' coordinates.

2.5 Database Management System

A DataBase Management System (DBMS) is a generalized software system that manages the database, providing facilities for the organization, access, and control of the data. The generalized term suggests that the DBMS is independent of individual applications and therefore can be employed by any user requiring access to the data contained in the database. The following section, Section 2.5.1, outlines the functions a user may expect to find within a DBMS whatever its data model is.

2.5.1 DBMS Functions

Codd [1982] has stated that eight major functions should be provided for the implementation of a comprehensive DBMS. These are:

- a- Data Storage, Retrieval, and Update. These functions should allow data to be entered, found and changed in a simple manner by many users. Different users may be provided with different views of the same data thus allowing them to store, retrieve, and update their data easily and efficiently.
- b- Data Dictionary/Directory (DD/D). This is defined, [McFadden & Hoffer, 1988], as the repository of all information about an organization's data. The DBMS should also maintain a user-accessible data dictionary/directory to make clear the different data structures used in the DBMS.
- c- Transaction Integrity. A transaction is a sequence of steps that constitute some well-defined activity. In processing a transaction, changes to the database should only be made if the transaction is processed successfully in its entirety. In this case, the changes are said to be committed. If, on the other hand, the transaction fails at any

point, then it is said to be aborted, and none of the changes should be made to the database.

To maintain transaction integrity, the DBMS must provide facilities for the user or application programmers to define transaction boundaries, that is, the logical beginning and end of transactions. The DBMS should then commit changes for successful transactions and reject changes for aborted ones.

- d- Recovery Services. The DBMS must be able to restore the database (or return it to a known condition) in the event of some system failure. Sources of system failure include power failure, disk head crashes, operator error and program errors. Typically, the whole database will be stored periodically to create a stable state which can be returned to when one of these conditions occurs.
- e- Concurrency Control. Since a database is shared by multiple users, two or more users may attempt to access the same data simultaneously. If two users attempt to update the same data record concurrently, erroneous results may occur, since the transactions may interfere with each other. Safeguards must be built into the DBMS to prevent or overcome such interference.
- f- Security Mechanisms. Data must be protected against accidental or intentional access, misuse or destruction. The DBMS provides mechanisms for controlling access to data and for defining what actions may be taken by each user.
- g- Data Communication Interface. Users often access a database by means of remote terminals in a telecommunications network. In spatial databases this means that the DBMS must have the ability to read data from different sources, or in different formats. Moreover, the data itself may be distributed amongst a number of machines, thus requiring the ability to find and transmit data around the network in a rapid manner.
- h- Integrity Services. The DBMS must provide facilities that assist users in maintaining the integrity of their data and protect it from reaching an inconsistent state. A variety of edit checks and integrity constraints can be designed into the DBMS and its software interfaces. These checks are normally administered through the data dictionary/directory.

Most contemporary database management systems provide all the functions named here, at least to some degree (although most do not yet provide a comprehensive set of integrity services). However, DBMS products differ in the manner in which the functions are performed. With some user-friendly products, the functions are performed more or less automatically by the DBMS, with little or no user involvement. With other products, the DBMS provides some facilities or interfaces, but the user must take major responsibility for defining the functions either directly or through application programs.

2.5.2 DBMS Software Components

The major software components of an operational DBMS environment are as shown in Fig. 2.11. This figure shows how a DBMS interfaces with other software components such as user programs and the access methods.

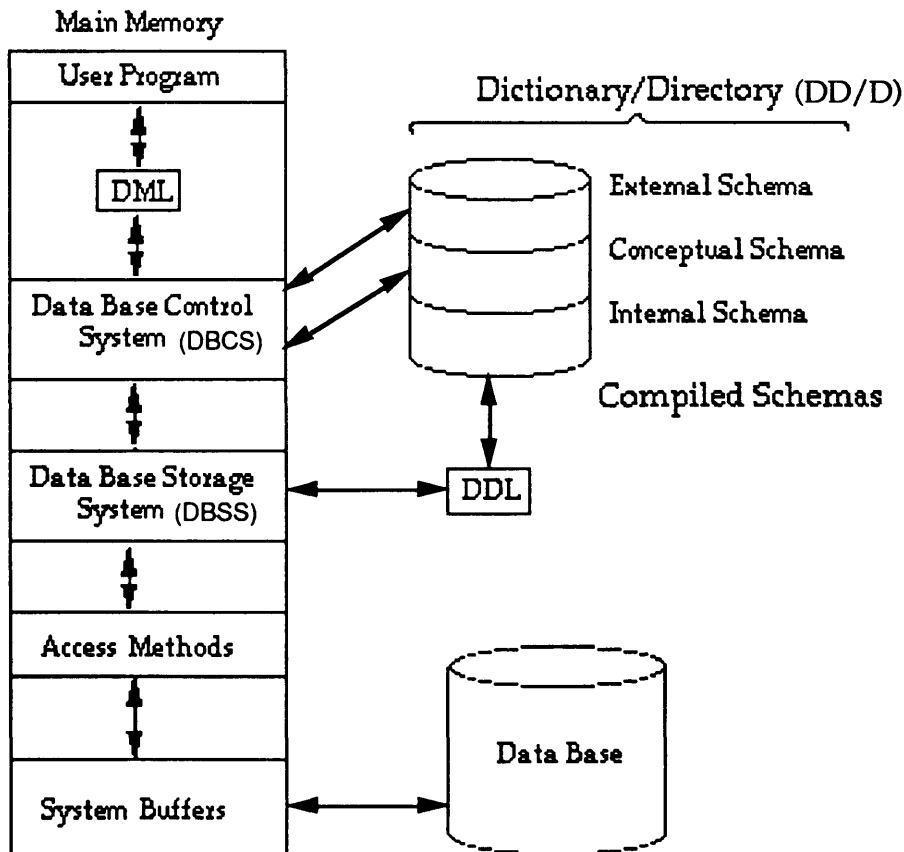


Fig. 2.11 DBMS interfaces with software components

The DBMS software, generally speaking, has two major operating components: a database control system and a database storage system. The Data Base Control System (DBCS) is a module that interfaces with user programs. It accepts calls for data which are written using

a special 'language' known as the Data Manipulation Language or DML which will be discussed in Section 2.6. Such calls include READ and WRITE commands. The DBCS examines the external and conceptual schemas to determine what conceptual records are required to satisfy the request. The DBCS then places a call to the database storage system to fill the request.

The Data Base Storage System (DBSS) manipulates the underlying storage files which were defined in an earlier stage using a descriptive 'language' called the Data Definition Language or DDL. It establishes and maintains the lists and indexes that are defined in the internal schema. If indexed files are used, the DBSS does not manage the physical input and output of data. It passes requests on to the appropriate access methods, which read data into and out of the system buffers. The requests delivered to this module are analyzed by matching the parameters of the request with a stored version of the database (called a schema) and a definition of the part of the database applicable to the program request (called a sub-schema).

In preparation for processing a database, all the software components shown in Fig. 2.11 are loaded into the main memory of the computer. Precompiled versions of all the software modules - user programs, DBCS, DBSS, and access methods - are stored on a system disk. Compiled versions of the three schemas - conceptual; external; and internal - are stored in a disk library. When a program is to be run, all these components are loaded into the main memory. They are linked together by an operating system module so that they can communicate with each other. System buffers are created by the operating system at this time.

These in general are the functions which any DBMS should offer whatever the data model is. However, the underlying structures which represent explicitly the real world differ according to the suitability of a particular database model to deal with the question in hand. With regard to the three types of database structures discussed earlier in this chapter, DBMSs are designed to match these types of databases. For example, the leading database management systems based on the hierarchical data model and still in use today is IBM's Information Management System (IMS) [IBM, 1978]. Furthermore, in network data modelling, the CODASYL (COntference of DAta SYstem Languages) approach represents a very good example of a network DBMS [CODASYL, 1971 & Olle, 1978]. Finally, Oracle is a typical DBMS which implements the relational data model [Oracle Corporation, 1986].

2.6 *Query Languages*

The users' interrogation of the database is done by a 'language' commonly known as a 'Query Language'. The various operations which can be undertaken by the end-user using this language can be grouped in four phases:-

- i) retrieval of data from the database;
- ii) the generation of statistics (e.g. on the quantity of data retrieved compared with the amount of data stored);
- iii) the extraction of data from the database (which is concerned with how the retrieved data is to be displayed, analysed or printed); and
- iv) the updating of data - this allows the end-user to perform manipulations upon the selected records and to return them to the system.

It should be noted here that retrieval (involving phases (i) to (iii)) is the main concern of query languages. Phase (iv) is properly the concern of the Data Manipulation Language, although these functions are combined in query languages like SQL.

Some query languages (such as SQL) also provide for the definition or description of the database objects by specifying schemas, sub-schemas, files (or relations), records and data items (attributes) [Date, 1981(a)], although these are usually thought to be the characteristics of a Data Definition Language (DDL) already described earlier in this chapter.

Providing a query language with a set of functions for updating and data definition as well as retrieval, creates an interface to the database which is well integrated. A query language can thus be defined as: "A system designed to support an interactive dialogue for the retrieval, display and, sometimes, update of records using variable criteria" [Gittins, 1986].

Query languages permit an interactive programmer to write record retrievals using expressions that specify which records are required, so that the programmer or user does not have to go through the process of record-by-record retrieval [McFadden & Hoffer, 1988]. Such query languages are helpful in that they can rapidly produce the result of a simple end user question, check on the contents of a database after a series of data maintenance program executions or provide a user with a prototype of the kind of report that could be produced by the report program.

Due to the fact that there are different types of database models (hierarchical; network and relational), by implication, query languages differ in correspondence with these different database models.

The Standard Query Language SQL represents a good example of a query language that has been widely accepted and used in relational database management system products. SQL allows data definition and data manipulation to be carried out through a list of commands with which the user can define tables, indexes and views (for data definition using DDL) and interrogate the databases using another list of commands (issued by the data manipulation language DML).

Table 2.1 illustrates some of these SQL commands with their functions. Table 2.2 presents some of those commands used by DL/I in the IMS hierarchical DBMS system and Table 2.3 illustrates some of COBOL DML commands.

Table 2.1 SQL Commands

| | |
|---------|--|
| SELECT: | Lists the columns to be projected into the table that will be the result of the command; |
| FROM: | Identifies the tables from which output columns will be projected and that possibly will be joined; |
| WHERE: | Includes the conditions for tuple/row selection within a single table or between tables implicitly joined. |

(Oracle Corporation, 1985)

Table 2.2. Summary of DL/I Operations

| | |
|------------------------------|---|
| GET UNIQUE (GU) | Direct retrieval of a segment |
| GET NEXT (GN) | Sequential retrieval |
| GET NEXT WITHIN PARENT (GNP) | Sequential retrieval under current parent |
| GET HOLD (GHU, GHN, GHNP) | As above, but allow subsequent DLET/REPL |
| REPLACE (REPL) | Replace existing segment |
| DELETE (DLET) | Delete existing segment |
| INSERT (ISRT) | Add new segment |

(Source: Date, 1981, 297)

Table 2.3. Typical COBOL DML Commands

| | |
|-----------------|---|
| FIND | Locates record in database |
| GET | Transfers record to working storage |
| OBTAIN | Combines FIND and GET |
| STORE | Puts a new record into the database and links it to all sets in which it is an automatic member |
| MODIFY | Changes data values in an existing record |
| CONNECT | Links an existing member record into a set occurrence |
| DISCONNECT | Removes (unlinks) an existing member record from its current set occurrence |
| RECONNECT | A combination of DISCONNECT and CONNECT to unlink a record from its current set and link it to a new set occurrence |
| ERASE | Deletes record from the database, DISCONNECTs it from all set occurrences in which it participates, and deletes other records for which this is an owner in set |
| COMMIT | Makes permanent all database updates made since the last COMMIT command executed. |
| ROLLBACK | Aborts all updates since last COMMIT and restores database to its status at the time of the last COMMIT |
| KEEP | Places concurrent access controls on database records |
| (CODASYL, 1971) | |

2.7 Vector Data Bases

As mentioned briefly in Section 1.6, the vector representation of data could be seen as a method of representing objects with maximum fidelity. It makes use of coordinate space allowing all positions, lengths, and dimensions to be defined precisely (within the restrictions imposed by the length of a computer word on the exact representation of a coordinate and the limitation of the basic step size of all vector display devices). However, all features can be reduced to the three main components (entities) of vector representation namely *points*, *lines* and *areas*.

2.7.1 Point Entity

Point entities represent all those geographic features that have only one pair of coordinates (X & Y). Together with their coordinates, point entities should have in their records descriptive factors (attributes) about what they are. For instance, a point can be represented by a symbol unrelated to any other information about features of other similar or different

types. In this case, a point record should hold information about that symbol, including its display size, and orientation. Whereas if the point is a location at which text should be displayed, the record should contain the text, the font and the size to be used in display, together with the orientation, and the justification.

2.7.2 Line Entity

Features built up by a series of straight line segments use this type of data representation. The simplest form of line is a straight line which only requires two pairs of coordinates (one pair at each end of the segment). Besides the coordinates, a line record should as well contain details about how the line should be displayed (solid, dashed, etc...), and when networking is preferred, the line record should hold a pointer field to the nodes which define the ends of the line.

2.7.3 Area Entity

The areas enclosed by polygons can be represented in various ways in a vector database. Because most kinds of thematic mapping used in geographic information systems have to do with polygons, the way in which these entities can be represented and manipulated has received considerable attention.

The aim of a polygon data structure is to be able to describe the topological properties of areas (that is their shapes, neighbours, and hierarchy) in such a way that the associated properties (i.e. the attributes) of these basic spatial building blocks can be displayed and manipulated as thematic map data. Before describing the ways in which a polygon data structure can be constructed, the requirements of polygon networks that are imposed by geographical data will be stated.

First, each component on a map will have a unique shape, perimeter, and area. There is no single standard or basic areal unit as is the case in raster systems. Second, geographic analyses require that the data structures be able to record the neighbours of each polygon in the same way that networking in utility applications requires connectivity. Third, polygons on thematic maps are not all at the same level (islands occur in lakes that are themselves on large islands, and so on, see Fig. 2.12).

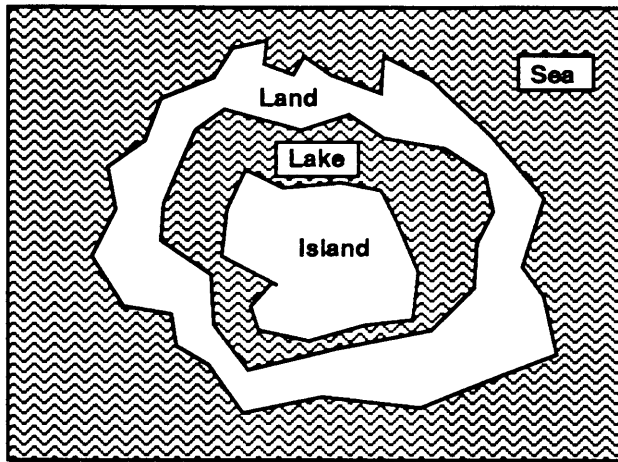


Fig. 2.12 The provision of islands & lakes

Islands and neighbours can only be properly handled by incorporating explicit topological relationships into the data structure. The topological structure can be built up in one of two ways; by creating the topological links during data input, or by using software to create the topology after the data has been input.

2.8 Discussion

As has been seen previously, handling data in vector format imposes several difficulties, the most important of which being the creation and maintenance of the structures that should hold the data in the database, together with different modules such as Data Entry, Cartographic Representation, Data Retrieval, and Data Output needed by the GIS system. There are also some other procedures that should be taken into consideration when GISs are being designed. These include the provision of map overlaying and merging; area calculation; search for polygon neighbourhood; point in polygon search; islands enclosure; and cartographic symbol representation. These all form important procedures within the GIS software environment, that should be present whether it be based on vector data or raster data.

The lack of flexibility in the classical data models and their deficiencies when undertaking complex applications, have led to rise in demand for a better programming environment where more facilities are provided. Two possible improvements present themselves. The first would be the use of an Object Oriented System, where the topological relationships between the geometrical elements of several objects or the construction of a single object

can be described explicitly. For example, a line can be a borderline of a building and of, say, two parcels. The object data geometry is stored independently of its cartographic representation. Then it can be mapped very easily with different graphics codes without the object data having to be edited.

The second option is the use of a database programming language which incorporates programming language facilities together with database management facilities. PS-algol is an example of such a language and has been put to the test in this thesis, which tries to explore the power that it possesses in order to be able to solve some of the problems outlined above. This language is described in Chapter 4. The results of this research will be discussed in detail in Chapters 5, 6, 7, 8, 9 and 10.

In the next chapter, Chapter 3, a survey has been carried out covering some of the data structures proposed or used by other people working in the GIS field to handle vector data for different GIS applications. Most of these are being implemented and marketed commercially by various companies active in this area.

CHAPTER 3

CHAPTER 3: STATE OF THE ART IN GIS

3.1 *Introduction*

A short introduction explaining the basic concepts and some of the requirements of a geographic information system (GIS) has already been given in Chapter 1. This was followed in Chapter 2 by a discussion of databases, database structures and database management systems which are the basic building blocks on which a GIS can be built. In this chapter, some further discussion of the different types of information system utilizing spatial information will be given and will be followed by a review of the various approaches which have been implemented to date, including the description and discussion of representative systems which are currently available on the market.

3.2 *Digital Mapping*

A first point which must be made is that there is a very large field of activity called digital mapping or automated cartography which is principally concerned with the computer-based production of maps. This activity takes place within the surveying and mapping industry and is the modern form of the classical area of map production concerned with the compilation and production of topographic maps, air and sea navigation charts and small scale maps and atlases. The reasons for the industry's widespread adoption of digital mapping techniques are numerous, but the principal ones can be summarized as follows [Petrie, 1990]:-

- i) Speeding up the process of map production is necessary to cope with the increased flow of data from field survey and photogrammetric sources and to shorten the period between the initial data collection and the availability of the resulting map in digital or hard-copy form;
- ii) Reducing (or even eliminating) the tedious cartographic work involved with map compilation and production such as draughting, scribing, mask-cutting, lettering and symbol generation and placement, which requires highly skilled personnel who are often difficult to find;
- iii) Ensuring that existing maps are kept current and up-to-date both in terms of their basic topographic information and of the specialist thematic information (e.g. that required

- by the public utilities) which is drawn, overlaid or annotated on them is a major objective both for map producing agencies and users; and
- iv) Reducing the cost of map, plan and chart production is another highly desirable requirement both for map producers and users, though it is one which has been quite hard to achieve due to the large capital costs of purchasing, installing, operating and maintaining high-accuracy, high-quality computer-based mapping equipment on the one hand and the costs of training and employing the specialist personnel needed to operate and maintain this equipment on the other hand.

In the field of digital mapping carried out at large scales for engineering projects, land registration and cadastral purposes, etc., the current degree of automation is extremely high. Data comes directly from field survey and photogrammetric instruments in digital form, is processed in a fairly simple computers and the final data is delivered in the form of large-format, vector-based maps in monochrome hard-copy form or their digital equivalent (or both).

A large number of systems such as Eclipse/Panterra, Moss, ProSurveyor, Wild Geomap, etc., are available as off-the-shelf systems or packages which are much used by land and engineering survey companies, civil engineering consultants and contractors, local government estates and roads departments, etc. Other similar packages have been produced for in-house use by survey companies such as Longdin & Browning, Mason Land Surveys, etc.

In this sector, another major activity is the digitizing of existing large-scale maps, principally the O.S. 1:1,250 and 1:2,500 scale series, which is carried out both by the Ordnance Survey itself in-house and by a number of approved contractors. This utilizes software packages such as Laser-Scan's LITES system, Map Data, etc., which basically are interactive digitizing/editing packages which convert the cartographic data on the maps into digital form for use by public utilities, local government organizations, etc., often within a GIS system. However, as can be seen from the above descriptions, the systems used for this process of data acquisition, data processing and map production, etc. can in no way be called information systems or GISs since they do not possess the DBMS, query language, data structures, analysis tools and other features that are characteristic of a GIS.

At the other end of the cartographic spectrum is the area of automated cartography

concerned with map and chart production and atlas production at small-scales, and characterized by the need to produce large volumes of multi-coloured maps using offset lithographic techniques. The data is almost always derived from larger-scale maps and other sources and so the digitizing of existing cartographic material is a major preoccupation of this sector. The basic cartographic operations of generalization and compilation followed by scribing, masking and photo-lettering have not proven to be easy to convert to digital operations. Also the requirement to generate final output as colour separations on film, including the need for tint screens and patterns with variable densities and orientations means that access to very accurate and expensive raster-based film plotters such as those made by Scitex, Intergraph/Optronics, Linotron, etc., is an absolute necessity. Also the need for high quality colour plots and proofs leads to the requirement for colour electrostatic plotters or thermal wax-transfer colour plotters, which again are very expensive to install and operate in an economic manner.

As can be seen from the above discussion, the areas of digital mapping and automated cartography are the subject of intensive activity and development at the present time. However, the matter of databases and database management systems, while important for larger national mapping agencies, are matters which do not have a high profile at the moment. Most of the national mapping agencies in the U.K. - e.g. the Ordnance Survey; the Mapping and Charting Establishment, R.E.; the R.A.F.'s AIDU aeronautical chart production organization; etc. - utilize the Sharebase relational database system mounted on Britton-Lee database engines for the management of their structured databases. These special-purpose, hardware-based database systems are reputed to have a throughput three to five times greater than a comparable purely software-based RDBMS.

3.3 Automated Mapping/Facilities Management (AM/FM) Systems

These systems are those employed by the various public utilities such as water, electricity, gas, telephone and sewage authorities which have large networks of pipelines or cables which are, in geographic terms, widespread and are also very complicated in terms of their structure. Their exact location and function need to be known for planning, operational and maintenance purposes. Much of this data needs to be recorded and displayed on maps and kept up-to-date for management purposes. Thus the public utilities have always been large customers for plans and maps. However, they also require their maps to be revised continually, both in terms of the basic topographic information which they contain and in

respect of their specialist information drawn, overlaid or annotated on them. The promise of being able to achieve this desired currency of information by adopting computer-based mapping techniques has therefore been a considerable factor for causing these large, technically-aware and capital intensive industries to become interested in the automated mapping (AM) component of an information system [Petrie 1990].

The other component of these public utility information systems is the facilities management (FM) aspect. All of these organizations share the common feature of wishing to manage their large networks of pipes or cables and the associated facilities such as generating centres; distribution, storage and control centres; sub-stations; etc. The former group (i.e. the networks) will have a mainly graphical representation whether it be the locational reference base provided by a map background or the diagrammatic representation of the networks themselves. Associated with this may be a smaller non-graphical element which gives details of dimensions, characteristics, etc., of the individual network components which will often comprise information in the form of alphanumeric or numeric tables. In the second group of facilities (i.e. the main generating, distribution and storage centres), the emphasis will be reversed. The graphic element may well be small and largely diagrammatic in nature, showing plant layout rather than cartographic information, while the non-graphic element in the form of textual and numeric information may be quite large.

Thus an AM/FM system is quite definitely a spatially-based information system which can handle both locational/graphical information and non-graphic information. Since many of these public utilities cover large areas of a country with complex distribution networks designed to serve large numbers of customers, these organizations have been early entrants into this field of spatial-based information systems. Indeed many of the systems available on the market are essentially AM/FM systems designed specifically for and targeted at the public utility sector, such is its size and importance.

3.4 *Land Information Systems (LIS)*

The concept of a land information system is essentially similar to that of the preceding type, the AM/FM type, in that it is designed to handle both spatially-located graphic data which is mainly derived from cartographic sources and a great variety of associated non-graphic data. However there are also several different features. The emphasis in an LIS is on

various aspects associated with land - land ownership, land registration, land valuation, land use, etc. Also it is largely based on areas and areal-based features rather than the lines and linear-based features of an AM/FM system. As will immediately be obvious, these quite different characteristics of the two systems mean that a spatially-based information system which is optimized for public utility applications, i.e. an AM/FM system, is most unlikely to be well suited for use as a land information system (LIS).

Once again, given the large number of organizations which are concerned with this field - e.g. cadastral and land registration offices, estate survey offices, property agencies, etc. - this is an area which is large enough to have encouraged the development of specialized LIS systems which are optimized for use in these areas. Those countries where national cadastral or land registration systems have long been established are especially large markets for this type of system.

3.5 Geographic Information System (GIS)

The term GIS is not a very easy one to define. At the one level, there is a tendency to use it (not always correctly) as a generic term which encompasses all spatially-based information systems, no matter what their particular orientation is or the specific field or application for which they were designed. On the other hand, here in the U.K., the term is often used, whether consciously or not, as an information system which is designed principally for use with socio-economic data, especially census data such as population data, employment data, agricultural data, etc., which has been collected on an areal basis. Often the cartographic aspects, especially those concerned with positional accuracy, and the incorporation currency and completeness of small topographic features, etc., have a secondary importance as compared with their status in a digital mapping, AM/FM, or LIS systems and the resulting maps are essentially of the polygon or area-based thematic type. By contrast, this second type of GIS is usually very well equipped or endowed with a rich selection of tools for the analysis of spatial data which are of great use and interest to geographers, planners, social scientists, etc. Once again, it must be said that there are quite a large number of systems which are designed specifically to address the needs of this particular group of users. Obviously it would be possible to go on to discuss further the semantics and the underlying philosophy of the term GIS but in the context of this present project, this would not be particularly useful or productive.

3.5.1 *Summary*

From what has been described and discussed above, it is clear that, essentially, all of these AM/FM, LIS and GIS systems are built on database management systems (DBMS). However, unlike traditional database management systems and procedures, the spatial attributes of the features incorporated in the database represent primary acquisitional data and there is a very heavy emphasis on the final results being presented in graphical, mainly cartographic form. Thus, the system can be viewed as a marriage between automated mapping and database management technologies, with the ability to search for records based on spatial location [Eastman, 1987].

In essence, the AM/FM, LIS and GIS systems are structured collections of spatially-located digital data (the database) which is controlled by a database management system (DBMS) and is usually envisaged or implemented as a query answering system. The emphasis will be on geographically referenced data which can be displayed graphically, i.e. in map form. Currently such systems are being purchased and implemented by government planning agencies, public utilities, and cadastral or land registration services on a very large scale. If the basic locational data collected by surveyors, photogrammetrists and cartographers or the specialist thematic data collected by engineers, land registrars, social scientists, etc. is to be used in such information systems, then it will have to be arranged or structured in a manner which suits the purpose of the particular AM/FM, LIS or GIS system with which it is associated [Petrie, 1990].

From the point of view of the overall system architecture, sometimes not too much difference can be seen between an AM/FM, an LIS and a GIS system. Often the differences arise mainly when the applications of the systems are envisaged and the appropriate tools and services have to be provided. However, in accordance with the classification and descriptions of the different approaches to building, structuring and managing databases used by computer scientists which have been outlined in Chapter 2, the following sections will review some of the systems available in the market within each of the main types of database systems described there - namely, the hierarchical, network and relational approaches. In addition, there will be a review of two systems (System 9 and GINIS) which employ an object oriented database system.

3.6 *Hierarchical Systems*

These systems were the first type to be implemented in the early 1970s when the first spatially based information systems were introduced. In spite of the fact that other types of DBMS have become popular since then, the hierarchically-based systems are still in widespread use and are still being sold in large numbers.

3.6.1 *Intergraph IGDS/DMRS*

Intergraph is a leading American company in the field of mapping and CAD systems. Originally called M & S Computing, Inc., the company first applied interactive computer graphics to mapping disciplines in 1973 with a digital mapping package. Since then, the company has continued to supply hardware and software tools to the mapping community. Later in the 1970s, it developed a suite of programs to handle the analyses of the data given by digital mapping. This was one of the very first packages which would nowadays be identified as being a land or geographic information system. Because of Intergraph's importance in the digital mapping, AM/FM, and LIS market place, an attempt will be made in this thesis to cover one of the company's main products in some detail.

3.6.1.1 *Overall IGDS/DMRS System Design*

On the system design side, Intergraph has, from the outset, used two main packages which even today, after many developments, still form the core of several of Intergraph's offerings in the GIS/LIS area and in the closely related area of AM/FM.

These two basic modules or systems, [M & S Computing, Inc., 1979], are:-

- i) Data Management & Retrieval System (DMRS); and
- ii) Interactive Graphics Design System (IGDS).

Their overall relationship and the interface between them is shown in Fig. 3.1.

As the name suggests, DMRS is essentially a database management system which handles the non-graphic elements of the system. In the case of a public utility, these would comprise the numeric and text-based information on ducts, cables and pipes, sub-stations, etc. Thus DMRS recognizes and controls the large amount of inventory data, facilities data,

land use data, or general attribute data that complements the graphic drawings, diagrams, maps or charts which are handled and produced by IGDS.

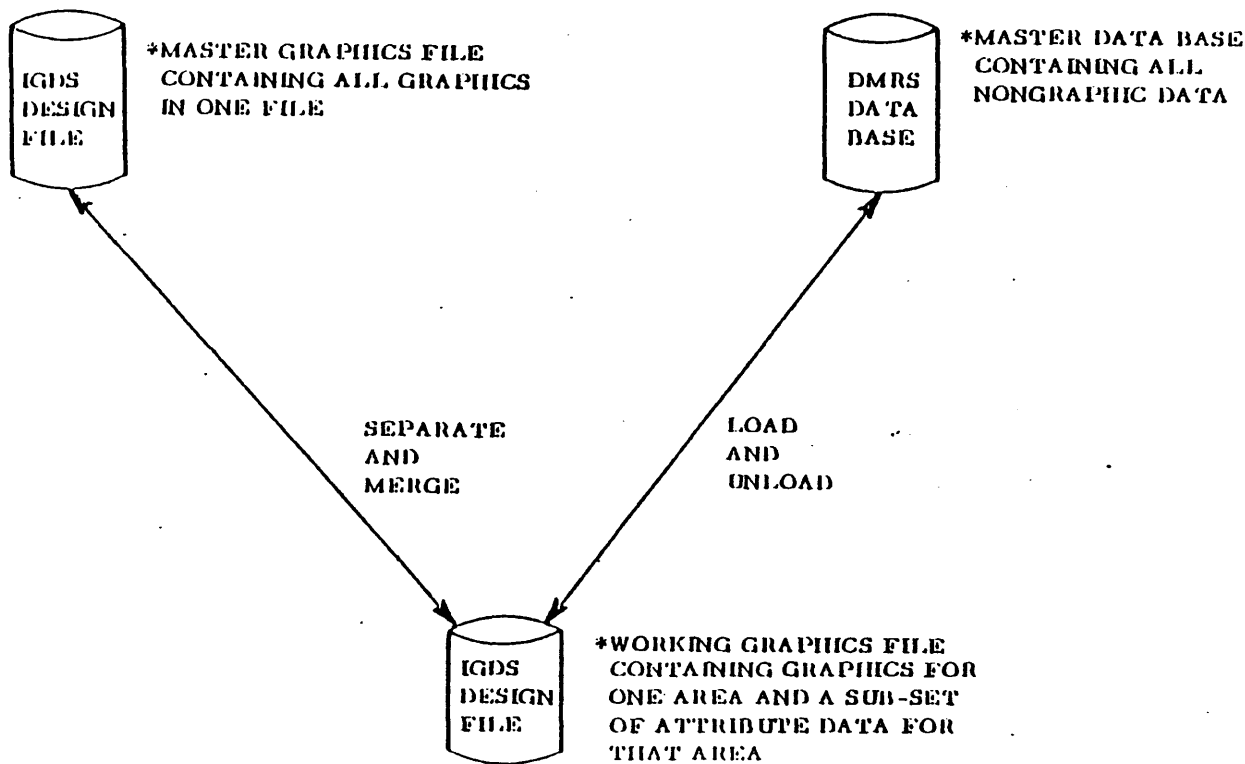


Fig. 3.1 IGDS/DMRS overall concept and interface

The IGDS part of the system includes both a Master Graphics File containing all the graphics relevant to a single project or area and a Working Graphics File which contains both the graphics elements for the specific area being worked on, together with the relevant attribute data for that area. Data verification and editing features for both graphic and non-graphic data as well as protection against failure and the ability to recover information have been incorporated into these basic software modules or systems as they have been developed further.

3.6.1.2 DMRS Overview & Components

The Data Management & Retrieval System overview is illustrated in Fig. 3.2.

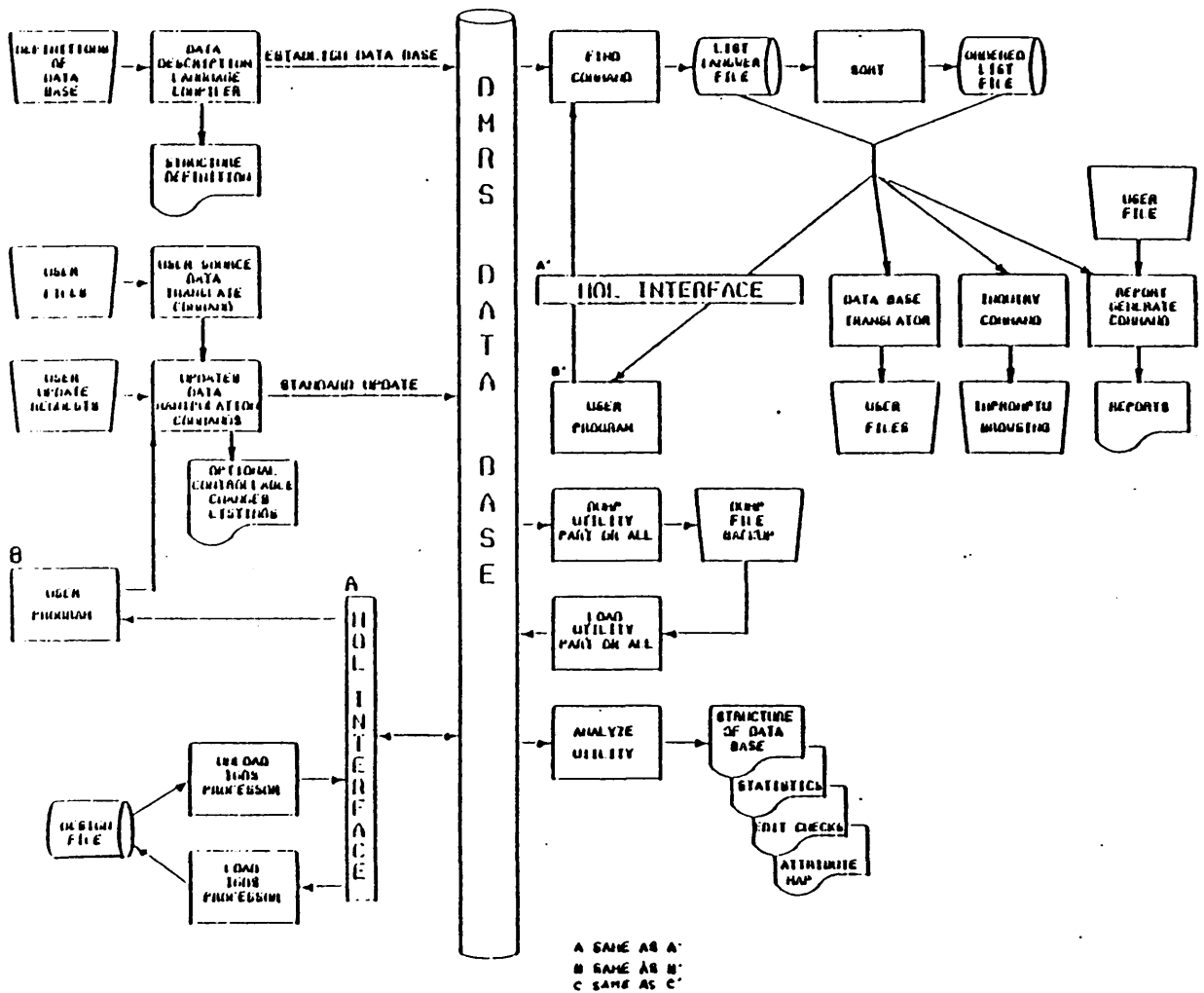


Fig. 3.2 DMRS overview and components

The major features of the DMRS system and its interfaces with the database are summarized below:-

- a) Definition and structuring of a database, including the Data Description Language (DDL) compiler;
- b) Database creation and maintenance, including the following commands:-
LOAD (whole or in part)
DUMP (whole or in part)
INSERT
CHANGE
DELETE
- c) Selection of data universe (FIND);

- d) Translators, including the following:-
 - Existing user files to database;
 - Database entries to user files;
 - IGDS selected data to database;
 - Database entries to IGDS graphic files.
- e) Inquiry for browsing and impromptu questions;
- f) Report generator for versatile reporting including
 - Report requests from interactive graphic user;
- g) User update requests;
- h) Sort;
- i) Linkages to IGDS;
- j) Analyze utility, this includes the following:
 - Integrity checking;
 - Statistics;
 - Structure maps

The diagram also includes the following items:-

- A and A' HOL Interface
- B and B' User programs
- C and C' User files

In addition, the following features are included in the system:-

- * Graphic polygon processing for land use management;
- * Processing of multiple databases;
- * Optimization of database storage;
- * Security of the database and each attribute;
- * Activity log; and
- * Input data verification and editing.

3.6.1.3 DMRS Database Organization [M & S Computing, Inc., 1978]

Each DMRS database is stored as a single file containing information about the file structure, the definition of codes and the definition of attribute security as well as the actual data. The data is stored in an optimized form. Database storage is allocated by bits such that

a single word of database storage may contain multiple data elements. This optimization occurs in a manner which is totally transparent to the user.

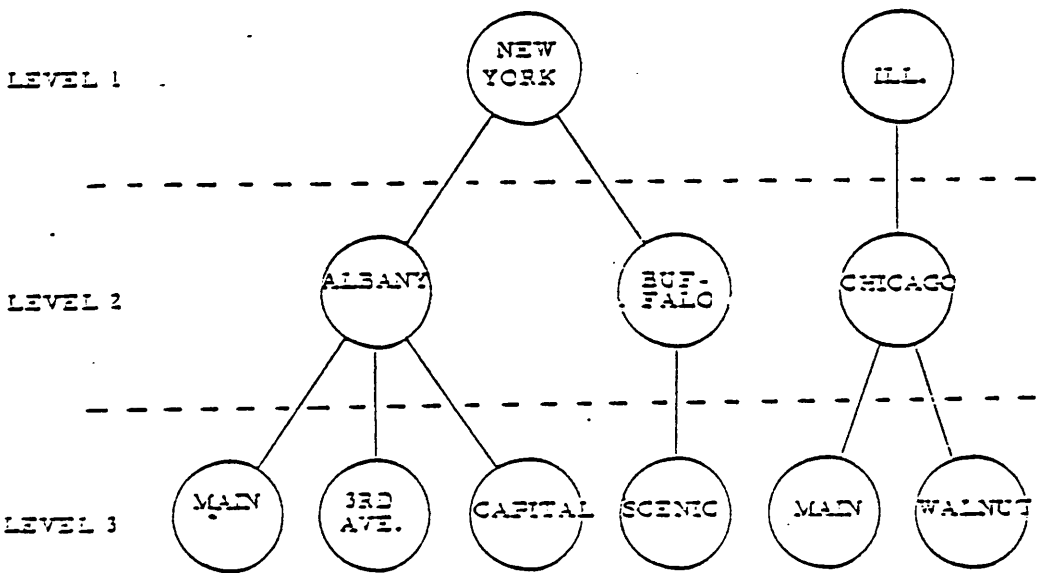


Fig. 3.3 DMRS database - logical organization

The data structure is organized hierarchically as illustrated in Fig. 3.3. It has no complicated indexing scheme, although all data elements are searchable (using a key). DMRS uses the so-called Disk Data Scanner to search the database and retrieve selected data elements at disk hardware speed. The speed and structure which are made possible by the hardware Disk Data Scanner also provide the interactive graphic response associated with IGDS.

DMRS supports multiple attribute files with each file containing the attributes associated with specific graphic elements which are being maintained in the IGDS graphic files. Each DMRS attribute file may contain a unique user-defined schema and typically maintains the attribute data associated with a related group of graphic elements. In this way, large amounts of attribute data can be supported in a manner that allows different classifications of the data to be maintained in different databases. This allows the user to maintain relatively small DMRS files, to maintain very fast retrieval response times and to support an almost unlimited amount of attribute data.

3.6.1.4 IGDS

IGDS is a complete, turnkey, interactive graphics system composed of an integrated configuration of hardware and software which is custom-tailored to meet the mapping and drafting requirements of municipal and utility organizations. However, in the context of the present discussion, its links to and its integration with the DMRS system will be emphasized, since it is the combination of the two which gives the GIS/LIS or AM/FM capability.

The graphics operations and applications supported by the IGDS system can be summarized as follows:-

- * Interactive graphics: Input, display, plotting, access, modification;
- * Data validation checking
- * Engineering design;
- * Polygon storage;
- * Polygon retrieval;
- * Report Generator (graphic statistical results also available);
- * Survey data input;
- * Stereoplotter input system;
- * Terrain data system;
- * Cross section/profile system; and
- * Survey system.

3.6.1.5 IGDS/DMRS System Overview & Interaction

IGDS and DMRS work together to provide an interactive graphics-oriented management information system capable of supplying information on land ownership and land use and the location and distribution of networks operated by the public utilities. In addition, it provides the tools required to carry out facilities management and land use analysis to provide informative answers to ad-hoc queries and to provide specialized management reports.

The IGDS/DMRS features include:

- * the definition of complex attribute files linked to graphic entities;
- * the facility to issue queries based on combinations of attributes and/or graphic criteria;

- * the generation of simple or complex reports which may vary from tabular lists to multi-summary reports;
- * the provision of reports and bills of material;
- * the use of multiple databases;
- * the ability to carry out polygon overlay processing for calculating resultant and remainder polygons based on attribute selection criteria; and
- * the ability to integrate existing data files into the system.

These features are especially useful for applications requiring large amounts of attribute data associated with the graphic database, as is the case with the utility services.

In systems with applications involving both IGDS and DMRS, the two databases must appear as a single coordinated production tool. A real-time link approach, shown in Fig. 3.4, is adopted as the central conceptual design for a standard IGDS/DMRS user.

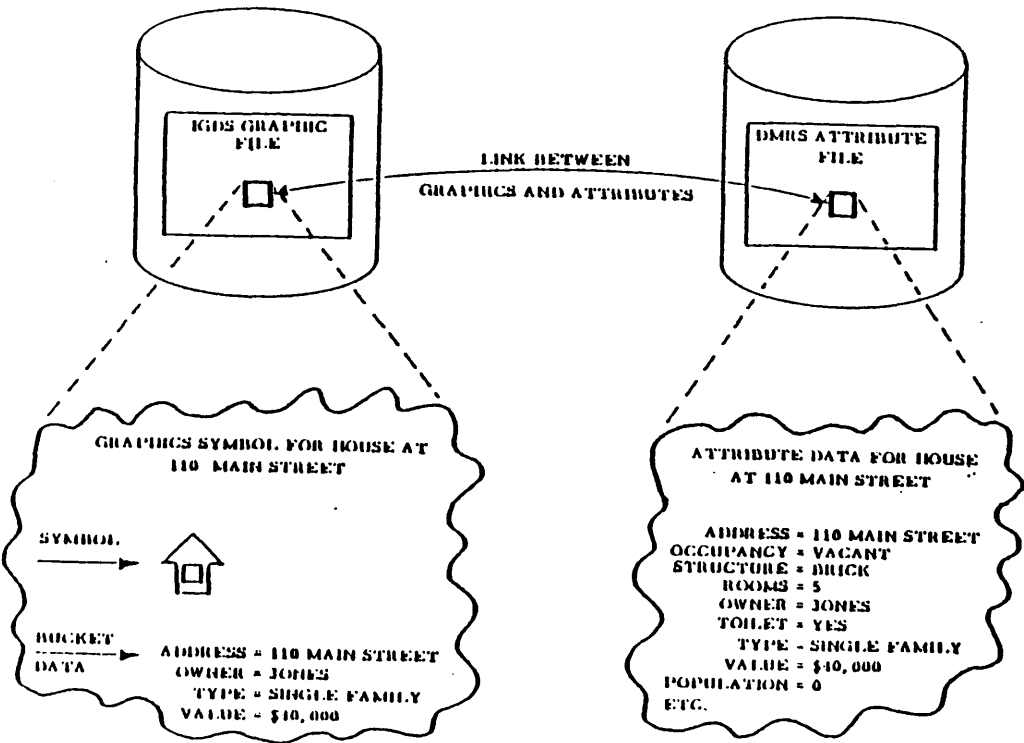


Fig. 3.4 The real-time link between IGDS and DMRS

As already mentioned, the DMRS attribute file generally contains non-graphic attribute data related to the graphic elements maintained in an IGDS graphic file. In most cases, there will

be more attribute data in the DMRS database for each graphic element than is desired graphically at one time. For this reason, a bucket of data which is a subset of DMRS attributes is defined and held by IGDS, (see Fig. 3.5). This bucket consists of a copy of a set of attributes physically attached to the graphic description of the elements within the IGDS database. This attribute bucket occupies the first part of the variable length 'associated element' field of an element's description. This field can be up to 75 words in length and can contain any combination of bucket data and associated element data.

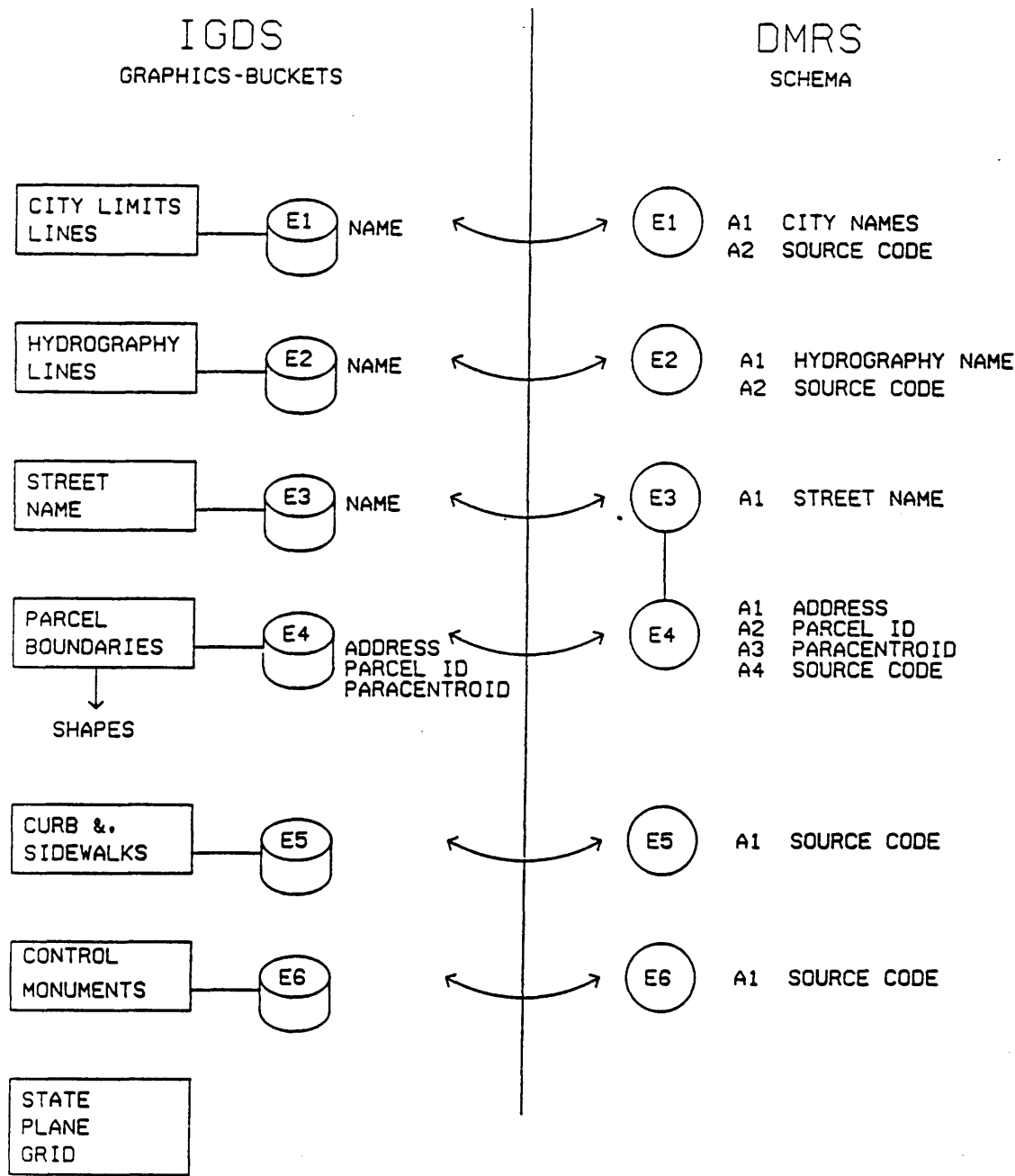


Fig. 3.5 The buckets held by the IGDS and the linkages with DMRS

To create a set of IGDS/DMRS supporting databases, the linkage between an IGDS graphic

element and its DMRS related attributes must be established by the IGDS operator using the concept of the bucket mentioned above.

When a graphic element is added to an IGDS design file, the operator must assign to it an attribute value or an entity name if it is to have a corresponding entry in the DMRS database. Note that initially, the element exists only in the IGDS design file and the corresponding attribute or 'entity' must be added to the DMRS database later on by a post-processing analysis assigned to the IGDS graphics element. This data is added to the element's bucket if one already exists, or a bucket is automatically generated if one does not exist. When the system generates a bucket, an identification number unique to the working design file is automatically assigned to it.

The bucket data is physically attached to the graphic data to form a single record within the IGDS database. This merging of the graphic and attribute data into a single record allows an instantaneous response to bucket queries and updates by taking advantage of the inherent serial operations of an interactive operator. In an interactive graphic environment, a graphic or attribute query or modification must be preceded by an identification procedure to define the graphic subject of the operator's requests. In IGDS, this identification sequence locates the graphic record in question, highlights the element to the operator, and maintains a copy of the record in memory to provide an instantaneous response to the operator's next activity.

The definition of buckets and their interaction with DMRS are controlled by the IGDS user. When the user assigns an attribute to a specific graphic element, IGDS automatically creates a bucket for that element, assigns a unique IGDS/DMRS linkage identification number, and adds the attribute data to the bucket. Subsequent attributes added to the element are merely added to the already existing bucket. Fig. 3.5 illustrates the linkage between an IGDS graphics element with its bucket and its corresponding set of attribute data in DMRS.

3.6.1.6 *The Interface Concept Using the Disk Data Scanner*

As mentioned previously, IGDS uses a proprietary hardware device called the Disk Data Scanner to selectively extract data from a disk file at the data transfer rate of the disk. This technique of hardware data extraction allows IGDS to select an individual line segment from a file of over 15,000 line segments in 700 milliseconds. This combination of

hardware and software allows the IGDS to support up to eight graphic workstations in an interactive environment.

DMRS, like IGDS, also derives its speed of response by optimizing the use of the Disk Data Scanner. The Scanner eliminates the software indexing complexities normally required to support a database management system by making every attribute of every element in the database directly retrievable by hardware. Each DMRS component used to build the database, update the database, and generate reports from the database has been designed to handle a large volume of data as fast as possible. Database operations typically require complex multiple queries involving combinations of attributes of various elements within the database. Each retrieval operation produces a list of elements, and the final result of a typical query requires that a resultant list be generated by a series of Boolean operations on intermediate retrieval lists.

3.6.1.7 *Creation of a Land Database Using the IGDS/DMRS System*

A diagram showing the series of steps involved in the creation or generation of a cartographically-oriented land database for use with the IGDS/DMRS system is given in Fig. 3.6. As shown in the boxes at the top right hand corner of the diagram, data may be input from aerial survey data acquired through photogrammetric operations carried out in stereo-plotting machines. This may be in the form of a traditional line map (or its digital equivalent) or an orthophotograph. Alternatively, as shown on the left side of the diagram, data may originate from existing maps, e.g. land use, land evaluation, land register or tax maps; topographic maps; etc. In both cases, the graphics data is entered into the IGDS database.

The necessity to fit data from all these different map sources which may be at different scales; may have different orientations; may be based on different (e.g. local, regional or national) coordinate systems; may have undergone undesired dimensional changes; etc., is recognized by the provision of what Intergraph terms an 'Elastic Body' transformation. In normal surveying and mapping terminology, essentially this is a polynomial transformation utilizing least squares techniques to handle redundant data. By using a series of suitably located control points, each set of input data is made to fit a common coordinate reference system. This is represented on the diagram by the boxes in the middle of Fig. 3.6.

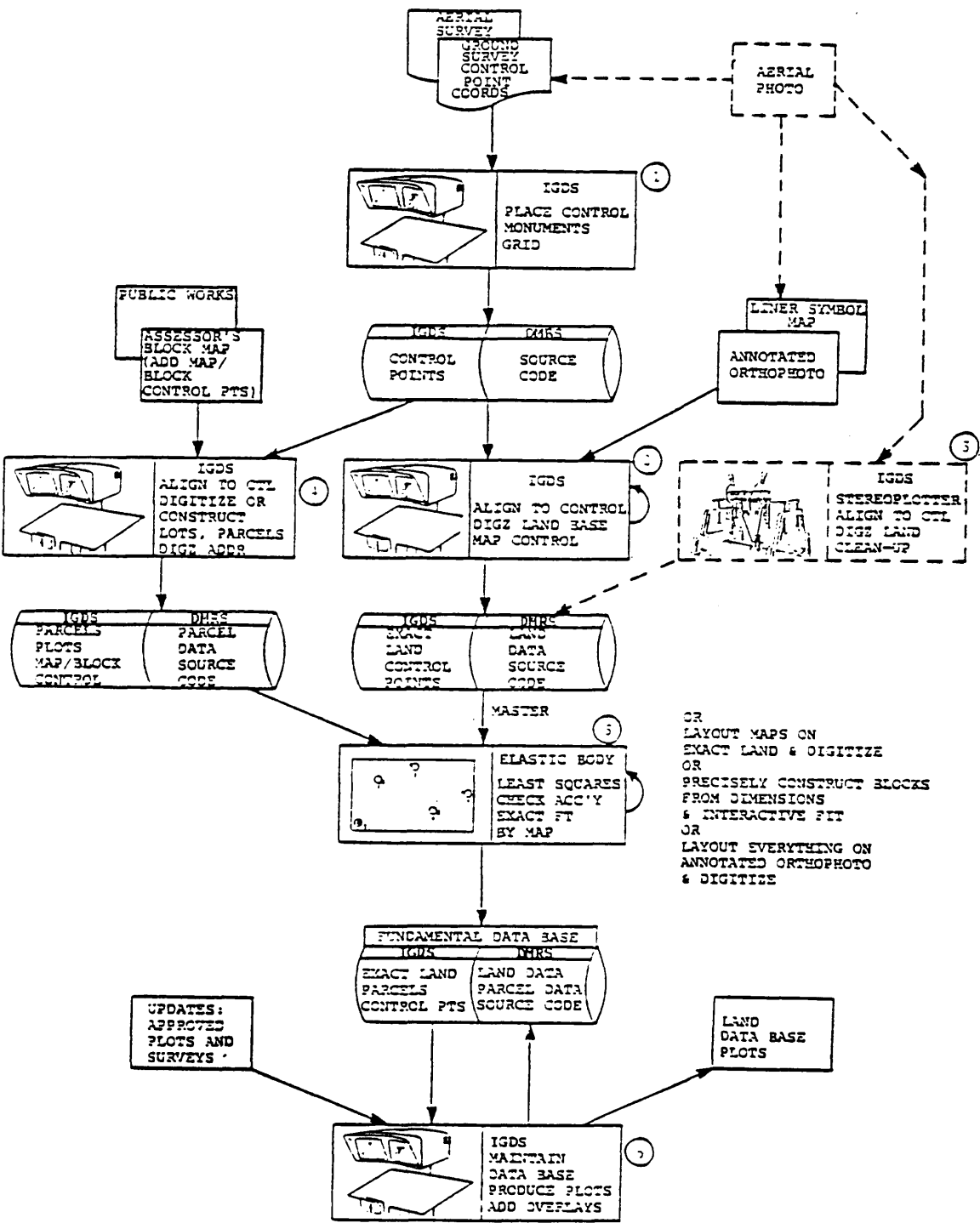


Fig. 3.6 Generation of land database

Finally, the boxes at the bottom of the diagram show the final 'fundamental database', to which updates and revisions can be made from new survey material and from which overlays, plots, etc., can be generated both for checking purposes and for use by the various classes of users.

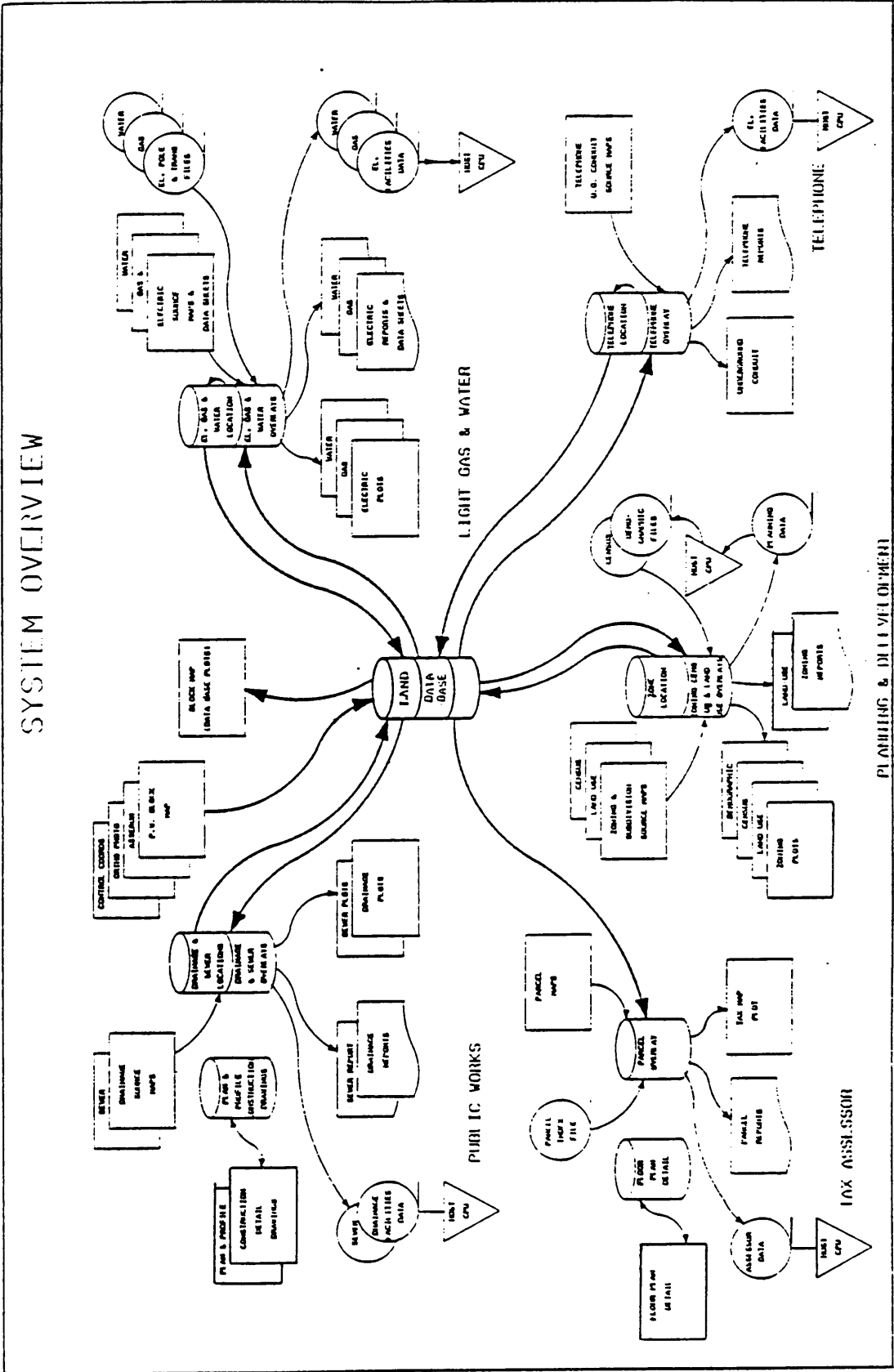


Fig. 3.7 IGDS/DMRS system overview seen from the applications point of view

Fig. 3.7 attempts to illustrate the major inter-relations of all the different sources and products which are commonly encountered in public works projects; utility (telephone, light, gas, water) applications; land parcel and land use mapping associated with cadastral, valuation and taxation registers and operations; and planning and development projects. In this context, the primary objective of the system is to create and maintain a common land base (the land database) in cartographic form for use by multiple agencies (Fig. 3.8). This will reduce the effort incurred by each department in an organization in maintaining its own land background in map form, as well as improving the accuracy and timeliness of the derived maps.

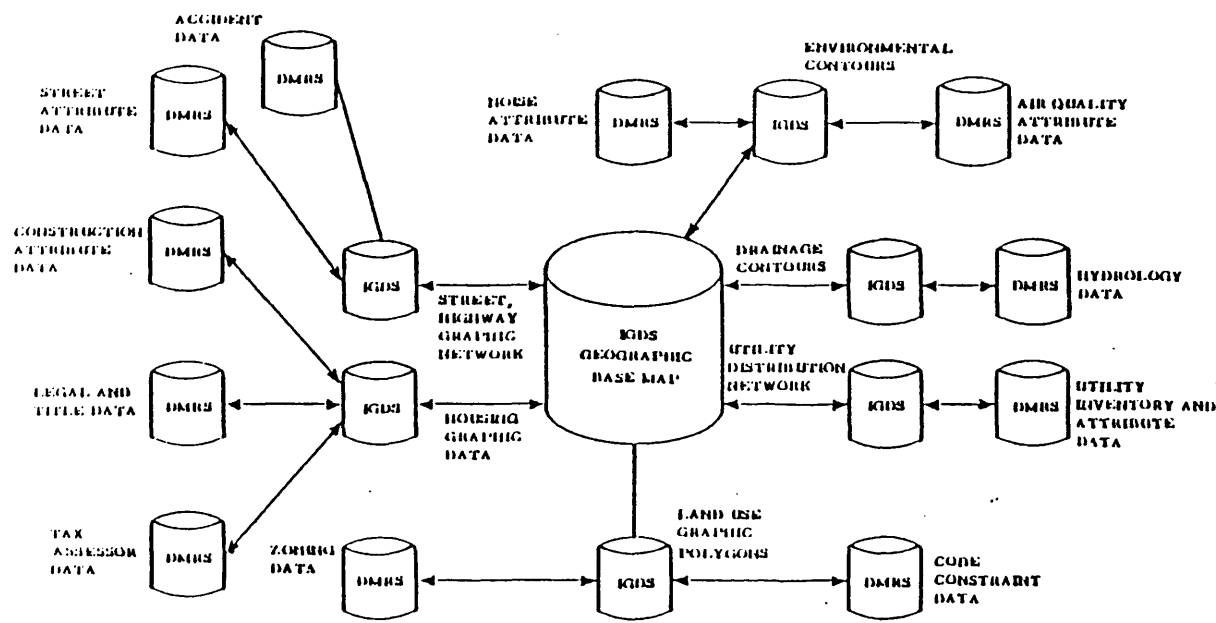


Fig. 3.8 IGDS/DMRS operation showing common database (geographic base map) together with the specialized databases required by each sub-Department or agency

The second objective of the IGDS/DMRS combination is to support inter-agency coordination of work by sharing information on the location of the facilities which are being deployed and used by several utilities. This is intended to reduce the incidence of damage to facilities, especially in underground or excavation operations.

The third objective is to support the mapping and facilities data management tasks required by each sub-department or agency. Reduced costs and improved service can result from creating and maintaining an integrated graphic and non-graphic database of the type represented by the IGDS/DMRS combination. If it is properly implemented and used, it

should be well suited to the engineering, design, operation, maintenance, analysis, and reporting functions which have to be carried out within each agency.

The fourth objective is to support the other drafting activities required by each sub-department or agency.

The overall approach of the Intergraph IGDS/DMRS combination is therefore as follows: to create and maintain a shared land base; to allow each sub-department or agency access to it to create their own individual overlay database; and to require each agency to post the location of completed facilities into the shared database. Individual agencies also have additional applications which are supported by the overall system if required.

3.6.1.8 *FRAMME*

The latest development in this area from Intergraph is the superimposition of an additional layer of software called FRAMME (Facilities Rule-Based and Application Model Management Environment) which sits between the IGDS/DMRS combination and the various applications - e.g. mapping, engineering and accounting - which are required by public utilities. (Fig. 3.9). This has been designed and implemented originally for the Southern Bell Telephone Company in the U.S.A. and is now being implemented as the basis of British Telecom's national AM/FM facility which will eventually cover the whole of the U.K.

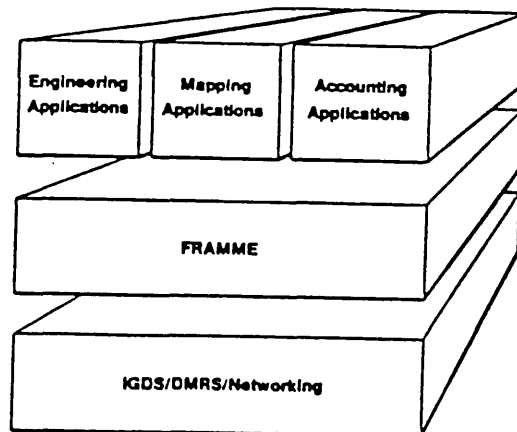


Fig. 3.9 The FRAMME software superimposed on the IGDS/DMRS system

The superimposed FRAMME software supplies the FM (Facility Management) component of the BT system. It is a so-called rules-based 4GL (4th Generation Language) which contains the standard rules and specifications for the company's records, including its own drawing symbologies, feature definitions, design menus, etc. A data dictionary accommodates this rule base, ensuring that: data entry, access and processing procedures are controlled and verified; proper relationships are established between facilities; and the data are structured to give the highest degree of flexibility in using the stored data. FRAMME also includes a feature definition language which defines the specific features to be accommodated in the system. For example, a pole can be represented graphically by a simple circle, while its associated non-graphic description would include its height, class, material and any other attributes pertaining to it [Parker, 1990]. The knowledge base of FRAMME provides a model that includes:

- i) the location of each item of plant;
- ii) the relationship between one item of plant and another; and
- iii) related information about each item of plant.

Each item (or feature) in the model has both a graphics and non-graphics representation which is handled as a logical unit of information, in this case, a logical unit of plant (LUP). The graphics items, maps, plans and diagrams etc., are stored in the IGDS part of the database. The non-graphics attributes of these items, e.g. the connectivity of plant, are stored in the DMRS database. Data can be stored or modified in the facilities model according to the definition contained in the data dictionary. The model analysis and the collection of attribute information is performed through database tracing. The final output from the FRAMME can be in the form of either a graphics (map or diagram) or a text-based report.

3.6.2 Synercom Informap

Synercom is a well known and established American company in the field of digital mapping. Synercom's earliest digital mapping system was called Informap and, in its initial form, employed a hierarchical database system. This was sold widely to public utilities, municipal/local government agencies and planning bodies, especially in the United States from the mid-1970s onwards. During the early 1980s, Synercom's Informap product was adopted by Wild Heerbrugg, the well known Swiss survey and photogrammetric

instrument manufacturer and system supplier, and developed to produce the Wildmap system based on the principle of continuous digital mapping. The principal user of the Wildmap version in the U.K. is BKS Surveys in Northern Ireland. Experience with this system is given by Byrne and Neil (1983) and Byrne (1986). However, within the U.K., the Informap system has been widely sold to public utilities by DEC and Laser-Scan who act as agents for Synercom and provide the necessary hardware and software support. The largest customer is British Gas, but the system is also used for network management by Wessex Water and Bristol Water.

Since the Informap package was not well provided with analytical tools or routines, a product entitled the Environmental Mapping Information System (EMIS) was produced which linked the Informap system with the Odyssey geographic analysis software produced by Harvard University. In the following sections, a review of Synercom's Informap will be presented, then a description of the EMIS software will also be given.

3.6.2.1 *Overview of Informap*

The overall arrangement of the original Informap system, its component modules are shown in Fig. 3.10. The software components of the system include [Wild, 1980(a) and Wild, 1980(b)]:-

- i) CAP/IN- Cartographic and Photogrammetric software. It controls the connection of analogue photogrammetric plotters to the computer system and records and organizes the data measured by these stereo-plotting machines.
- ii) MAP/IN- Mapping Interactive Graphics Digitizing System. This is a real time, interactive graphics computer system that facilitates the entry of geographic and associated tabular data in the form of maps, aerial photographs, and records.
- ii) INFORM- Information System for Mapping and Records Management. A database management and report generation system designed specifically for geographically organized graphic and related attribute data. INFORM organizes all of the individual maps and associated records input through CAP/IN and MAP/IN into a 'Continuous Digital Map' to permit retrieval of maps and information by polygonal area, street address, or other geographic identifier.

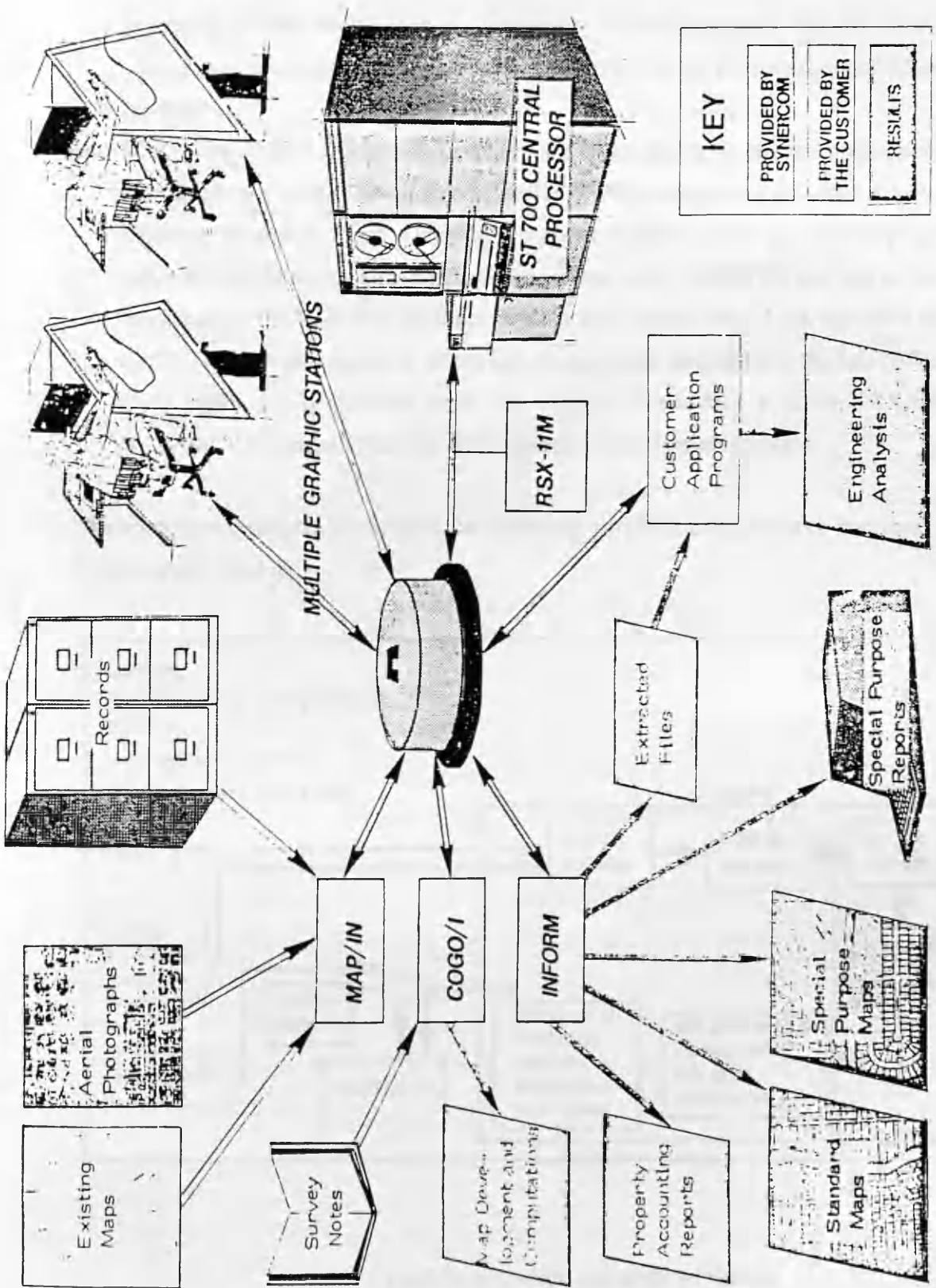


Fig. 3.10 Overview of Informap and its components

- iii) COGO/I- Interactive Coordinate Geometry System. An optional interactive coordinate geometry system which supports geometric and trigonometric constructions and adjustments in support of high precision data input to the Information and Mapping System.
- iv) RSX-11M- Digital Equipment Corporation's multi-programming real-time operating system for the earlier 16-bit PDP11 series of mini-computers provides a real-time response through dynamic allocation of system resources. User application programs and other database management systems can run under RSX-11M and can be readily interfaced to the INFORM database through files. While RSX-11M was used on the PDP11/70 mini-computers on which Informap was run originally in the late 1970s and early 1980s, the system has since been adapted to use DEC's 32-bit VAX, Micro VAX and VAXstation machines which use the VMS operating system.

The hardware arrangement used in the Wildmap version of Informap is shown in Fig. 3.11(a) and (b) below.

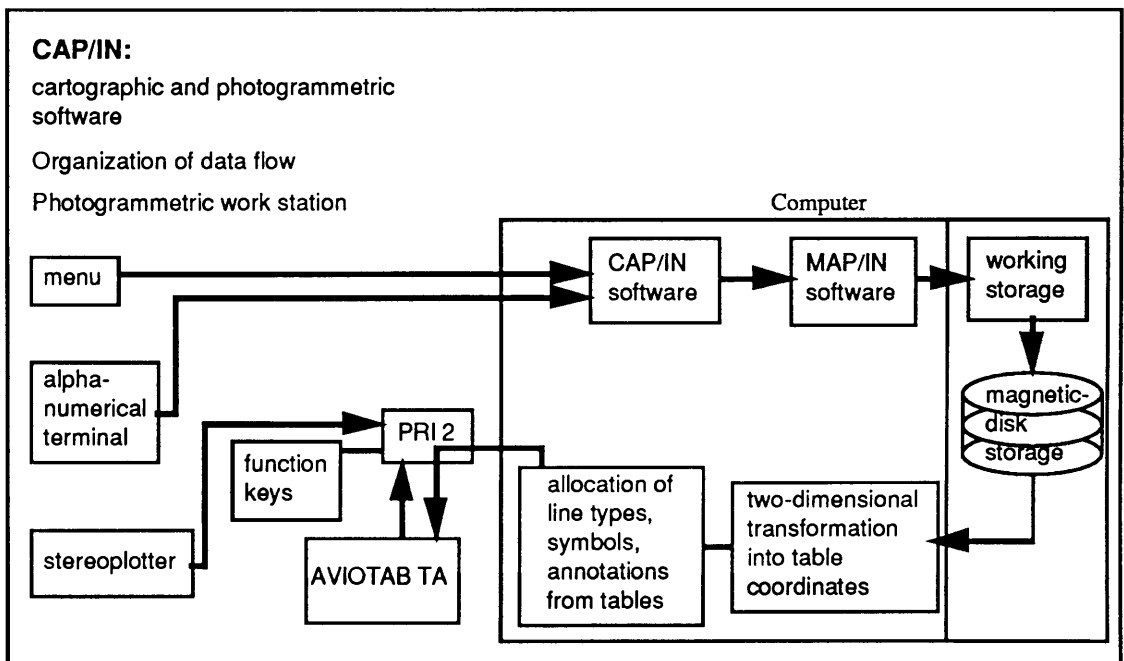


Fig. 3.11 (a) CAP/IN -> MAP/IN in Wildmap

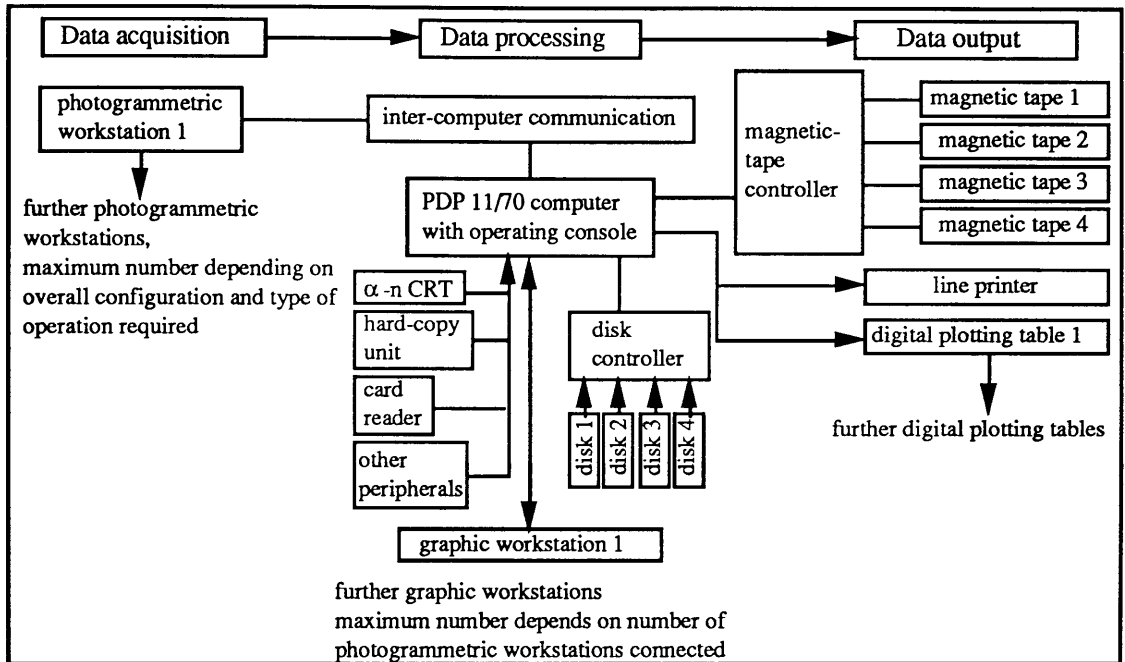


Fig. 3.11(b) The hardware arrangement as used by Wildmap

3.6.2.2 Informap Data Base Organization

Informap's database structure was again, like most information systems designed in the 1970s, hierarchical, with multiple entry keys and is shown graphically by Fig. 3.12.

Since most user requests are geographic in nature, location is the first level of organization. This is accomplished by partitioning the entire area served into a grid of regularly-shaped elemental areas called facets. All the data required to describe the facilities and the geographic features contained within the boundaries of each facet are stored separately within the database. This permits direct access retrieval of data by a wide variety of graphics keys such as:-

- i) Location;
- ii) Map Name (where map name indicates location);
- iii) Area Name (city, district, region, etc.);
- iv) Street Address;
- v) Street Intersection;
- vi) Known Landmark.

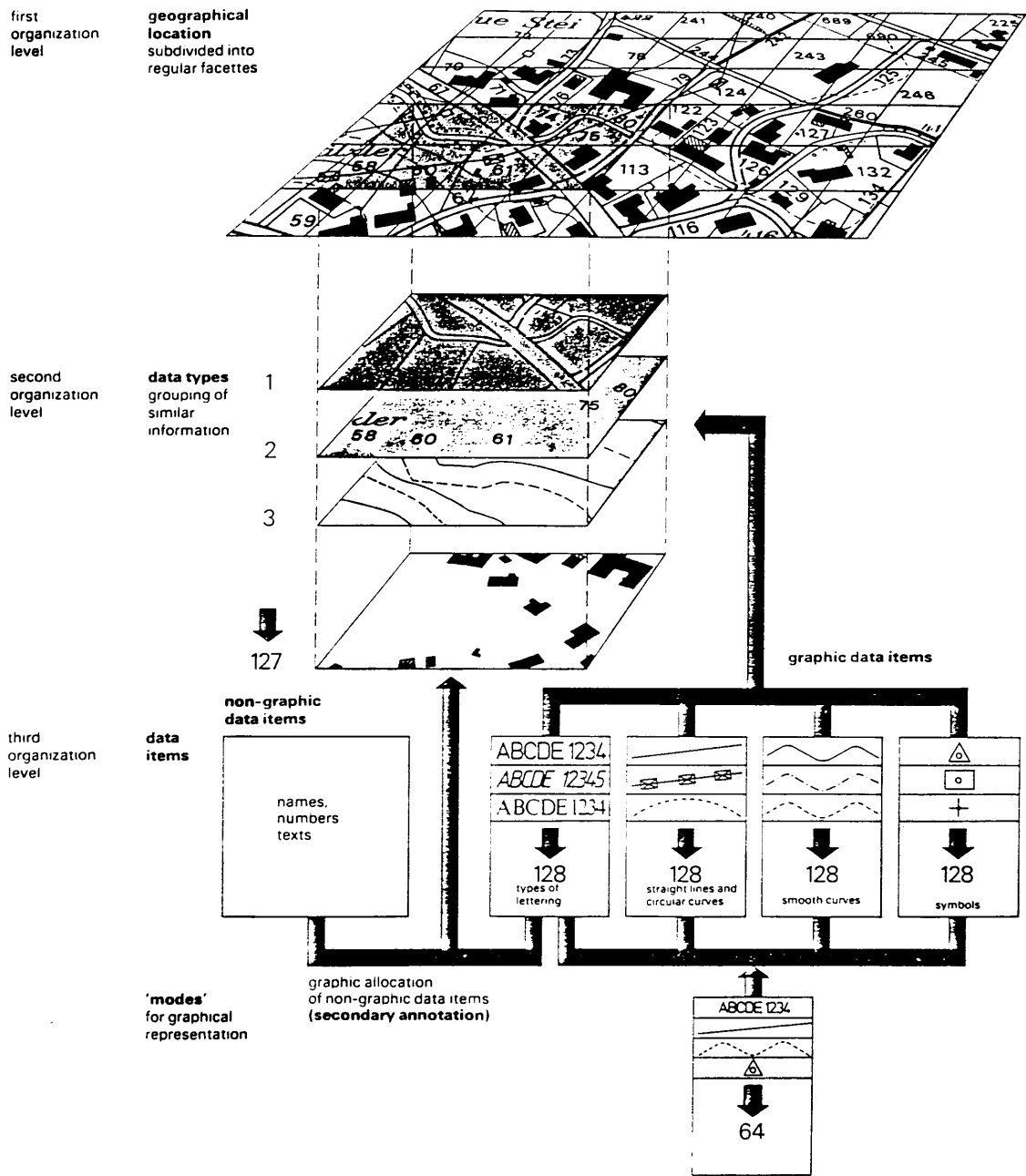


Fig. 3.12 The data structure of Informap

Within each facet, data is further organized into groups of like data called data types. Each data type represents a certain kind or class of graphics or facilities elements such as cables, conduits, water mains, base map features annotation, etc. Each data type can be viewed as a layer such as a property layer, a water mains layer and a gas mains layer. There is no inherent limit to the number of map information layers that Informap can create and hold, although no more than 128 can be accessed at any one time. The layers that are active at any

particular time are defined in a schema. Thus within a municipality, the fire department might define a schema for the specific layers that it needs, the police department would define another schema, and so on. Thus this second hierarchical level permits the Informap database manager to directly access a particular kind of data held in a layer within a facet without having to search through unrelated data.

Each data type may be further divided into a series of data items. A data item describes a particular property or attribute of a data type, such as its location, size, operating condition, date installed, or work order number. Each data type or layer can have as many as 64 attributes. The same set of data items are used to describe each element (e.g. each cable, each terminal, etc.), for a particular data type.

A further level of the database organization is that, as in the Intergraph IGDS/DMRS systems, each data type in Informap is subdivided into graphic and non-graphic data items. Graphic data items contain the information that is essential to describe each element of the data type graphically, such as its location, coordinates, symbol, line type, etc. For display, these graphic data items or attributes can be depicted in any one of 128 line types, 128 symbol types or 128 types of lettering, each of which can appear in any of 8 colours. Non-graphic data items describe attributes such as names, numbers and text. The form and format of these non-graphic data items are user defined by a variable database schema. Informap provides the ability to display these data items graphically as secondary annotation if required.

The graphic portion of each data type is separated from the corresponding non-graphic portion, and a direct link is provided so that INFORM may examine all data items for a data type only when necessary.

3.6.2.3 *Odyssey*

Odyssey is not itself a stand-alone GIS system, but a geographical analysis package written in Fortran and developed over a period of several years by researchers at Harvard University's Laboratory for Computer Graphics and Spatial Analysis. Its importance here is that it has been combined with the previously described Informap to produce the EMIS system which will be described in the next Section. The package comprises several modules, each of which performs different analytical functions. Each of these functions can

be run separately by issuing commands on a terminal.

The package is an integrated software system for geographic analysis. The Odyssey series of program modules can prepare, edit, manipulate and display geographically based areal data. The resulting data of the analysis can be reported in the form of two-dimensional shaded maps showing thematic attributes in black and white or colour and the map can be scaled, shifted, rotated or sectioned and the map elements can be arranged in any way in the display space. On the other hand, the three-dimensional module displays the spatial distribution of a quantitative attribute as a set of raised polygonal prisms, with hidden lines removed. The same symbolisms used in two-dimensional maps may be used for shading prism tops and sides.

Furthermore, Odyssey contains a module that performs common modifications to an Odyssey cartographic file. A map may be generalized, transformed into any one of ten common projections, rescaled, registered to another coverage using linear transformations, split into subsets, or aggregated with other coverages. Attribute files may be split or aggregated appropriately as required. Geographic overlay is the most advanced analytic capability of Odyssey. It implements the polygon overlay of two or more cartographic files of a region creating a new file of the resulting boundary network with complete identification of the boundary intersections and new polygons, and the recording of the parent polygons. Congruent features are coalesced and insignificant areas eliminated using a user-specified tolerance parameter. Attributes can be allocated in proportion to the area of each polygon (either as density or raw count values), or a user may specify a separate factor on which to base the estimations of attributes.

The modules and their functions, according to Gatrell and Charlton (1987), are listed in brief in the table below (Table3.1), while the relationships between these modules are shown in Fig. 3.13 [Anon, 1985]:

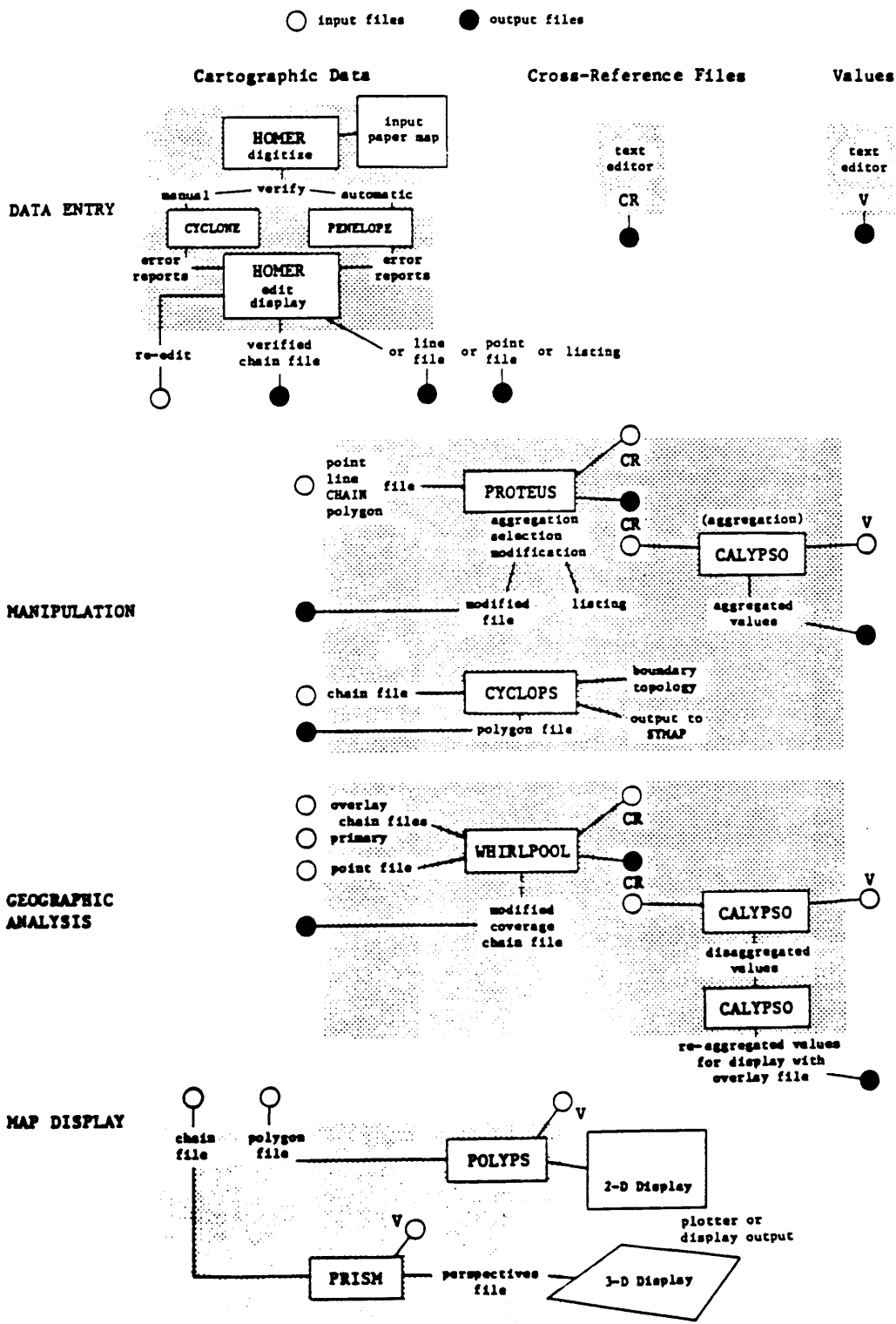


Fig. 3.13 Odyssey system overview

Table 3.1: ODYSSEY MODULES & FUNCTIONS

| <u>Name</u> | <u>Description</u> |
|-------------|--|
| HOMER | Digitizing and editing of cartographic data; |
| CYCLONE | Analysis of errors in chain files; |
| CALYPSO | Analysis of zonal data (values files); cross-aggregation; |
| PENELOPE | Automatic creation of structured chain files from ordinary line files; |
| CYCLOPS | Construction of polygons from a chain file; |
| PROTEUS | Cartographic data manipulation; |
| WHIRLPOOL | Polygon overlay, point in polygon searching, etc.; |
| POLYPS | Two-dimensional mapping using polygon and attribute ('values') files; |
| PRISM | Three-dimensional mapping of polygon (zonal) data. |

It is very clear from the diagram (Fig. 3.13) and what has been written about Odyssey, that this package deals with simple data files as such rather than with the data provided by an organized database. Thus the various functions which are achieved by using a DBMS in other systems are missing in Odyssey. This means that the introduction of data is an operation which has to be continuously implemented afresh since the data is not held in a single managed database but in a series of unconnected and unrelated files. Thus, the user is asked to issue the commands as to which module and which data set needs to be called at a certain stage. Needless to say, this rather restricts its use, though in fact, since it was a very early attempt to provide a geographically based analysis package, it has a very wide range of users, especially in North America. In the U.K., various universities (Durham, Lancaster, Glasgow, etc.) have installed the system which has also been used by the ESRC Data Archive at the University of Essex.

3.6.2.4 *Environmental Mapping Information System (EMIS)*

As mentioned previously, EMIS is an integration of Synercom's Informap mapping information management software with Harvard University's Odyssey geographic analysis software. Fig. 3.15 illustrates the Informap II/Odyssey integration [Synercom, 1984].

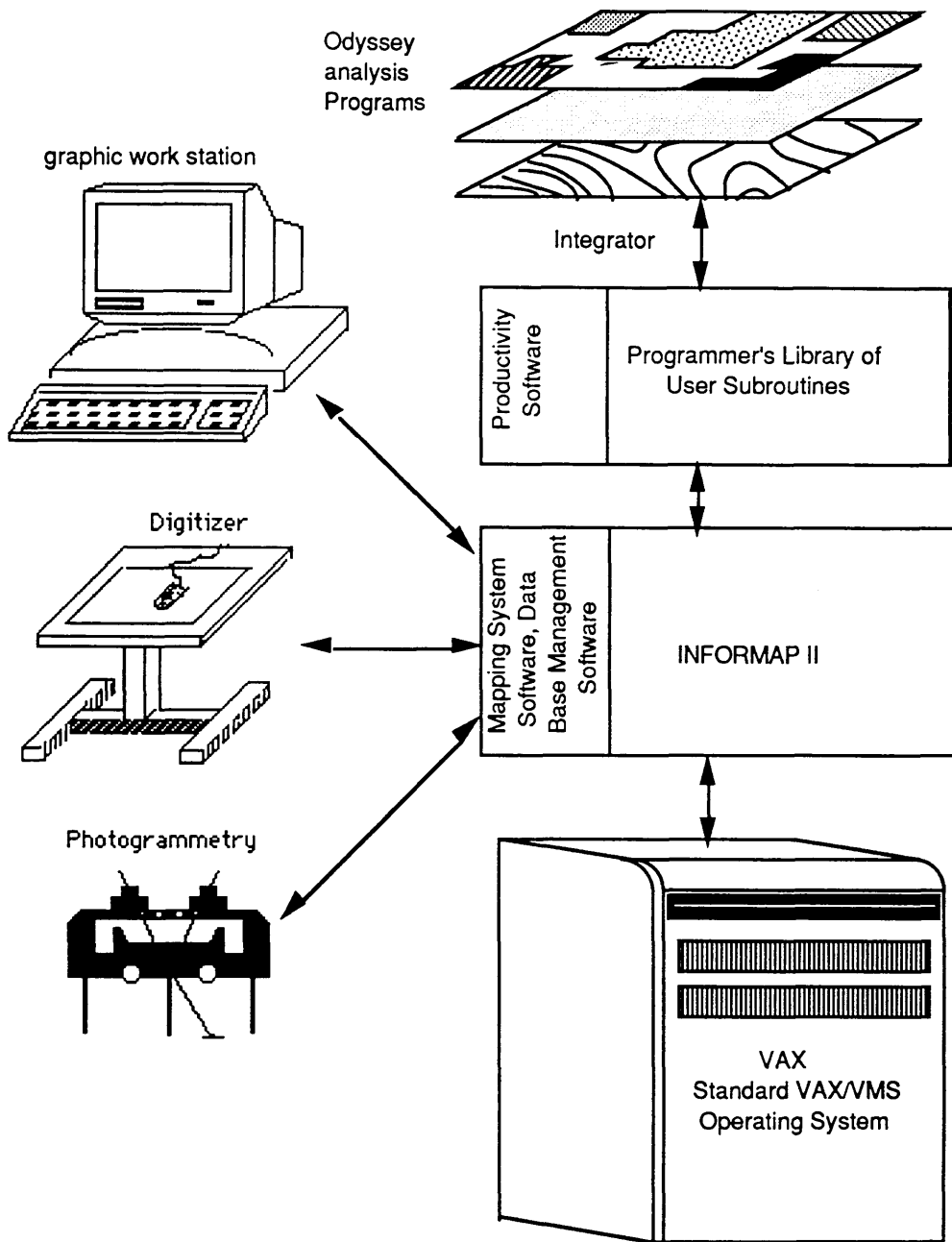


Fig. 3.14 Informap II/ODYSSEY integration in EMIS

Besides developing the EMIS concept, Synercom seems also to be developing Informap II further in the direction of becoming a relational-based GIS/LIS, complete with a query language and complex reporting features. The basic system described in Section 3.6.2 is now termed Infomapper. Building on this, a second level, Infomanager adds advanced map maintenance and enhanced data management procedures to the basic facilities of Infomapper. Finally, the Infoquest module provides the relational database management system (RDBMS) complete with the query language and report facilities.

3.7 Network Based Systems

Geographic or Land Information Systems based on the network data model are very few in number and only a single system has been included here as an example.

3.7.1 Strings

Strings is a relatively simple polygon-based GIS developed by GeoBased Systems, Inc. which is written in Fortran [Coe & Quigley, 1986]. The Strings system is designed to simulate the cartographic process during map encoding and, at the same time, use a network database management system needed to carry out map analyses. The system has the cartographic features of a CAD/CAM drafting system as well as the file structure and software design needed for area calculation, overlaying, networking and other analytical features which help support the answering of queries to assist in decision making.

The structure of Strings is made up of four basic files, the first being the centroid file, the elements of which represent polygon seeds and contain pointers to the next file, which is the file containing line and point features. Each polygon record is assigned its 'start line record number', being the line to the west of the seed point given during the polygon build process. The start line number record in the centroid file points to the correct 'line record number' in the line file. The second file is the line file, where each line record is assigned with a top and bottom 'exit line number' based on the 'line exit angle', which is calculated and stored in the third file, the intersection file. Intersections are created automatically wherever a line begins and ends. These are then stored in the intersection file. The fourth file is not mentioned in the article by Coe and Quigley, but presumably it is the file where data are kept about the different maps to be stored in the database. Table 3.2 shows the links between the three files discussed above and Fig. 3.15 shows diagrammatically the terms used to define a polygon.

| LINES | | INTERSECTIONS | CENTROIDS |
|---------------------|-----|------------------------|---------------------|
| Record# | | Record# | Record# |
| Top Exit Line | | 1st Line Record # | Start Line Record # |
| Bottom Exit Line | <-- | Exit Angle of 1st Line | (if polygon) |
| Top Intersection | | 2nd Line Record # | |
| Bottom Intersection | | Exit Angle of 2nd Line | |

Table 3.2 The three files forming the structure of Strings

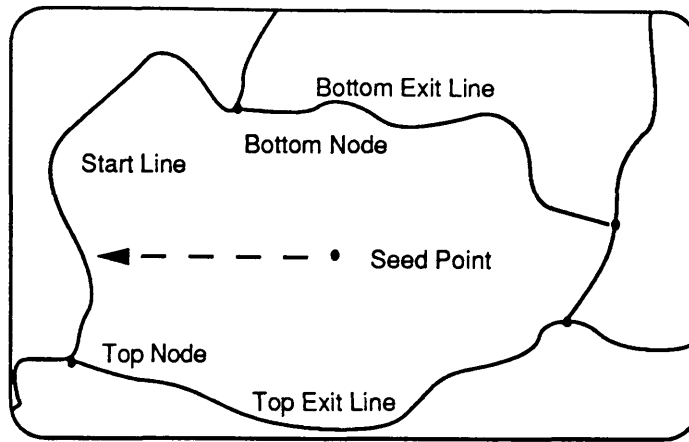


Fig. 3.15 Polygons are specified by a user-entered point (seed) and defined by a chaining process

A polygon is then defined by following the exit lines through the line file until the beginning intersection number is reached, thus closing off the polygon. Polygons are then specified by a user-entered point and defined by a chaining process.

Due to this hierarchy in the structure of the database, problems can be found at different stages of the process. Problems can arise should errors be made, for example, during area calculation. Area calculation is done by a separate program but when the data are entered into the same data file of the database, the occurrence of an error during the run of this program can cause the loss of the previously entered centroids. When this happens, the time spent (to re-enter and edit the centroids) could be quite considerable. Finally, an inherent property of network models is the limitations imposed when updating since this will cost time in re-arranging pointers from and to different files. When complex models are in question, as in spatial models, the resulting pointers are a considerable overhead. Due to the inter-dependency of data, the more complex the data, the more planning time is needed on a practice file to search out all options.

From the above description, it can be seen that the orientation of Strings is towards data which is related to geographical areas rather than to the networks of points and lines which are the main concern of the Intergraph IGDS/DMRS system and Infromap systems oriented towards the needs of public utilities. Thus the maps generated by Strings are most likely to be of the area/polygon-based thematic type.

3.8 *Relational Systems*

In parallel with the developments in database management systems in general, a variety of geographically-based information systems have been developed recently which are based on the relational model of DBMS.

3.8.1 *SysScan DNMS*

SysScan is considered to be one of the leading European companies in the field of digital mapping and geographically-based information systems. It was formed by a combination of interests, but principally Kongsberg Vapenfabbrik (Norway) and MBB (West Germany). Recently the MBB interest has been replaced and the company's operations are nowadays centred on Kongsberg (Norway) and Bracknell (U.K.).

The company's mapping-related products are numerous and very widely used. SysScan's earlier systems were intended purely for digital mapping applications. These were developed further into what might be termed a mapping information system which includes databases and a database management system.

As the name implies, the Distribution Network Management System (DNMS), is oriented towards the public utility sector of the market place, i.e. it can be viewed as being an AM/FM system. The overall system architecture of DNMS is given in Fig. 3.16 [SysScan, 1988].

The input and output functions are shown in the boxes located at the top of the diagram respectively. The central core of DNMS is shown in Fig. 3.16 comprising three elements:-

- i) the integrated relational database;
- ii) the application language; and
- iii) the application functions.

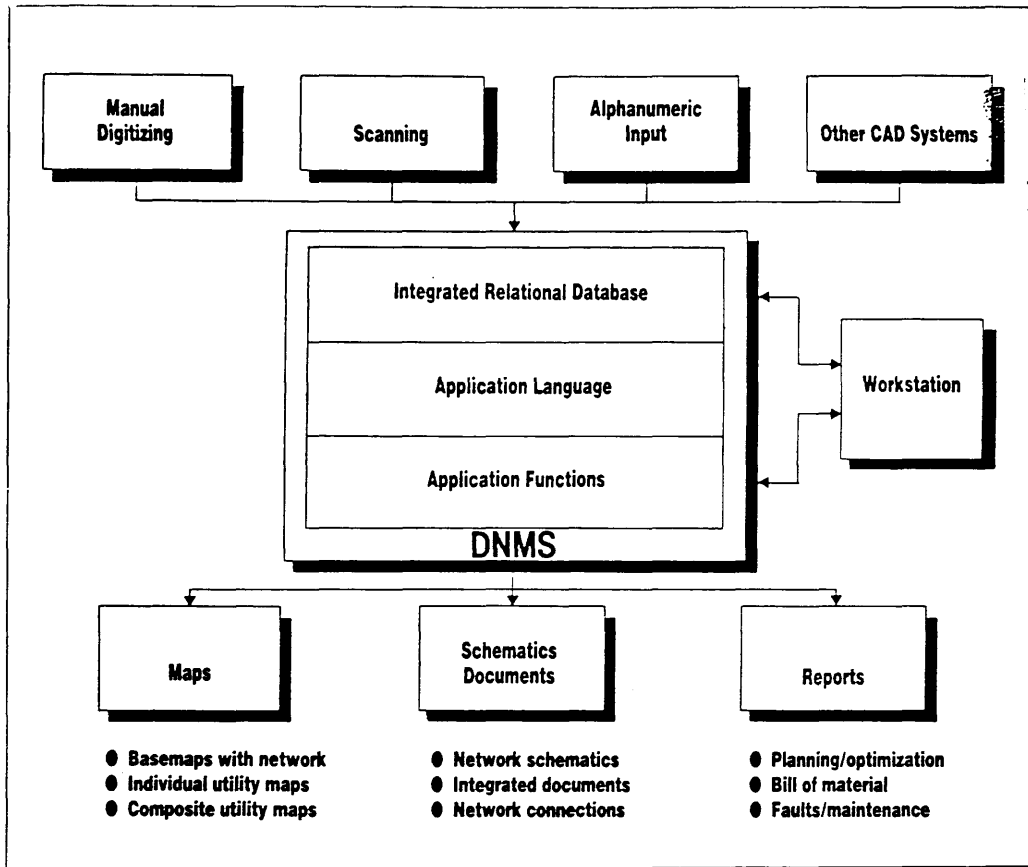


Fig. 3.16 The DNMS system architecture

3.8.1.1 DNMS Input/Output

DNMS is designed to cover a range of functional requirements. In terms of data input, these are:-

- the manual digitizing of existing maps;
- the automated raster digitizing of existing maps;
- the input of non-graphic information; and
- the transfer of data from other CAD systems.

These are represented in the boxes located along the top of Fig. 3.16.

On the output side, DNMS provides facilities for the plotting of maps and other documents using a variety of printers and plotters. These are represented by the row of boxes at the bottom of Fig. 3.16.

3.8.1.2 DNMS Data Structure

The data structure in DNMS includes several data types which are designed to provide a complete description of a utility network. These include:-

- a) positional coordinates for all the network elements;
- b) related non-graphic data; and
- c) data describing the network logic.

DNMS manages this information with the structure and degree of detail required by the user.

3.8.1.3 System Architecture

DNMS gives the user access to an Application Control Language (ACL) which is a tool for the individual tailoring of the package to suit the needs of the user. With ACL, the user can implement and fully integrate additional application functions within the standard product. This enables applications to be developed exclusively to suit an organization's own method of operation within the standard framework of network applications which will apply to utilities such as electricity, water, gas, etc.

Furthermore, the database content is accessible by means of a query language. The result of such queries can be presented in user-defined reports, facilitated by a report generator. DNMS can also communicate with or be linked to other systems that perform other functions on the same data.

Obviously DNMS is a relatively integrated system based on the relational data model which has been produced wholly in-house by SysScan. In this respect, it differs from many of the other relational-based GIS/LIS systems which are based on existing well-known non-graphic RDBMSs on top of which the graphics part of the overall system is constructed and to which it is linked via a suitable interface.

3.8.2 ESRI ARC/INFO

The ARC/INFO system is a very well-known GIS package which has been produced by the Environmental Systems Research Institute (ESRI) of Redlands, California. In numeric terms, it probably has the largest and most widespread base of installations though in terms

of value, probably Intergraph is the market leader in the AM/FM and GIS/LIS field. ESRI is essentially a software house and systems integrator and supplier, buying in computing platforms, peripherals, etc., and porting and integrating its software to run on these different hardware items. Thus in this respect, it differs substantially from other suppliers such as Intergraph, SysScan, Wild, etc., which are also involved in the manufacture of certain hardware items - e.g. InterPro and InterAct workstations, InterMap Analytic stereo-plotters, Anatech and Optronics raster scanners and plotters, etc., (in the case of Intergraph); Vera workstations and Kartoscan scanners (in the case of SysScan); and Aviolyt and S9-AP analytical stereo-plotters and Aviotab flatbed plotters (in the case of Wild). Furthermore, unlike Intergraph's systems which can only run on either Intergraph or DEC hardware, or Wild's which can only be implemented on DEC or Sun computers, the ARC/INFO system has been implemented on a very wide range of machines. These include mainframes (e.g. IBM 3090); multi-user mini-computers (DEC, Prime, and Data General); graphics workstations (e.g. Sun, Apollo, Hewlett Packard, IBM 6150); and microcomputers (IBM PC/AT and PS/2 and clones).

Historically, the ARC/INFO system has used the Henco INFO relational software for the management of its spatial attribute data. However, while the new release of Version 4.0 continues to support the INFO product, it also allows the user to integrate other Relational Data Base Management Systems such as Oracle, RDB, Ingres and DBase II to create the overall system. A new interface tool, the Relation Data Base Interface (RDBI), provides a capability to explicitly integrate any of these RDBMSs directly with the ARC part of the system which handles the graphics side of the GIS, (see Fig. 3.17). However, it happens that, at present, the microcomputer version (PC ARC/INFO) is only available with INFO.

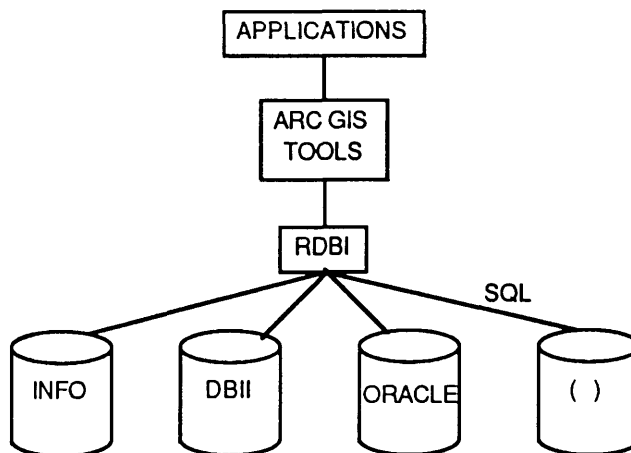


Fig. 3.17 ARC interface tool

3.8.2.1 The Software Tools

In ARC/INFO, the software and associated hardware are used for five basic GIS functions, these are [Dangermond, 1986] :-

- * Data Entry;
- * Data Analysis;
- * Data Manipulation;
- * Database Query; and
- * Data Display and Report Generation.

These five GIS functions are supported by a large number of the kind of 'software tools', Table. 3.3 illustrates these tools. ARC/INFO is a geographic information system which can be used for a wide variety of applications. It includes over 300 software procedures for handling all commonly encountered problems with geographic data. These procedural subroutines are used to build the tools shown in Table. 3.3 below.

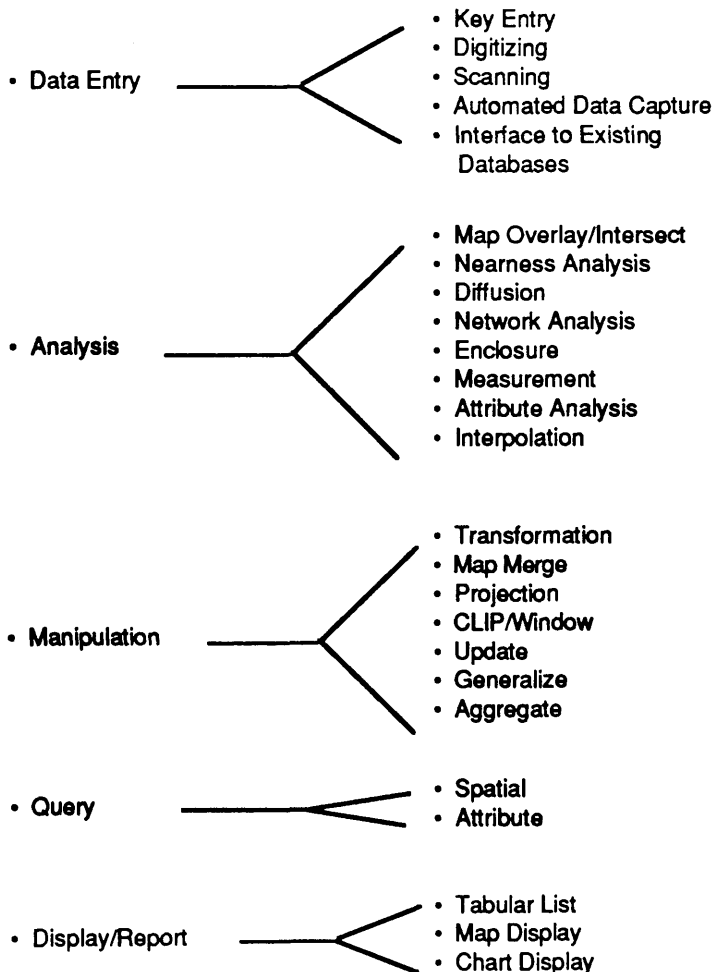


Table. 3.3 ARC/INFO Tools

3.8.2.2 The GIS Model

ARC/INFO is using a topological and relational DBMS model. Those features which are to be depicted cartographically are first defined as points, lines, polygons together with their attributes. More complex groupings of features, such as a group of islands or chains of arcs, are indexed with additional relationships within the database. The system organizes all of the feature attributes in related attribute tables which are managed by the RDBMS. The software tools mentioned above are used to enter, update, analyze, manipulate, extract, symbolize, display and report the information contained in the database. Users can create graphics by relating graphic symbols and shading to point, line, and polygon features. Text is related with text fonts in a similar manner. This allows the user the flexibility to associate the symbology of choice with cartographic features, based on one or any combination of attributes in the database.

A generalized picture showing the relationships of these software tools is presented in Fig. 3.18.

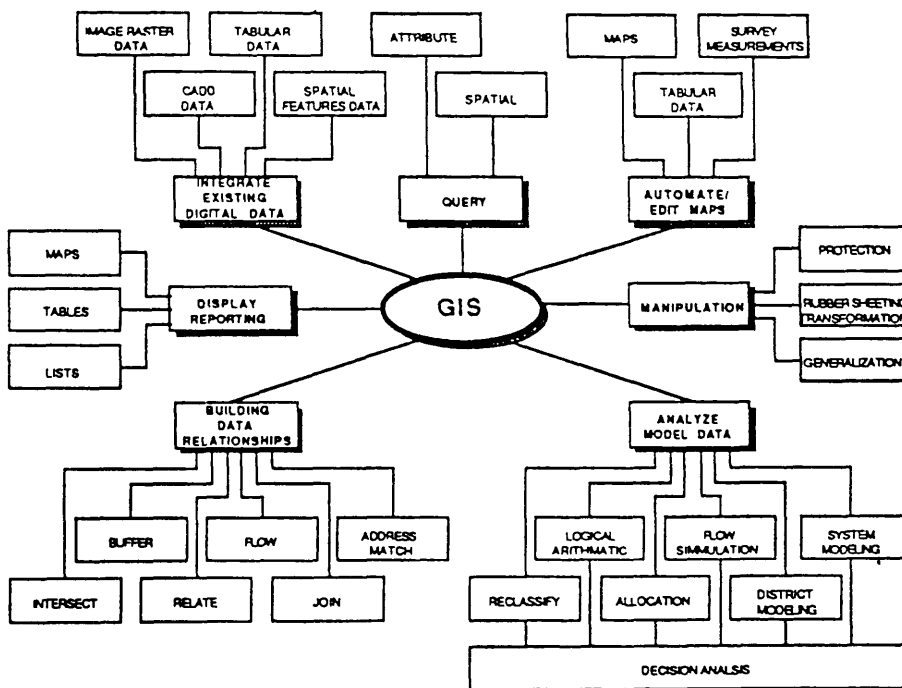


Fig. 3.18 General overview of ARC/INFO

3.8.2.3 ARC/INFO Approach

As a general remark, one can say that ESRI seems to emphasize the generation of polygon-based thematic maps with the ARC/INFO package. Thus most of its users appear to be

interested in mapping distributions, occurrences and intensities of a particular feature from socio-economic data sets on an areal basis. Typical applications appear to be the generation of thematic maps based on census or other comprehensive statistical data on areas such as wards, districts, counties, provinces, states, regions, etc., as required by municipal, regional and national government organizations concerned with administrative and planning. These authorities are able to exploit the rich selection of analytical tools such as map overlay, nearest neighbour analysis and queries based on the use of SQL to provide information on the association of different features contained in individual tables.

The main product sold by ESRI (ARC/INFO) is composed of two tightly integrated software components which are: ARC, which is used to manage the cartographic data, and INFO, used for managing the tabular, numeric and textual data describing the attributes of the cartographic features. As shown in Fig. 3.19, these two components are linked together and are also linked individually to the software tools outlined in Section 3.8.2.1. Both the cartographic and attribute data are stored in a series of related tables. The features present in each of the two components are linked together using "feature identifiers" which provide a common index between the two data types. The user can invoke either ARC or INFO procedures while working on the database.

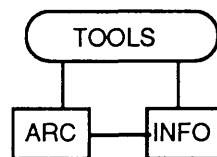


Fig. 3.19

For query purposes, ARC can be considered as sitting on top of INFO and providing a topological pathway to the relationally structured database. All spatial objects (points, lines, polygons) are stored in a series of tables in ARC. The ARC files contain coordinate and topological data, whereas their attributes are implemented in feature tables managed by the INFO component. Thus INFO is used to query, manipulate and generate reports on the non-graphic data stored in the feature attribute tables.

Turning to individual features of the package, an interactive graphics editor ARCEDIT is provided. This has two functions: it interactively edits the spatial database, and it places text and symbols in the appropriate positions on high quality graphics displays. ARCPLOT is used to compose maps on graphic displays for output on plotters. All additional

information such as legends, borders and titles can be added to the displays and the result then sent to the plotter. The MAP LIBRARIAN module aids the tracking and control of many different data sets.

As mentioned above, ARC/INFO supports a full polygon, link and node topological data model. The vector spatial data are stored in a relational database as are the feature attribute data. The COGO (coordinate geometry) program supports direct land-survey data input and ADS (ARC digitizing system) supports digitizer input from cartographic sources. The creation of the topological data model is fully automatic on input. A unique reference number links a feature's spatial data with its descriptive data.

The network software performs analyses on networks modelled using ARC/INFO. Two of the major functions are routing and allocation. *Routing* determines optimum paths for movement of resources through a network; *allocation* finds the nearest centre (minimum travel cost) for each link in a network to best serve the network. These two functions can be used on networks such as city streets, waterways and telephone lines for operations such as vehicle routing, optimum facilities siting and time-distance flow analysis. However, as already discussed previously in Section 3.8.2.1, this represents only one individual example of the very wide range of software tools available with ARC/INFO.

3.9 *Object Oriented Systems*

Again, in accordance with the developments in the DBMS field in general, a parallel effort to develop GIS/LIS systems using the object-oriented approach has also been taking place over the last few years. As will be seen, the object-oriented approach involves storing the functionality of the objects together with the objects themselves at the same level, although the objects inherit some of the characteristics of the classes to which they belong. At the same time, it is quite possible for an object-oriented database system to utilize some of the more traditional types of data structures at a lower level within the system. Thus, for example, some of the classes of attribute data may be organized in a hierarchical manner or in relations within the overall object-oriented approach.

3.9.1 SysScan *GINIS*

As already mentioned in Section 3.8.1, SysScan offers numerous products in the mapping information systems field. At the core of many of these systems resides the software package known as GINIS, an acronym formed from the title Graphic Interactive Information System [SysScan, 1986].

3.9.1.1 *GINIS- Overall System*

GINIS is a program system which is designed to carry out the structuring and manipulation of cartographic data. The functions and data structure of the GINIS system enable both graphic and non-graphic data to be handled efficiently within a mapping environment. All relevant map information, such as points, lines, curves, areas, symbols, text, objects, logical layers and associated non-graphic data can be input, maintained, processed and output using the GINIS system. Fig. 3.20 gives an overview of the GINIS system.

As will be seen, the central core of GINIS is the so-called Mapman module which gives direct access to the two databases - the one holding the graphic/cartographic data; the other, the non-graphic (attribute/text/numeric) data. Along the top of the diagram are a series of boxes representing the various input modules handling survey, cartographic and photogrammetric data as well as data in SIF (Standard Interchange Format) from other systems. All of these give data into Mapman. At bottom left of Fig. 3.20, connections to and from an external database management system (DBMS) such as DEC's relational database management system (RDB) are shown. In the lower central part of the diagram, the output driver programs to produce hard-copy plots on Kongsberg, Versatec and Calcomp (CC) plotters are shown. Finally, along the right hand side of the diagram, various GINIS options for manipulating cartographic data are shown.

From this description and the figure, it can be seen that the main orientation of GINIS is towards mapping applications either in the surveying and mapping industry or in the public utilities sector. There are a minimum of analysis tools. However, with the use of two databases - the one for graphic data and the other for non-graphic data - together with a database management system and links between them, GINIS would, by most counts, be considered a land or mapping information system rather than a simple digital mapping system.

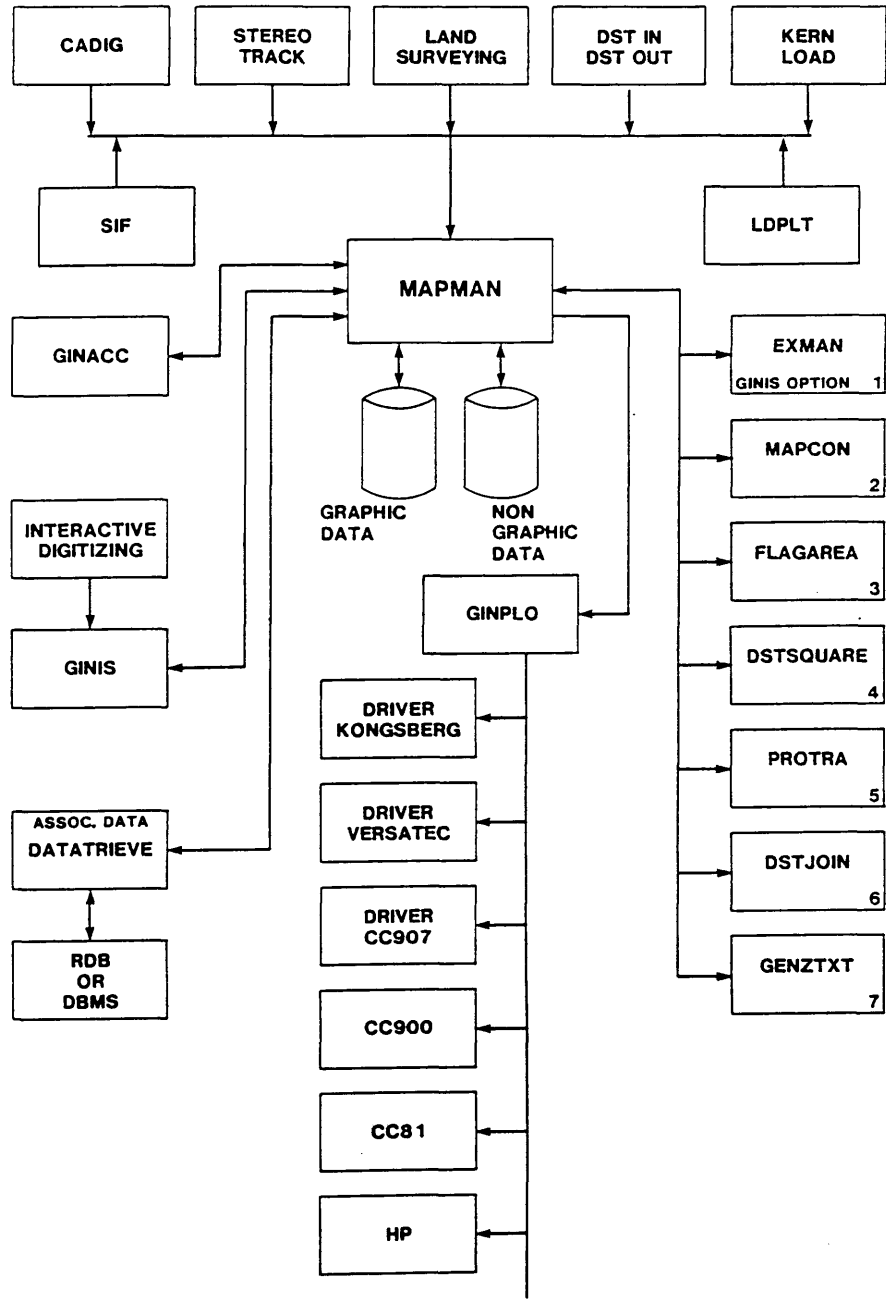


Fig. 3.20 GINIS system overview

3.9.1.2 System Architecture

As discussed above, GINIS provides a wide range of interactive and batch processing modules for entering, structuring, editing, processing, manipulating and retrieving both graphic and non-graphic data. Fig. 3.21 illustrates the system architecture and the links between the GINIS system and these databases and libraries.

3.9.1.3 GINIS Data Structure

The GINIS-Graphic Data Structure (GDS) is based on a hierarchical model as illustrated in Fig. 3.23, [Radwan, Kure & Al-Harthi, 1988]. However, the system can also be described as object oriented, whereby all objects and all information assigned to them (either as spatial or functional descriptors) are treated as individual units or objects.

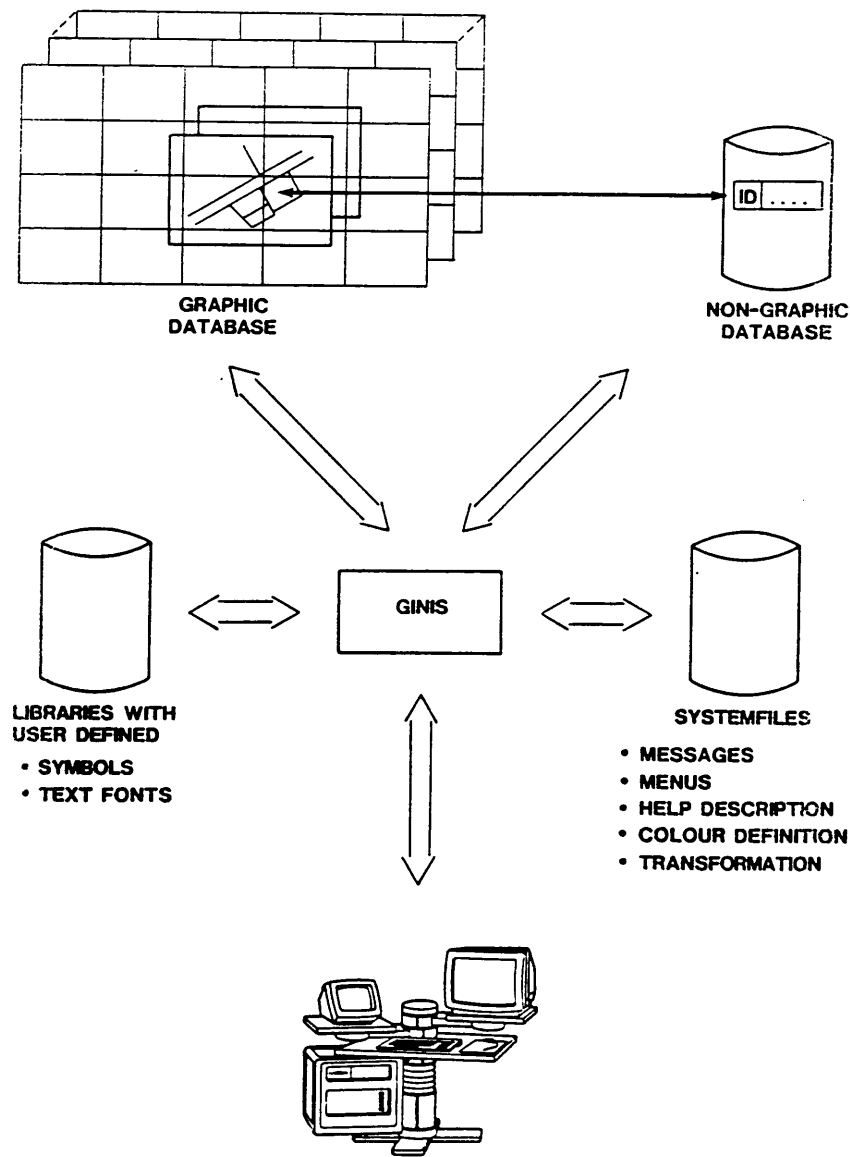


Fig. 3.21 GINIS system architecture

Fig. 3.22 and Fig. 3.23 below show the organization of the 'information structure'. Data items are organized into three different classes namely:

- a) topographic entities in a logical hierarchy with:-

- i- the first being the theme/class data set, represented by 'C';
- ii- second is the object data set, represented by 'O'; and finally
- iii- the graphical primitives (nodes/arcs) represented by 'G' and lying at the lowest level of the hierarchy;

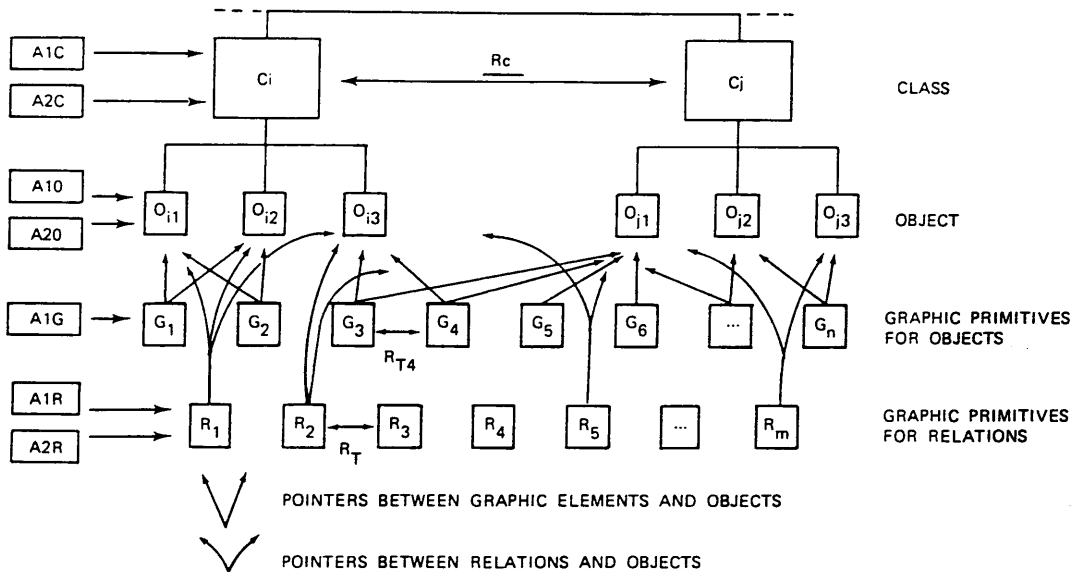


Fig. 3.22 The GDS data structure

- b) attributes also divided into two classes:-
 - i- spatial attributes represented by 'A1'; and
 - ii- non-spatial attributes represented by 'A2';
- c) relationships or associations, which are divided into four classes:-
 - i- topological;
 - ii- proximity;
 - iii- functional; and
 - iv- phenomenological.

These data elements are stored in separate units or blocks:-

- i) logical blocks for the data classes of type C;
- ii) geometric blocks for type G;
- iii) object blocks for type O;
- iv) non-graphical descriptors 'A2' assigned with associate-data blocks; and

v) finally the relationships are assigned with relation-blocks.

Fig. 3.23 shows the diagrammatic representation of the main two blocks, the Logical and Geometric blocks.

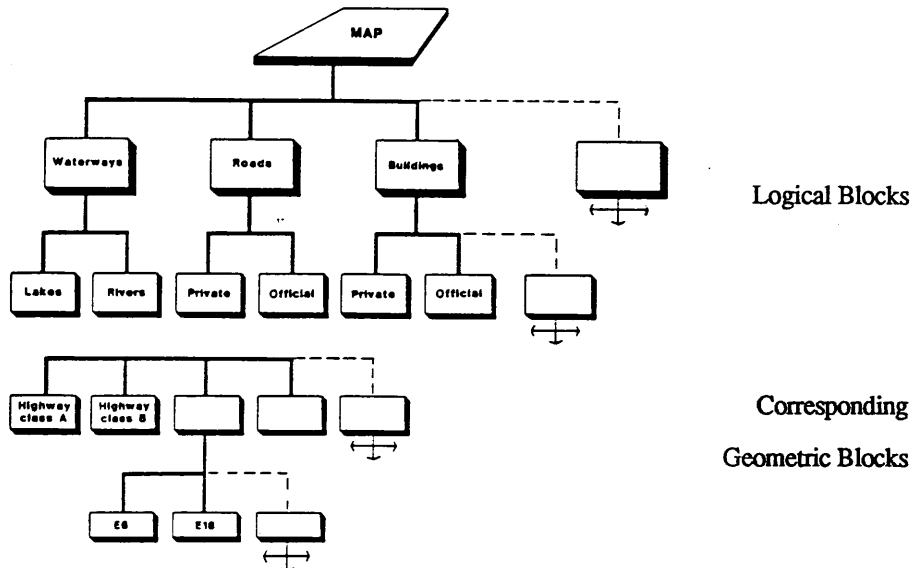


Fig. 3.23 Logical and Geometric Blocks

The upper part of the diagram shows the logical blocks for the classes represented by the boxes marked 'Ci' in Fig. 3.22. This includes a quite simple hierarchical classification of the topographic features contained in the map. The lower part of the diagram shows the corresponding geometric blocks which relate to the graphics primitives marked G1...Gn in Fig. 3.22. The logical blocks are linked to each other through up, down and side pointers, thus providing a logical meaning to the other data blocks to which they are connected within the network structure. This would allow not only objects, but also the spatial and functional descriptors, together with relationships to be hierarchically structured.

If data items, and the blocks describing them, are linked to more than one data block on a higher hierarchical level, then they have various logical significances although they have only been stored once. This provides the possibility of handling common portions of layered data sets in a non-redundant manner while the geometrical consistency supports cross-layer spatial analysis queries.

The above does not take care of queries related to non-topological relationships. This

problem was solved by having the geographic entities, their logical classes, their spatial and functional properties and their interrelationships presented in terms of objects.

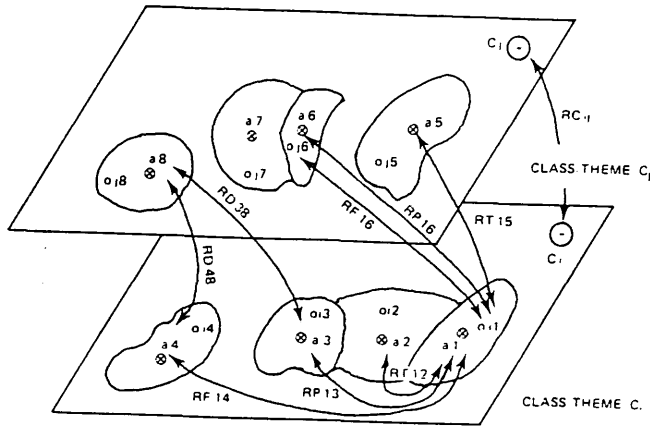


Fig. 3.24

This is illustrated in Fig. 3.24 above, where the points C_i and C_j represent the class vertices for the class data sets C_i and C_j respectively; the reference points a_1, a_2, \dots represent the object vertices for objects O_{i1}, O_{i2}, \dots of the types \underline{Q} or \underline{G} and the arcs (links) represent the various types of relationships established between these elements.

When two objects are linked by more than one relationship, e.g. the proximity and functional relationships RP_{i16} and RF_{i16} between the objects O_{i1}, O_{i6} , the links between the two vertices will be represented by a multi-graph [Bouille, 1978]. Further, some of the links will refer to 'virtual' objects that do not exist in reality. These are V-links, as opposed to the R-links that are associated with real objects.

These relation-graphic primitives can be structured in a similar way to the real graphic elements, i.e. further grouped, classified and hierarchically organized in a 'relation-hierarchy', comparable to the 'object-hierarchy' described earlier. These structural elements will be stored in the GINIS Graphic Data Structure (GDS) in blocks of the geometric block type mentioned before. These blocks (called relation-blocks) have links to associate data blocks which carry additional information of the A2R type for 'information carrier links'. These information items include a relation identifier, the access to objects participating in it, its rôle and its significance, etc. Furthermore, the various graphical and non-graphical manipulation functions of the GINIS system (access, extract, display, modify, delete, etc.) can be applied to these 'relationships' in a similar manner to the 'real' graphical elements.

In summary, it can be seen that the SysScan GINIS system is a powerful mapping information system with an emphasis on map production and the map-related requirements of public utility organizations. It appears to use an object-oriented approach, but with an underlying hierarchical structure implemented in terms of the classes and objects and also in terms of the logical blocks for data classes and the geometric blocks used for the graphical primitives.

3.9.2 Wild System 9

System 9 is a fairly new geographical information system which was originally developed by Wild Heerbrugg, the Swiss surveying and photogrammetric systems suppliers. Quite recently, the main rights to the system have been sold to the computer systems manufacturer and supplier, Prime, which has always been prominent in the fields of computer graphics and CAD/CAM, through its ownership of the U.K.-based Medusa CAD/CAM and modelling products and of the Computervision company in the USA, which is probably the biggest specialist systems^{supplier} in the computer graphics field after Intergraph.

System 9 organizes geographically-related information into a topologically structured, object-oriented, relational database system. With System 9, three-dimensional geographic databases can be built up from an existing base map sheet, photogrammetric data, and survey data. Also, it can generate a number of graphical products on either high-resolution monitors or plotters, to produce a variety of outputs such as utility maps, contour maps and profiles [Charlwood et al., 1987].

The system is available only on the family of Sun 3 graphics work stations running under the Unix operating system and utilizing the Sun window management station. A variety of work stations is available for digitizing (S9-D), editing (S9-E), for data capture using photogrammetric instruments (S9-AP), etc. Each work station can have its own project database but if several work stations are linked together by an Ethernet, the database can be distributed over many machines.

3.9.2.1 System 9 Components

An overall view of the system is provided by Fig. 3.25.

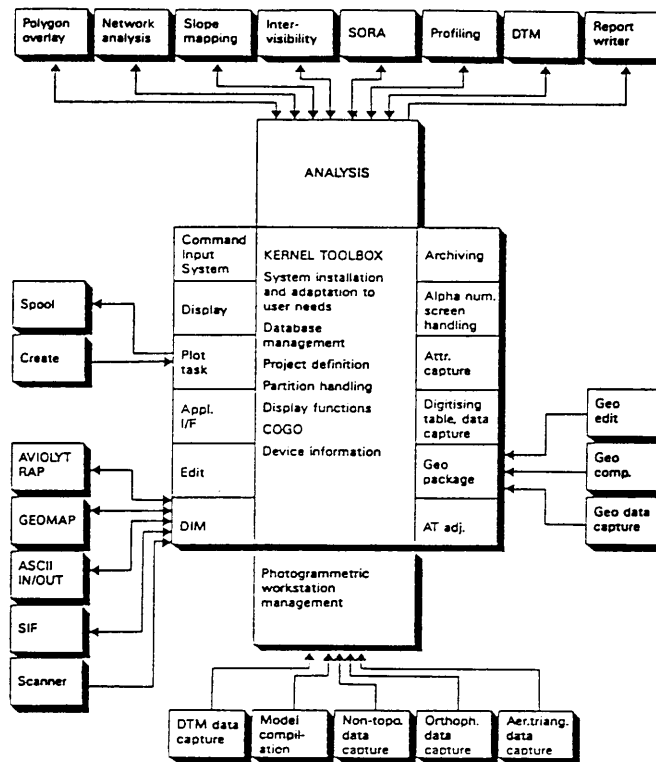


Fig. 3.25 System 9 overall view

Beside the Kernel toolbox which is central to all System 9 operations, there are numerous software modules grouped around it. The modules shown at the foot of this figure are related to the management of the input from field survey and photogrammetric data capture operations. The data interchange modules (DIM) shown in Fig. 3.25 are designed specifically to allow rapid transfer of files of data between existing digital mapping systems, such as Wild's existing RAP (photogrammetric-based), Geomap (field survey-based), and Wildmap systems and those CADD-based systems which utilize the SIF (Standard Interface Format) for data exchange. At the top of Fig. 3.25, a number of analysis programs are shown; these act as modules for standard tasks such as polygon overlay, network analysis, slope mapping, etc. Finally, at the right hand side of the figure, the Geo modules for the capture and editing of data held on existing maps are shown.

System 9 supports a full topological data model of polygons, links and nodes. The non-graphic elements of the model and all attribute data are stored using the EMPRESS relational database management system. Thus like ARC/INFO, the RDBMS used for the storage and handling of the attribute data is an existing commercial product. However, the software design differs from other GIS packages in being 'object oriented'. The software is

divided into modules not solely by operation but also by *object*, which comprises the data structure plus the procedures required to operate on it (Fig. 3.25).

Within System 9, several organizational groups of information are recognized: projects, partitions, feature classes, features and themes. Fig. 3.26 shows the hierarchical organization of data in System 9. A project is the highest level of data organization, representing the entire database for a geographical area. It comprises two components - the spatial/attribute database, and the database definition which specifies the project structure such as feature classes and themes.

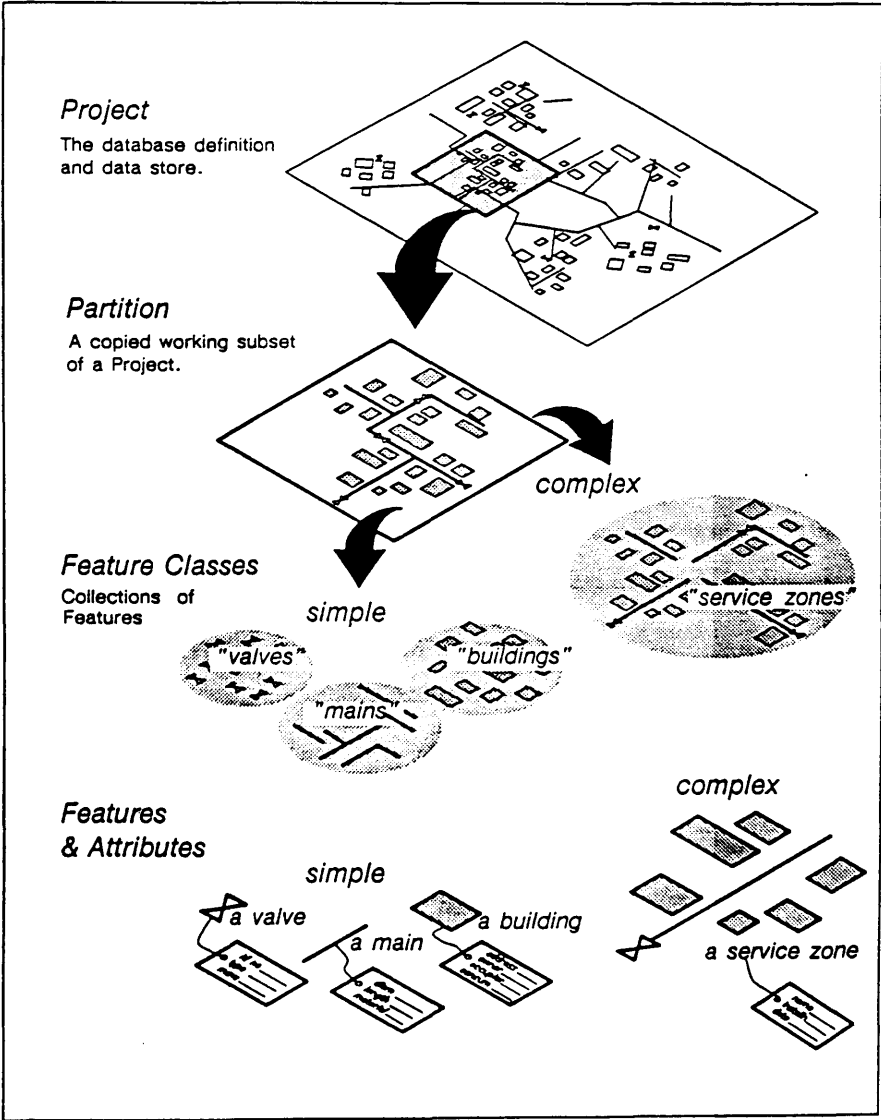


Fig. 3.26 The hierarchical organization of Data in System 9

The database is created and updated by means of partitions, working subsets of a project which are extracted according to the type of work to be done. When editing is complete, the

partition is merged back into the project database to effect the update. A partition definition describes spatial extent, contents and their representation. The merit of the partition structure is that it makes it safe for different sections within an organization to work on data from the same common project.

At the levels below partitions are the feature classes and features. Individual drains or sewers are considered as simple features. A logical group of simple features such as a drainage area is a complex feature. A complex feature can contain any number of other complex features. All complex features of a particular type comprise a complex feature class. Complex features can have attributes associated with them in the same way as simple features. At the level below the features are their attributes. The attributes of each of the complex features apply to all the simple features which it references, hence obviating duplication. Below the attributes are the geometrical primitives which comprise all the basic building blocks from which all the geometrical features can be constructed.

Finally the theme definition determines how features such as colour, symbols, line types, etc., are to be displayed. Since many theme definitions are permitted, any data set can be displayed differently to suit different users and requirements.

3.9.2.2 Data Base Management System

System 9 uses the EMPRESS database management system which structures information using relational database methods. At the highest level, the DBMS provides project databases - the master stores for all geographic data. Each project database can contain many partition databases which can be accessed and modified through an application. The database management system supports the structure of projects and partitions, as well as the simple query language through which operators can interrogate the databases.

3.9.2.3 Topology & Data Sharing

Nodes play a central rôle in System 9, since they are the only geometric primitives which have coordinate information directly associated with them. Lines are not defined in terms of geographic coordinates, but by pointers to their topological nodes. And surfaces are defined by pointers to the lines surrounding the surface. Note that these pointers are created

and maintained automatically by System 9. With this approach redundancy is cut to a minimum. Also editing and updating becomes an easier matter. As a result, information common to more than one element should not be duplicated in the database. Such replication of data can lead to huge problems of database maintenance, since it requires the consistent and simultaneous revision of many instances of the same element, when in reality only a single element is changed- e.g. a valve is shifted; a road is widened, etc.

The topological consistency of the System 9 database is maintained by automatically structuring the node-line-surface relationships at the time of data capture. System 9 incorporates extensive checking procedures to ensure that the correct topological relations are established from the start: for example, lines will be checked for continuity, and surfaces will be checked for closure. Entities which fail to satisfy such checks will be flagged as being in error. This procedure differentiates System 9 from many other systems in which topological checking is carried out as a separate operation following data input.

3.9.2.4 System 9 Data Base

As mentioned briefly in the introduction given above in Section 3.9.1.1, the GIS implemented by System 9 can be reached through geographic databases known as Projects and Partitions. What follows is a more detailed description and discussion of the Project and Partition databases and of the feature classes contained within them [Simon, 1987].

3.9.2.5 Projects

A Project database can be divided into two logical components; a project definition component and a data store. The project definition consists of entities known as feature classes, attributes, and themes which describe the characteristics and structure of the project. The data store component contains all the information relating to the size, type, position, and attribute values of all the geographic features.

3.9.2.6 Partitions

A partition database is the access mechanism to System 9 both for the data capture workstations and for the user to carry out work on a specific area. For a defined area, the feature classes, feature attributes, themes, and geographic data are all carried over into a

partition database by a checkout process which automatically extracts all the necessary information from the project database. Through the various data capture applications, the user is able to define and modify geographic data, and provide all the necessary attributes for every feature in the partition. Once data capture is complete, the partition is checked into the project; this function automatically merges partition data with existing project data while performing topological checking to ensure data integrity. Besides their rôle during data capture, partitions are employed extensively during analysis. The great merit of the partition structure is that it makes it safe for different sections to work on data from the same common project.

3.9.2.7 *Features & Feature Classes*

It will be seen that feature classes impose a structure on the geographic data through an association of features and their component primitives. Thus a project is defined through a list of complex feature classes, surfaces, lines, and nodes. Each feature class represents a category of features which may be entered in a project, and can have a series of attributes associated with it.

3.9.2.8 *Simple and Complex Features*

Groupings of related objects which may be established on the basis of location, spatial relationships or common attributes, in System 9 are identified as complex features. They are defined as features which contain other features. This distinguishes them from simple features - which then are not allowed to contain other features. All complex features of a particular type, comprise a complex feature class. Similarly, all simple features of a particular type, are said to comprise a simple feature class. However, the definition of a complex feature is not restricted to include only simple features as constituent components. It is possible as well to define complex features which comprise other complex features. Fig. 3.27 illustrates the composition of complex features and their classes and attributes in System 9.

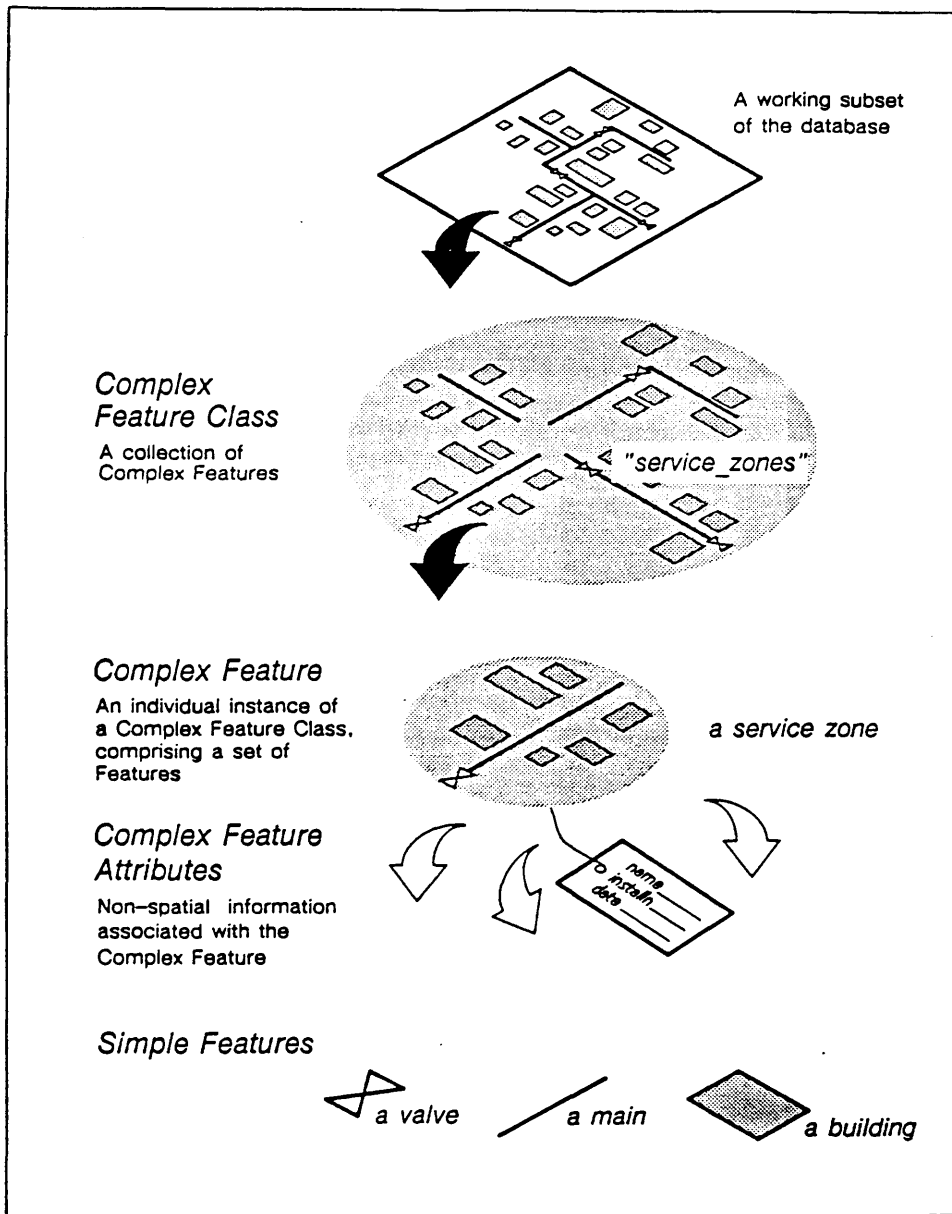


Fig. 3.27 The composition of complex features in System 9

3.9.2.9 Attributes

The attributes represent the desired characteristics of features (text or numeric data). Attribute values are stored, and may be displayed or output, along with their associated feature. Through attributes, a very detailed characterization of individual features is possible.

3.9.2.10 Geometric Primitives

All geometric features in the System 9 data structure are built up from geometric primitives, referred to as nodes, lines, surfaces and spaghetti. A node is stored as a set of X, Y, and optionally Z coordinates in a three-dimensional database. A line primitive is a geometric element defined by two end-nodes, allowing its true path to be described by one or more line segments which together form a closed polygon. The fourth geometric primitive, spaghetti, exists to enable System 9 to model features where no topological structure is required.

System 9 holds these geometric primitives as recognizable geographic phenomena or objects belonging to generic groups of objects which then can be identified and named in the real world. Fig. 3.28 illustrates the concept of features and feature classes in System 9. Categories such as "roads", "pipes" and so on are known as "feature classes", and the individual instances of geographic objects as "features". All features within a particular class will have the same topological structure, and the same set of attributes.

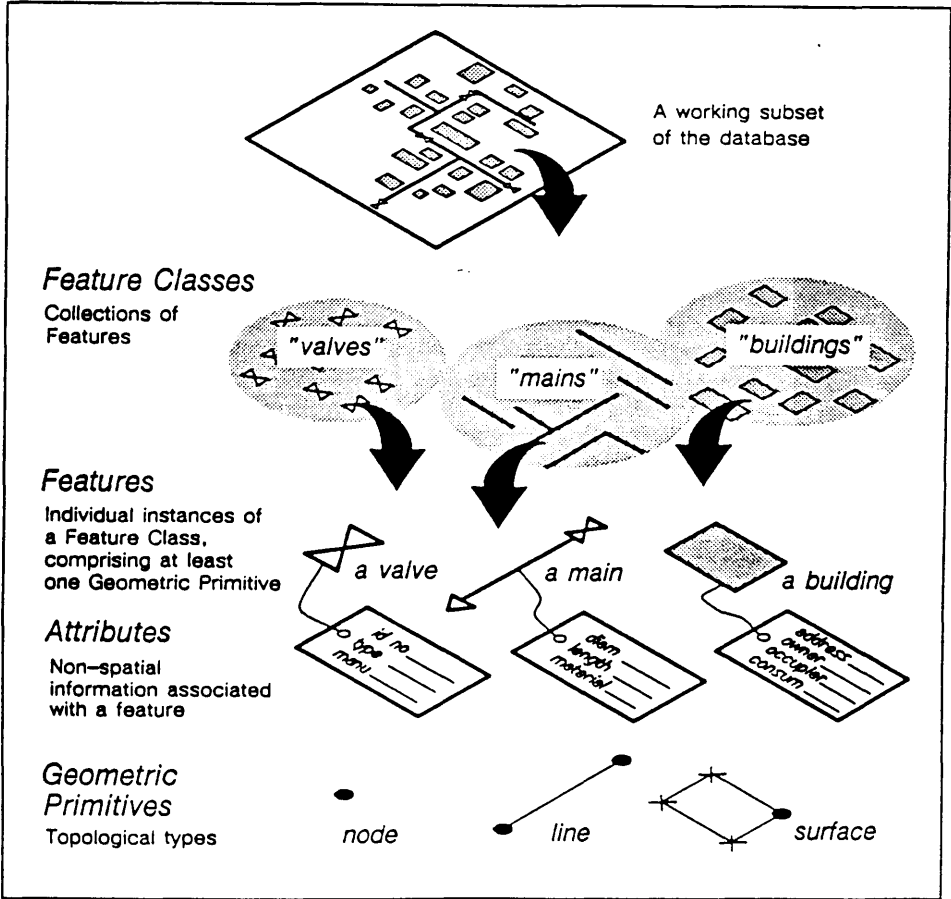


Fig. 3.28 The representation of features in System 9

3.9.2.11 Themes

The definition of a theme specifies which feature classes are to be presented and how they are to be represented. A theme definition determines the appearance of graphical reports, including complete specifications of colour, texture, line patterns, and fonts. The theme definitions give the user a high degree of control over the visual content of graphical reports and allows him to display the same data in different manners, by applying different themes to the same information.

3.9.2.12 Summary

In summary, it can be said that, in many ways, System 9 represents the current state-of-the-art in the GIS field with many interesting concepts, structures, and features, especially the manipulations offered by the object-oriented approach. Within this approach, it has a complex structure, in that it also utilizes a hierarchical organization of the database classes running through different levels from projects, partitions, features classes, features, attributes down to themes. At the same time, it utilizes a relational approach to the storage of both graphic and non-graphic data in the form of tables, a proprietary RDBMS (Empress) to manage these data and a query language with which the database can be interrogated. Its modernity is in fact reflected by the fact that it has only recently reached the market place and so it has, as yet, a very small penetration, although this is also the result of the recent sale of the major interest in the system to Prime and the consequent loss of momentum in marketing and sales.

3.10 Discussion

Table 3.4 presents a brief summary of basic information about some of the systems available in the North American market and formed part of a survey carried out in April 1989 by H. Parker (1989). It will be seen that quite a number of GIS/LIS products available in Europe, e.g. the SysScan products, ICL Planes system, etc., are missing from the Table. Also the well-known Canadian CARIS system from USL is not included in the Table. However, in view of its currency, it still contains many points of interest. Parker reported in this survey that most of the current GIS systems using external DBMSs were based on the relational approach. The reason for this is of course fairly obvious in that quite

a number of the customers for GIS/LIS systems have already implemented commonly available RDBMS systems such as RDB, Informix, Oracle, Ingres, and others which are built to manage simple non-graphic data in the form of text and numerical tables. Thus when such organizations decide to purchase a GIS or LIS, these organizations may wish to buy a product which utilizes their existing RDBMS as its base.

As already seen, this is the approach being taken by ESRI with its ARC+RDBMS product. Also a similar approach is being implemented by Laser-Scan in the U.K. with its newly introduced Metropolis (LIS) and Environmental GIS (EGIS) products which basically are developments of its existing digital mapping products (e.g. LITES) together with the SQL query language interfaced to an existing RDBMS. In addition, a number of analysis tools are used to carry out the required analytical functions which may be required by the customer. While some suppliers such as ESRI and Laser-Scan actually use the RDBMS such as INFO, Oracle, RDB, Ingres, etc. to handle the non-graphic elements of the GIS, others simply provide an interface which allows data transfer between the GIS and the RDBMS. While these are favourable aspects of this particular approach, taking into consideration the very special requirements of a geographic information system, it will be apparent that the use of these commercially based RDBMS systems also has some drawbacks, since these systems were simply not designed to handle databases with the complexity and volume of geographic data.

The survey of GIS software by H. Parker mentioned above covered over 62 systems reported to be truly GISs or related systems. Table 3.5 summarizes the percentages of systems that perform some of the more important analytical GIS functions. Each class of functions listed may have one or more subfunctions within it. Each subfunction usually translates to a specific system command.

| System Name | Computing Environment | System Type | First Installed | Number of Users | Data Structure(s) | DBMS Interfaces |
|------------------|---------------------------------|--------------------|-----------------|-----------------|----------------------------|-------------------------------|
| ARC/INFO | DEC, PRIME DG, IBM, etc. | GIS | 1981 | nr | Vector | Info, Oracle Ingres |
| Deltamap | HP9000, SUN APOLLO, etc. | GIS | 1986 | 100+ | Vector/Raster TIN | Oracle, Ingres Informix |
| ERDAS | PCs, SUN, VAX | GIS, IP | 1979 | 900+ | Raster | Info |
| GeoVision | VAX, ULTRIX, SUN, IBM-RT/AIX | GIS, FM | 1976 | 47 | Raster, Vector Quadtree | Oracle |
| - 'GeoPro' | PCs, Macintosh | AM | 1988 | 2 | Vector | SQL & DBF |
| - WOW | PCs | GIS | 1985 | 1200+ | Vector | nr |
| GFIS | IBM S/370 | GIS | 1977 | 180+ | Vector | IMS/DLI, SQL/DB2 |
| GDS | VAX, Ultrix | GIS, AM | 1980 | 800+ | Object(vec) | Oracle, etc. |
| IGDS/DMRS | VAX | CAD,CAE FM, GIS | 1973 | 1371 | Vector,Raster | Informix |
| Infocam | VAX | GIS | nr | 23 | Raster,Quad. | Oracle |
| Informap | VAX | GIS | 1975 | nr | Vector | SQL-based |
| Laser-Scan | VAX | GIS | 1985 | 150 | Vector, Raster | RDB |
| MicroStation GIS | Intergraph/Unix | GIS | 1989 | 11 | Vector, Raster | Oracle,Ingres Informix |
| PC ARC/INFO | PCs/PS-2/DOS | GIS | 1987 | nr | Vector | Info |
| SICAD | Siemens/Unix | GIS | 1978 | 250 | Raster, Vector | DB2, Quad. etc. |
| SPANS | PCs/DOS/OS2 | GIS | 1985 | 400 | Raster, Vector Quad. | nr |
| Strings | PCs/DOS | GIS/FM | 1979 | 150 | Vector | Ingres,Sybase, Britton Lee |
| System 9 | SUN/UNIX | GIS/IP | 1987 | 25 | Vector | Empress |
| TIGRIS | Intergraph | GIS | 1988 | 16 | Vector, Raster | na |

Source [Parker, 1989]

Table 3.4 A survey of GIS systems

Table 3.5 Percentage of systems capable of specific GIS analytical functions

| Class of Function | No. of Subfunctions | % of Systems* |
|-------------------------------|---------------------|---------------|
| Distance Measurement | 3 | 74 - 94% |
| Buffering | 5 | 78 - 90% |
| Map Algebra | 5 | 36 - 78% |
| Boolean Operations | 2 | 80 - 82% |
| COGO Computations | 1 | 40% |
| Network Tracing | 1 | 44% |
| Remote Sensing Image Analysis | 1 | 26% |
| Terrain Analysis | 8 | 26 - 60% |
| Polygon Operations | 6 | 18 - 82% |

* A range is shown in cases where the number of subfunctions within a function class exceeded one, because not all systems perform the same combination of subfunctions. Source: [Parker, 1989]

From this, it can be seen that there is a great diversity in the range of analysis functions which are provided in a GIS. This presumably stems from the fact that each system is aimed at a specific sector of the market and only attempts to meet the specific functional requirements of that sector. If this observation is correct, then it has very important consequences for the customers and potential users of a specific system.

It will also be seen that currently, the relational approach is favoured by most system suppliers because of its relative simplicity and suitability for the purposes of carrying out spatial data analysis. As D. J. Peuquet (1987) has mentioned, "Developments in this field were driven by a need for efficiency in a practical, implementational context. The rational basis behind the initial development of the relational database concept was to provide a unified and consistent model for structuring the data with minimal redundancy". Hence, at the present time, the relational database model has been developed and used extensively and has become the most common and most successful approach developed within the field of geographic information systems.

But several shortcomings which are inherent with this approach have been discovered in it. The most obvious one, is that the actual system implementations have often proved in practice to be too slow for databases of large sizes and complex data types. Another is the limitations inherent in such a system when it is used with models with regular ,

homogeneous structures.

3.11 *Conclusion*

This chapter has reviewed the current situation in the closely related GIS/LIS and AM/FM fields, taking particular examples from each of the main approaches to a DBMS to illustrate the practical implementation of each particular approach. Obviously there are a great variety of systems available but all have their advantages and disadvantages. Since the field is growing rapidly and the disadvantages and difficulties of existing systems are becoming apparent, this means that there is still plenty of opportunity for the development of fresh approaches and new ideas, which is a spur and great encouragement to undertake further research and development in this area. However, it is by no means easy to make advances since, as will have become apparent from the review conducted above, in fact, the requirements which must be satisfied by the DBMS are, by any standards, enormous. This results from the need to manage widely dispersed data, together with the wish to accommodate many highly complex data types. In particular, the requirements in these areas have increased dramatically over the last few years.

It is enough to know that, in implementing a GIS/LIS or AM/FM system, the programmer (or, in fact, the team of programmers) have to tackle several systems at one time, namely the data structure of the main data, their graphic attributes, their interlink and relations, and finally, the user views of the data. In addition to these, there is also the matter of the use and integration of query languages like SQL which are commonly used to access the databases. It would be far easier for the implementor of a geographic information system to tackle a single complex problem rather than three or more. However, a possible way of solving the difficulties is the use of a database language, in which the programmer does not need to worry about many of these problems, since they are taken care of by the features of the language itself. This allows the programmer to concentrate on the major issues which have to be implemented and solved by the GIS.

For this reason, the author has investigated the use of a database language, and in particular PS-algol, for the implementation of a GIS in the search for a better, and more relaxed design environment.

CHAPTER 4

CHAPTER 4: PS-ALGOL, THE LANGUAGE

4.1 *Introduction*

PS-algol is the result of considerable effort spent in the design and implementation of a database programming language. Although PS-algol is still an experimental language, it has already shown a very good ability to compete with well known languages in terms of what it offers programmers, both on the basis of its simplicity (in learning and in applying it) and its power (in handling data).

PS-algol has been developed jointly at the Universities of Glasgow and St. Andrews by the Persistent Programming Research Group (PPRG) headed by Prof. M. Atkinson in the Department of Computing Science at the University of Glasgow, and by Professor R. Morrison at the Department of Computational Science at the University of St. Andrews [Atkinson & Morrison, 1986 and Atkinson, Morrison and Pratten, 1986].

This language is still, as mentioned earlier, experimental and subject to development in various ways before one can judge its full capabilities. On the other hand, its use in this project can in no way be claimed to have exploited the full power of the facilities which are already available in the language.

What follows is an overview of the language, with a general look at the aims behind the introduction of a new language. This is followed by a description of its various facilities including some not usually found in other languages.

4.2 *Language Design and Aspects*

The aims of the designer of a programming language are to provide facilities which make it easier for programmers to write and maintain programs. Some properties aimed at by designers are :

- * the language should be sufficiently powerful to express the programmer's intentions;
- * the language should be easy to learn, remember and utilize;
- * the language should not introduce arbitrary distinctions between similar concepts;

- * the syntax of the language should make programs easy to read and understand; and
- * the language should give assistance in detecting errors as soon as possible after they occur.

To achieve these goals, the following essential principles should be adopted [Atkinson et al, 1984]:

- a- The principle of data type completeness;
 - b- The principle of abstraction; and
 - c- The principle of correspondence.
-
- a- The principle of data type completeness states that all data types must have the same "civil rights" and that the rules for using the data types must be complete, with no gaps. This does not mean that all operators in the language need to be defined on all data types but rather that general rules should have no exceptions. This principle is important in making the language powerful and in making it easy to learn, understand and remember. The avoidance of special cases leads to a simpler yet more powerful language. To illustrate this point, one can note for instance, that a Pascal programmer must remember that procedures may not be passed as parameters to other procedures. Thus Pascal is both weaker and more complex than a language for which this is not so.
 - b- Abstraction is the process of extracting the general structure to allow some details to be ignored. This principle is invoked by identifying the semantically meaningful syntactic categories and providing abstractions over them. For example, a given operation is represented by a particular series of instructions. If the series can be wrapped up into some kind of subprogram, subsequent usage of the operation can then refer to the sub-program and ignore the series of instructions contained within it. Languages which support the ability to abstract composite objects rather than be confined to using primitive constructs greatly shorten programs and make them more comprehensible.
 - c- The principle of correspondence states that the rules for introducing and using names should be the same everywhere in a program. Often this refers to the rules for declaring names in program blocks, and those for naming the parameters of procedures. But it may be applied equally to the rules for introducing the names of

fields in records, or any other names, such as the names of modules. A language with a consistent naming scheme will enhance program readability.

Thus the language should be made as compact as possible by removing arbitrary distinctions as long as it does not remove real distinctions in the process.

Some examples of arbitrary distinctions which only confuse the programmer are:

- different methods of manipulating long-term and short-term data values;
- restricting apparently general purpose operations to particular values of the data; and
- requiring the use of an entirely different language for some sections of the program.

[Cooper, 1989]

One example of aiding error detection is by giving the language a type system which is enforced at all times. All data values are categorized into types, such as integer; real; string or character, and their usage is restricted to those operations appropriate to their type. Failure to keep to these restrictions results in errors which are reported to the programmer to assist debugging.

These can either be reported when the program is compiled or when the data value is used at run-time. The former is called static type checking and results in the much earlier detection of error. The latter is called dynamic type checking and results in a much freer style of programming. Thus there is a tension between these two desirable properties. A mixture of the two may well be best [Morrison et al., 1985].

To organize a language design, it appears to be best to decide first which objects the language will operate on. This can be further divided into identifying the atomic objects, and then the constructor mechanisms which allow composite objects to be built from them. Next, one identifies the operations which may be applied to the objects, identifies the supported abstractions, introduces a store to hold the objects and finally packages these concepts in a simple syntax.

As with any design project, designing languages involves laborious work and many iteration. One important concern, in this case, is to improve the parsimony of concepts. If two concepts in the language look similar, the designer must investigate whether there is a more primitive concept that will serve both roles or whether the distinction between them is

of sufficient importance to be retained.

Thus, in general, programming language design tries to create languages which improve software production. The design of database programming languages has the more specific aim of creating languages which are appropriate for data-intensive operations and allow efficient software to be written with this in mind.

4.3 Applying the Language Design Principles to a Data Base Language

A number of characteristics or features of the design of a language which is aimed specifically at database applications will also be discussed below.

4.3.1 The Idea of Persistence

The term persistence is used to describe that specific property of a particular data value that determines how long it will be kept. It is an orthogonal property of data, in that, in principle, any data item may exist for an arbitrary length of time. For these purposes, existence is equated with being potentially accessible by a programmer, or by the user through some program operation. The data associated with different procedure activation already has variable persistence; since the innermost activation are most transient, while these used in the global scope of the program are those which are most persistent. The data held in files and databases have been stored there in order that they may have a longer persistence. The property of persistence can therefore be thought of as a continuous variable describing one aspect of that data.

Traditional programming languages have provided facilities for the manipulation of data whose lifetime does not extend beyond the activation of the program. On the other hand, if data is assumed to survive a program activation, then some file I/O or database management system interface is needed. This results in a view in which data can either be classed as short term data and would be manipulated by the programming language facilities or the data would be long term data, in which case, it would be manipulated by the file system or the database management system. Values of different persistence are usually treated differently. For example, they may be named differently (as in the case of files), or objects of different types may have different facilities (e.g. procedures cannot be stored in Pascal). Furthermore, the mapping between the two types of data is usually done in part by the file

system or the DBMS and in part by explicit user transaction code which has to be written and included in each program.

A consequence of this view of the data is the need for a considerable amount of program code concerned with transferring data to and from files or the DBMS. This leads to much space and time taken up by the code required to perform translations between the program's form of data and the form used for the long term storage medium. For example, graphs modelled in a programming language need to be explicitly flattened and rebuilt in order to write them out to or to read them back in from the file store.

Thus what is needed is to make the quality of persistence orthogonal to type and naming. A language which applies this rule is called a persistent language.

4.3.2 *The Data Objects & the Principle of Completeness*

First, the objects which are to be manipulated need to be identified. Until recently, the candidate objects have been: sets, rings, networks and relations. However, the newer data models also imply new candidate objects. But still the principle of Data Type Completeness (DTC) dictated that the rules governing their manipulation should be consistent. The examination of existing languages which follows will show that this principle has not been applied universally. Some data types have been allowed to have only persistent instances, others (those that already existed in a "parent" language) have been allowed only transient instances. It is intended to show that adherence to the principle of data type completeness should lead to better database programming languages.

The failure to adhere to this principle has a high cost. For example, large parts of programs are concerned with making the transition between the two data words (persistent and non-persistent), where the programmer has to organize a translation, which takes a considerable amount of code and CPU time. It can have an even higher cost; a typical programming task requires that the programmer understand a topic in the real world, and constructs a model of it in a program. Understanding the problem and modelling it in a program is hard enough. Adding the requirement that the data stored between different sessions has to be in a different form, and the programmer's intellectual difficulties are much increased. Not only that, the programmer also has to implement and manipulate the mapping from the stored model to the program model. Thus, instead of having to visualize only one mapping,

the programmer has to manage three, making his task three times more difficult [Atkinson et al., 1984].

Apart from incurring these large learning and program design costs, failure to adhere to the principle of completeness also has very severe maintenance costs. It is easy for different programmers to visualize different relationships between two data models. Consequently someone undertaking maintenance may easily misunderstand and introduce severe errors into the software, errors which may manifest themselves much later in the form of corrupt persistent data.

To some extent, the present interest in integrity constraints [Date, 1981(a)] and their centralization [Nijssen, 1980] is an attempt to treat the symptoms of this defect rather than the cause. Such an approach will never cover all conceptual errors, since it is not feasible to recognize all potential errors, and devise rules which prevent operations which may cause such errors without preventing legitimate operations. Even where this is possible, enforcing such a complete set of rules would not be a feasible engineering task.

A yet more fundamental objection to not adhering to the principle of data type completeness can be identified. When someone builds a database, it can be viewed as a model of the real world. Similarly, when a useful program runs, it can be thought of as manipulating a real world model. What is likely to be useful for one modelling activity is likely to be useful for the other. It would have been expected that proven modelling techniques in the programming languages (here called data type declarations and abstract data types) would be useful in the database. Similarly, improved concepts in modelling developed for databases would be useful if incorporated in programming languages. So the PPRG felt that there was a sound philosophical basis and good engineering reasons for trying to bring together, or at least reconcile, the developments in these two areas.

It is clearly wasteful if different code has to be written and maintained to achieve the same effects - in the one case on persistent data, and, in the other, on transient data. It is believed that the language should be so defined that a procedure contained in it may be written without knowing whether it will be supplied with persistent or transient data as the actual values of its parameters. This is called persistence independent programming and it is believed all satisfactory languages will need to support persistence independence.

4.3.3 The Conceptual Store

Many languages make the concept of a store in which to hold representations of the manipulated objects explicit. For example, Fortran introduces such a store in common blocks and as variables. The Algols have a store associated with the nested activation of scopes. Pascal and Algol68 have added another store, the HEAP, which, in the one case, is explicitly relinquished and, in the other, is conventionally recovered by garbage collection. (Actually this is not strictly a property of the language definition).

It is not essential that the language should have the notion of a store (and by implication that of representation). The applicative languages avoid the whole concept of a store. Notionally they have only values, the results of expressions. This approach seems inappropriate for database applications. It does not seem helpful to visualize recording the change to one person's salary as the creation of a copy of the whole payroll, making a new record for that person in passing. (It is important to note, however, that this was the model of persistent storage which applied when data processing depended on magnetic tapes). The practical implementation of an applicative language must detect and avoid the copying of massive structures. However, this is not simple. Both interpretations of the phrase data sharing are required; many users should be able to access and see changes made to a given item of data, updating it within constraints, and many data items should be able to refer to a single instance of a data item, so that updates to it are reflected in all those contexts.

What is sought for the provision of persistence is an adequate abstraction for the composite store of main memory and backing store devices (predominantly discs). One existing abstraction is that of virtual memory. This abstraction covers the properties of size and speed, but not that of longevity. *The introduction of mapped files* [Organick, 1972] *has extended this abstraction into that of persistence*, by linking it with a persistent naming scheme (the filing system). This extension is limited to a time scale where the program does not change to the extent of changing its data structure definition.

Databases may have other notions of store, since different people looking at its contents will see them differently, and will have different entitlements to operate on different parts of the store. Most programming languages have denied responsibility for the concurrency and protection of data, leaving it to the operating system which controls the data with less precision. But programming languages are now approaching this issue, both in the

integration of the program development environment and in the provision of tools for the assembly of modules into systems, as well as in the modular concept itself [Jones and Liskove, 1978].

Thus, the provision of persistence is regarded as orthogonal to all other properties of data. For this, the Principle of Independence has been defined as being: "The persistence of a data is independent of how the program manipulates that data object". Which means, all code should be written so that it will work with the same interpretation independent of the persistence of the data on which it operates. This reduces the number of conceptual mappings from three to one. Having this, an extension to the Principle of Data Type Completeness would be : " All data objects should be allowed the full range of persistence".

4.3.4 *Binding*

In traditional programming languages, operating systems and file systems have a number of binding mechanisms which are often not easy to comprehend or to use. A binding mechanism has four components by which it can be categorized, which are :

- i- what does the name bind to ?
- ii- when is the binding performed ?
- iii- what scoping is involved ?
- iv- when is the type checking performed ?

Each of these questions is discussed below.

- i- Names of variables bind to locations whose value may change without altering the binding. Constant names bind to values.
- ii- The binding of both variables and constants is usually performed when the location is created. That is, it is carried out at run time unless the location or value is manifest (a literal), in which case, binding may be performed at compile time. Manifest constants can be seen in Pascal (for example) and manifest locations are the variables in Fortran. In block structured languages, it is usual for variables to bind to locations created at run time. Constants whose values are created dynamically can be seen in S-algol.

- iii- Names may be scoped statically in their compile time environment or dynamically by the run time system.
- iv- Dynamic type checking occurs when the run time system executes code to ensure that the data is of the correct type. This typically occurs in "read" statements and in projections out of a union.

There are potentially 16 different methods of binding commonly in use in modern computer systems based on the four binding choices given before. The most static form of binding occurs where only manifest constants are allowed with static type checking. The most dynamic form allows variables with non- manifest values, dynamic scoping and dynamic type checking [Atkinson, Morrison and Pratten, 1986].

The term Flexible Incremental Binding (FIB) has been introduced to describe the mixture of bindings, which are expected to obey the Principle of Correspondence. In fact, most languages have more than one binding mechanism.

Dynamic binding does have costs. Programs which depend on it may contain errors which could have been detected in a static binding system. This is obviously not acceptable in some situations. Programs may also run more slowly because of the checks required at run time. Many programs require to have run time checking, even in languages with static binding. For example, array indices need to be checked. The lack of this kind of checking is a common defect in implementations of the Pascal and C languages.

On the other hand, dynamic binding does have one clear advantage, that is it gives data independence. With static systems, the data is bound to the program at compile time. Therefore any changes made to the data require the programs using that data to be recompiled. This is not the case with systems using dynamic binding.

4.3.5 *The Persistent Store & Store Interface*

Obviously the actual store provided for persistent data and the interface provided to access this store are matters of considerable importance in the present discussion on the desirable characteristics of a database programming language [Morrison et al., 1985].

4.3.5.1 *The Persistent Store*

PS-algol's persistent store consists entirely of legal PS-algol data objects. The store is partitioned into individual databases to allow concurrency control and protection when sharing persistent data. Each database has a root data structure. This is a complex object which contains pointers that allow access to the other data objects in the database. These data objects may themselves be complex and point to further objects. The interface to the persistent store need only provide a method of accessing the root data structure of a database since any object in the database may now be retrieved by following pointer chains. Since the pointer data type may point to any structure class and any data type can be a field of a structure, there is no restriction on the data types that can be held in a database.

4.3.5.2 *The Persistent Store Interface*

The interface to the PS-algol persistent store is implemented by two procedures. The first is:

```
let open.database = proc(string database.name,password,mode -> pnttr)
```

This procedure attempts to open the database with the name 'database.name' in the mode ("read" or "write") given by 'mode'. Passwords are associated with each database to provide some security when sharing databases. The result of this procedure is a pointer to the root data structure of the database or, if unsuccessful, a pointer to an "error.record". An "error.record" is a data structure containing information describing why the open command failed.

This is sufficient to provide access to any object in the persistent store. Automatic transfer of data from the long term persistent store is performed by the persistent object management system when the data is accessed. The access of the data in the persistent store is performed in exactly the same manner as in the main store, the object manager knowing the difference so as to leave the transfer transparent to the user.

It is often desirable to ensure that updates to persistent data occur in total or not at all. A mechanism that implements atomic transactions is therefore provided by the second procedure :

```
let commit = proc( -> pnttr )
```

When the first database is opened, a transaction is started. Ordinarily data objects are

copied from the persistent store when they are first used and changes to them are made locally. If any of these data objects have been changed, a "commit" command will copy them back into their databases. Any newly created objects reachable from these changed objects will also be copied into the persistent store. They have space allocated for them in the database of the object pointing at them. If data objects from databases that were not opened in write mode have been changed, a "commit" command will fail. This ensures that the persistent store is always in a consistent state.

If for any reason, a "commit" command should fail, then its effects will be removed before any other use is made of the databases being updating. In this way, PS-algol provides a secure transaction mechanism on its persistent store.

If the "commit" command fails, the pointer which is returned is a pointer to an "error.record" and nil otherwise. The error record contains information about why the "commit" failed so that the program can do something sensible which may be to try to execute the "commit" command again or to give the user an error message.

4.4 Language Syntax

The syntax of PS-algol is specified by a set of rules or productions. Each production specifies the manner in which a particular syntactic category (e.g. a clause) can be formed. The syntax of PS-algol can be described in about sixty productions which can be referred to in the "PS-algol Reference Manual" [PPRR 12, 1987] and in "An Introduction to PS-algol Programming" [PPRR 31, 1986].

4.4.1 Identifiers and Object Declarations

In PS-algol, an identifier may be given to a data object, a procedure parameter, a structure field and a structure class. Data objects are declared by a 'let' statement.

Before an identifier can be used in PS-algol, it must be declared. The action of declaring an identifier associates it with a location of a certain type which can hold values that the identifier may take. In PS-algol, the programmer may specify whether the value is constant or variable. A constant may be manipulated in exactly the same manner as a variable except

that it may not be updated. When introducing an identifier, the programmer must indicate the identifier, the type of the data object, whether it is variable or constant and its initial value. For example, a variable can be declared as follows:

let $a := 5$

where a is an integer with the initial value assigned to it being 5.

On the other hand, constants are declared as follows:

let $b = 10$

in this case, b is an integer of constant value equal to 10. Constant values cannot be updated but can be manipulated in exactly the same manner as variables.

The difference in the declaration procedure of variables and constants is clear, since the first employs the operator "==" and the other "=" respectively.

4.4.2 Compound Data Objects

PS-algol allows the programmer to group together data objects into larger compound objects. There are three such object types in PS-algol: Vectors, Structures and Images. Images are collections of pixels. Vectors, structures and images have full 'civil rights' the same as any other data object in PS-algol.

All compound data objects in PS-algol have pointer semantics. That is, when a compound data object is created, a pointer to the locations that make up the object is also created. The object is always referred to by the pointer which may be passed around and tested for equality. The location containing the pointer and the constituent parts of the compound data objects may be independently constant or variable.

To comply with the principle of data type completeness, all objects, be they integers, strings, graphical objects, procedures, vectors, and structures, are all declared with the statement 'let'.

4.4.2.1 Vectors

A vector provides a method of grouping together objects of the same type. Since PS-algol does not allow undefined values, all the initial values of the elements of the vector must be

defined beforehand. These elements could be of any type recognized by PS-algol. For example:

```
let this.vector := @1 of int [1,2,3,4]
```

This statement introduces the variable 'this.vector' as a vector of four integers (***int**) with values 1,2,3 and 4 respectively. This method is used to initialize vectors of different values. Another method is used when initial values are of no importance since they are to be changed anyway, for example:

```
let x := vector 1::10 of 1
```

This declares a variable vector 'x' of ten integers with initial values equal to '1'.

Vectors can be of any type, not only vectors of strings and integers, but also vectors of procedures (principle of data type completeness). There can also be vectors of vectors, thus permitting arbitrary multi-dimensional arrays to be declared. For example:

```
let p = vector 1::n,1::m of 0.0
```

This means that 'p' is a two-dimensional array of 'n' elements, and each element is a vector of reals with 'm' elements.

Furthermore, since the size of a vector is not part of its type, multi-dimensional vectors which are not necessarily rectangular can also be created. For example, the Pascal triangle, shown below,

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
```

can be represented in PS-algol as :

```
let triangular.array = @1 of *cint[ @1 of cint[1],
                                   @1 of cint[1,1],
                                   @1 of cint[1,2,1],
                                   @1 of cint[1,3,3,1],
                                   @1 of cint[1,4,6,4,1] ]
```

the type ***cint** used in the example shows that each element in the vector **triangular.array** is itself a vector of elements of type **cint**.

Furthermore, there are two functions (**lwb** & **upb**) attributed for use with vectors to interrogate for the bounds of a certain vector. For example, if the vector 'x' is declared as follows:

let x = @1 of [1,5,7,-1,8]

then the interrogation of the upper bound of vector 'x' will be, **upb(x)** results in '8', and the lower bound, **lwb(x)** results in '1'.

4.4.2.2 Structures

Objects of different types can be grouped together into a structure. To ensure strong typing in the language, the structure class and the fields have to be given names (which have to be unique in the block). The fields may be constant or variable. Before a structure can be created, the nature of its structure class must be declared. For example:

structure building(cint postal.number; cstring street.name; cbool abandoned)

declares a structure class 'building', with three fields of type "cint", "cstring", and "cbool" respectively. The order in the structure is significant. The following expression will be of type **pntr** (pointer).

building(17, "Lilybank Gardens", false)

Two binary operators, (**is** & **isnt**), are attributed with structures. These operators are used to check whether or not a pointer is of a certain class. If the pointer is of the same class, **is** gives the result **true** and **isnt** gives **false**. For example:

K is building

Two structure classes, which may be in separate programs, are considered the same if they have the same class name, field names and types in a one-to-one correspondence.

4.4.2.3 Images

An image is a rectangular grid of pixels. Images may be created and manipulated using the raster operations provided in the language. Images are first class data objects and may be assigned, passed as parameters or returned as results. The raster operations are performed by considering the image as bitmaps and altering each bit in the destination image according to the source bit and the operation. The **limit** operation allows the user to set up windows in images.

Image expressions are discussed in more detail in sub-section 4.5.

4.4.2.4 Assignment and Equality of Pointers

Equality is defined on all data objects in PS-algol. In the case of compound objects; equality means the equality of the pointer. That is, for two vectors or structures to be equal, they must be the same incarnation of the same object. It follows that assignment of a compound data object means copying the pointer only.

4.4.3 Procedures

Procedures in PS-algol constitute the abstractions over expressions, if they return a value, and clauses of type "void" if they do not. In accordance with the principle of correspondence, any method of introducing a name in a declaration has an equivalent form as a parameter.

Thus, in declarations of data objects, giving a name an initial value is equivalent to assigning the actual parameter value to the formal parameter. Since this is the only type of declaration for data objects in the language, it is also the only parameter passing mode and is commonly known as 'call by value'.

Like declarations, the formal parameters representing data objects must have a name, a type and an indication of whether they are variable or constant. A procedure which returns a value must also specify its return type.

Structure classes and associated fields may also be passed as parameters to complete the principle of correspondence. For type checking, the argument and result types of the procedures and the field types of structure classes must be given in full when they are passed as parameters. Whereas the constancy of a formal parameter of a procedure (which is itself a formal parameter) is immaterial, it is not so with structure fields, and to avoid the possibility of altering a constant, the constancy attribute of the fields must be the same in this case. There must be a one-to-one correspondence between the actual and formal parameters and their type.

In PS-algol, all identifiers must be declared before they can be used and an identifier comes into scope immediately after its declaration. This is awkward when recursive procedure definitions are involved. Thus, the following procedure declaration :

```
let fac := proc(int n -> int); if n=0 then 1 else n*fac(n-1)
```

is not a recursive definition and is only legal if 'fac' has already been declared in an outer scope. To get recursion, the following is being used;

```
let fac := proc(int n -> int); nullproc
fac := proc(int n -> int); if n=0 then 1 else n*fac(n-1)
```

4.4.4 Data Persistence

As has been discussed before, data persistence is the length of time that the data exists. In PS-algol, any data item is allowed the full range of persistence. It is necessary for the programmer, however, to identify which data is to persist and the specific database in which it should persist. This section describes the mechanisms for the storage and retrieval of persistent data.

These mechanisms are available via a set of standard procedures which are described later in the section. The actual transfer of data is automatic, data being brought into the program's active heap when the program attempts to access it. Furthermore, the data may still be accessible, being migrated back at times which are left to the discretion of the implementor or on the instance of the programmer.

The procedures are divided into two groups, - (i) the group concerned with identifying the relationship between data and databases, and the implementation of transactions, and (ii) the group concerned with providing a new data structure or tables.

4.4.4.1 Tables

Tables are a system supported data structure in PS-algol. They are commonly used and are needed for building databases (in fact, the root object of a database is a table), but they may also be used for transient structures. A table stores an updatable mapping from keys to values. The keys may be integers or strings, and the values are pointers to instances of any structures. The implementor will probably have used B-trees or some adaptive hashing technique such as hashed trees to implement these maps. For example, the expression:

```
let the.map = table()
```

creates a table and assigns 'the.map' as a pointer to it.

There are two procedures (**s.enter** and **i.enter**) used to modify the entries in a table given as the parameters table. A table may contain entries whose keys are integers or entries whose keys are strings, where a key of one type never matches a key of the other. A new association is recorded in the table between the key and the value. This supersedes any previous association for that key which was held in the table. If the value is **nil**, the effect is to remove any existing entry for the given key from the table. For example:

i.enter(1, this.database, my.table) for integer keys, and

s.enter("the.key", other.database, your.table) for string keys.

There are another two procedures (**i.lookup** and **s.lookup**) to return the value associated with the given key from the given table. If there is no entry for that key, the result is **nil**. For example:

let *a.table* = **i.lookup**(1, this.database) for integer keys, and

let *a.table* = **s.lookup**('the.key', other.database) for string keys.

4.4.4.2 Data Base Procedures

All data which persists longer than a program execution is held in some database, any given item being in only one database. Pointers may refer to items in other databases. A database is identified by a database name which will often have the same syntactic form as that used for identifiers. A database may be created by:

let create.database = **proc**(string database.name, password -> **pntr**)

'password' being a string which must be quoted correctly to gain access.

A check is made that no other existing database has the same name as the one to be created. If this is successful, a database and a pointer are created and opened for writing. a table is inserted in the database and a pointer to it returned . If the 'create' is unsuccessful, then an error-record will be returned. A database may be opened by :

let open.database = **proc**(string database.name, password,mode -> **pntr**)

As before, 'password' is a string which must be quoted to gain access in the mode requested while 'mode' must have one of the values "read" or "write". A database may be opened by many users for reading or by one user for writing.

A check is made that the database is compatible with any others which are open, whether the user has quoted the password correctly and whether other programs which are using this database are in a mode incompatible with that requested. Any other databases that may

be referenced by objects in that database are recursively opened in read mode so that every object encountered will refer to an object in an open database. If any of these checks or the recursive open operation fails, then an exception will be raised, otherwise the result is a table.

Whatever the mode in which the database was opened, the programmer may now access data in that database, and change the data so accessed. However, no changes are recorded unless and until the program executes a call for 'commit'

let *commit* = proc(-> pnttr)

This procedure commits the changes made so far to all the databases opened by the program. Either all or none of the changes will be recorded, so the programmer can use this as a device for ensuring that the databases are consistent. It is only after a 'commit' command that the changes made to the data can be observed by any other programs. Note that only the changes prior to the last 'commit' of a program, if any, are recorded in the databases. It is an error to perform 'commit' when any database containing changed objects is not open for writing. In this case, an error-record will be returned. If the 'commit' is successful, the result is the **nil** pointer.

4.4.4.3 Data Base Conventions

The value returned by the commands 'open.database' or 'create.database' is a pointer to a table, conventionally called the root table. Data will be preserved for as long as it is reachable from some entry in a root table via the transitive closure of all references.

4.5 Graphics in PS-algol

The PS-algol graphics facilities provide an integrated method of manipulating both line drawings and images. Vector-based line drawings have the type "picture" and raster-based bitmaps, the type "image" [Morrison et al 1986].

The picture drawing facilities of PS-algol are a particular implementation of the Outline system which in turn took many ideas from GPL/1 [Smith, 1971]. These allow the user to produce line drawings in an infinite two-dimensional real space. The relationships between different parts of a picture are defined by mathematical transformations, therefore "pictures" are usually built up from a number of sub- pictures. "Pictures" may be mapped on to an

image thus providing flexibility in the way that line drawings may be manipulated.

An image is a rectangular grid of pixels of some 'intensity' or 'colour'. Images may be manipulated by raster operations provided by the language. These correspond to those generally available on systems with bitmap displays.

4.5.1 Pictures

In PS-algol, the picture description is represented by the 'picture' data type. The simplest picture is a point. For example :

```
let point = [ 0.1, 2.0 ]
```

represents the point with x-coordinate 0.1 and y-coordinate 2.0 in two-dimensional space. All the operations provided on pictures return a picture as their result, so arbitrarily complex pictures may be described and operated on [Morrison et al., 1985].

Points in pictures are implicitly ordered. The binary operators on pictures operate between the last point of the first picture and the first point of the second picture. In the resulting picture, the first point is the first point of the first picture and the last point is the last point of the second picture.

There are two binary operators on pictures, namely join '^' and combine '&'. The effect of the 'join' operator is to produce a picture that is made up of its two operands with a line from the last point of the first operand to the first point of the second operand. 'Combine' operates in a similar way without adding the joining line. For example:

```
let PIC1 = [1,2] ^ [3,4]
```

This will create a picture *PIC1* of a line joining the two points (1,2) and (3,4), Fig. 4.1.



Fig. 4.1 The picture *PIC1*

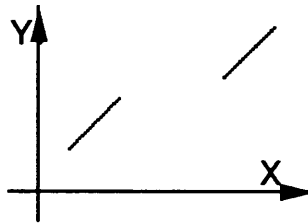
If *PIC2* is another picture for a line, as shown in Fig. 4.2, then

```
let PIC2 = [5,4] ^ [6,5]
```

Fig. 4.2 The picture *PIC2*

Going on from these individual lines represented by *PIC1* and *PIC2*, then using combine '&' will make a new picture *PIC* with both lines being components of the picture shown in Fig. 4.3.

`let PIC = PIC1 & PIC2`

Fig. 4.3 The picture *PIC*

In addition to the binary operators, pictures may also be transformed by shifting, rotating and scaling. For example, a shift (or translation) is implemented as follows:

`shift p by x.shift,y.shift`

will produce a new picture by adding *x.shift* to every x-coordinate and *y.shift* to every y-coordinate in the picture *p*. Furthermore a rotation is implemented by the statement

`rotate p by no.of.degrees`

which will produce a new picture by rotating the picture *p* clockwise about the origin by the specified *no.of.degrees*. Finally scaling can be carried out using the statement

`scale p by x.scaling,y.scaling`

which will produce a new picture by multiplying the x and y-coordinates of every point in the picture *p* by *x.scaling* and *y.scaling* respectively.

Text can be included in pictures using the text statement. This takes a string of characters and a base line and constructs the picture of those characters along the base line.

`let p = text "hello !" from 1.1 to 2,1`

The characters will always be drawn from the first to last point of the base line. As a

consequence, text can be inverted by ending the base line on the left of its starting position.

In order to view a picture, it is necessary to map it onto an output device. The PS-algol standard function 'draw' is provided to map pictures onto images. For example, the statement

draw(an.image,a.pic,0,3,0,3)

will draw the section of the picture *a.pic* bounded by the box specified by the points (0.0, 0.0) and (3.0, 3.0) on the image *an.image*.

4.5.2 Pixels

The pixel is a basic data type in PS-algol and is used to construct images [Morrison et al., 1985]. Two pixel literals are predefined in the language. These are **on** and **off**. These are said to have depth one, since they are only one pixel plane deep. Thus :

let a = on

creates a pixel *a* with a depth of 1. Pixels may be concatenated in order to create pixels with a depth greater than one, for example

let b = on & off & off & on

which creates a pixel *b* with a depth of 4. The expression on the right hand side of the above declaration is called a pixel sequence or simply a pixel.

4.5.3 Images

Pixels are used to construct images. Images are initialized with a pixel expression and have the same depth as that expression. Images also have x and y dimensions, for example

let c = image 5 by 10 of on

creates an image *c* with 5 pixels in the x direction and 10 in the y direction, all initially on. The origin of all images is in the lower left corner, which has the address 0,0. In this case, the depth is 1. Full 3-dimensional images may also be created, for example

let d = image 64 by 32 of on & off & on & on

creates an image which has depth 4.

In order to introduce the concept of images and the operations carried out on them, the discussion will be restricted to images with a depth of 1. Images are first class data objects and may be assigned, passed as parameters or returned as results. For example

let b := a

will assign the image *a* to the new one *b*. In order to map the operations usual on bitmapped screens, the assignment does not make a new copy of *a* but merely copies the pointer to it. Thus the image acts like a vector or pointer on assignment.

PS-algol supports eight imaging operations. These are : **ror**, **rand**, **xor**, **copy**, **nand**, **nor**, **not**, and **xnor**. Thus, for example,

xor *b* onto *a*

performs a raster operation of *b* onto *a* using xor. Notice that *a* is altered 'in situ'. Both images have origin (0,0) and automatic clipping is performed at the extremities of the destination image.

It is often desirable to set up windows in images. The PS-algol **limit** operation allows this.

Thus the expression **let *c* = limit *a* to 1 by 5 at 3,2**

sets *c* to be that part of *a* which starts at 3,2 and has size of 1 by 5. *c* has an origin of 0,0 in itself and is therefore a window on *a*. Rastering sections of images onto sections of other images can be performed by expressions such as :

xor limit *a* to 1 by 4 at 6,5 onto

limit *a* to 3 by 4 at 9,10

Automatic clipping of the edge of the limited region is performed. If the starting point of the limited region is omitted, then 0,0 is used for the purpose, and if the size of the region is omitted, then it is taken as the maximum possible; that is from the starting point to the edges of the host image. Limited regions of limited regions may also be defined.

The standard identifier *screen* is bound to an image representing the output *screen*. Performing a raster operation onto the image *screen* alters what may be seen by the user, e.g.

xor *a* onto limit *screen* to 4 by 5 at 4,7

will write the raster image *a* onto the defined section of the screen. This will be visible to the user. The standard identifier *cursor* is bound to an image representing the cursor. This allows the cursor to be manipulated in the same manner as any other image in the system.

4.6 *Management of Data in PS-algol*

The main point in PS-algol is that it is of little concern to the programmer how the data is actually stored. The use of the '**commit**' command causes all data reachable from a database root to be stored. Commit is implemented so that the transaction is guaranteed to

be atomic, that is either all the specified updating to the data will be completed successfully or no change will occur.

Data in PS-algol is retrieved only as it is used. If an object is retrieved with complex components, these components will only be retrieved if they are already used. For example, during data retrieval, if the user selected to retrieve a combination of data so that data retrieved should be of type 'areas' AND of layer 'building', then although, the search will be carried out over all area types and building layers, only items satisfying both conditions will be retrieved.

Furthermore, the backing store needs to be purged of all 'garbage' of old now unreachable objects when it gets filled up.

4.7 Conclusion

By adhering to the principles of data type completeness, abstraction and correspondence, together with furnishing the language with storage facilities, an immediate outcome is that the PS-algol language has greatly facilitated programmers' tasks in many aspects. The persistent store has simplified data storage immensely and first-class procedures provided a method for making functional abstractions. Also, the ability to manipulate procedures in the same way as other objects within the language is a consequence of the principle of the data type completeness. The persistent store is used to maintain a procedures library systematically, thus developing a method for modular programming in a large application.

The graphics facilities of PS-algol not only allow graphical data to be stored in a database with the same ease as text and numbers, but also provide a set of operations with which user interfaces can be constructed.

The pointer type of the language permits the construction of object structures of arbitrary complexity, which makes PS-algol a flexible tool for modelling objects.

Based on these facilities, the language seems appealing to develop systems to handle large amounts of complex geographical data.

CHAPTER 5

1. The first part of the chapter is devoted to the study of the

properties of the

of the function

the function is continuous on the interval

the function is differentiable on the interval

the function is concave up on the interval

the function is concave down on the interval

the function has a local maximum at

the function has a local minimum at

the function is increasing on the interval

the function is decreasing on the interval

the function has an inflection point at

the function has a vertical asymptote at

the function has a horizontal asymptote at

CHAPTER 5: SYSTEM ARCHITECTURE

5.1 *Introduction*

A GIS may be seen as the end-product of the process of creating a model of geo-related information from the real world. The topography, the natural resources and environmental conditions existing in an area and also its administrative and technical infra-structure, may be described by the model in a formal way that can be implemented by database designers. The resulting database feeds application programs which provide ways for decision makers to understand the relationships among geo-related phenomena. However, there is a great diversity of data views that must be made available to the various kinds of user. This makes it necessary to break down the model of the real world into primitive units for long-term storage. These can then be handled easily and checked for consistency and serve as building blocks for short-term applications.

5.2 *General Considerations in Data Structuring*

There are three levels for topographic modelling which have to be considered when data is introduced to the computer. The first level is the real world model. The second level consists of the internal mapping and structuring of the data within the system which manages the data (i.e. the data model for the database management system). Finally there is the physical file structure which exists in a particular computer environment.

The initial modelling phase requires a conceptual understanding of the various processes involved in utilizing the data, since redundancy has to be avoided between different user perceptions of real world phenomena when identifying and structuring the data elements. This structuring includes aspects such as the arrangement of elements; analysis of linkages between elements; analysis of the aggregation of elements into various classes; deciding how attribute information can be assigned to the elements; and finally deciding how functional associations (non-graphical relationships) can be established between elements.

The use of a topological model for the conceptual modelling of topography is now very common. A particular advantage is that in describing topographic features by sets of primitive topological elements (points, lines, and polygons) and storing these without any

redundancy, the spatial relationships of connectivity and adjacency between elements are expressed explicitly and are included as data items in the data structure.

In principle, the topological relationships, be they horizontal (on a layer level) or vertical (cross referencing between layers), are probably a minimum requirement for the expansion of the simple model. The inclusion of some additional relationships will depend upon a full analysis of the user requirements in a particular context or application, and as a consequence of this, may lead to the inclusion of more relationships in the data structure. This database management system will become more complex and the computer overheads (storage and response time) will increase, particularly in the case of interactive user queries. It should be noted here that, in some systems, the provision of a DBMS is not a strict requirement since some designers are able to establish their own database management systems which allow them to map the conceptual model and its data structure directly into the physical file level.

Data models provide the abstraction, the perception, and the description of the real world topography in a computer environment. The fidelity achieved in representing this abstraction will reflect the success of this representation. So this abstraction of reality should only incorporate those elements considered to be relevant for the user applications, and should utilize the best possible way of presenting these elements for use in the analysis to follow.

However, using PS-algol, the GIS system designer will only need to concentrate on the modelling between the real world and the internal data structure. This modelling should take into consideration the relationship between the different types and elements of the data. Since PS-algol is a database language, the GIS designer is already offered a "DBMS" which takes care of most of the DBMS functions as stated by Codd [1982] and mentioned earlier in Section 2.5.1 . Thus the programmer can concentrate on the high level functions of his own application.

5.3 Data Structure for the Project

In this thesis, the author has tried to create a data model which represents the real world as closely as possible. So, at the top of the hierarchy of the system is the directory where all information about the data, such as country name, package used for digitization, maximum

and minimum coordinate values and map scale, are kept. Next comes the data structure used to describe the model which in fact is composed of two main data structures, one hierarchical and the other relational. Fig. 5.1 illustrates the directory and the two data structures.

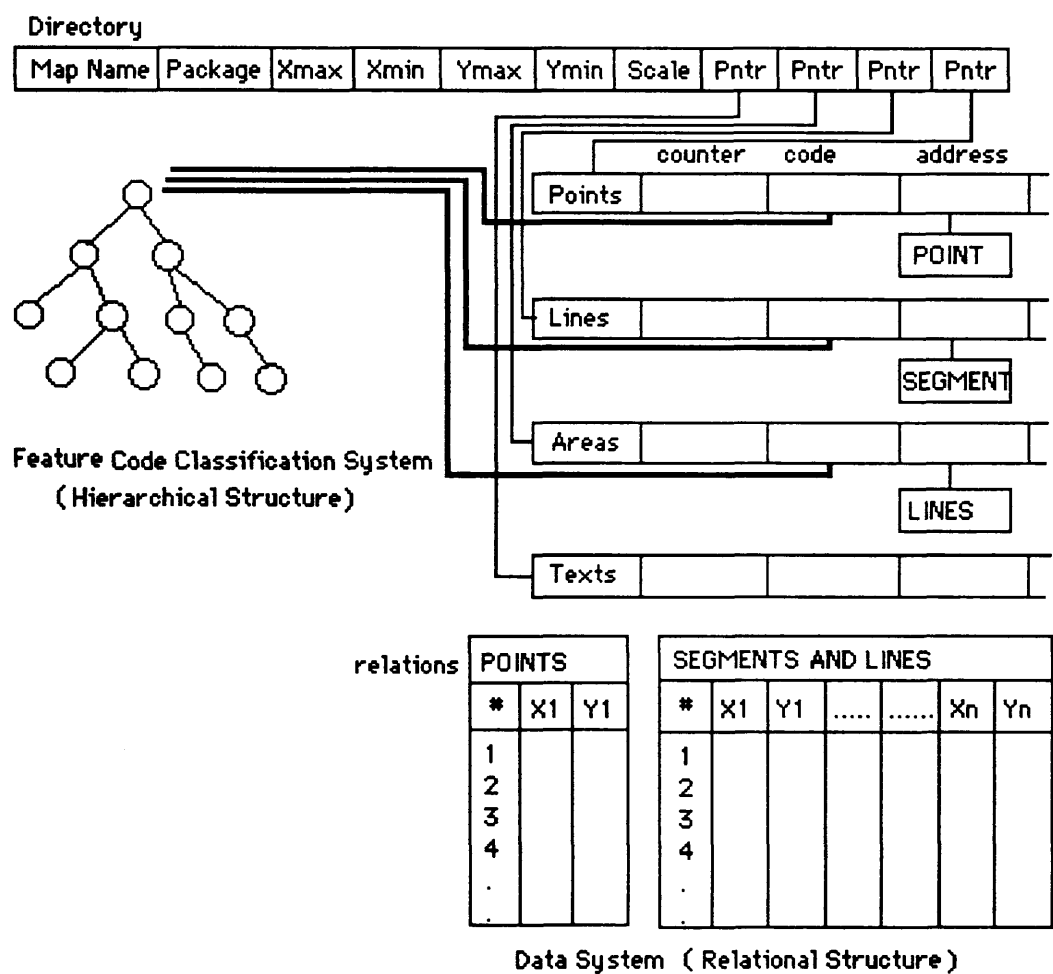


Fig. 5.1 Suggested data structure

5.3.1 Hierarchical Data Structure Used for Feature Coding

As has been discussed previously in Section 2.4.1, hierarchical data structures can be extremely simple and efficient if the limitations imposed in accessing and updating data are at their minimum. Furthermore, in hierarchical data structures, pointers are kept in a predefined manner so that they do not represent any complexity in the building of the structure.

A given application can be represented well in such a way if the data are only accessible

through well known and defined keys, and if the data are not updatable, in the sense that very few updates are required. In this case, the obvious choice to represent these data is with a hierarchical data structure.

Within the rules stated above, the coding system associated with the feature classification proposed for this project is thus qualified to be a candidate for using such a data structure, because:

- all the data attributes should only be accessed through their key identifiers (their code numbers);
- the need to update the feature classification system is very unlikely to be frequent, and, if this does happen, this will not result in any complications to the prototype system because of the stand-alone characteristic of the classification system; and
- data redundancy is nil because of non-dependency of the data in the different layers of the system.

5.3.1.1 Feature Coding System

To avoid lengthy records by including all the attributes of all feature classes, an explicit coding system for classifying features has been established. The organization of this classification is that every feature belongs to an overall class and to a specific category within that same class. Then the feature itself may consist of a set of attributes. This concept is illustrated by the diagram in Fig. 5.2, in which the classification is seen to resemble a tree structure. For this reason, a separate hierarchical database was established to contain the feature classification. It is then linked to the main relational data structure via a key in each tuple. A full description and listing of the classification coding system can be found in Appendix A.

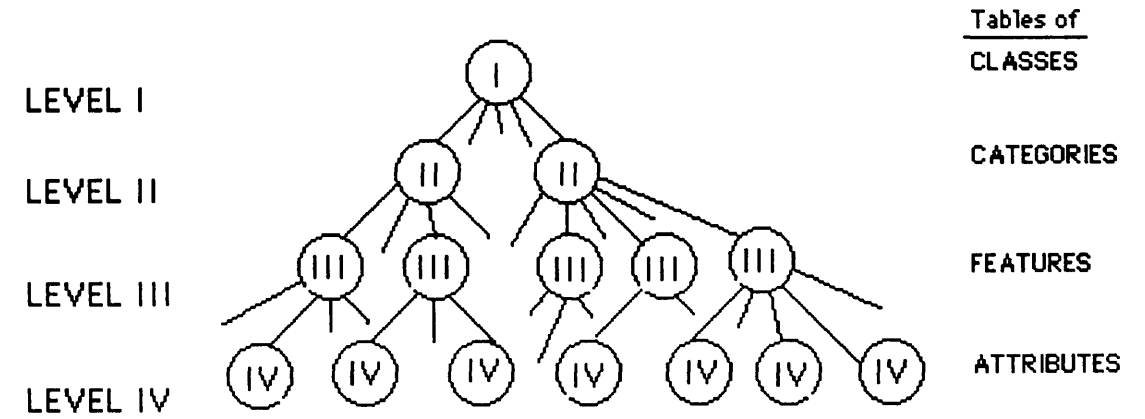


Fig. 5.2 Diagrammatic representation of the feature coding system

The next obvious step would be to provide this data structure and to store it in the database for later use. It will be seen that use is made of a hierarchical data structure for the following reasons :

- * the nature of the classification system;
- * the pre-determined pathways through the classification;
- * the static-like status of the classification system except in a few cases where the classification system has to be updated.

Each element at all levels of the classification has been assigned within a table identified by a name and containing an identifier, a code and a pointer to the sub-level table which follows in the classification hierarchy. The idea is illustrated in Fig. 5.3. It will be noted however that Level IV does not contain any pointer since there is no further level to point to.

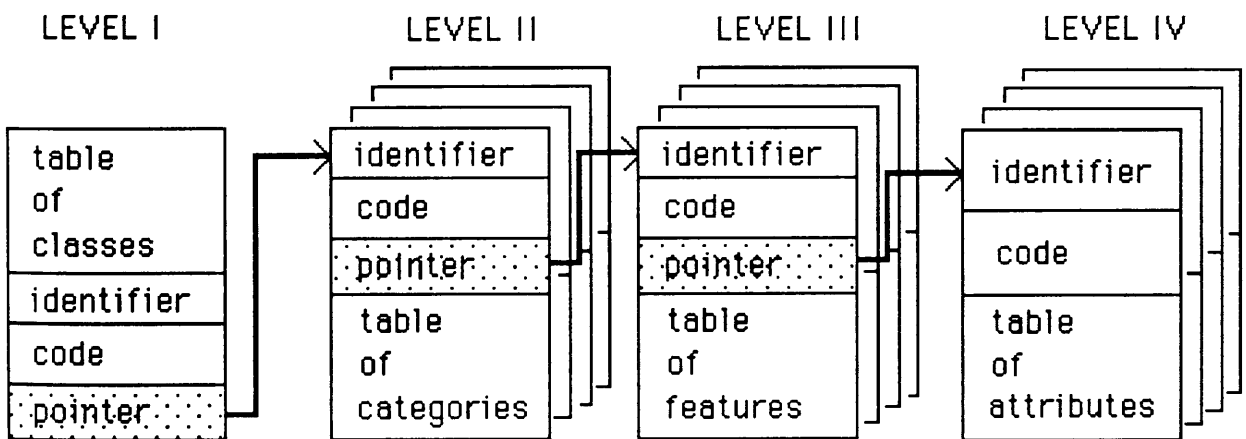


Fig. 5.3 The hierarchy of the feature coding system

Codes are associated with specific features through a coding procedure, which allows the user to choose codes by scanning through the database using a hierarchy of menus. From these he can then select the most convenient class, category, feature, and attribute to suit the phenomenon or subject in question. The next stage is to assign this code within the main data structure which will contain the rest of the details about each feature. The discussion of this will occur later on in this chapter.

The reason behind employing a separate configuration for the data classification and then linking it to the main data structure is the benefit gained in reducing the number of fields

required in the relational data structure. It also avoids data redundancy, and since the hierarchical data structure used is "static", the retrieval of codes and their attributes is quite fast.

5.3.2 Relational Data Structure Used for Data Entities

The following advantages can be drawn from using a relational data structure :

- * avoidance of any hierarchy within the structure;
- * avoidance of use of pointers which place a large overhead on the system;
- * simplicity in building new tuples for new items;
- * the ability to access any field within any existing tuple;
- * the strength of applying boolean logic for data retrieval;
- * flexibility in issuing queries.

[Deuker, 1985 and Van Roessel & Fornight, 1985]

On the other hand, the disadvantages that one can experience from the use of a relational data structure include the need for sequential searches of the different tuples, and the existence of redundant data to serve as keys instead of the use of pointers. However, the searching operations can be improved upon in environmental data structures by employing segmentation of the data, by using indexes, and building different relations to hold different types of data.

As discussed earlier, topographic phenomena as represented in map form can be reduced to the three basic or primitive elements (entities) of points, lines, and areas. According to this taxonomy, relations holding information about the elements of one type should be held separately. This does not imply in any way that elements should not be linked, or have no cross-referencing with other types, since these can be implemented through key fields to indexes which play the role of a mediator between the three types. In fact, there is a fourth type, which is a special relation which holds text data, required for attributes, names, etc.

As has been discussed in Chapter 4, pictures and images can be included and assigned in the same way as any other data object. Because of this, picture fields were assigned in all relations (points, lines, and polygons), so that, whenever there is a requirement to retrieve any feature or group of features, there is no need to issue the command to re-draw or to call

up other storage descriptions to retrieve the pictures. With this facility being an integral feature of PS-algol, pictures are stored alongside the rest of the data, and can be retrieved and manipulated like any other field in the record.

Another facility, which is supported by PS-algol and which has been used in constructing the data structures of the different entities, is the possibility of assigning vectors to fields. For example, after having found the neighbours of a particular polygon, these can be stored in a vector held in one field of a record under the data structure for polygons.

However, the translation of these structures into physical files is carried out automatically by the PS-algol language. Obviously this has great advantages in allowing the work undertaken in the present project to be concentrated on developing the application software rather being occupied with the physical allocation of the different data in different files.

As has been mentioned before, there are four types of data entities (points, lines, areas and text) with which all map features will be associated and to which attributes can be attached. Since there are different attributes for each type, both in terms of their numbers and their types, four different data structures have been devised to suit these different entities.

5.3.2.1 Data Structure for Point Entities

For point entities, the record contains the following fields shown in Fig. 5.4:

| counter | code | nodes | process | dimension | name | comment | shape | link |
|---------|---------|----------|---------|-----------|--------|---------|---------|---------|
| integer | integer | integers | boolean | real | string | string | picture | pointer |

Fig. 5.4 'point' data structure

These comprise the following:-

- * counter (of type integer);
- * identification code (of type integer);
- * vector of Keys to Nodes (of type integers) - these will be the value of a code of another entity;
- * signal for processing (of type boolean), whether or not it has been enhanced;
- * a dimension (of type real) for instance depth, radius or some other size not represented

on the map;

- * name (of type string);
- * comment (of type string);
- * picture (of type picture);
- * a pointer to a nil table for future expansion of the system (of type pntr).

5.3.2.2 Data Structure for Line Entities

The records for line entities given in Fig. 5.5, are different and thus :

| | | | | | | |
|---------|---------|----------|---------|--------|---------|------------|
| counter | code | segment | process | name | comment | dimension1 |
| integer | integer | integers | boolean | string | string | real |

| | | | |
|------------|-----------|-----------|---------|
| dimension2 | shape (O) | shape (E) | link |
| real | picture | picture | pointer |

Fig. 5.5 'line' data structure

- * counter (of type integer);
- * identification code (of type integer);
- * vector of Keys to Segments Table (of type integers);
- * signal for processing (of type boolean);
- * name (of type string);
- * comment (of type string);
- * first dimension (of type real);
- * second dimension (of type real);
- (the two dimensions might correspond to length and width for instance)
- * shape of original data (of type pic);
- * enhanced cartographic shape (of type pic);
- * a pointer to a nil table for future expansion of the system (of type pntr).

5.3.2.3 Data Structure for Area Entities

Area entities are more explicitly described in their structures than the previous entities and

they contain the following fields, shown in Fig. 5.6 :

| | | | | | | | | |
|---------|---------|----------|---------|--------|---------|----------|----------|--|
| counter | code | segment | process | name | comment | centroid | postcode | |
| integer | integer | integers | boolean | string | string | reals | string | |

| | | | | | | |
|----------|------------|------------|------------|-----------|----------|---------|
| islands | neighbours | dimension1 | dimension2 | shape (O) | shape(E) | link |
| integers | integers | real | real | picture | picture | pointer |

Fig. 5.6 'polygon' data structure

- * counter (of type integer);
- * identification code (of type integer);
- * vector of Keys to Segments Table (of type integers);
- * signal for processing (of type boolean);
- * name (of type string);
- * comment (of type string);
- * centroid X coordinate (of type real);
- * centroid Y coordinate (of type real);
- * shape of the original data (of type pic);
- * area of the polygon (of type real);
- * length of the perimeter of the polygon (of type real);
- * post code (of type string);
- * number of islands (of type integer);
- * vector of islands (of type integer);
- * number of neighbours (of type integer);
- * vector of neighbours (of type integer);
- * enhanced cartographic shape (of type pic);
- * a pointer to a nil table for future expansion of the system (of type pntr).

5.3.2.4 Data Structure for Text Entities

Finally text entities have the following attributes which are given in Fig. 5.7 :

| | | | | |
|---------|----------|---------------|--------|--------|
| code | nodes | justification | text | font |
| integer | integers | integer | string | string |

Fig. 5.7 'text' data structure

- * a code to identify to which layer the text belongs (of type integer);
- * a code to identify text justification (of type integer);
- * X coordinate (of type real);
- * Y coordinate (of type real);
- * the text (of type string);
- * the font to be used (of type string).

5.4 Overall System Configuration

The advantages drawn from the use of pre-determined pathways in the hierarchical data structure devised for the feature codes or attributes coupled with those drawn from the use of a relational data structure to deal with the entities will, when combined, boost the system response in both directions, i.e. in data retrieval and in query processing. This type of combination is known as a hybrid, in which the aim is to exploit the benefits of both approaches.

In constructing this system, different databases were built in order to facilitate dividing the main program into modules and to speed up the processing of these different modules. The displays of the different menus for the different operations are also stored in one of these databases.

The overall system is divided into four main modules, the first being termed Data Entry; the second, Cartographic Representation; the third, Data Retrieval; and the fourth, Data Output. These modules are saved as compiled procedures in a database called "%\$Modules" and they are passed to the main program through a "structure". The detailed discussion of all these modules is given in Chapters 7 through to 11.

Fig. 5.8 illustrates the layout and the interaction of the different modules and databases used in the system, together with the PS-algol DBMS.

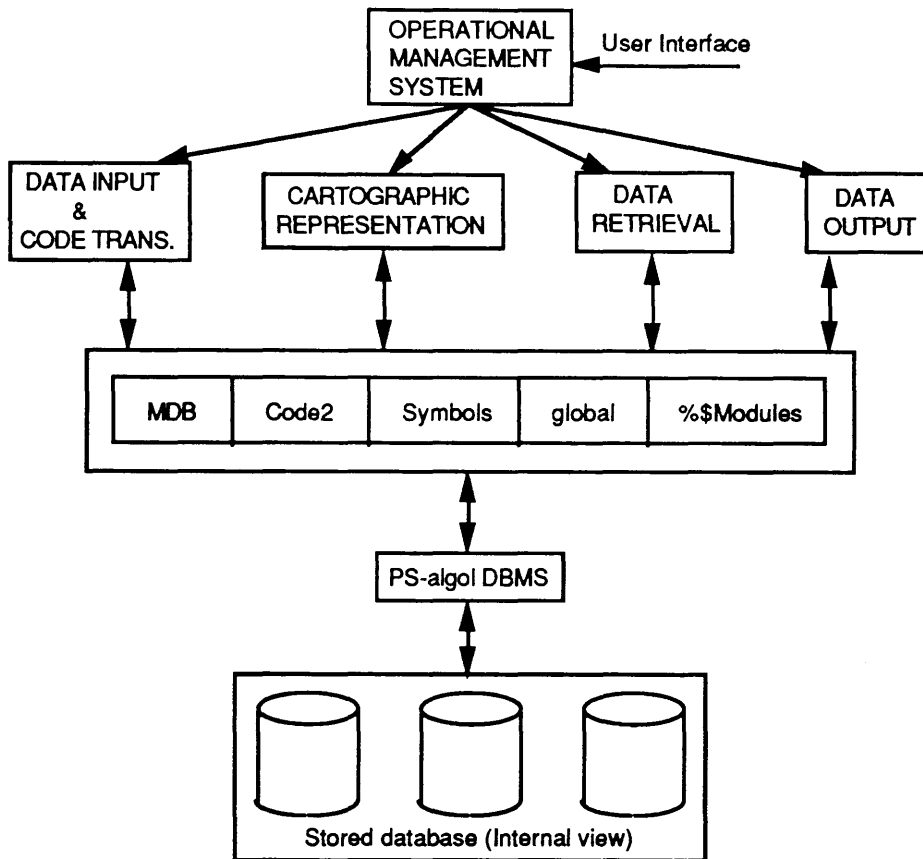


Fig. 5.8 System architecture

5.5 Individual Data Bases Within The Prototype GIS

(i) The main database is the one that holds the different data structures and is called "*MDB*", *MDB* has at its highest rank a table called 'Maps' that contains the names of the different maps. Each map has a pointer associated with it. This points to another table that contains the main details about a particular map, such as the name of the map; the package used in digitizing (or the source of the data); the map serial number; the scale; the direction of north; the grid size; the minimum and the maximum coordinate values and finally pointers to the data structures of the four sets of entities.

(ii) There is another database holding the different symbols to be used for cartographic display, called "*Symbols*". These are held in "picture" type..

(iii) Furthermore, yet another database called "*global*", holds almost all the menus to be displayed during the operation of the program. These menus are saved as images in a table called "*Images*" within "*global*". Furthermore, "*global*" keeps track of the numbers of

items entered into the system using four counters.

(iv) Finally, the database that holds the feature coding system is called "*Code2*" (as mentioned above), linked to each item in the different data structures via keys. A more detailed description of the programs that create these databases is given below.

5.5.1 *The Main Data Base (MDB)*

This database, MDB, is created at the start of the main program (see Appendix H). The program first checks whether the database already exists or not. If the database does exist, then the program opens it; otherwise the program creates it.

The hierarchy of the database contents is then declared. The structure 'maps' which will hold the details of the different maps is declared first. The maps entered into the system are stored in a directory table called 'Maps'. This directory structure points to four sub-structures, which contain the different types of features. The structures needed to hold the different types of data (points, lines, areas and text) are then declared and named as: P-holder; L-holder; A-holder and T-holder respectively. This is illustrated by the representation in Fig. 5.9.

5.5.2 *'Symbols' Data Base*

As the name implies, the '*Symbols*' database is used to store the different shapes of symbols which are used in area filling and in displaying point features.

Symbols stored in this database are of two types; Areal and Point. Areal symbols, in this particular case taken from Ordnance Survey maps, include eleven symbols to represent different types of vegetation, e.g. bracken, conifer, coppice, deciduous, heath, marsh, orchard, reeds, rough-grass, saltings and scrub. It would be equally easy to implement a family of areal symbols dealing with different types of building, e.g. public building, industrial building, residential building, etc.

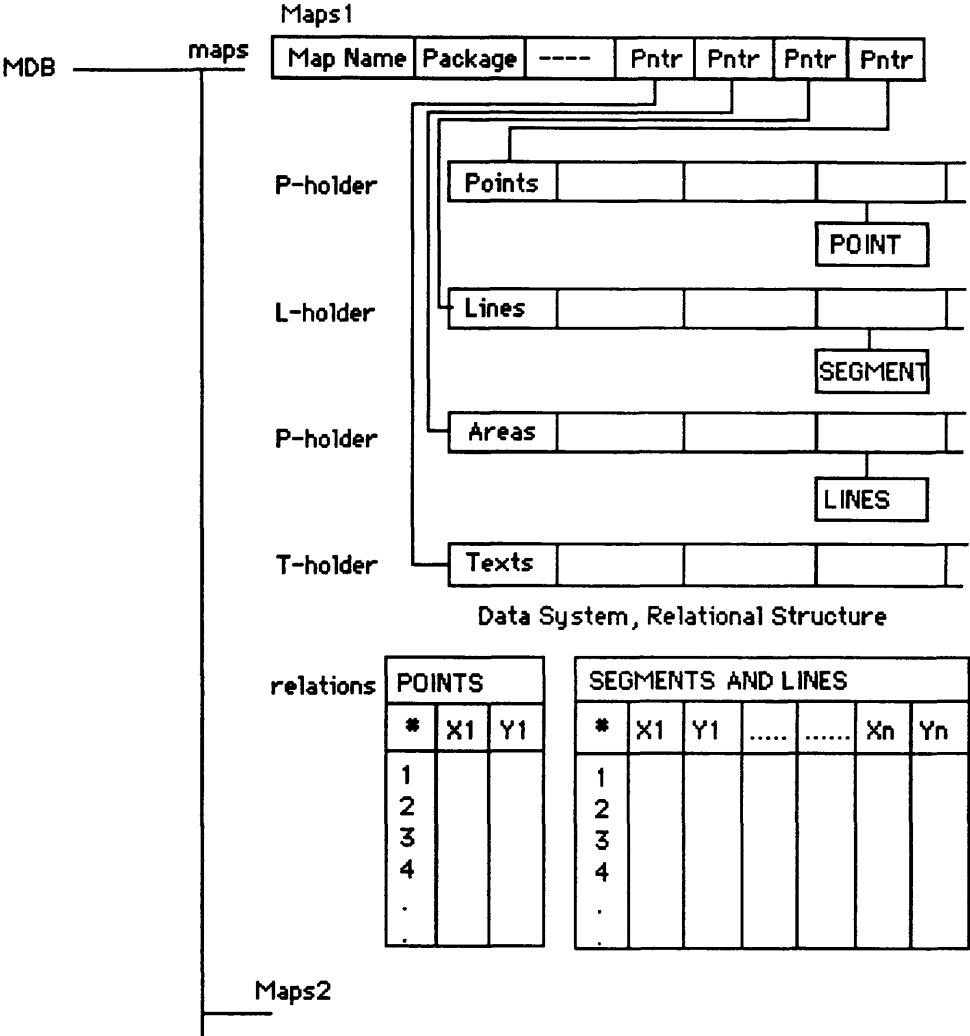


Fig. 5.9 Schematical representation of the MDB

Point symbols, on the other hand, comprise regular geometrical shapes or figures such as circles, squares, triangles, etc. which are used to represent cartographically dimensionless features such as triangulation marks, manholes and so on. Fig. 5.10 shows the combined display menu of these symbols .

These symbols are created by two different programs, one for each type, because of the different natures of the symbols themselves and the different ways by which they are generated.

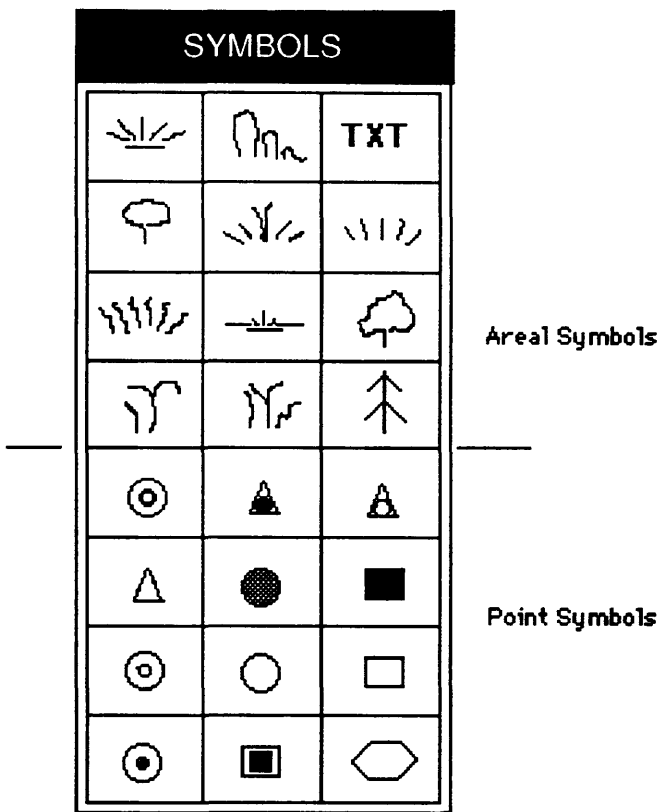


Fig. 5.10 The Symbol menu

5.5.2.1 Areal Symbols

The approach adopted to obtain the shape of these symbols was by digitizing them off an existing map [Yoeli, 1982]. The symbols' coordinate values were then made absolute. Fig. 5.11 shows the symbol for 'orchard' with reference to a coordinate system of origin 'O'. The aim then is to shift this coordinate origin to the minimum Y-value and averaged X-value of the symbol. This is done through a transformation procedure run over the coordinates of all the symbols [Harrington 1987]. The transformation procedure was written in Quick Basic on an IBM-compatible micro-computer in the Department of Geography and Topographic Science. Then all the coordinate values were transferred to the Unix environment of the graphics work station ready to be manipulated and stored in the database. Symbols' data comes in the form of eleven separate files, one file for each symbol. Then the program 'agrsym.S' was written to create the database 'Symbols' (the listing of the program can be found in Appendix B).

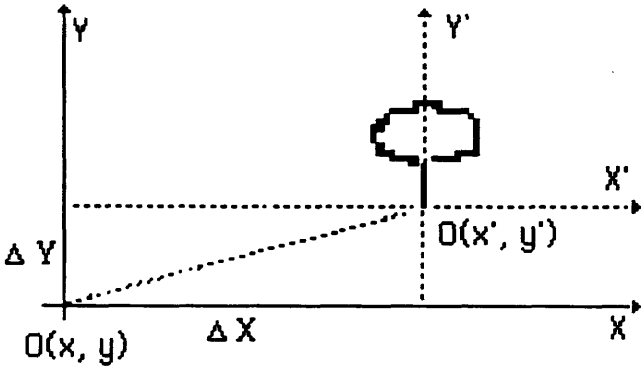


Fig. 5.11 Axis Shifting of the origin of a symbol

The program '*agrsym.S*' makes use of two procedures written for the main program. These procedures are; '*message.proc*' and '*stringtoint*'. Next, the program creates the database 'Symbols' using the statement 'create.database' and declares a structure '*agrsym*' to hold the areal symbols. Then the program starts reading symbols by first requesting the name of the file where the data of a particular symbol resides. Once this has been successfully read, a message to proceed with another symbol is displayed and so on until all the areal symbols have been consumed. Fig. 5.12 shows the flowchart of program '*agrsym.S*'.

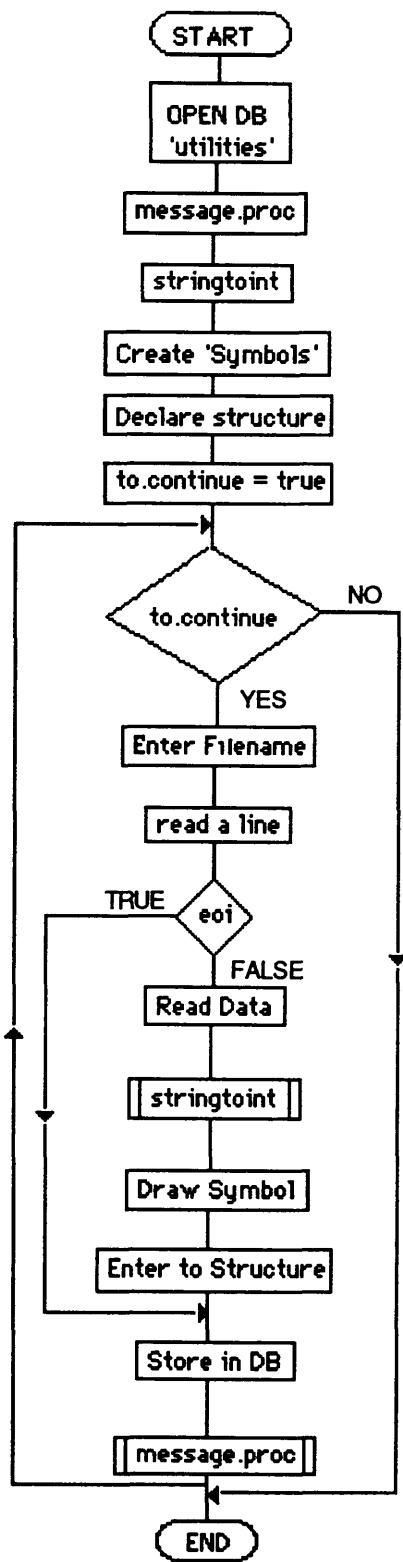


Fig. 5.12 Program 'agsym.S' flowchart

5.5.2.2 Point Symbols

Unlike areal symbols, point symbols are created from mathematical formulae. For this reason, the procedure '*polygon*' was borrowed from the main program to do the job, since this procedure is able to draw all the geometrical shapes needed. Furthermore, since the database '*Symbols*' has already been created in program '*agrsym.S*', program '*geosym.S*' only opens it, but declares another structure '*geosym*' to hold these symbols. The procedure is formed mainly of numerous calls to procedure '*polygon*' to draw the different shapes of each of the symbols.

There are twelve different geometrical shapes, all of which are created by the same procedure in twelve different calls. Then they are stored in the database. Fig. 5.10 shows all the symbols as displayed in menu form.

5.5.3 '*global*' Data Base

'*global*' database is created by a program called '*Menus*'. This program makes use of several procedures intended for the Cartographic Representation module in the main program. These procedures are; *sorting*; *linepara*; *lineint*; *perpenline*; *Lparaline*; *Rparaline*; *drawline*; *dashing*; *hatchpoly*; *doblseg*; *dobline*; *ddline*; *thkline* and *borderline*. A detailed description of each of these procedures will be presented in Chapter 8. Furthermore, two more procedures, '*rec*' and '*Rec*', which are general graphics procedures taken from the Operational Management System, have been used extensively and will be described in Chapter 6. This database is created by the OMS (see Appendix H).

'*Menus*' creates the menus to be used in the main program and stores them in a table called '*Images*' in '*global*'. The menus involved are: '*typemenu*'; '*angmenu*'; '*scalmenu*'; '*defaultmenu*'; '*linemenu*'; and '*Symbols Menu*'.

5.5.3.1 '*typemenu*'

The '*typemenu*' procedure creates the menu which allows the user to choose a type for the line to hatch a certain polygon. This is done by using three procedures, namely: '*drawline*'; '*dashing*' and '*Rec*'. The key identifier to access this menu in the database is '*Hlinetype*'. Fig. 5.13 shows the display resulting from this procedure.

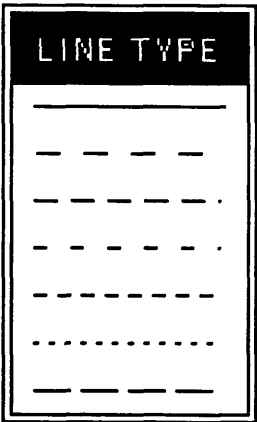


Fig. 5.13 Line Types

5.5.3.2 'angmenu'

The procedure 'angmenu' makes use of two global procedures in the main program 'rec' and 'Rec'. Fig. 5.14 shows the shape of the menu.

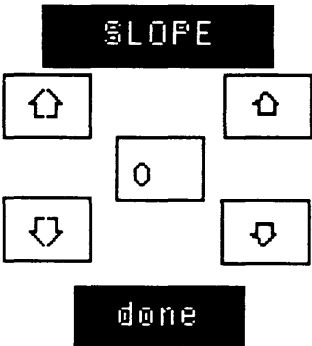


Fig. 5.14 The display of 'angmenu'

The aim of this procedure is to allow the user to choose a suitable angle for polygon hatching. The angle can vary from 0° to 360° and is changed by selecting the arrowed buttons. The large arrow buttons on the left increase and decrease the angle by 15°. The smaller buttons change the angle by 5°. The key to this menu in the database is 'Hangle'.

5.5.3.3 'scalmenu'

This procedure displays a dialogue box as shown in Fig. 5.15. The middle box displays the symbol to be scaled. This box displays the symbol at the scale at which it was stored in the database. The upper box shows the effect of scaling. Scaling is carried out by selecting the arrows to increase or decrease the scale by 20%. The upper and lower arrows increase

and decrease the scale of the symbol in the Y-direction. The left and the right arrows decrease and increase the scale in the X-direction.

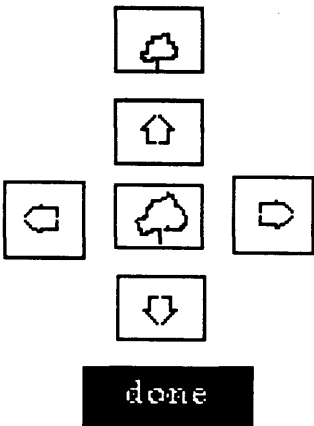


Fig. 5.15 The Scaling menu

The key to this menu is 'Scaling'.

5.5.3.4 'hatchspace'

This procedure uses three other procedures 'Rec'; 'drawline' and 'hatchpoly'. The menu produced allows the user to choose a suitable spacing between the lines used for hatching certain types of polygon. Fig. 5.16 shows the display of this procedure, and the key to this menu is 'Hspacing'.

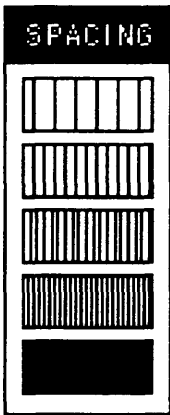


Fig. 5.16 Options for line spacing

5.5.3.5 'default.menu'

'default.menu' uses three procedures which are 'Rec'; 'rec' and 'hatchpoly'. 'default.menu' shows the default settings of the polygon hatching factors for the type of the line; the spacing between lines and the gradient at which these lines should be drawn.

The user can then choose these settings or alternatively change any (or all) of them. Fig. 5.17 shows the menu as it is displayed on the screen. The key identifier to this menu in database 'global' is 'Hdefault'.

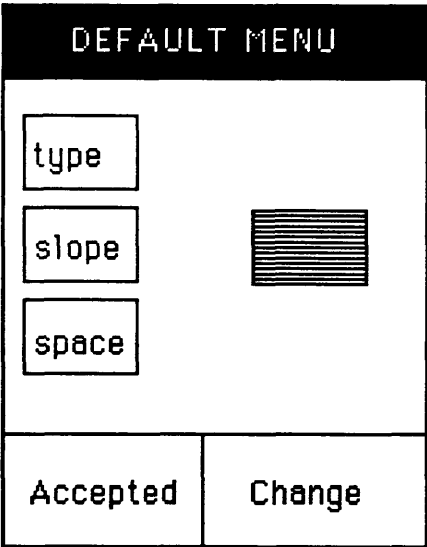


Fig. 5.17 The default menu

5.5.3.6 'linemenu'

Lines drawn on a map display or plot convey different meanings to the user by the way in which they are presented. Procedure 'linemenu' allows the user to choose one of several cartographic representations for lines. Fig. 5.18 shows the types of line made available by invoking this procedure. The procedure itself calls some eight other procedures to draw these types of line. These procedures are: 'Rec'; 'drawline'; 'doblseg'; 'dashing'; 'thkline'; 'ddline'; 'borderline' and 'railine'.

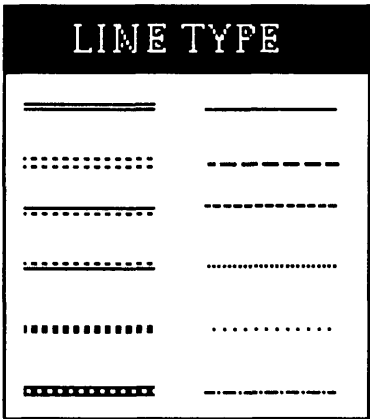
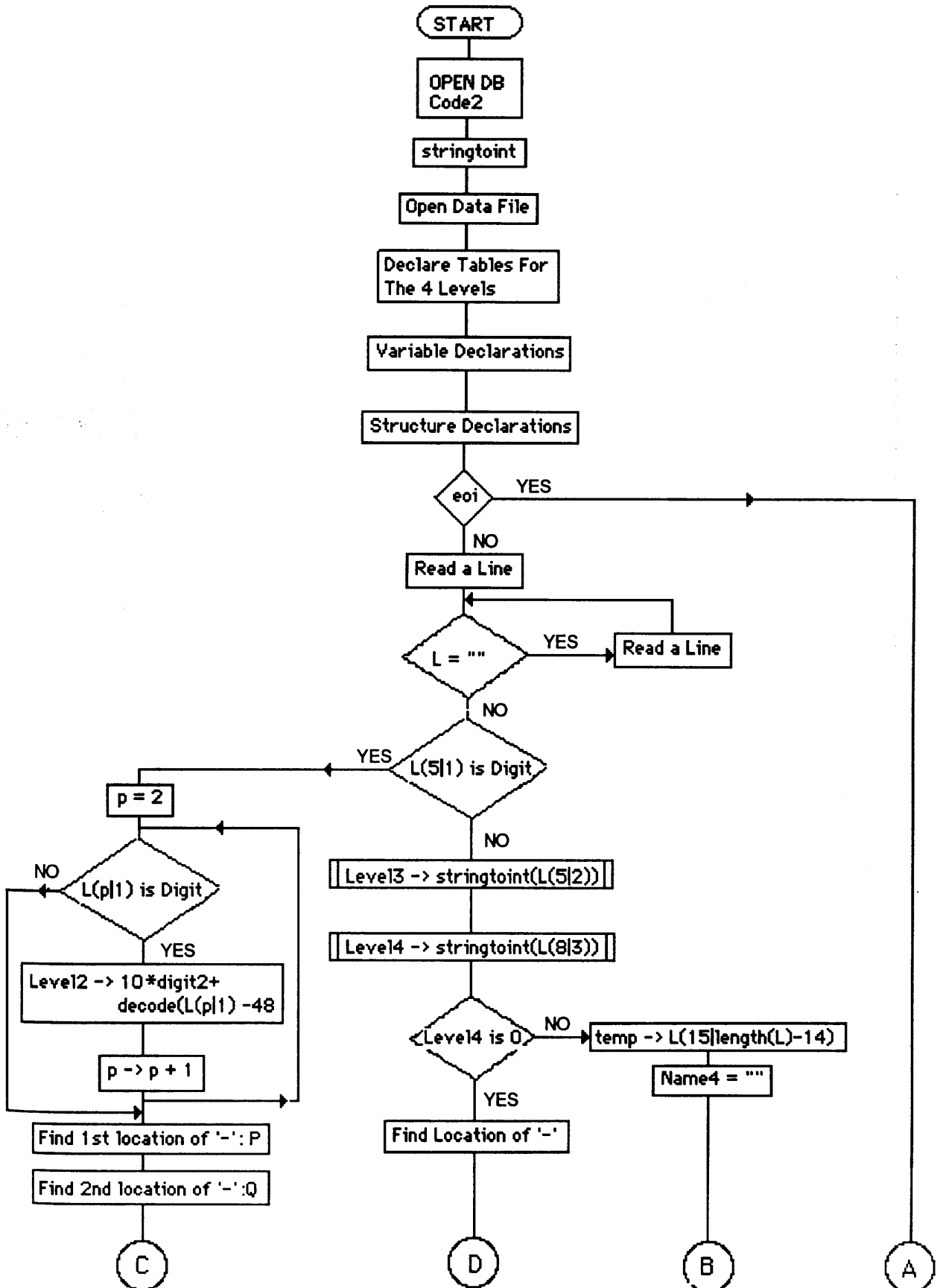


Fig. 5.18 The menu of Lines

The key to this menu in the database is 'LineMenu'.

5.5.4 'Code2' Data Base

The feature coding system adopted in this project is derived from the "National Standards For The Exchange Of Digital Topographic Data" published by the Canadian Council on Surveying and Mapping in 1984. The listing of this feature coding system is in Appendix A. The program that loads this system into the database 'Code2' is called 'incode'. It starts by creating the database and declaring the four levels (tables) of the classification (see Section 5.3.2.1) together with the structures needed to hold the data. Only one procedure is needed to be called from this program. This translates the digital numbers read as characters from the file containing the feature coding system into integers (these integers are the keys to the features' description).



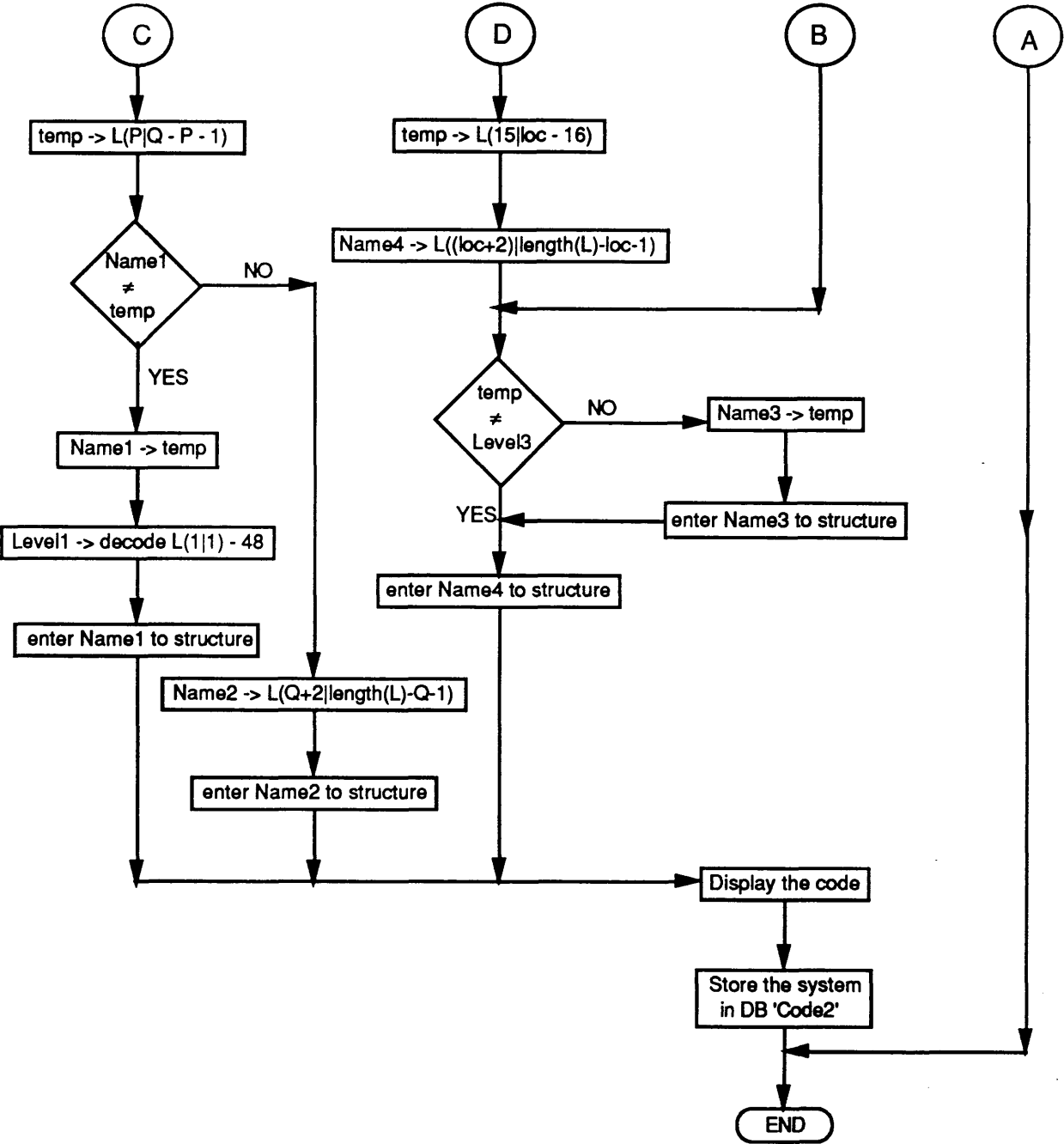


Fig. 5.19 The flowchart of program 'incode'

After the whole coding system has been read successfully, the program 'commits' it to the database.

5.5.5 '%\$Modules'

The trend in programming development is towards analyzing a program and splitting it into small manageable units. In fact, PS-algol is well suited to doing this [Cooper, 1987] and so offers a most efficient way of carrying out large programming tasks. In other languages,

various functions of the program are carried out by procedures. Since procedures are first-class objects in PS-algol, they may be manipulated in the same way as other data types - in particular, they can be assigned to variables, passed as parameters to other procedures and be stored in databases.

The benefit of doing so is that, once a program has been divided into procedures, each one of them can be put into a different source file and be compiled separately with consequent savings in debugging time.

In fact, wherever the procedures are small or where they share a lot of data, it would have been better to code more than one procedure in an individual module. In this case, the procedures may be packaged together into a single structure for storage in the database.

Program '*Globals*' has been written to store all global procedures in a database called '*%\$Modules*' because all the modules make use of them, (- the listing of the program is in Appendix C). After listing all the global procedures contained in the program, a packaging structure called '*listGlobals*' is declared, then the procedures are packaged and entered into the database.

5.6 *Summary*

It is clear from Fig. 5.8, that, in order to be able to manipulate all these modules in an interactive manner, an overall Operational Management System (OMS) is needed, which takes care of calling the different modules that the user may need during a working session. A description of this system and the tasks which it carries out, together with the global procedures that are needed in a global scope, are described in Chapter 6.

CHAPTER 6

CHAPTER 6: OPERATIONAL MANAGEMENT SYSTEM

6.1 *Introduction*

The new trend in computer programming is towards simplicity of interaction between the computer and the user. Programs fulfilling this objective (or rôle) are termed 'user friendly'. The aim behind the user interface being 'friendly' is to make the program easier to operate and to reduce the possibilities of entering wrong data in response to questions or on issuing commands. Menu driven systems are one example of such systems which are widely used nowadays. Indeed, a menu-driven approach was adopted in constructing the GIS system developed during the research carried out for this thesis, since the language used for programming allows such facilities to be constructed relatively easily [Brown & Dearle, 1986].

6.2 *Operational Management System (OMS)*

The operational management system can be described as a set of procedure calls to perform certain jobs in accordance with user commands, including the recording of all the results obtained during processing. At the outset of an operational session, the OMS will display a menu of the principal operations available in the system. These operations are Data Entry; Cartographic Representation; Data Retrieval and Data Output. The functions carried out by each of these operations are described in Chapters 7 to 10. When the user chooses a particular operation, say Data Entry, the OMS will then call all the appropriate procedures from the persistent store. At the end of the session, data being handled or generated by the system will be returned to the store if no errors have occurred.

6.3 *OMS Components*

The OMS forms the main body of the system, and is composed mainly of procedure calls to the four main modules forming the framework within which all these calls are performed (Fig. 6.1). These four modules correspond to the four operations defined in the previous section (6.2) above.

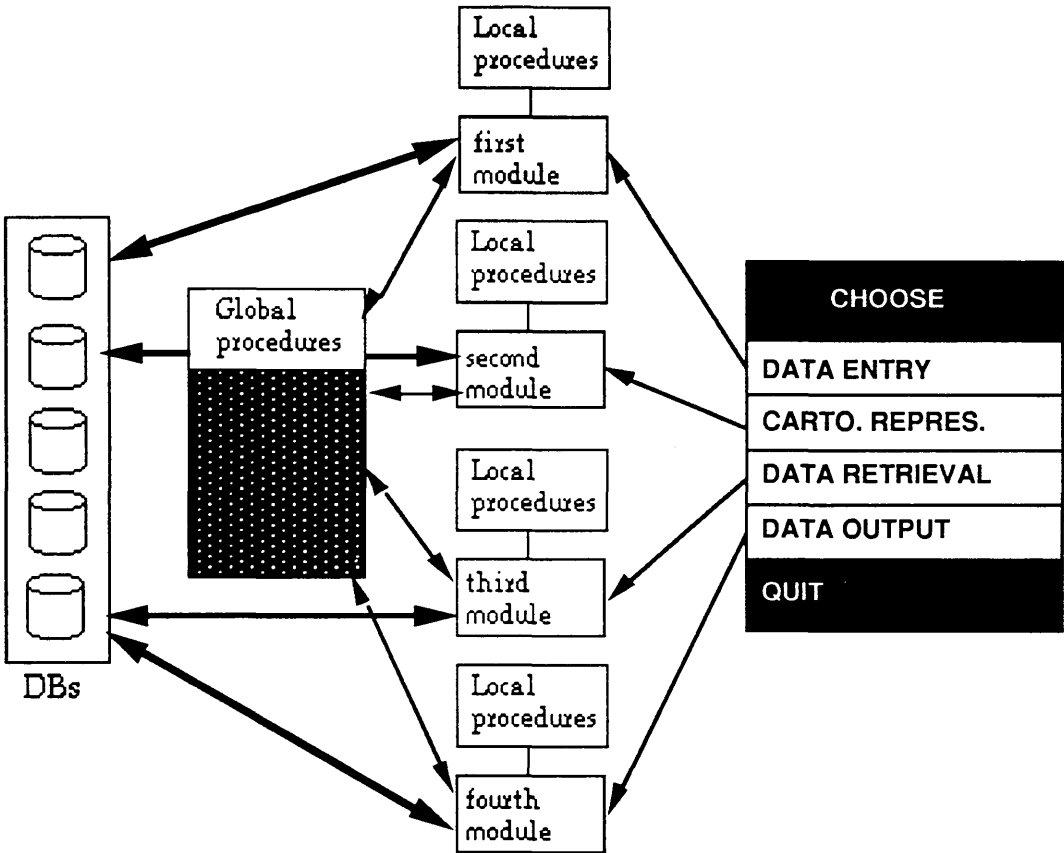


Fig. 6.1 OMS linkages

Procedures used in the system are of two types; the first being called 'module' procedures and the second, 'global' procedures. Module procedures are those used within the limited scope of a particular module, and cannot be used on a wider scope. For example, a procedure declared within the Data Entry scope cannot be used within the scope of Data Retrieval. An example of a module procedure is the procedure used to read data from original data files. This procedure is only needed in the Data Entry module, because no other modules deal with the original data file.

Global procedures are defined as being those procedures that can be called from any scope in the system as represented by Fig. 6.2. An example of a global procedure is the zooming procedure which enables the user to enlarge part of the screen to make clearer the manipulation of the data within that specified area of the screen.

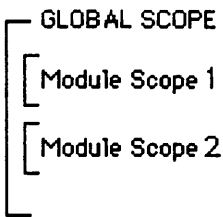


Fig. 6.2 Nesting of scopes

In the following sections, a description of the global procedures is presented together with their functions.

6.4 *OMS Description*

As previously discussed, OMS is composed mainly of four procedures. These procedures deal with the various databases and other data objects. Arising from this, links should be made to databases and data objects should be declared beforehand. Thus, the main body of the program (Appendix H) is concerned with opening databases and declaring the structures needed as such, followed by the data object declarations and the presentation of the generic screen shown in Fig. 6.3, which allows a selection from the four different modules.

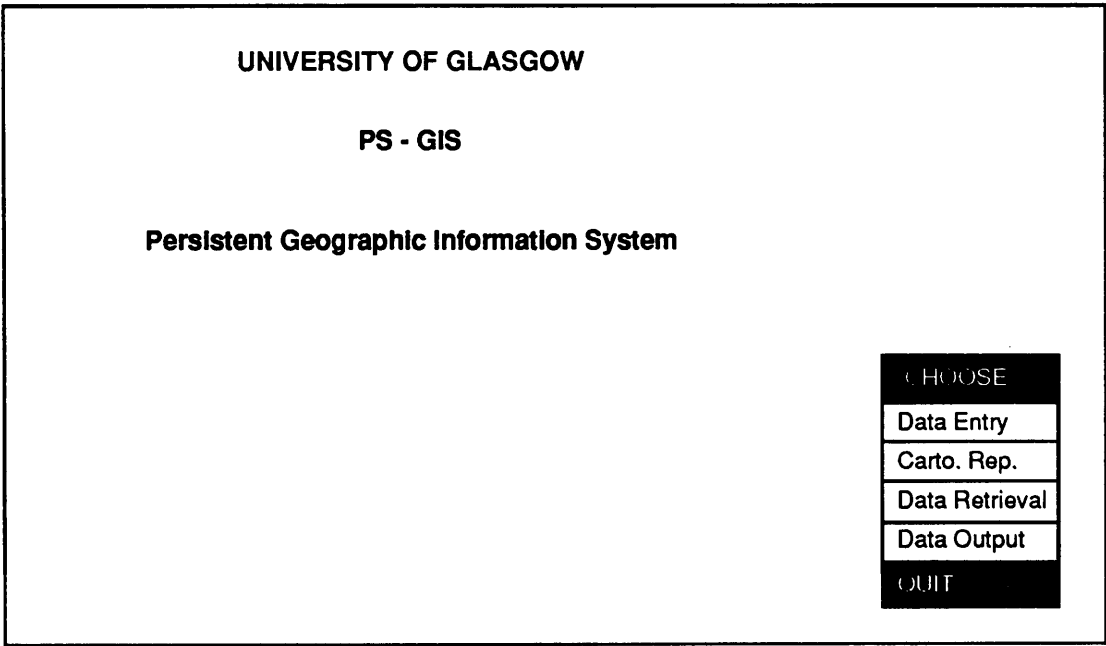


Fig. 6.3 The generic screen of the system

At the start of the program, the utility databases (available with PS-algol) are called. Such utility databases hold the different fonts, and the menu-making facilities. Fonts are available in a PS-algol database called 'FONTS', of which only eleven were used as will be seen in Chapter 8.

The other utility database is called 'rutilities' and contains several important procedures [Cooper, 1988] which are:

| <u>Procedure</u> | <u>Function</u> |
|------------------|---|
| s.editor | is a text editing procedure in a screen window |
| error.message | is a procedure which displays a message in a screen window |
| more | is a procedure which allows the paged display of long texts |
| form.generate | this procedure generates a dialogue box as a set of light buttons |
| set.up.choose | this procedure creates a menu which allows the user to choose from a set of objects |
| table.to.text | makes a string vector of the keys of a table |

(Source: PPRR 56, 1988)

The next step is to call the system database (MDB) if it already exists, or create it if it does not (see subsection 4.4.4.2). This database is the one that holds the geographical data, the detailed description of which will be given in Chapter 7. Furthermore, the other databases ('global'; 'Symbol' and 'Code2') are also opened at this stage. Following this is the declaration of the various structures used by the program and the global identifiers. The OMS program then calls the system modules using a menu with five options, the first four represent the four modules (Data Entry; Cartographic Representation; Data Retrieval and Data Output) with the fifth option being the 'Quit' command.

6.5 *Global Procedures*

In a system with more than one module, global objects are those which are shared by more than one module. The system is designed so that there is a hierarchy of scopes of different data objects, of which the main two scopes are the global and the module scopes.

Any data object declared within the global scope and outside the module scope will persist and be used during the activation of the program. On the other hand, objects declared within a module scope will only persist during the module activation, unless stored in the

persistent store. For example, procedure 'distance' is declared within the 'Cartographic Representation' module, so this procedure is only callable from within this module and cannot be called, say, from the 'Data Entry' module. On the other hand, procedure 'stringtoint' can be called from any module in the system because of its declaration being in the global scope of the main program.

6.5.1 Global Procedures Description

Global procedures are also declared at this stage. There are twenty two of them used by the four modules of the system. Global procedures are usually utilities which are needed to perform frequently required tasks. These procedures are first declared and then compiled. The compiled version is then stored in a single structure in a database called 'Global.Proc'. They are called by retrieving the structure and extracting the procedures according to need within the modules. In the following list, the individual global procedures are described.

| <u>Procedure</u> | <u>Function</u> |
|--|--|
| rec(int x, y, xd, yd) | This procedure draws a rectangle with its centre at a point 'x,y' and with dimensions equal to 'xd' and 'yd' |
| Rec(x, y, xd, yd, text, font, position, backlit) | Draws a rectangle at a point 'x,y' with dimensions equal to 'xd, yd', a header 'text', justification 'position' (left, middle and right) and the choice of reverse video display (boolean: true or false). A sample result having 'OPTIONS' as 'text', 'middle' for justification and 'true' for backlit is shown in Fig. 6.4. |

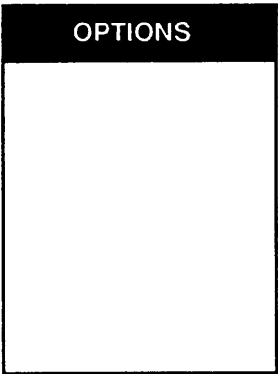


Fig. 6.4 Display result of an example of the 'Rec' procedure

`icon(x, y, text)`

This procedure takes the 'x,y' coordinates specified for a point together with an item of text and returns a picture of an icon containing that text. The size of the icon is determined by the procedure. Fig. 6.5 shows an example of the use of procedure 'icon' with 'text' being the word 'Cancel'. Once the cursor is placed over the icon and the button is clicked, the icon will be displayed in reverse video.



Fig. 6.5 Result of procedure 'icon' before and after clicking

`text.write(x, y, text, font, window)`

This procedure displays a text item in a particular window with a particular font at a specified location having coordinate values 'x,y' as the starting point of this text.

`first.screen()`

The procedure is responsible for the displaying of the generic screen of the system together with a menu of the four modules available. Fig. 6.3 shows the result of this procedure.

`zoomin(picture, grid, range, counter, offsetx, offsety, window)`

This procedure enlarges the display of a particular area of the screen specified by clicking the mouse at one location and releasing the button at another. The procedure needs to specify the picture and (if present) the grid, together with the range of the picture (in ground coordinates) in order to compute the transformation. The 'x' and 'y' offsets are used to display part of a particular window. The counter allows continuous and successive zoomings.

`zoomout(picture, grid, range, offsetx, offsety, window)`

This procedure reverses the effect of the previous one.

`checkin(vectorx, vectory, scopex1, scopey1, scopex2, scopey2)`

This procedure checks whether a particular point, line or polygon (whose data are supplied by vectorx and vectory) lies within a chosen part of the display bordered by scopex1, scopey1, scopex2 and scopey2 respectively, and returns a boolean result (true if it does or false if it does not).

`minmax(vectorV)`

This procedure takes a vector of reals, finds the minimum and the maximum values and returns a vector of two elements, the first being the minimum and the second the maximum.

`drawline(x1, y1, x2, y2)`

This procedure 'creates' a line between the two specified points (x1, y1) and (x2, y2) and returns a picture of it.

`polygon(x, y, r, resolution)`

The polygon procedure is a multi-purpose one, since it is able to 'create' a multiplicity of shapes, namely: triangles, rhombuses, hexagons and circles of different resolution. The 'resolution' factor is a string and can be any one of the following:

| | | | |
|-------|-----------------------------|-------|----------------------------|
| 'tr' | triangle | 'vl' | very low resolution circle |
| 'rec' | rhombus | 'low' | low resolution circle |
| 'hx' | hexagon | 'hi' | high resolution circle |
| 'df' | very high resolution circle | | |

In each case, it returns a picture of the shape required. The parameters 'x' and 'y' give the position and 'r' the radius of the figure.

`north.dir(angle, window)`

The procedure takes the north bearing and 'creates' a picture of the North direction. Fig. 6.6 shows the shape of the arrow-based symbol which indicates the North direction.

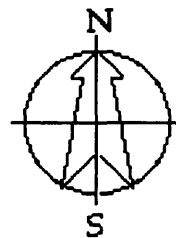


Fig. 6.6 The North symbol

`stringtoint(string)`

This procedure takes a string of digits and returns an integer number.

`stringtoreal(string)`

This procedure takes a string of digits including a decimal point and

returns a real number.

Highlight(vectorx, vectory)

The job of this procedure is to emphasize a point, a line or a polygon to attract the attention of the user to the chosen feature. This is distinguished by drawing small hexagons at the vertices of the line/polygon (or one hexagon centred at the point x,y coordinate values in case of a point feature) as in Fig 6.7.

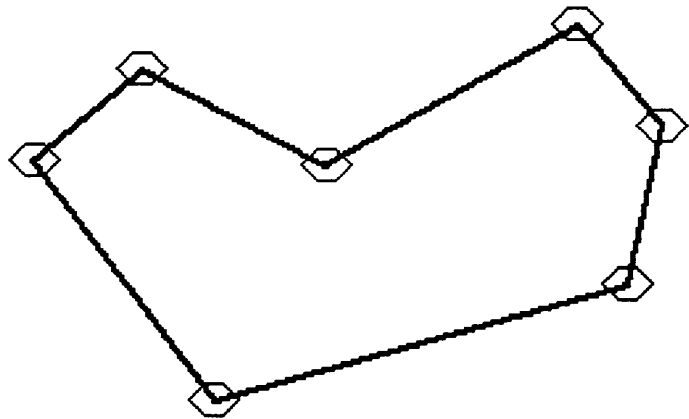


Fig. 6.7 Highlighting a polygon

message.proc(string1, string2, string3, x, y, xd, yd)

The task implemented via this procedure is to allow the user to communicate with the screen via the mouse. It provides a question (string1) and two possible answers (string2 and string3) which could, for example, be 'YES' and 'NO'. 'x' and 'y' represent the coordinates of the lower left corner of the rectangle (in which the message should be displayed) and 'xd' and 'yd' are its dimensions. This procedure returns a boolean result. Fig. 6.8 shows an example of a message resulting from this procedure giving the user the option whether to proceed with a session or to quit.

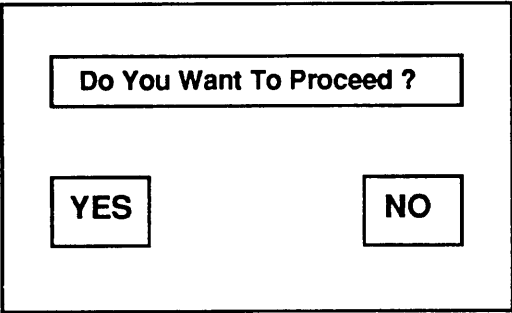


Fig. 6.8 A typical display of the result of the procedure 'message.proc'

prepform(string)

This procedure 'creates' the layout of the screen for the Cartographic Representation and Data Retrieval modules and draws the selected layout on the screen. It is supplied with the heading or title of the module. Fig. 6.9 represents the layout of the screen resulting from this procedure.

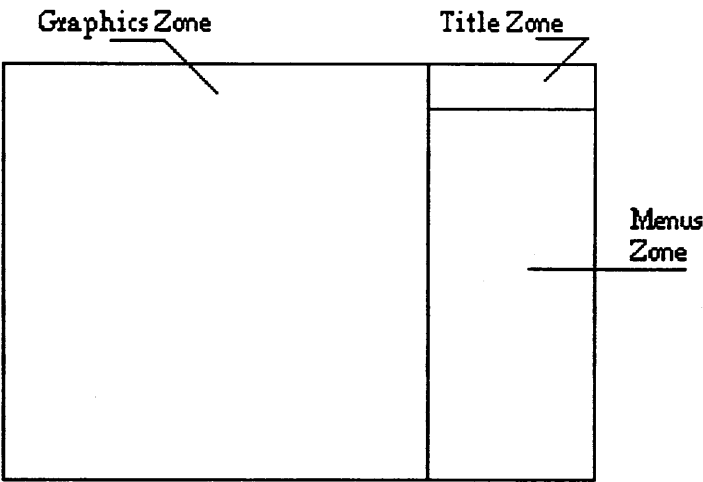


Fig. 6.9 The layout of the screen

feature.type()

This procedure activates a selection menu of feature types. The type that can be chosen is one of the three types, point, line or polygon, and returns a string reflecting the choice. Fig. 6.10 shows the display resulting from this procedure.

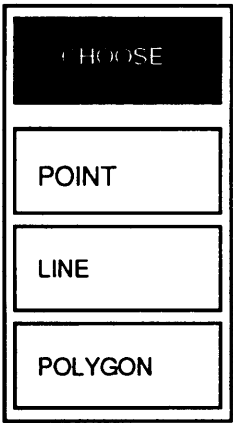


Fig. 6.10 The display of the 'feature.type' procedure

trans.code()

This procedure represents the first module of the system, within which the calls to all the procedures concerned with data entry are

performed. This will be explained in Chapter 7.

`data.prep()`

This procedure represents the second module of the system. It calls all the procedures required for cartographic representation. The procedures used within this module are detailed in Chapter 8.

`retrieve()`

This is the third module procedure and calls the procedures needed for manipulating queries in a menu-driven fashion and interaction. Information retrieval is also performed at this stage. The descriptions of these procedures are given in Chapter 9.

`result()`

This is designed to assist in sending the results of data processing to different hard copy output channels. The output devices which are supported are laser printers and plotters. The results are output via transitional files and will be described in Chapter 10.

6.6 *Summary*

In this chapter, the reasons for employing an Operational Management System (OMS) to handle the user interface with the proposed GIS system have been discussed. Furthermore, this chapter has also presented a description of the various scopes together with a brief description of each of the global procedures (and their functions) which are used in the OMS. This introduces the framework within which the four main modules will be discussed. These will be described in the next four chapters.

CHAPTER 7

CHAPTER 7: DATA ENTRY AND CODE TRANSFER MODULE

7.1 *Introduction*

Data that represent the terrain surface, come in two distinct formats - raster and vector. The work undertaken in this thesis deals solely with data in vector format, and so only vector data can be accommodated in the system as it stands at the moment. The addition of raster data has been taken into account in the basic design of the system but time has not been available to implement this facility.

7.2 *Data Formats*

Digital data in vector format can have different arrangements according to the organization from which the data were obtained. They also depend on the characteristics of the system and the software package used for digitization. Different mapping organizations and users also employ different data formats, but the trend nowadays is toward unification into one standard format. In the U.K., the principal standard format used by the national mapping organization, the Ordnance Survey (OS), is the Ordnance Survey Transfer Format (OSTF). A new and more highly developed format agreed by the OS and the main user community is in the course of being introduced. This is known as the National Transfer Format (NTF). It must also be said however, that many other formats are used, notably the Standard Interface Format (SIF) used by Intergraph and other leading system suppliers; the Data Exchange Format (DXF) used by the popular CAD package Autocad; the MOSS format which has been adopted widely by those organizations concerned with civil engineering design and landscape architecture; etc. [Petrie, 1990].

In the meantime, various packages have been developed which aim at encoding maps digitally with a reasonable accuracy using very good quality software and which come complete with a menu-driven interface. Map Data (produced by Map Data Management Ltd. of Kendal) is one of these software packages and has been used to furnish the data for this project.

7.3 Digitization and Data Structuring

Digitization took place in the Department of Geography and Topographic Science at the University of Glasgow using Map Data version 3.0 mounted on an Apricot Point Seven micro-computer equipped with a 10Mb hard disk and linked to a large-format GTCO digitizing tablet. The data which has been used to test the system has been acquired by digitizing a part of the 1:10,000 scale Ordnance Survey map sheet NS 77 NE for a 5km x 5km area located in Strathclyde Region, Scotland covering part of the town of Cumbernauld. The lower left corner is defined by the coordinate values: $X_L = 75,000\text{m}$ and $Y_L = 75,000\text{m}$, while the opposite corner coordinates are: $X_R = 80,000\text{m}$ and $Y_R = 80,000\text{m}$.

The format of the file of the resulting digitized data consists of (a) the heading of the file followed by (b) sections for each of the digitized features (see Table 7.1). The heading section has three lines allocated to it. These are as follows:-

(a) Heading

- i) The first line is used to indicate the area being digitized. The user can, at the start of the digitizing procedure, enter the name of the area (comprising up to 40 characters) to enable the data set to be easily identified.
- ii) The second line is used to locate the file containing the control points, i.e. the coordinates of the grid intersections and the edges of the map.
- iii) The third line contains the coordinate values (X, Y, and Z) of the centre point of the map.

(b) Data Describing Features

For each digitized feature, there is a set of lines as follows:-

- i) The first line contains the segment number and a space allowing comments to be inserted, which might be of help in describing the digitized feature. The term segment number as used in Map Data is a little bit confusing, since in fact it covers both individual points and line segments.
- ii) The next line contains a code which identifies and classifies the type of the features. The 'code' is a number between 1 and 127 together with a 'sub-code' of the same range as that of the code. The code and the sub-code are separated by a full-stop, e.g. "127.127". The meaning of the codes may vary from one organization to another.
- iii) Finally there is a set of lines giving the X, Y, and Z coordinate values for each segment in a digitized line. These continue until the next feature is reached when a

new segment number will be given.

Table 7.1 illustrates an example of data resulting from the digitization procedure using the Map Data package.

Table 7.1 The typical Map Data format

| | | | |
|-------------|-----------|------------|-------|
| Cumbernauld | | | |
| cumcon | | | |
| 77500.000 | 78500.000 | 0.00000000 | m m m |
| segment 1 | ** | ** | |
| code 6.1 | | | |
| 77002.120 | 78696.620 | 0.00000000 | |
| 77095.440 | 78739.940 | 0.00000000 | |
| 77521.080 | 78871.140 | 0.00000000 | |
| 77610.160 | 78823.720 | 0.00000000 | |
| segment 2 | ** | ** | |
| code 6.7 | | | |
| 77123.060 | 78080.240 | 0.00000000 | |
| segment 3 | ** | ** | |
| code 6.7 | | | |
| 77688.940 | 78196.100 | 0.00000000 | |

In terms of the actual procedure which is followed in map digitizing and the related data structuring, digitizing can be carried out in one of two ways, the first being the 'link and node' method, while the second is called the 'spaghetti' method [Parker, 1990].

The link and node method recognizes links and nodes only. Nodes represent either point features or intersections between line or area features. Links represent the grouping of segments which connect two nodes. Linear features are represented by a series of links (equivalent to the segments discussed in Chapter 2). Point features are positioned at nodes and areas (polygons) are represented by closed chains of links (termed segments and lines in the previous discussion in this thesis).

Theoretically, the structure used in the 'link and node' model provides the ability to handle the majority of retrieval demands in a GIS, such as network analysis; the extraction of

specific feature classes; the generation of thematic maps; etc. On the other hand, using this method with its attendant data structure, needs a lot of preparation and is time consuming, because the operator will need to:

- give names to the different polygons on the map (usually there are hundreds);
- specify the nodes at which the digitizing of a line starts and ends;
- ensure that digitization proceeds in a single direction throughout the operation;
- specify those areas lying to the left and right of the line; and
- specify polygons as groupings of links rather than as entities.

Not all data processing packages can cope with this model.

On the other hand, in the spaghetti method, digitization is carried out in a manner whereby connectivity references are neglected. Features are digitized independently of one other and may only be related subsequently by comparing the digitized values of these features. However, the advantages of this method are the simplicity of the digitizing operation and the fact that several layers of the same base map can be digitized separately and held as separate units. The drawbacks resulting from this method are mainly concerned with connectivity and the duplication (redundancy) which takes place when digitizing polygons. Fig. 7.1 and Table 7.2 show the polygons 'I' and 'II' and their representation in both the link and node and spaghetti methods.

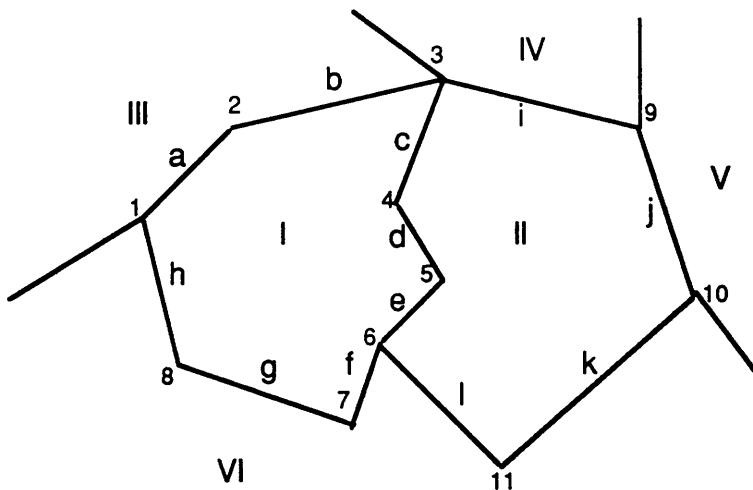


Fig. 7.1 A sketch showing an area of two polygons

| Polygons Digitising Method | I | | | II | | |
|----------------------------------|----------|---------------------------|-------|----------|---------------------|-------|
| | Links | Segments | Nodes | Links | Segments | Nodes |
| <u>LINK & NODE</u> | 1st Link | a, b | 1, 3 | 4th Link | i | 3, 9 |
| | 2nd Link | c, d, e | 3, 6 | 5th Link | j | 9, 10 |
| | 3rd Link | f, g, h | 6, 1 | 6th Link | k, l | 10, 6 |
| | | | | 7th Link | e, d, c | 6, 3 |
| <u>SPAGHETTI</u> | | a, b, c, d, e, f, g, h | | | i, j, k, l, e, d, c | |

Table. 7.2 'Link & Node' and 'Spaghetti' methods of digitizing polygons

Based on the above discussion, the spaghetti method was used for this particular project, since the concentration has been on system design and implementation. In this case, the initial data capture should be made as easy as possible. Indeed, it is the programmer's task to sort out the problems encountered during the later stages of data processing, such as creating links between different features held in different layers, and removing the resulting slivers and gaps resulting from the duplication in carrying out the digitization operation [Burrough, 1985]. Furthermore, the system should be able to handle this sort of data, since it is so prevalent.

In fact, the system developed in this project does not let the user edit the input data, but on the other hand, the package used for digitization is well equipped to do so. Moreover, the possibility of recalling existing digitized points has already been provided in the Map Data package, which, if used correctly, will reduce the problem of slivers and gaps to a minimum. So, with careful digitizing, there should be no problem with regard to those features which are inherent with this method of digitization. Thus, the system devised for this project assumes that the original data is error free.

7.4'*trans.code*' Module

This module takes a file of digitized data and stores it in the persistent store with all its features coded. It contains two parts: one which permits some global configuration to be given to the data by the user; while the other reads the data from the file and allows the user to code each feature, one at a time. Fig. 7.2 illustrates the overall operations carried out by 'trans.code' module, namely data entry and code transfer. The coding part of this module is organized so that coding may be interrupted at any stage and returned to in a subsequent

run of the program.

The data entry and code transformation module is based on three types of procedures- (i) global procedures (low level); (ii) modular procedures (mid level); and (iii) local procedures (high level). Fig. 7.3 illustrates the interlinks between these three types of procedures. Finally, Fig. 7.4 illustrates the line of actions carried out in this module by a flowchart.

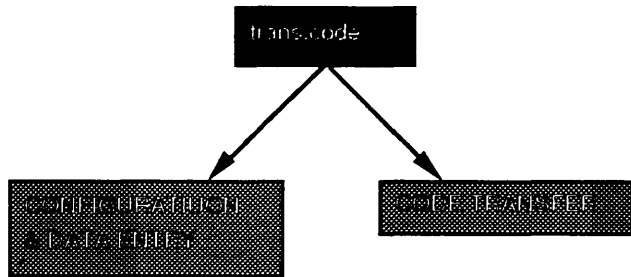


Fig. 7.2 Operations of 'trans.code' module

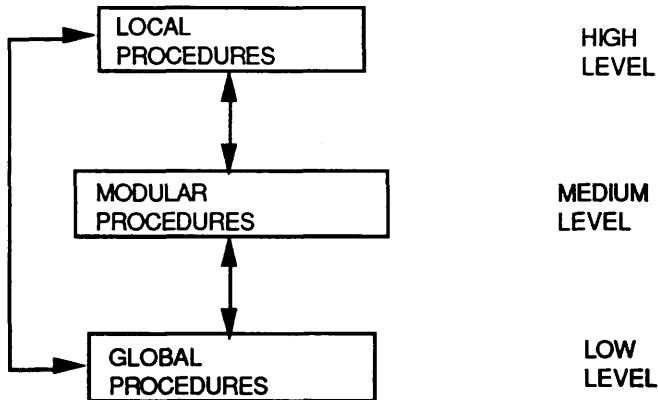


Fig. 7.3 Types of procedures in this module

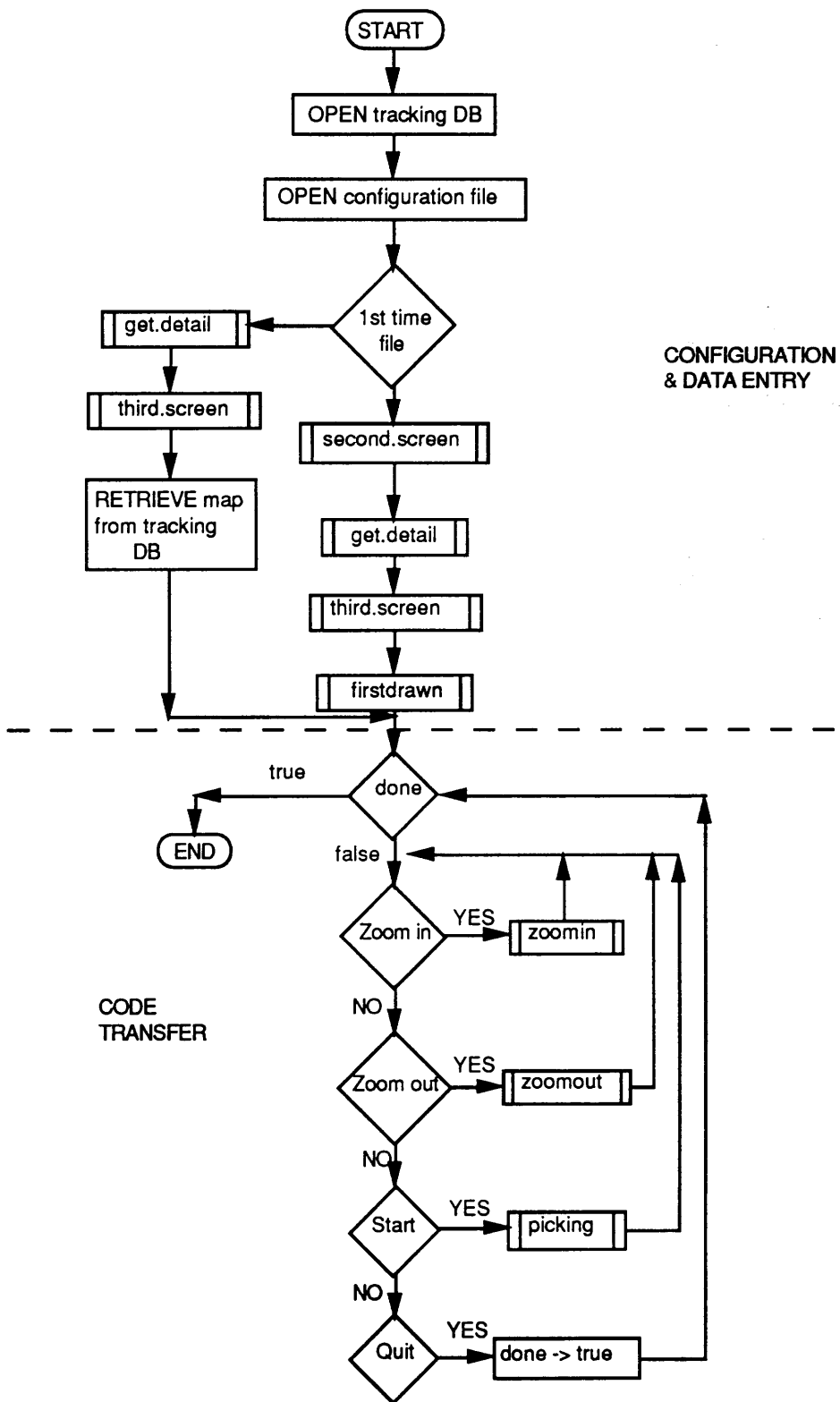


Fig. 7.4 Flow chart of module 'trans,code'

In the next sections, general descriptions of the two main operations - Configuration & Data Entry and Code Transfer - will be presented in Sections 7.4.1 and 7.4.2 respectively and afterwards, descriptions of the procedures involved will also be provided in Section 7.5.

7.4.1 *The Configuration & Data Entry Operation*

At the outset of this module, the system will check whether a configuration file exists or not. This file records some important details about the data to be manipulated in the system. The discussion of this file will appear later on in this chapter.

When this module is selected for the first time, no such configuration file exists and, at this stage, the layout of the screen changes as determined by the procedure '*second.screen*'. A series of questions are displayed on the screen, the responses to which are intended to provide details about the data which the system needs to have if work is to be carried out on the data. These questions are:-

- 1- Name of Package Used For Digitization
- 2- Map of (Volume Set Name)
- 3- X, Y Coords. of the Lower Corner
- 4- X, Y Coords. of the Opposite Corner
- 5- Grid Interval
- 6- Bearing from North (in degrees)
- 7- Total Number of Physically Separated Files
- 8- Individual File Number
- 9- Source File Name
- 10- Scale of the Map
- 11- Number of Features' Coordinates

Once all these questions have been answered, a box appears on the right hand side of the screen allowing the user to go through these questions again correcting any errors that might have occurred during the first time of answering or to proceed to the next step if no correction is needed. Fig. 7.5 shows the typical display on the screen at this stage.

| PS - GIS | |
|---|---------------------|
| Name of Package used in digitization | : MapData |
| Map of (Volume Set Name) | : Cumbernauld |
| Serial Number of Volume Set | : NE77NS |
| X, Y Coords of the Lower Corner | : 75000.00,75000.00 |
| X, Y Coords of the Opposite Corner | : 80000.00,80000.00 |
| Grid Interval | : 500 |
| Bearing from North (in degrees) | : 0 |
| Total Number of Physically Separated Files | : 5 |
| Individual File Number | : 1 |
| Source File Name | : mapcumrd |
| Scale of the Map | : 1:10000 |
| Number of Features' Coordinates (2 or 3) | : 2 |

RETRY

YES NO




Fig. 7.5 The configuration data entry

Upon entering the answers to these questions, they will be recorded in the '*MDB*' data base and in a configuration file called '*config*' so that this phase can be skipped in subsequent sessions. Then the system will show another layout of the screen containing three different types of display. The screen is divided into four zones; (i) a graphics zone, to display the features' drawings; (ii) a text zone, to display details of the features that were extracted from the original data together with the corresponding coordinate values; (iii) another text zone to allow the display of the file details; and (iv) a menu zone which serves to display the different menus needed during the process. The procedure entitled '*third.screen*' is responsible for the display of these four zones. Fig. 7.6 shows the outline of the divided screen.

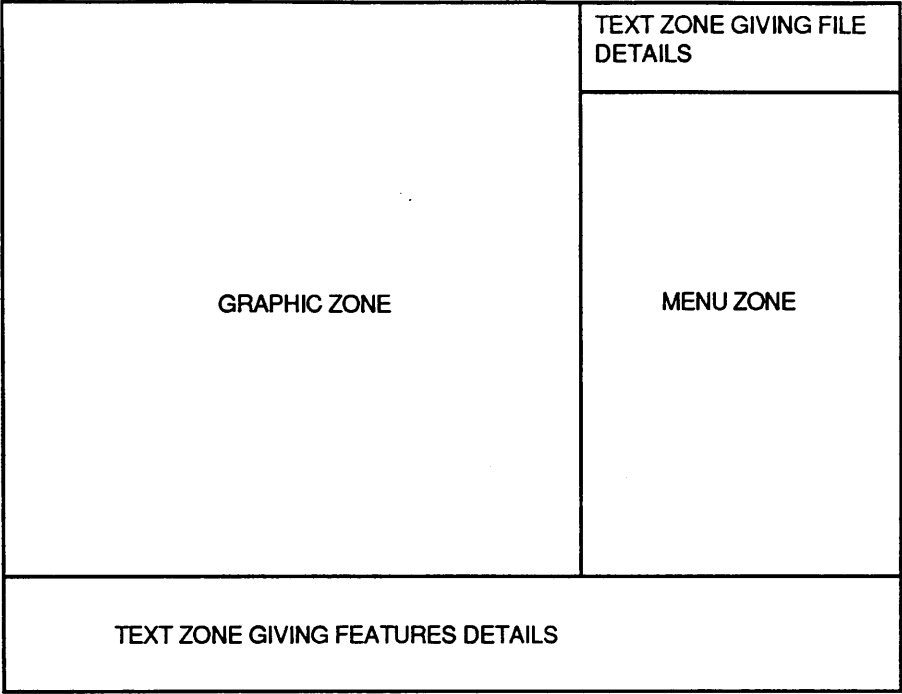


Fig. 7.6 Screen division

Next the system starts reading automatically the file specified in response to question No 9 and displays its contents on the screen. This is done through the procedure *'first.drawn'*. After completing the drawings, a snapshot of the displayed features is recorded and stored in the data base *'global'* which is accessed via a key identifier named *'dr'* in a table called *'Variables'*.

Since some of the data in the original file are textual and others are numerical, and since the system will read all of them as being of type **'string'**, a differentiation needs to take place at this stage. Thus, answers 3, 4, 5, 6, 7, 8, 10 and 11 need to be transformed back to their original types, which is **'real'** for answers 3, 4, and 10; and **'integer'** for the rest. This is done through the procedure *'get.detail'* which reads the file details from the configuration file and, in turn, calls two other procedures named *'stringtoreal'* and *'stringtoint'*.

On the other hand, if the module had been activated in a previous session, then the system will bypass the questions phase and will, instead, read the file details from the configuration file *'config'* and display the features graphically from the stored map in the database. The user will notice the substantial difference in speed when displaying the map on the screen on the second occasion.

7.4.2 The Code Transfer Operation

Coding systems play a major rôle in processing the acquired data. They help in structuring these data into the databases, identifying their types, and in carrying out query processing and data retrievals. The more comprehensive and well structured the coding system is, the simpler the processing of data. This part of the program allows the user to associate a code with each feature depicted on the map.

Once the map has been read and displayed, the user is offered four processing options in a menu form. These options are: '*Zoom In*'; '*Zoom Out*'; '*Start*' and '*Quit*'. The description of the activities of each of these options is given in the remainder of this Section.

In this module, three main procedures are used. One of these called '*picking*' is declared at this module level while the other two - '*zoomin*' and '*zoomout*' - are of global scope.

7.4.2.1 Option '*Zoom In*'

The description of this procedure has already been presented in Chapter 6, Section 6.5.2. It is used here to select a smaller area of the graphic to code.

7.4.2.2 Option '*Zoom Out*'

The description of this procedure has also been presented in Chapter 6, Section 6.5.2. It is used here to reverse the action of the previous option, so that the user will be able to '*zoom out*' to the original display scale on the screen and is then in a position where he can '*zoom in*' on a different part of the map.

7.4.2.3 Option '*Start*'

When the option '*Start*' is chosen from the menu, the procedure '*picking*' is called which starts picking features from the map sequentially and highlighting them, so the user can identify which feature has been chosen. Then the various operations needed to be executed with them are carried out. Fig. 7.7 illustrates the flowchart of this procedure.

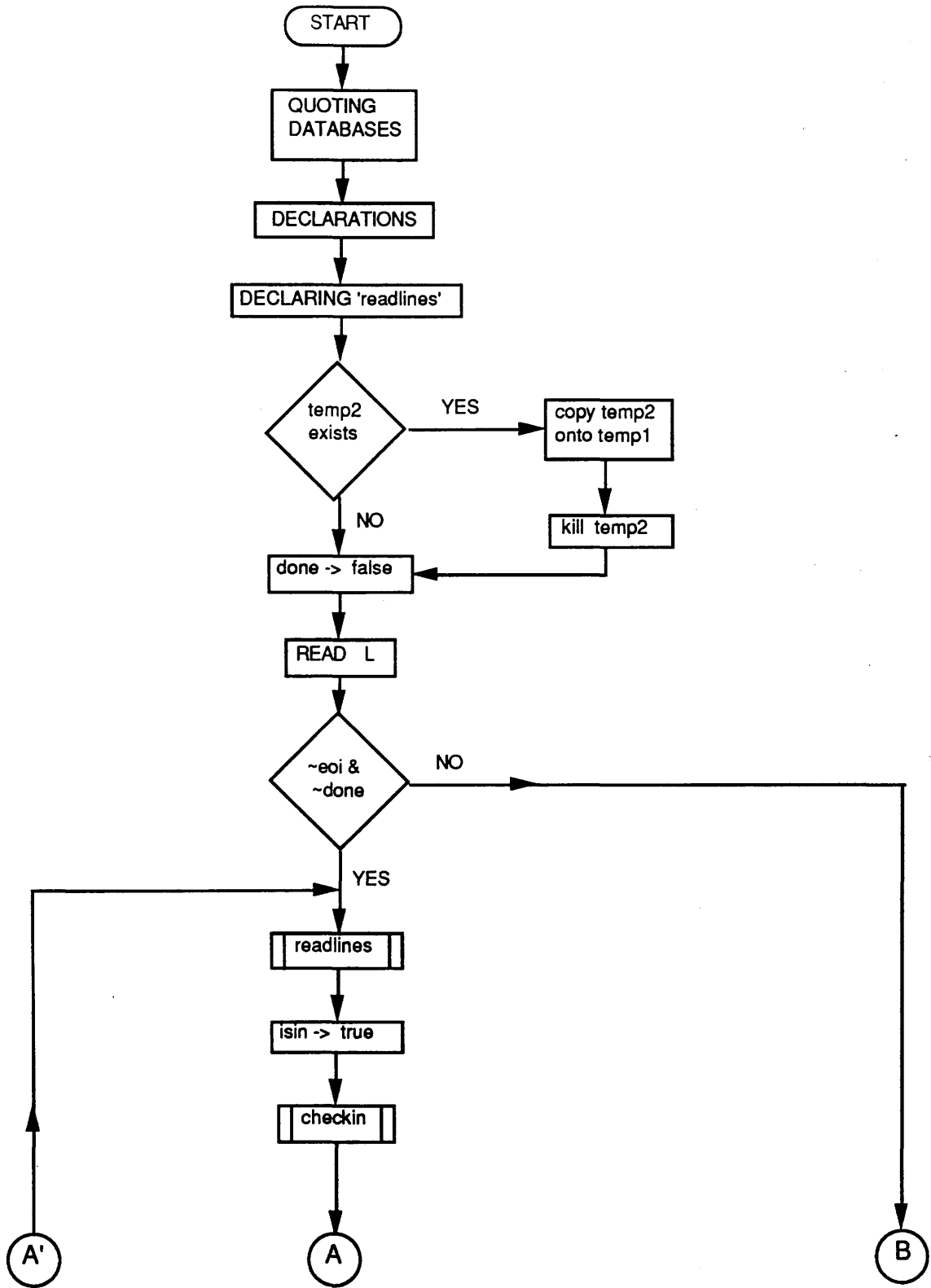
7.4.2.4 Operational Aspects of Configuration & Data Entry

At the beginning of this module, the '*global*' and '*MDB*' databases are checked for any specific details that might have been recorded for the particular map in hand. If the details described in Section 7.3.1 are found, then the system will retrieve the data for this map and assign the features to four temporal vectors - *Avec*; *Lvec*; *Pvec* and *Tvec* - according to their data types. A total counter is established by summing the dimensions of these four vectors.

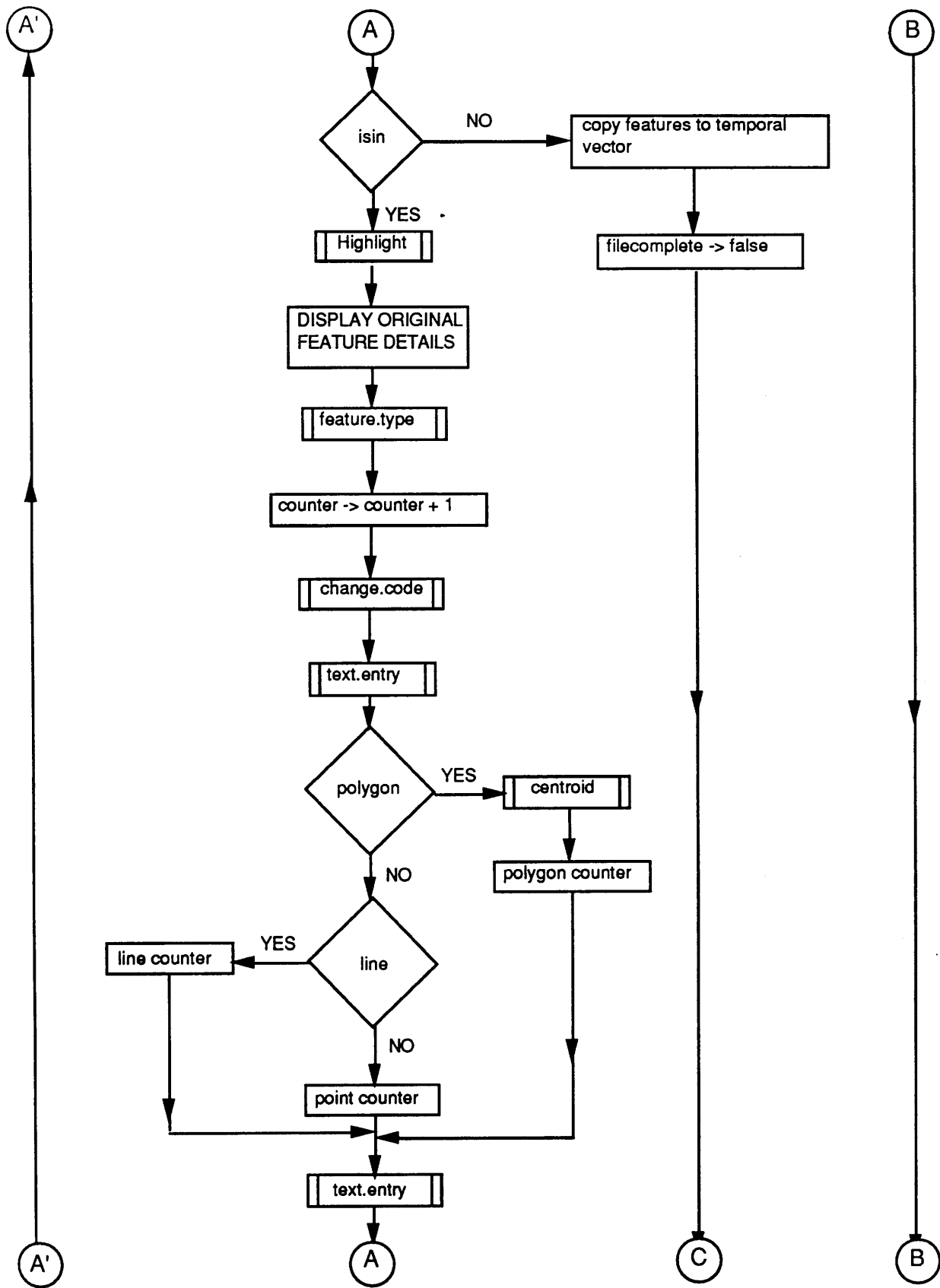
Since data entry and code transfer will often constitute a lengthy task to carry out in one session, the user might wish to quit the job at any time. This is allowed in this system by keeping track of all the operations taking place during a session and storing this information in three places as follows:-

- (i) The '*global*' database stores the counters of the four types of data in a table called 'thevar'.
- (ii) Data that have been dealt with during a working session are recorded in a temporary vector called 'temporal'.
- (iii) Features that have not yet been processed are stored in a temporary file named 'filename.temp2'.

At the start of any '*code.transfer*' session, a temporary file, called '*filename.temp1*', is set up which contains all those features which have yet to be coded. In the first run, this will be a copy of the original file. Subsequently it will be a copy of the '*filename.temp2*' file left by the previous session. After the copying of '*filename.temp2*' into '*filename.temp1*', file '*filename.temp2*' is erased.



Cont.



Cont.

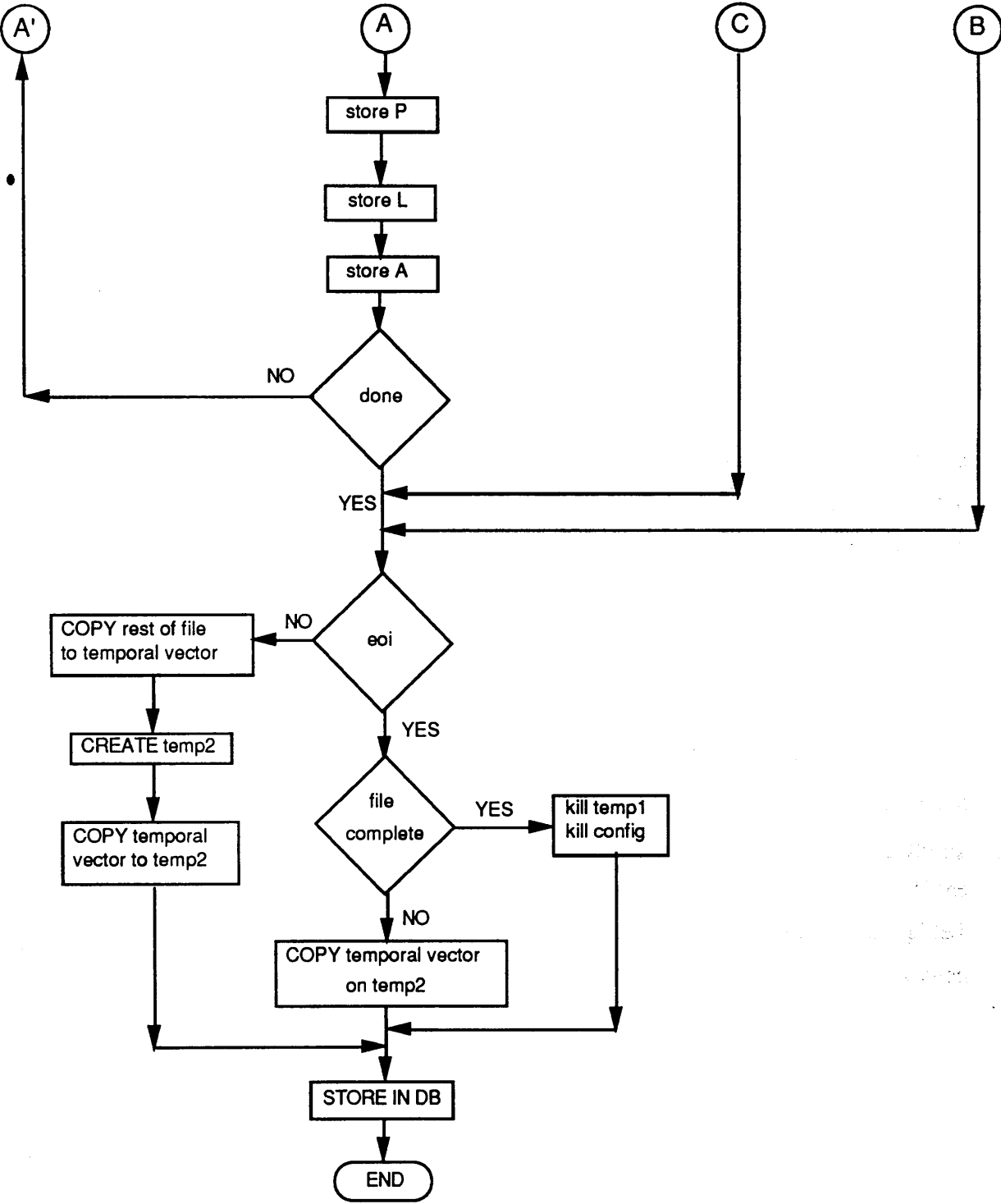


Fig. 7.7 Flowchart of procedure 'picking'

The declaration of all the variables needed throughout the module then follows together with the declaration of one local procedure, called '*readlines*'. This is a very simple procedure which, when given a file name, reads data from this file in a sequence of lines

containing X, Y values. While reading, the procedure checks the start and the end of every 'grouping' of segments (or link) and then copies their coordinate values into two vectors declared in the module scope. At the end, it returns all the lines read as a vector of strings.

Having read all the data relating to a feature, these data are then subjected to several operations. To start with, suppose that the user had chosen to zoom in on a particular zone of the map on the screen, then not all the features of the map can be seen. The system skips all the features that do not appear on the screen leaving them until a later stage. This checking is done by a procedure called '*checkin*' which returns a boolean value (true if the feature can be viewed or false if it cannot).

Any feature which lies entirely outside the screen limits has all its components recorded into the temporary vector automatically. But if it is partly displayed, then a message asks the user whether or not he can clearly identify the object, Fig. 7.8 shows the display of such a case on the screen. However, if the object cannot be identified, the feature components are also copied into the temporary vector. On the other hand, if the object lies in its entirety within the screen limits or if it can be identified by the user, it will be subjected to further operations.

The object to be processed should be distinguished from other objects on the screen, so that the user may know which object is undergoing processing. This is done by highlighting that particular feature by drawing small hexagons at each of its different vertices. If the object had been passed through the procedure '*checkin*', it would have been 'highlighted' there. The procedure responsible for doing so is called '*Highlight*' and has already been described in Chapter 6, Section 6.5.2.

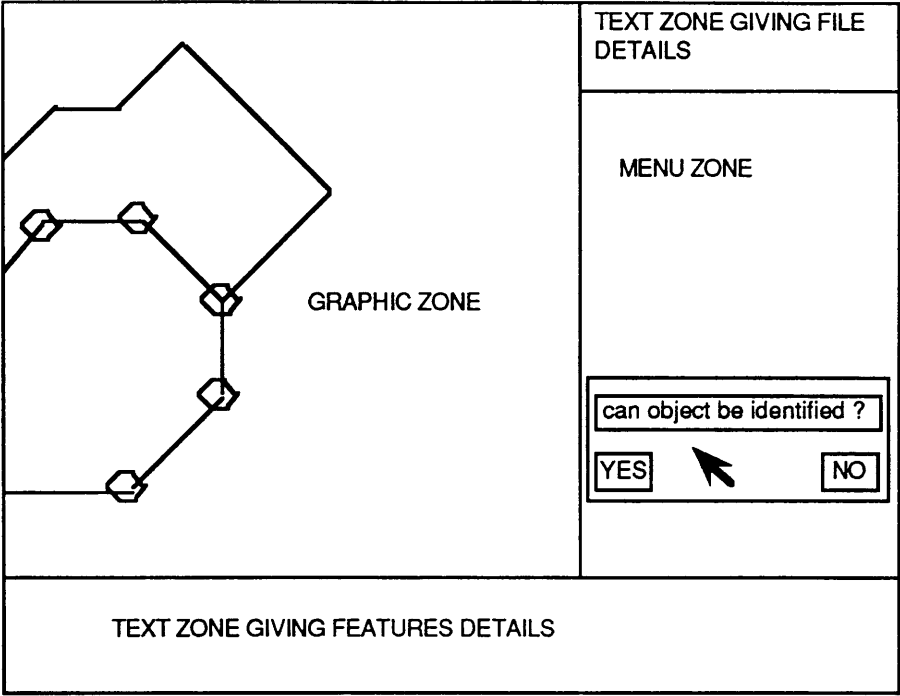


Fig. 7.8 'Highlight' in action

Next, the original details about the object (feature) are displayed in the text zone of the screen. Following this, a type menu is displayed in the menu zone to allow the user to specify the type to which this particular object belongs (see Fig. 7.9). Once this has been done, the counter of that particular feature type is increased by one.

Classification of features is the next stage. This is done by calling the procedure '*change.code*'. This procedure creates four successive menus to comply with the four levels of classification discussed in Chapter 5. Thus '*change.code*' opens database 'Code2' and retrieves sequentially the attributes from each level, subsequently forming the menus. From these, the user can choose those attributes which best describe the feature using the four menus and thus forming the feature code. Once a menu has been created, it is then stored in a temporary structure called 'menu'. This will increase quite substantially the speed of displaying this same menu on the second or any subsequent occasion.

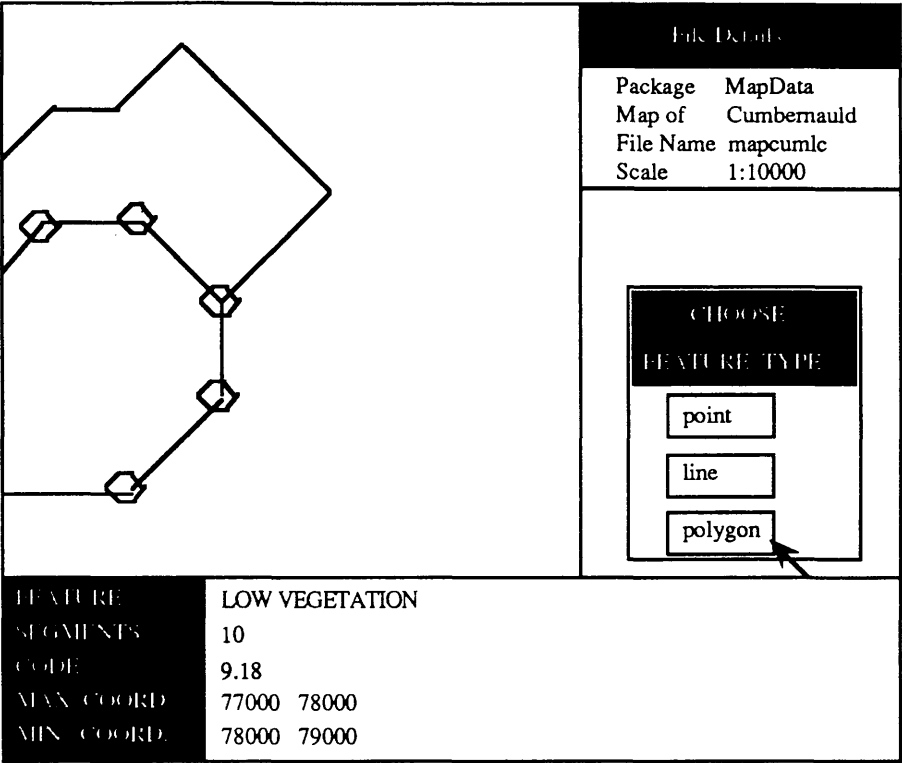


Fig. 7.9 Example of the use of the different zones of the screen

If the feature is of polygon type, further information is needed about it. The system will ask the user to introduce a 'centroid' which is done by positioning the pointer and then clicking the mouse anywhere inside the boundaries of the polygon (this is checked by the system as well). This will help in identifying the polygon and will help in determining the neighbouring and any enclosed polygons as well.

Finally, textual information, such as its name, or any comments about the feature, is then entered using the procedure 'text.entry'.

At the end, all the newly entered data are sorted and placed in different structures according to their types, and the user is given the option whether to proceed to the next feature or to quit the session. If the user decides to proceed, the operation continues by picking another object. On the other hand, should the user decide to quit, then the data counters are stored in the 'global' database, and the part of the entry file which has not been read yet is copied into file 'filename.temp2'. Then file 'filename.temp1' is removed (killed).

7.5 Procedures Used In This Module

There are some thirty procedures used in this module. Of these, sixteen are globals and the remainder are modules. In this Section, only the module procedures are described, the global procedures having been described in Chapter 6, Section 6.5.2. Fig. 7.10 shows the interlinks between the different procedures in this module.

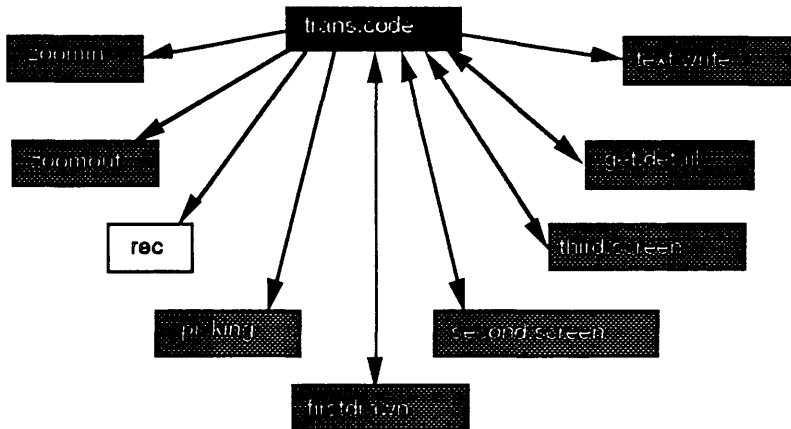


Fig. 7.10 The module interlinks

| <u>Procedure</u> | <u>Function</u> |
|------------------|-----------------|
|------------------|-----------------|

| | |
|--|--|
| <code>loc.print(Number, Xlocation, Y-clearance, text)</code> | |
|--|--|

The aim of this procedure is to place a piece of text in a particular place on the screen. It works in conjunction with the procedure 'second.screen'. The identifiers passed to 'loc.print' are:

- 'Number' : which is the number of text items in the list;
- 'Xlocation' : which is the distance at which the text should be placed measured in screen units starting from the left edge of the screen;
- 'Y-clearance' : which is the allowed spacing in the vertical direction (in screen units) between two consecutive lines of text; and
- 'text' : is the text to be displayed.

| | |
|------------------------------|--|
| <code>second.screen()</code> | |
|------------------------------|--|

This procedure, together with 'loc.print', serves to display the list of questions to be answered by the user about the map details. It makes use of an editing procedure supplied by the 'utilities' database, and returns the answers as a vector of strings. After all the answers have been supplied, 'second.screen' calls up the procedure 'message.proc' to allow the user to alter the information should any error occur. Finally, 'second.screen' copies these details into the file 'config'. Fig. 7.11

shows this procedure's calls to other procedures.

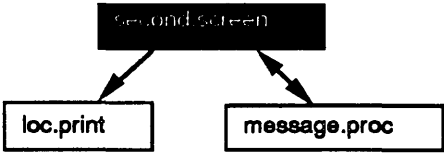


Fig. 7.11 'second.screen' calls

get.detail(Answers)

This procedure takes the answers of '*second.screen*' and transforms them into reals and integers according to their original types. It returns a vector of all transformed data called '*Tanswers*'. Fig. 7.12 shows the calls made by this procedure.

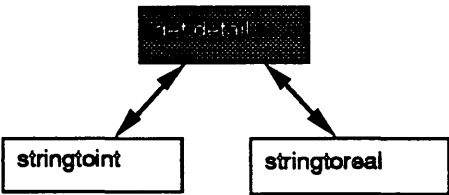


Fig. 7.12 'get.detail' calls

third.screen(Tanswers)

This procedure divides the screen into four zones to allow three different types of display:- graphics; texts and menus. Fig. 7.6 shows the design of the screen implemented by this procedure. The upper part of the menu zone is allocated to display the file details. The procedure then draws the grid of the map and displays the north direction. Fig. 7.13 illustrates the procedure's calls to other procedures.

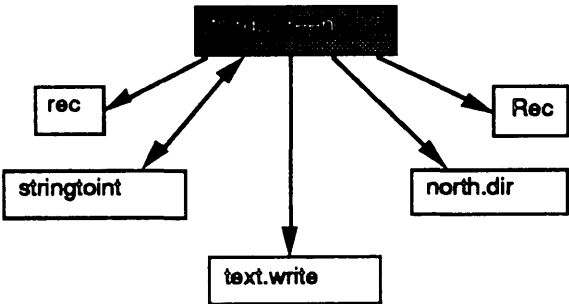


Fig. 7.13 'third.screen' calls

drawfile(filename, Grid, bool)

As the name implies, '*drawfile*' is a procedure that reads a file into the system and draws it on the screen if the '*bool*' is true. The procedure starts by creating a table which, later on, will store the picture of the whole map in the database. This table is called '*Variables*' and is stored

in the 'global' database. Then it declares the different variables needed within the procedure scope. Also it declares a temporary vector which will hold all the data file's contents as strings, and, at the end of the procedure, it stores this vector in a file called '*filename.temp1*'. This is done to ensure that the original data file remains untampered with. It can be removed at a later stage after all the data of the file has been passed through the second module (described in Chapter 8). Within this procedure scope, another procedure called '*readlines*' is declared. It searches for the beginning and end of each chain (i.e. each group of segments forming a complete feature) within the file and reads its various coordinate values and then creates pictures from them. After having read all the file, the pictures of all the features are grouped into one total picture, which is then stored in the database.

`first.drawn(xmin, xmax, ymin, ymax)`

The jobs carried out using this procedure are to call the '*drawfile*' procedure; to assist in passing the coordinate values of the corners of the map; and to scale them to the window limits. Fig. 7.14 illustrates the calls made by this procedure.

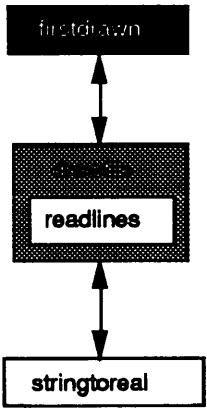


Fig. 7.14 'firstdrawn' calls

`change.code()`

It is the job of this procedure to attribute codes to all features according to the classification system implemented in this project. Fig. 7.15 illustrates the calls needed within this procedure.

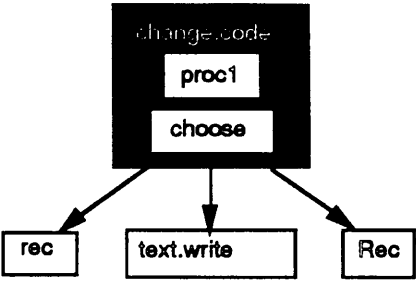


Fig. 7.15 'change.code' calls

At the beginning, the procedure creates a window in which details regarding the progress of the coding operation are displayed to the user. Then it makes use of two other procedures declared at this procedure scope. The aim of the first one, called '*scan.table*', is to scan through a table and extract or list all the features which it encounters, returning a vector of pointers and their counts. The second procedure is a recursive procedure and is called '*choose*'. It calls '*scan.table*' four times, each representing one level of the classification scheme. It then forms a menu of what it finds in each level. Any chosen element of this text-based menu returns a pointer to a lower level table, which in turn is then scanned and so on. Since each element of the menus is assigned with a particular code, traversing through the hierarchy from level 1 to level 4 will thus form the total code of the feature. This code is the value returned by the procedure '*change.code*'.

Fig. 7.16 shows the sequence of the operations as viewed on the screen.

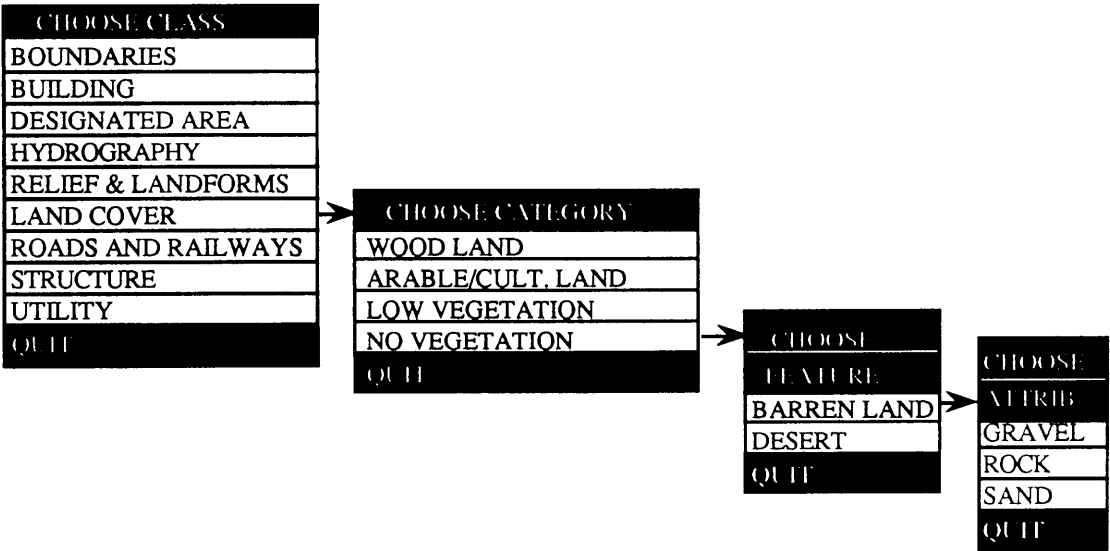


Fig. 7.16 The coding sequence using menus

text.entry(text1, text2)

This procedure is aimed at allowing text to be entered into the system. It makes use of a procedure named '*s.editor*' which is called from the '*utilities*' database. 'text1' is used to represent an instruction to the user about the text that should be entered and 'text2' contains the original version of the text which may be modified. This helps the user not to re-enter the same text more than once. Fig. 7.17 shows the calls made by this procedure.

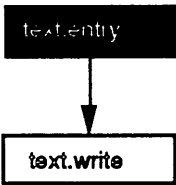


Fig. 7.17 'text.entry' calls

centroid(vecx, vecy, xmin, xmax, ymin, ymax)

This procedure aims at introducing centroids to polygons to make it easy to initiate links with other features. (This will be explained in more detail in Chapter 8). It starts by displaying a message asking the user to locate the centroid by placing the pointer at an appropriate position somewhere in the middle of the polygon and then clicking the mouse. After receiving the coordinate values from the mouse (in screen coordinates), they are then transformed into map coordinates. Then the procedure checks whether these coordinate values lie inside the polygon. If they do, the procedure returns these values to the main body of the module '*trans.code*' which stores these values in the polygon's record. Otherwise it displays a message to the user to try again. Fig. 7.18 shows the '*centroid*' calls to other procedures.

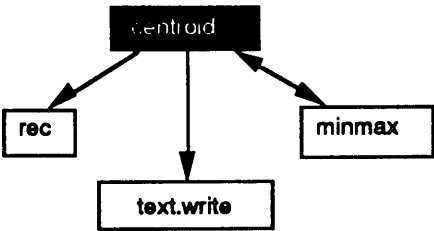


Fig. 7.18 'centroid' calls

text.window()

This procedure's job is to display the details of the map as they are retrieved from the '*MDB*' data base.

picking(file.details)

Fig. 7.7 illustrates the line of action of this procedure as a flowchart while Fig. 7.19 shows the calls made by this procedure to other procedures.

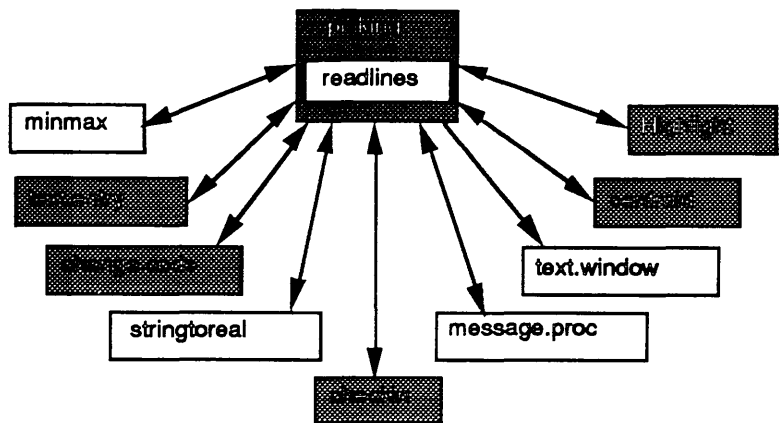


Fig. 7.19 'picking' calls

At the start, the procedure opens database '*global*' to retrieve details about the file being read. It then opens database '*MDB*' and establishes counters for the different types of features and a total counter for all of them. Next, declarations of all variables needed in the procedure are made. As has been stated before in the discussion of procedure '*drawfile*', the sequential selection of features is carried out from file '*filename.temp1*'. Before this is done, the existence of the file '*filename.temp2*' is checked. If it does exist, the procedure will copy this file into file '*filename.temp1*' and erase '*filename.temp2*'. In the next step, the procedure will create a temporary vector which will record any feature which the system is not able to process. This vector is called '*dummy.vec*'. Data are then read into the system using the procedure '*readlines*', the same procedure as that used previously by the procedure '*drawfile*'. A loop is then repeated under two conditions, the first being that the reading of data has not reached the end of the file, and the second that the user has not issued a quit command.

As the procedure reads a feature, various operations are carried out on it. At the beginning, the minimum and maximum extent of the feature are determined using the procedure '*minmax*' and then the results are

sent to the procedure '*checkin*' to determine whether the feature can be identified or not. If it can, then the procedure '*Highlight*' is called and the details of the feature, read from the original file, are displayed in the text zone of the screen. Then a series of calls are made to the following procedures:- '*feature.type*'; '*change.code*'; '*text.entry*'; '*centroid*'; and '*polygon*'. The features and their attributes are stored in one of the following structures:- A-holder; L-holder and P-holder (at this stage, texts are being treated as though they were of type '*points*'). However, if the feature cannot be identified, then it will be copied to '*dummy.vec*' and, at the end of the session, this vector is copied into file '*filename.temp2*'. Then the user is offered the choice of either proceeding or quitting. If the user decides to continue, then another feature is selected. On the other hand, if the user decides to quit, then the three structures are stored in the '*MDB*' database.

7.6 Summary

In this chapter, a detailed description of the Data Entry and Code Transfer module has been provided. Details have been given about the digitizing method used in this system and the subsequent code transfer operations. Following this, a listing has been provided of all the various procedures used in this module together with a brief description of their individual functions. At the end of running this module, data which have been passed through the code transfer operation are then stored in the database and are ready for the processing activities which can be executed by the two remaining system modules - Cartographic Representation and Data Retrieval. These will be discussed in Chapters 8 and 9 respectively.

CHAPTER 8

CHAPTER 8: CARTOGRAPHIC REPRESENTATION MODULE

8.1 *Introduction*

In this module, the main concern is to represent the data in a cartographic form so as to allow the final representation of the analysis to be displayed and understood in the same way as any other map. Various cartographic representation activities are introduced for the different types of features by use of separate menus containing all the possible ways of representation which have been implemented in the system. For example, in the present system, polygons have two different styles of representation - hatching or filling - each of which is presented in a separate menu. Whereas for lines, representations can be chosen from a single menu containing several different line styles. Point entities also have only a single menu to choose from but this menu is linked to three different procedures. Fig. 8.1 illustrates the different operations needed to generate the symbols available for each of the three main feature types stated earlier.

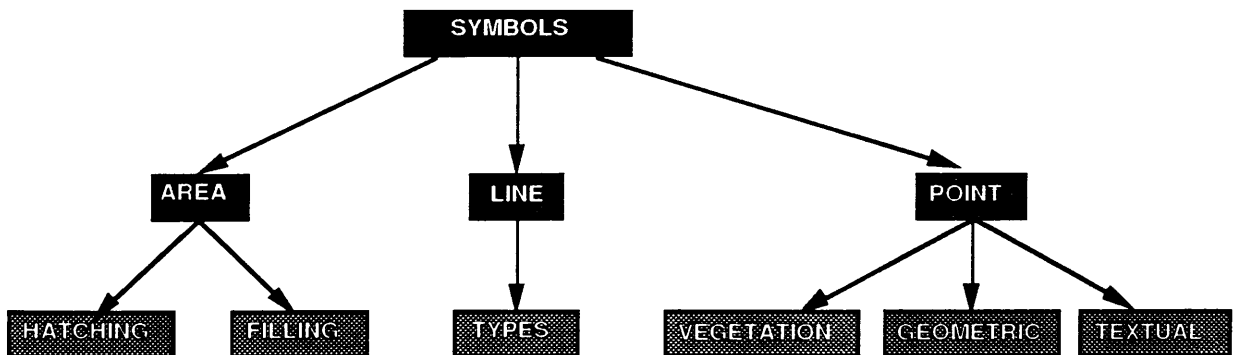


Fig. 8.1 Types of symbolisation

8.2 *Cartographic Representation of Features*

At the start of this module, and before any cartographic treatment has been applied to any feature, a snap shot of the whole of the screen is recorded in a temporary 'image'. The system then displays a menu offering a choice between the three types of data, as shown in Fig. 8.2. If, for example, 'POLYGON' is chosen from the menu, first a search of the database will be carried out for polygon features only and then those procedures related to polygon features are called. Fig. 8.3 shows the module's calls, while Fig. 8.5 gives the flowchart for this module.

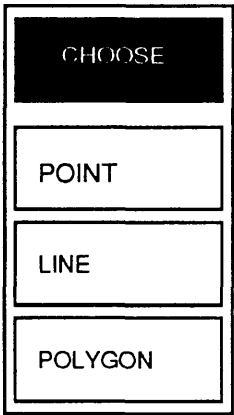


Fig.8.2 The type menu

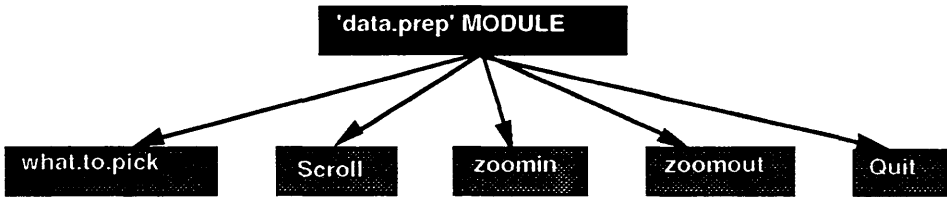


Fig. 8.3 The module calls

The cartographic representation of features is divided into three sections in accordance with the three types of feature handled by the GIS system. Fig. 8.4 shows the calls required to select the specific class or type of feature needed.

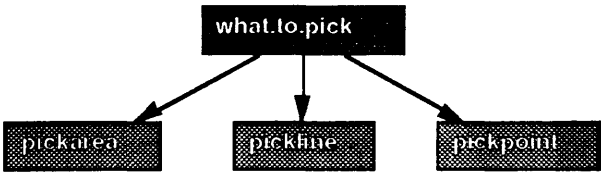


Fig. 8.4 Types of feature selections

Firstly, the description of those general procedures needed by all three sections are presented in Section 8.2.1. The description of the procedures available for polygon representation then follow in Section 8.2.2. Next, those procedures required for the representation of line features are described in Section 8.2.3. Finally those procedures required for point representation are examined in Section 8.2.4. The procedures used in this module will now be described.

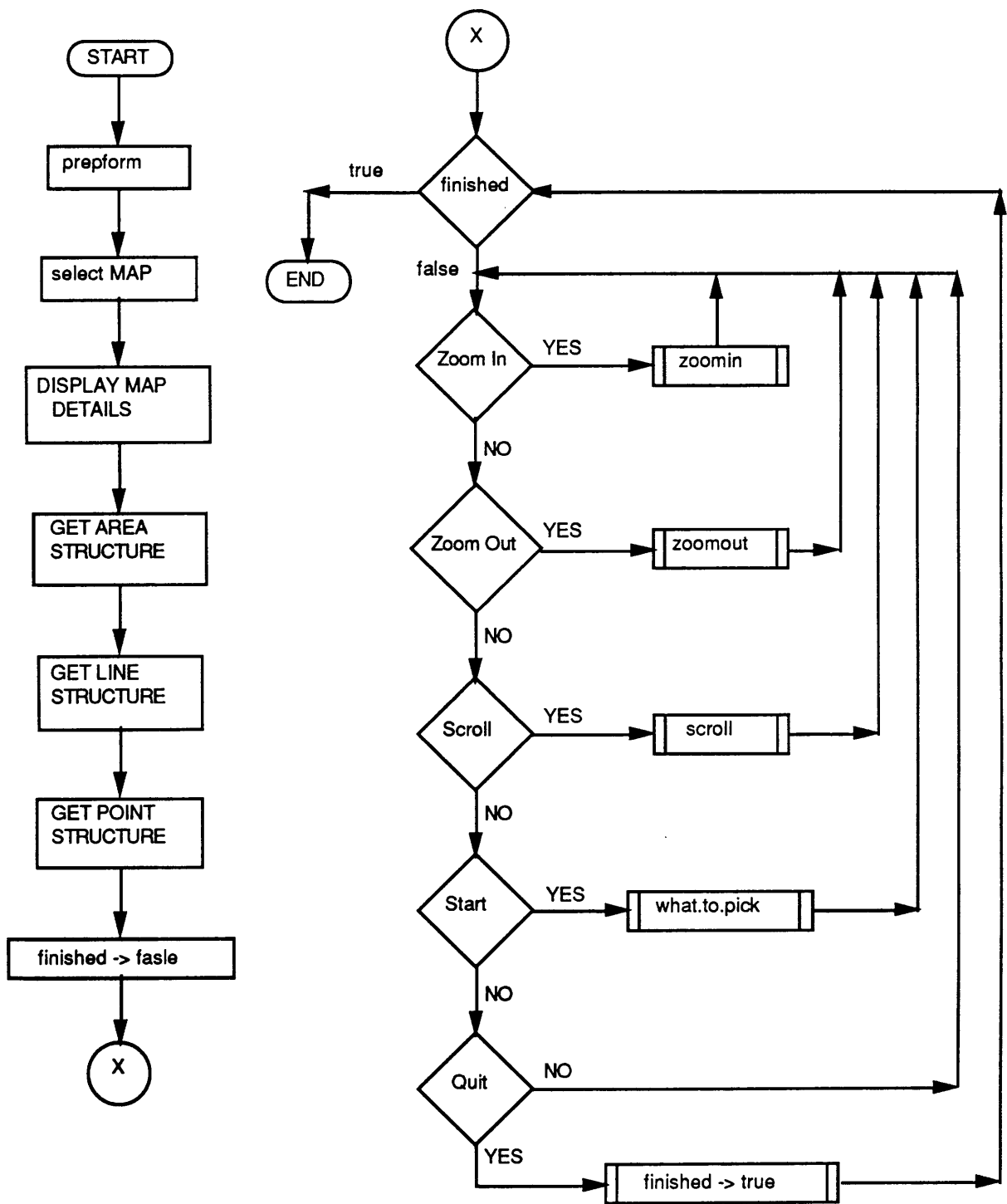


Fig. 8.5 The flowchart of the module 'data.prep'

8.2.1 General Procedures

The general procedures employed in this module play the same rôle as the global procedures used for the overall system. They are defined as module utility procedures so

that they can be called from anywhere within the module scope. There are five of these procedures, namely : 'linepara'; 'lineint'; 'perpenline'; 'drawline' and 'dashing'.

Procedure

Function

linepara(xA, yA, xB, yB)

This procedure is intended to determine the coefficients of a line. Since a line can be represented by an equation of the type: ' $y = mx + n$ ', the coefficients ' m ' and ' n ' can be determined for a line passing through two points $A(x_A, y_A)$ and $B(x_B, y_B)$ by the two equations. (see Fig. 8.6)

$$m = Dy / Dx = (y_B - y_A) / (x_B - x_A)$$

$$\text{and } y_i = (y_A * x_B - y_B * x_A) / (x_B - x_A)$$

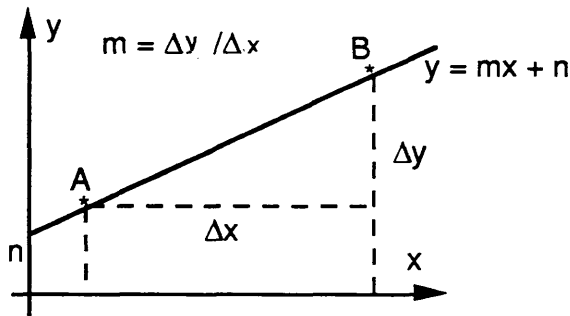


Fig. 8.6 The parameters of a line passing through 'A' & 'B'

The procedure then returns these parameters in a vector of two elements.

lineint(a, b, c, d)

Since given the coefficients of two lines, their point of intersection, if any, can be determined by the following two equations. (see Fig. 8.7)

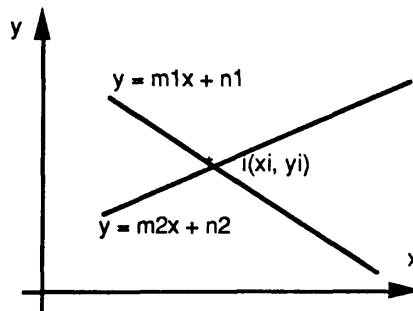


Fig. 8.7 The intersection 'i' of the two lines 'y1' and 'y2'

$$x_i = (n_2 - n_1) / (m_1 - m_2)$$

and $y_i = (m_1 * n_2 - n_1 * m_2) / (m_1 - m_2)$

The coordinate values of the point of intersection 'i' are then returned in a vector of two elements.

`perpenline(x, y, m)`

The aim of this procedure is to determine the coefficients of a line 'W' perpendicular to another known line 'V' passing through a specified point 'P' (see Fig. 8.8). This procedure then takes the coordinate values of 'P' and the gradient 'm' of 'V', and calculates the coefficients of 'W' as follows:

$$m' = -1/m$$

and $n' = y - m' * x$

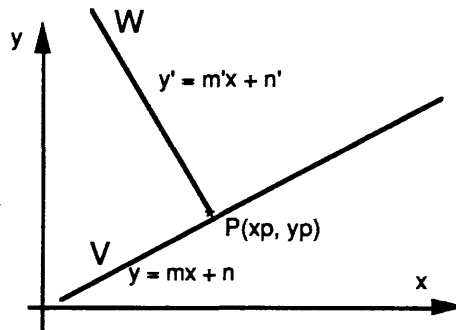


Fig. 8.8 'W' perpendicular to 'V' at 'P'

The procedure returns these values as a vector of two elements.

`drawline()`

This procedure has been described earlier in Chapter 6, Section 6.5.2

`dashing(x1, y1, x2, y2, dash, gap)`

This is another graphics procedure which returns a picture of a dashed straight line segment joining two points $A(x_A, y_A)$ and $B(x_B, y_B)$, using the dash and gap values specified for the drawing. The procedure calculates first the distance between the two points and then determines the angle ' α ' formed between the line passing through these two points and the horizontal, (see Fig. 8.9). Next the sizes of the gaps and dashes are adjusted so that the length of the segment is a multiple of a period (a period is one dash and one gap). For a better display, each dashed line should start with a half dash and end with another half dash, thus forming one complete dash.

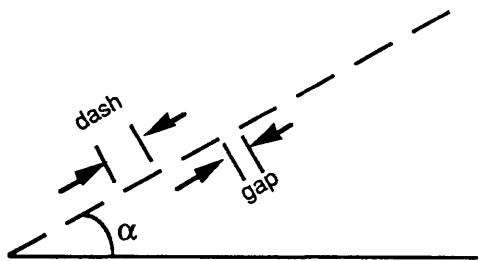


Fig. 8.9 Dashes & gaps

The coordinate values of the two points forming a dash are then calculated and stored in two vectors, one for the x-values and the other for the y-values. Finally a picture of the segment is returned using procedure 'drawline'. Fig. 8.10 shows the procedure call.

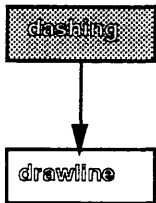


Fig. 8.10 'dashing' call

8.2.2 Polygon Representation

If the option 'POLYGON' has been chosen from the menu displayed in Fig. 8.2, then the procedure 'pickarea' is activated. This then retrieves the structure 'A-holder' from the database and, in a sequential manner, starts selecting polygons and highlighting them. After a polygon has been selected, another menu is displayed showing two different methods - hatching and filling - which may be used for polygon fill representation. Fig. 8.11 shows the menu displayed at the outset of choosing 'pickarea'.

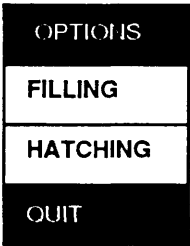


Fig. 8.11 Menu of options for polygons

There is a list of procedures which are used in conjunction with polygon processing and called by 'pickarea'. These are: 'Highlight'; 'minmax'; 'text.write'; 'message.proc'; 'nabor'; 'area.prepare'; 'area'; 'perimeter'; 'inclave'; 'polycheck' and 'area.menu'. Fig. 8.12 illustrates these calls, in which those procedures represented by filled boxes call

lower-level procedures, while the procedures in plain boxes do not.

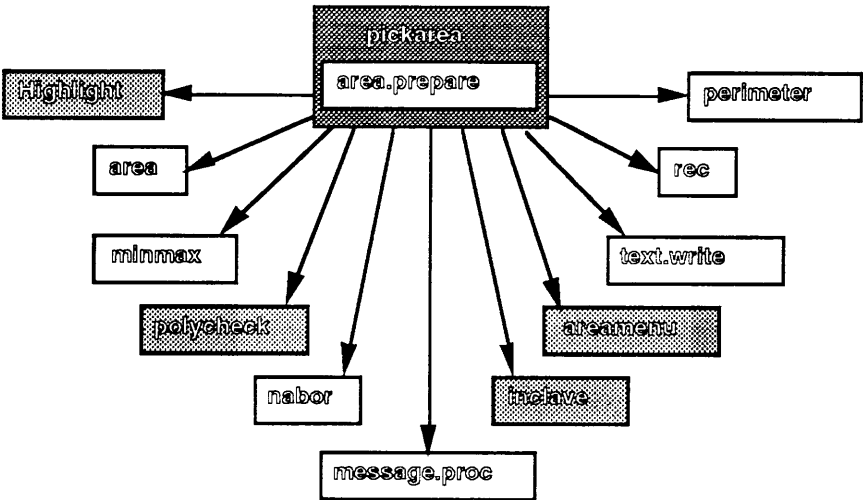


Fig. 8.12 'pickarea' calls

The first five procedures have already been discussed in Chapter 6, Section 6.5.2, and the last three make use of the lower level procedures. Thus *'inclave'* calls *'linepara'*; *'lineint'* and *'minmax'*; *'polycheck'* calls *'innerangle'*; and finally *'areamenu'*, has two jobs, the first is to hatch polygons and the second is to flood polygons with patterns, an operation which is also known as filling. This *'areamenu'* procedure calls any one of sixteen lower level procedures. These are *'linepara'*; *'lineint'*; *'sorting'*; *'drawline'*; *'dashing'*; *'hatchpoly'*; *'hatchtype'*; *'hatchangle'*; *'hatchspace'*; *'rec'*; *'default.menu'*; *'hatch'*; *'plotsym'*; *'symscale'*; *'filling'* and *'fill'*. The polygon hatching procedures (*hatch*, *hatchpoly*, *hatchtype*, *hatchangle*, *hatchspace*, etc.) will be described later in Section 8.2.2.1, while the polygon filling procedures (*symscale*, *plotsym*, *filling*, *fill*) will be described later in Section 8.2.2.2. Fig. 8.13 shows these *'areamenu'* calls.

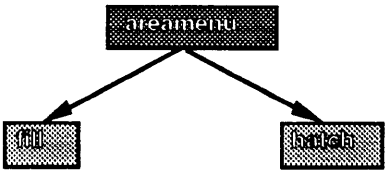


Fig. 8.13 'areamenu' calls

However, the description of procedures starts with those of the higher level which are called by *'pick.area'* mentioned above and shown in Fig. 8.12.

Procedure

Function

area(vecx, vecy)

This procedure calculates the area of a polygon by simple calculation of the area lying underneath each segment of the polygon regardless of its sign, Fig. 8.14 shows the process of the calculation for an individual segment. It first determines the least y value (ymin), on which the calculations of areas will be based. Then, for every segment, the differences in the x-direction (Dx) and in the y-direction (Dy) of the end points are determined, and whether these values are negative or positive. Fig. 8.15 shows how areas can be either positive or negative with respect to the difference in the x-direction.

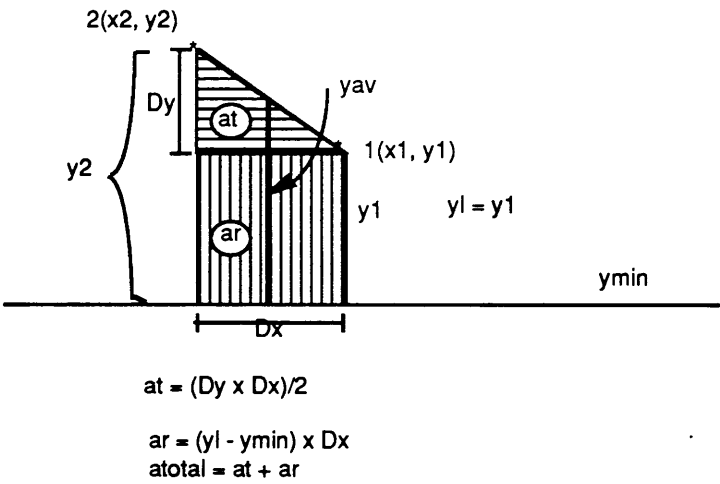


Fig. 8.14 The calculation of area under a segment

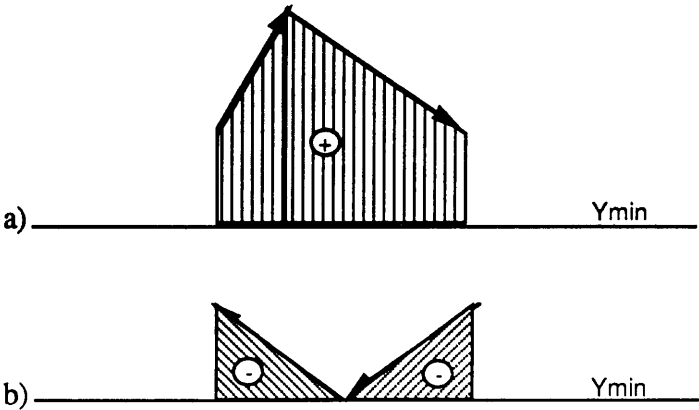


Fig. 8.15 The direction of the polygon lines decides the sign of area calculations

As each segment passes through the area calculation process, areas are accumulated including whether they are positive or negative and finally the whole area is obtained. Fig. 8.15(a) shows the positively calculated areas while Fig. 8.15(b) shows the negatively calculated ones. Finally Fig. 8.16 illustrates the area resulted from the calculation procedure.

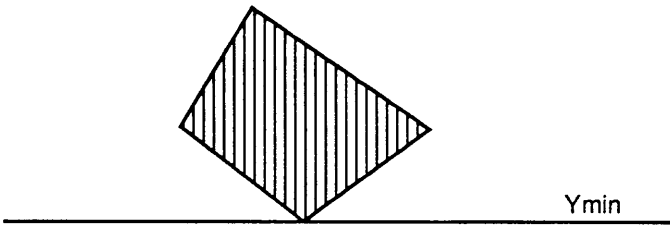


Fig. 8.16 The vertical hatchings represents the final area

The procedure returns the total area value of the polygon.

`perimeter(vecx, vecy)`

Knowing that the distance between two points can be determined by the equation:

$$\text{length} = \sqrt{Dx^2 + Dy^2}$$
 (see Fig. 8.17)

then, by accumulating the lengths of all the segments forming the polygon, the length of the perimeter is obtained.

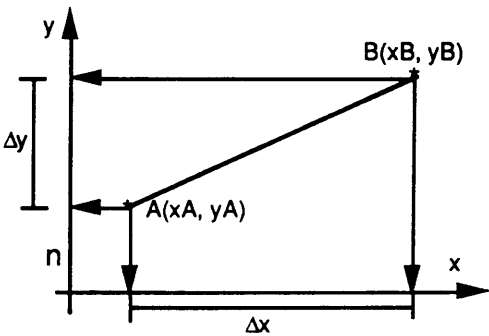


Fig. 8.17 Length determination

`nabor(vecx1, vecx2)`

This procedure takes the two vectors of two polygons and checks whether or not these two polygons are neighbours. This procedure follows an earlier check on the two polygons made in procedure 'inclave' as to whether or not one of them is contained within the other. The procedure 'nabor' checks the compatibility of the

segments in both polygons. If any two segments, comprising one from each polygon, match, then these two polygons are neighbours and the procedure returns 'true'. Otherwise it returns 'false'.

sorting(veca, vecb)

This is not a normal sorting procedure, because only the first vector, 'veca', is to be sorted as usual whereas the second vector, 'vecb', is to be arranged in accordance with 'veca'. For example, the numbers within 'veca' and 'vecb' given below represent dummy values to be sorted by this procedure.

| <u>i</u> | <u>veca</u> | <u>vecb</u> |
|----------|-------------|-------------|
| 1 | 5 | 4 |
| 2 | 2 | 7 |
| 3 | 10 | 8 |
| 4 | 3 | 12 |

Since each of these vectors represents a set of coordinate values, they should always have the same link. Thus, for example veca(i) should always be coupled with vecb(i), i.e. from the example above, 5 should always be coupled with 4, and 10 with 8 and so on. So, after sorting the values in 'veca' into a numerical order, these vectors become :-

| <u>i</u> | <u>veca</u> | <u>vecb</u> |
|----------|-------------|-------------|
| 1 | 2 | 7 |
| 2 | 3 | 12 |
| 3 | 5 | 4 |
| 4 | 10 | 8 |

This is done by using the standard 'swap' procedure and a sorting procedure 'sort' which checks the values of 'veca' and places them in an order of ascending values. Whenever two values in 'veca' are interchanged, their corresponding values in 'vecb' are also interchanged. A record of the interchanging activities is used to keep track of the interchanging activities taking place and when this becomes zero, the sorting operation is complete.

inclave(vecx1, vecy1, vecx2, vecy2, oxc, oyc)

The aim of this procedure is to determine whether or not a polygon is enclosed totally by another. A series of checks are made for this

purpose. The first check is to compare the maximum and minimum coordinate values of both polygons to determine which is likely to be enclosed within which. Then these maxima and minima are compared for overlapping. If they are apart, then these polygons fail this procedure. Next, checks are made for the segment intersections of both polygons. If any are found, then these polygons do not meet the conditions for enclosure. A further check is made to determine whether or not the two polygons are in the situation illustrated in Fig. 8.18.

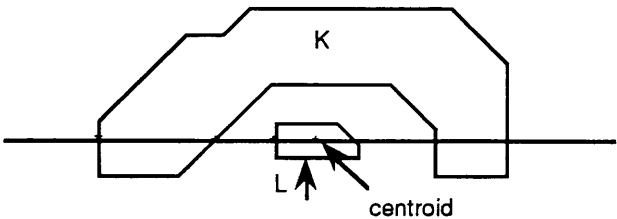


Fig. 8.18 A special case for a non-enclosed polygon

These two polygons meet the above conditions but still, 'L' is not enclosed by 'K'. Thus, an imaginary horizontal line passing through the centroid of 'L' is generated, and the intersections of the line with both polygons are established. So, if the number of intersections on the one side of the centroid of 'L', is a multiple of two, then 'L' does not lie inside 'K' and the procedure returns 'false'. Otherwise it returns 'true'.

Procedure 'inclave' uses three other procedures as illustrated in Fig. 8.19.

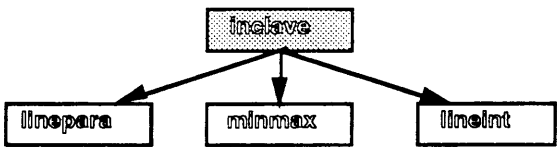


Fig. 8.19 'inclave' procedure calls

`polycheck(vecX, vecY)`

This procedure checks whether or not an individual polygon had been closed and whether a gap still exists in its perimeter. The following examples of polygons shown in Fig. 8.20 represent polygons which have not been closed.

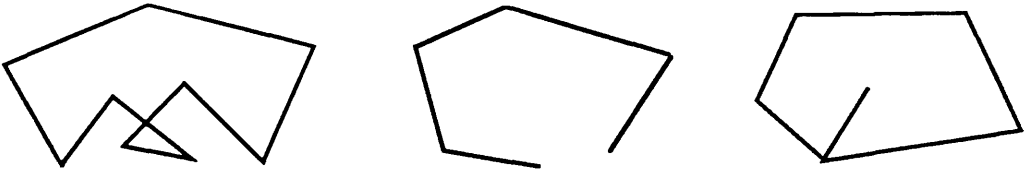


Fig. 8.20 Non-closed polygons

A polygon is said to be closed if the sum of its inner angles satisfy the following condition:

$$\text{sum of angles} = 90 * ((2 * \text{No of sides}) - 4)$$

The calculation of angles is made within a loop by calling the procedure 'innerangle' for each pair of segments, (see Fig. 8.21). These are then summed and tested to ensure that the result of the above equation to an accurate to $\pm 0.001^\circ$, this is done due to the rounding of numbers during the mathematical operations resulted from the precision of the processor.

This procedure can cope with polygons where their vertices were digitised in either direction (clockwise or anticlockwise). The procedure then returns 'true' if the polygon is closed or 'false' if it is not.

`innerangle(x1, y1, x2, y2)`

Given two segments, the inner angle ' α ' formed by these two segments, shown in Fig. 8.21, can then be calculated using the bearings of the segments as follows:

$$\alpha = \beta_2 - \beta_1$$

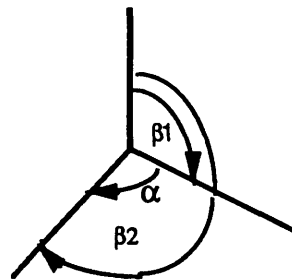


Fig. 8.21 The bearings of a vertex

The procedure returns the calculated angle as real.

8.2.2.1 Hatching Polygons

Hatching can be carried out according to the specific characteristics selected from a default menu which is displayed immediately after the option 'HATCHING' has been chosen from the menu in Fig. 8.11. The user can either accept the default settings for hatching or change the parameters.

If the user chooses the default settings, as displayed in Fig. 8.22, then the program will proceed with the calculation of the different intersections and transformations needed. Otherwise, if the user is not satisfied with the default settings, these can be changed by first selecting the command 'Change' from the menu using the pointer and the mouse. If this is done, the user can change any of the three setting parameters at a time, these being:

- a- the line type used for hatching - implemented by placing the pointer in the box marked 'type' and then clicking the mouse;
- b- the orientation of the hatched pattern (the angle at which the lines are drawn); - which is activated by placing the pointer in the box marked 'slope' and then clicking the mouse; and
- c- the density or spacing allowed between the lines - which is selected by placing the pointer in the box marked 'space' and then clicking the mouse.

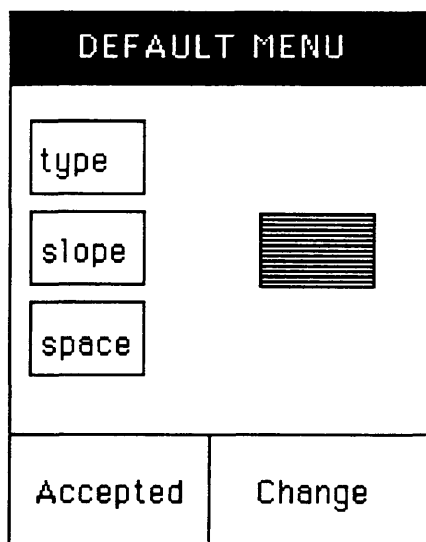


Fig. 8.22 The default menu for selection of parameters for hatching

a) If the line type is being chosen, then a menu of the different line types which may be used for polygon hatching is displayed (see Fig. 8.23). The user may then choose the type that is regarded as the most suitable for the particular polygon in hand.

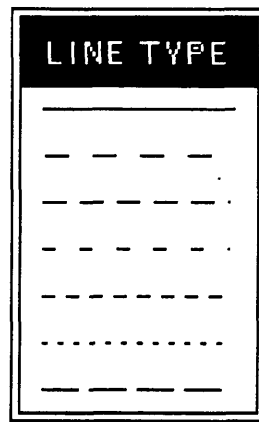


Fig. 8.23 The line type menu for polygon hatching

b) On the other hand, if the orientation setting is chosen, then a window for the change of slope is displayed in which the user can alter the default setting of the slope clockwise or anti-clockwise, in increments of 5° or 15° . Fig. 8.24 shows the menu with a set of arrows which are used to alter the orientation of the line pattern.

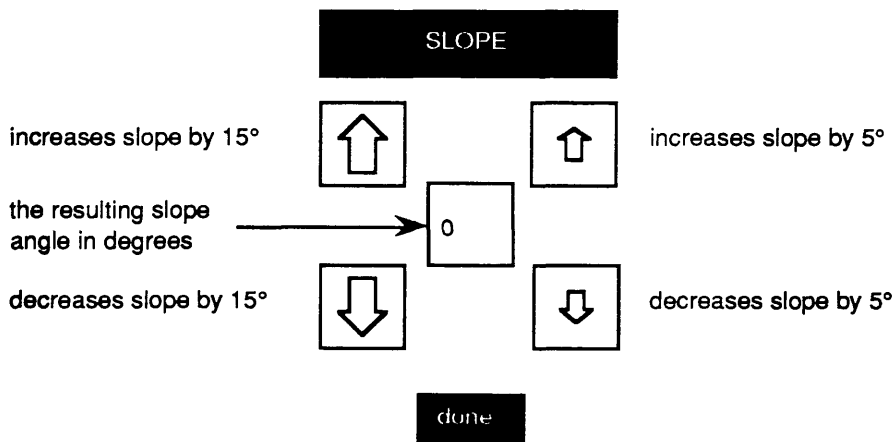


Fig. 8.24 The menu used for selecting the orientation of the hatched lines

c) The third option is to choose to change the spacing between lines. This also can be achieved using a third menu displaying different spacing settings as shown in Fig. 8.25, from which a specific one can be chosen.

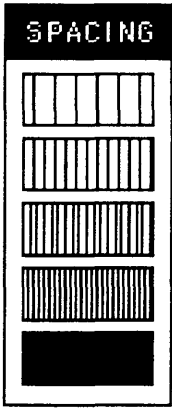


Fig. 8.25 The spacing menu

After having chosen the parameters required for hatching a specific polygon, the system will carry out the job using the specified line type, slope and spacing.

The procedures used to implement hatching are as described below:

| <u>Procedure</u> | <u>Function</u> |
|------------------|--|
| default.menu() | This procedure retrieves and activates procedure 'Default.Menu' from the database. It enables the user to choose alternative settings for hatching polygons. Otherwise the default settings are retained. The procedure returns the particular values selected for hatching in a vector of four elements. |
| hatchtype() | This procedure retrieves and activates the menu 'Htype' from the database which enables the user to select a suitable line type for polygon hatching. It returns information about the particular line type chosen, i.e. the lengths of the dashes and gaps (giving a negative value if the line is continuous). |
| hatchangle() | This procedure retrieves and activates the menu 'Hangle' from the database which enables the user to choose the orientation, i.e. the angle at which hatching lines should be drawn. It returns the value of the chosen angle in degrees. |

`hatchspace()`

This procedure retrieves and activates the menu 'Hspace' from the database, thus allowing the user to choose a convenient spacing between hatching lines. It returns the value of the chosen spacing.

`hatchpoly(vecx, vecy, vecI, space, angle, dash, gap)` [Harrington, 1987]

This procedure is used to actually implement the polygon hatching after the selection of the settings has been made by the user. It makes use of five other procedures, namely: '*linepara*'; '*lineint*'; '*sorting*'; '*drawline*' and '*dashing*'. These calls are illustrated in Fig. 8.26.

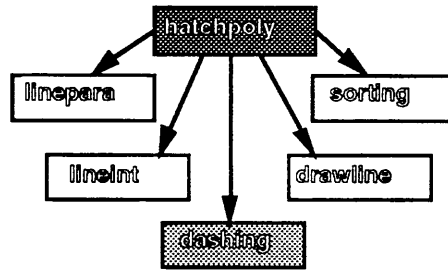


Fig. 8.26 The calls of procedure 'hatchpoly'

The procedure takes two vectors for the x- and y-coordinate values which are the coordinate values of the vertices of the specific polygon to be hatched. It also utilises the addresses for other smaller polygons that may exist within this polygon (given in vector '*vecI*') so that a void may be left in the hatching pattern. Also it takes the particular values selected for the line type, the gradient (or slope) of hatching and spacing between the lines, already discussed above under the headings *hatchtype*, *hatchangle* and *hatchspace* and performs the hatching operation. If the slope is a non-zero value, a transformation procedure is applied to all the coordinate values and the original vectors are replaced by their transformed values in the rotated coordinate system x' , y' (see Fig. 8.27).

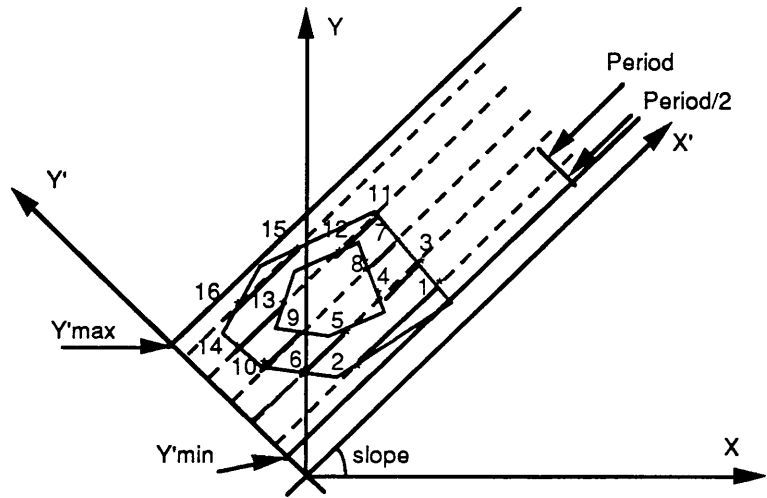


Fig. 8.27 The steps in the hatching procedure

In this system, the hatch lines are parallel to the x' axis. In order to find the number of hatch lines required to cover the whole area, the values 'ymin' and 'ymax' are found. Knowing the spacing between the hatch lines chosen from '*hatchspace*', the ordinate of the first hatch line, 'starty' can be identified as:

$$\text{starty} = \text{ymin} - (((\text{ymax} - \text{ymin}) - (\text{NoOfLines} * \text{space})) / \text{NoOfLines}) + \text{space}/2$$

This is done to distribute the hatch lines evenly all over the area.

The intersection points of every hatch line with the polygon sides are then found. This is done in two nested loops for each hatch line. A check for intersection with all sides of the polygon is made. If intersections are found, they are then checked to see whether or not they occurred between the end points of the segments. If they do, the intersections are then recorded into two vectors for point intersections, one for the x- and the other for the y-values. These two vectors are then sent to the '*sorting*' procedure, and afterwards the hatching lines are formed according to their chosen particulars. At the end of this operation, the procedure returns a picture of all the hatched lines of the polygon. Fig. 8.28 illustrates examples of the output resulting from this procedure.

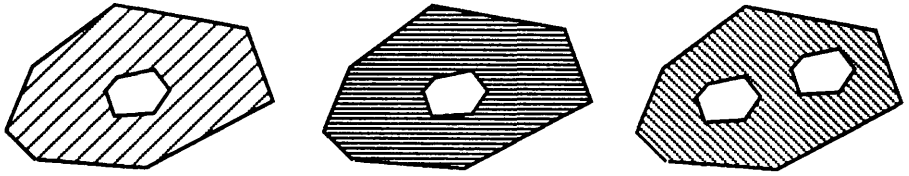


Fig. 8.28 Examples of the output from the hatching procedure

`hatch(vecx, vecy)`

This procedure is designed to manage the calls to other procedures to hatch certain polygons. Fig. 8.28 shows the calls made by it.

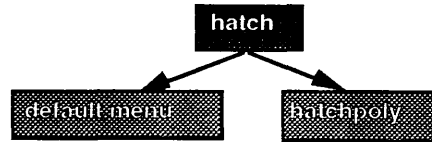


Fig. 8.29 The calls of procedure 'hatch'

First, it calls procedure '*default.menu*' to obtain the particulars of the hatch lines to be used and then it calls procedure '*hatchpoly*' to perform the task of hatching the polygon as specified by its coordinate values in vectors '*vecx*' and '*vecy*'.

8.2.2.2 Filling Polygons

The other alternative for polygon representation is filling. Filling is concerned with flooding a particular polygon with patterns. With spatial data, these patterns are often vegetational patterns such as those for grass, trees, etc. In this system, eleven vegetational patterns are available, namely those for bracken, coppice, heath, marsh, conifer woodland, deciduous woodland, orchard, reeds, rough-grass, saltings, and scrub. These patterns are chosen from a text menu and can then be scaled to fit to the specified scale of the map.

The descriptions of the various procedures used to fill polygons for area representation are presented below.

Procedure

Function

`drsym(symbolname, posx, posy, xscale, yscale)`

This procedure is not listed as one of those called by either of the three procedures '*inclave*'; '*polycheck*' and '*areamenu*' because it is not called directly by them. However, it is needed by the two higher level procedures '*symscale*' and '*plotsym*'. The task

implemented by this procedure is to read the data of a specific symbol, to scale it in both directions according to the factors 'xscale' and 'yscale' and to generate a picture of it. In turn, it calls the procedure 'stringtoint' to translate each of the symbols (which are read as characters) into the corresponding set of numerical coordinate values which describe the particular symbol and are held in the database. The procedure returns the picture of the symbol.

symscale(symbolname)

This procedure takes a symbol name and places that symbol in the menu shown in Fig. 8.30, where the user can then scale the symbol in both the x- and y-directions.

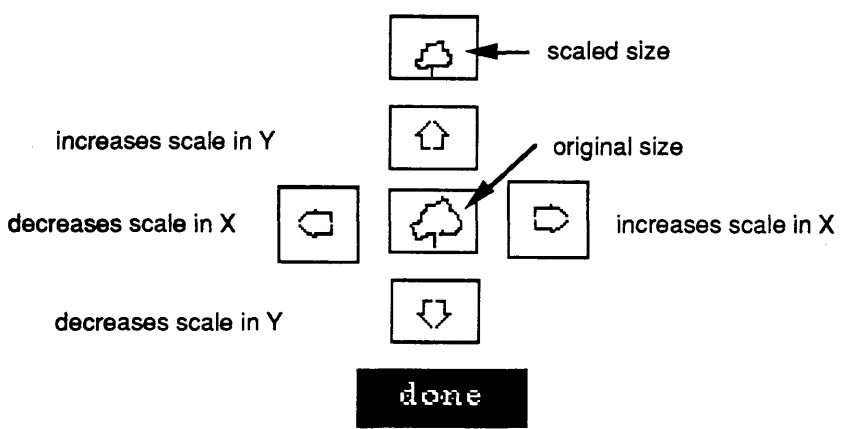


Fig. 8.30 The scaling menu

The use of the right arrow increases the scale in the x-direction while the left arrow decreases it. Similarly, the use of the up arrow increases the scale in the y-direction while the down arrow decreases it. The modifications made to the scale of the symbol can be followed in the uppermost box of the menu. At the end, the procedure returns the scaling factors obtained in each direction as a vector of two elements.

plotsym(x_a, y_a, length, pitch, xscale, yscale, symbolname)

The aim of this procedure is to place a series of symbols along a stretch ('length') of line starting from a point A(x_A, y_A) at intervals according to the selected pitch. The symbol is scaled by the xscale and yscale factors. The procedure returns a picture of the whole line of symbols (see Fig. 8.31).

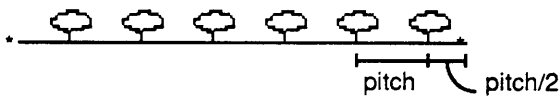


Fig. 8.31 This figure illustrates the layout of the 'plotsym' procedure
filling(vecx, vecy, vecI, spacing, symbol) [McGregor & Watt, 1986]

Filling a polygon with a specified pattern of symbols is the task of this procedure. It uses five other procedures to carry out this task. These are: 'symyscale', 'linepara'; 'lineint'; 'sorting' and 'plotsym'. Fig. 8.32 illustrates these calls.

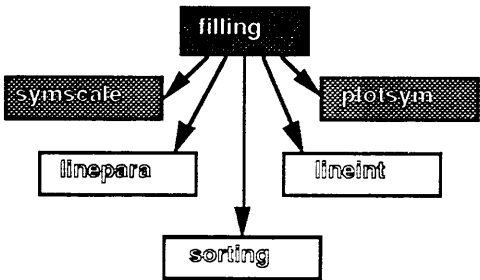


Fig. 8.32 The calls of procedure 'filling'

At the beginning, the procedure retrieves the table of available symbols from the database and displays them in the form of a text-based menu (Fig. 8.33) from which the user will be able to choose the symbol needed. Once a symbol has been selected, a scaling menu appears to scale that symbol to fit in with the scale of the map.

| CHOOSE |
|-------------|
| BRACKEN |
| CONIFER |
| COPPICE |
| DECIDUOUS |
| HEATH |
| MARSH |
| ORCHARD |
| REEDS |
| ROUGH-GRASS |
| SALTINGS |
| SCRUB |
| QUIT |

Fig. 8.33 The menu used to select the filling symbols

The steps which are followed to implement the filling operation are similar to those used to hatch polygons except that, instead of drawing the continuous or broken lines used in hatching, each line used in a fill pattern is divided into periods or pitches where each

period represents the location where a symbol should be placed. After the calculation of these locations (periods), a picture of the symbols is created and returned using 'plotsym' as shown in Fig. 8.34.

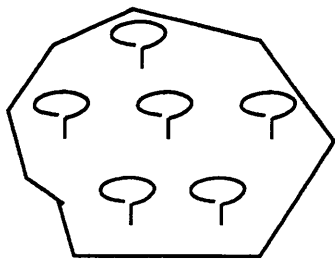


Fig. 8.34 The output of the procedure used for area filling

fill(vecx, vecy, vecI, spacing)

The job of this procedure is to manage the calls to the filling procedure and to display the returned picture on the screen.

8.2.3 Line Representation

When the option 'LINE' has been chosen from the menu in Fig. 8.2, procedure 'pickline' is activated and retrieves the structure 'L-holder' from the database. Then, in a sequential manner, the procedure highlights the selected feature. For line representation, the user will be able to choose any one of the different line types available from the line type menu shown in Fig. 8.35 (which is different to that used before for polygon hatching).

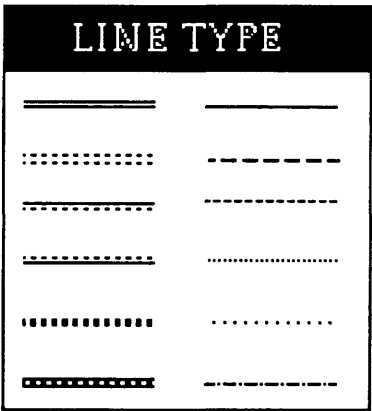


Fig. 8.35 The line type menu for linear representation

Procedure Function

doblseg(x_A, y_A, x_B, y_B, width, dash, gap)

This is a graphics procedure which enables the program to represent

a segment joining two points $A(x_A, y_A)$ and $B(x_B, y_B)$ as two parallel lines spaced from each other by a distance equal to the value contained in 'width'. These two lines can either be continuous or dashed. Fig. 8.36 shows the effects of this procedure.



Fig. 8.36 The principle of the 'doblseg' procedure

This procedure uses the two other procedures, '*drawline*' and '*dashing*' described earlier, and returns a picture of the drawing.

`dobline(vecx, vecy, width, type, dash, gap)`

This is another graphics procedure which aims at representing a chain of segments as double lines on either side of the original segments (which are then omitted), and spaced apart by the 'width' as shown in Fig. 8.37.

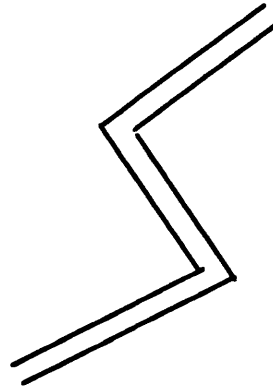


Fig. 8.37 The result of procedure 'dobline'

This is done in six steps:

(i) The first of these is to obtain the equation of the line that passes through the first two points, shown as line 'A1' in Fig. 8.38.

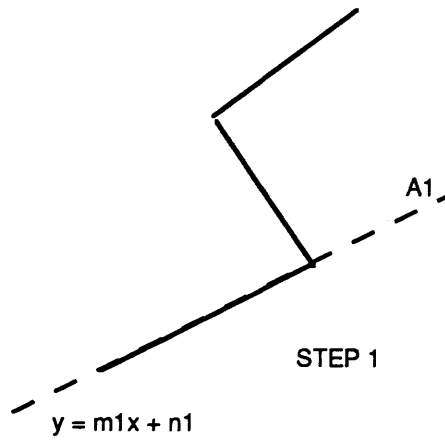


Fig. 8.38 Step 1

(ii) Secondly, the equation of a line 'B1' perpendicular to 'A1' is obtained using the procedure *'perpenline'* as shown in Fig. 8.39.

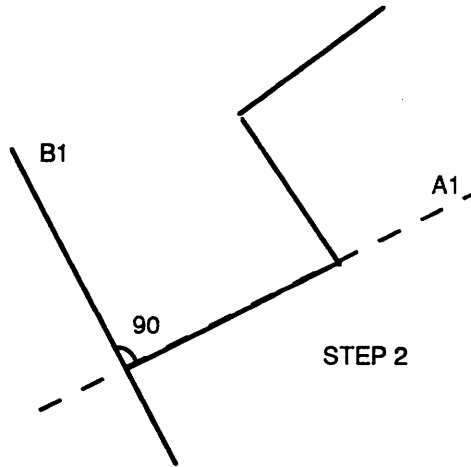


Fig. 8.39 Step 2

(iii) In the third step, two parallel lines, 'A1L' and 'A1R', are then established and spaced by 'width/2' on either side of 'A1', as shown in Fig. 8.40.

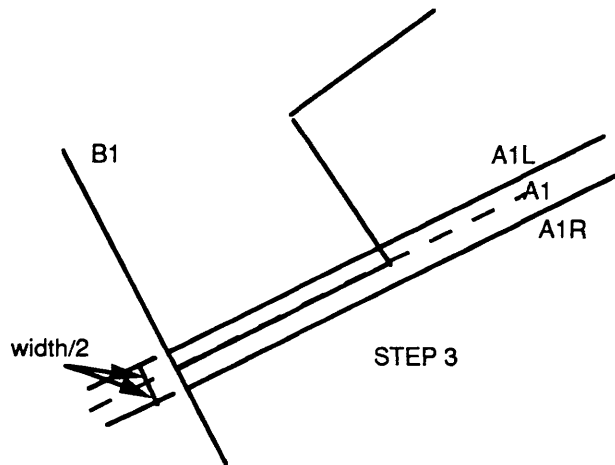


Fig. 8.40 Step 3

(iv) Next, in the fourth step, the intersections of these parallel lines, 'A1L' and 'A1R', with the perpendicular 'B1' are calculated and stored in two vectors 'VL' and 'VR', one for the left intersections and the other for the right intersections (see Fig. 8.41).

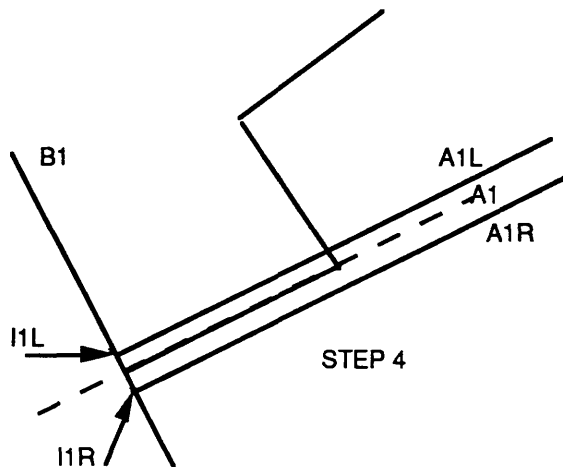


Fig. 8.41 Step 4

(v) In the fifth step, the previous steps (i) to (iv) are repeated for the second segment, which will end having the lines 'A2'; 'B2'; 'A2L' and 'A2R'. The intersection of 'A2L' with 'A1L' is then calculated and stored in 'VL' while that of 'A2R' and 'A1R' is stored in 'VR' (see Fig. 8.42).

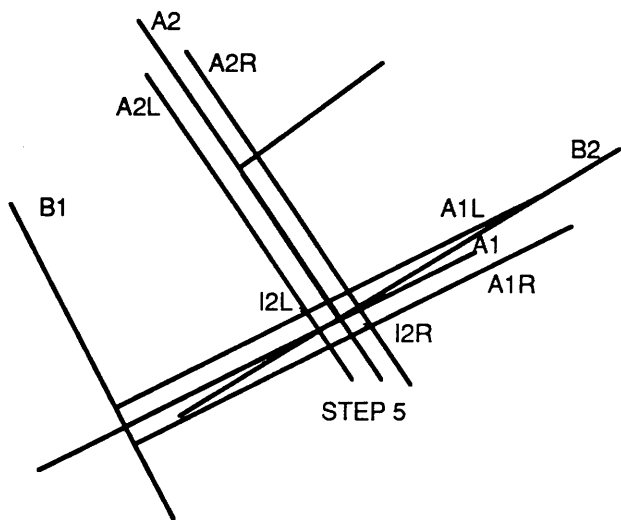


Fig. 8.42 Step 5

(vi) Finally in the sixth operation, steps (i) to (v) are repeated for all the segments and at the end, the parallel segments are drawn in a loop joining these segments with the type of lines specified by 'type' (a number selected between 1 and 4). Fig. 8.43 illustrates the result from this particular operation.

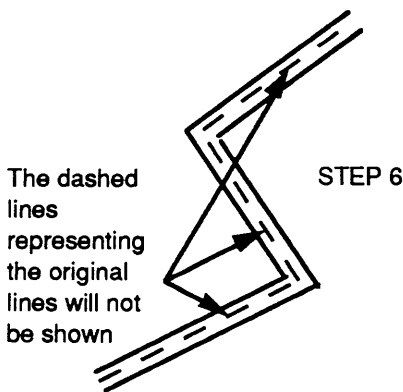


Fig. 8.43 Step 6

The four types of lines which can be obtained from this procedure are as shown in Fig. 8.44.



Fig. 8.44 The four line styles available for double lines

thkline(vecx, vecy, width)

This graphics procedure produces a thick line as shown in Fig. 8.45.



Fig. 8.45 The result of procedure 'thkline'

This is done by calling procedure '*doblseg*' from within a loop and specifying the '*width*'. At each call, the width is decreased so that when width becomes zero, the whole space between the first two lines is filled.

`ddline(vecx, vecy, width, gap, dash)`

This graphics procedure produces a thick dashed line as can be seen in Fig. 8.46.



Fig. 8.46 The result of procedure 'ddline'

The design of this procedure is the same as that of '*thkline*' except that, instead of using continuous lines in each call to procedure '*doblseg*', '*ddline*' uses a dashed line.

`railine(vecx, vecy, width)`

This procedure is used to represent railway lines. First, the procedure calls the subsidiary procedure '*ddline*' to draw the first part of the shape, which comprises the thick dashed line, as described above and shown in Fig. 8.46.

Secondly, '*dobline*' is then called to draw two bounding lines on either side of the thick dashed line, so ending up with the required representation of this line symbol (see Fig. 8.47).



Fig. 8.47 The result of procedure 'railine'

`bordering(vecx, vecy, width)`

This procedure imports a 'dash/dot' appearance to a given line. It is implemented in a similar way to that used in '*dashing*', except that there is a counter for the dashes. Whenever this counter is a multiple of two, the procedure then draws a 'dot' rather than a dash as shown in Fig. 8.48.



Fig. 8.48 The result of procedure 'bordering'

8.2.4 Point Representation

Having chosen the 'POINT' type from the menu in Fig. 8.2, procedure '*pickpoint*' is activated which in turn retrieves the structure 'P-holder' from the database, thus allowing access to all the various features stored there. It then calls three other procedures; '*text.write*'; '*chpoint*' and '*message.proc*' to implement the cartographic representation of point features. Fig. 8.49 illustrates these calls.

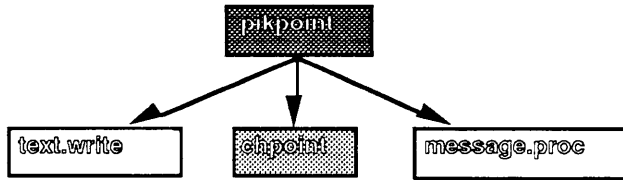


Fig. 8.49 The calls of procedure '*pickpoint*'

Since points represent those features occurring on the Earth's surface which are too small to be distinguished by dimensions on a map at a specific scale, they need to be represented symbolically. At the beginning of this procedure, a menu is displayed with the different symbols which are available to represent features in this system. In the point representation menu shown in Fig. 8.50 (and previously in Fig. 5.10), there are three different classes of symbol types, which are:

- a- Vegetational;
- b- Geometrical; and
- c- Textual.

Vegetational symbols are those used to represent the individual occurrence of a vegetation feature such as a tree at a specific location, and they are the same as those mentioned earlier in polygon filling. Geometrical symbols usually represent individual entities, such as cities (which may be represented by a circle), triangulation marks, and so on. These two types of symbols are stored in two different tables in the database and are called by two different procedures. The third available type is textual where text can be introduced and be laid on top of the graphical images using different fonts.

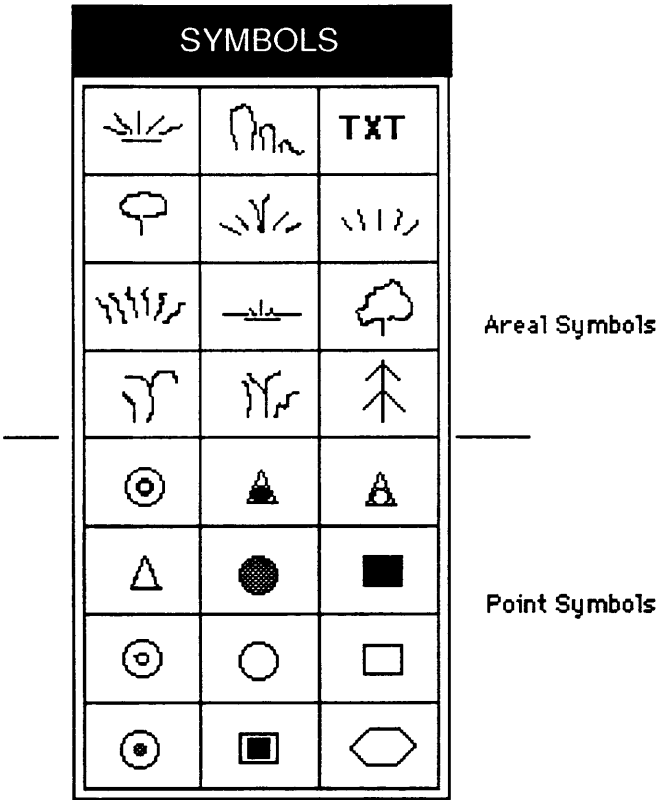


Fig. 8.50 The menu used for symbol selection

Procedure 'chpoint' uses another procedure called 'Fonting' which manages texts. 'chpoint' and 'Fonting' are described below:

Procedure
chpoint(x, y)

Function
Given the x- and y-coordinate values of the position where a symbol should be placed, this procedure retrieves the picture-based menu shown in Fig. 8.50 above, which allows the user to select the most appropriate symbol. Then the procedure retrieves that particular symbol from the database and displays it at the location specified by the 'x' and 'y' coordinate values.

Fonting(x, y)

At the start of this procedure, the fonts available with PS-algol are retrieved and are divided into two sets for the reason of display convenience. Several procedures are then declared within the 'Fonting' scope to allow for text entry and choice of fonts. These

are: 'text.writing'; 'chosefont'; 'fonting' and 'retrieve'. They are described below individually. Fig. 8.51 illustrates the procedure's calls while Fig. 8.52 shows the display resulted from 'Fonting' at the 'text.entry' stage.

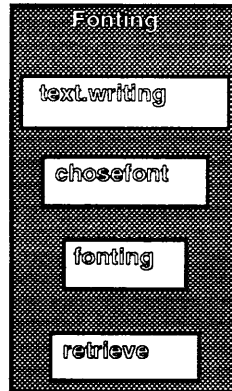


Fig. 8.51 The calls of procedure 'Fonting'

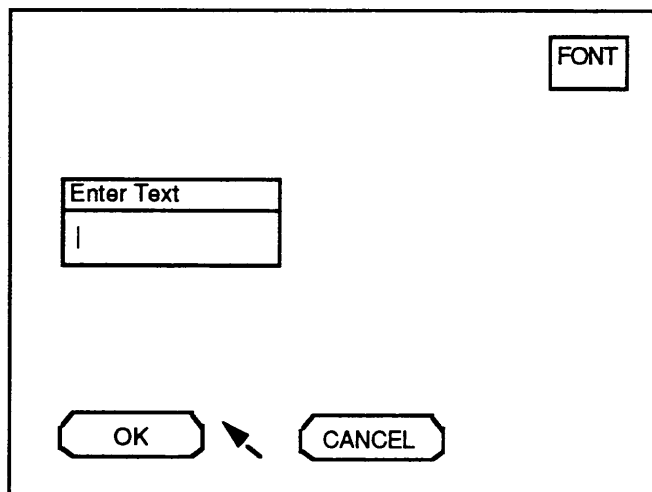


Fig. 8.52 The figure illustrates the window and the individual boxes used to enter text and choose the convenient font

text.writing(x, y, text, font, window)

The job implemented by this procedure is to take the 'text' supplied to this procedure in the specified 'font' and place it in a particular 'window' on the screen at the determined location 'x' and 'y'.

chosefont(which.font.menu)

This procedure allows the user to select a convenient font for the text out of the twelve available fonts in a pull down menu. Fig. 8.53 shows the display of the font selection window. It will be noticed that only six of these fonts are available in the menu at any one time.

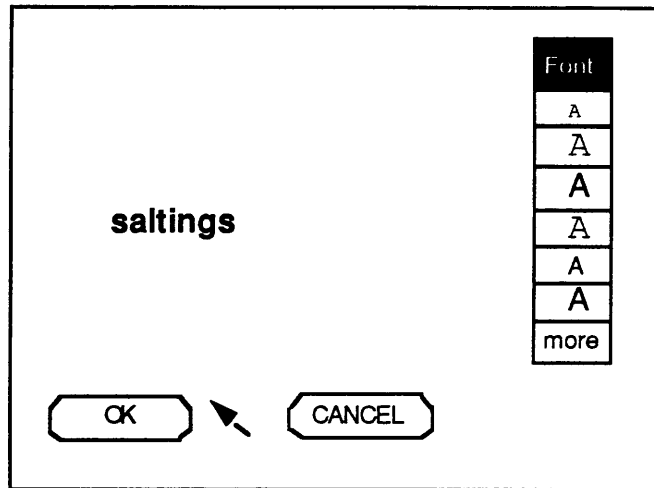


Fig. 8.53 Fonts are available from a pull-down menu

The procedure then returns the selected font.

fonting(text)

This procedure takes the entered text and calls procedure '*chosefont*' to select a font and then displays that text in the chosen font using procedure '*text.writing*'. The procedure returns the final chosen font.

retrieve()

The task implemented by this procedure is to activate the font window which enables the user to select a font type. It first allows the user to enter text using the procedure '*s.editor*' taken from the '*utilities*' database and by selecting any one of the three available commands : 'OK'; 'Cancel' and 'Font', the user can (i) assign the default font to the text entered; (ii) re-enter text; and (iii) choose a different font style from the pull-down menu.

8.3 *Scrolling*

The scrolling facility, which implements the operation otherwise known as panning, works hand in hand with zooming. This results from the fact that scrolling allows the user to move to areas which were hidden due to zooming activities. As has been discussed in Chapter 6, Section 6.5.2, if a window smaller than the image portrayed on the screen is specified, this image is clipped automatically. Then if a viewport equal to the screen area is specified and the contents of the window are displayed, the clipped image is then

magnified. Furthermore, scrolling is achieved by defining a series of windows, each smaller than the full size of the screen.

The six procedures used to perform scrolling are: '*Scrolling*'; '*transform*'; '*Ushift*'; '*Dshift*'; '*Lshift*' and '*Rshift*'. Fig. 8.54 illustrates the links between these procedures. However, only '*Scrolling*'; '*transform*' and '*Lshift*' will be described here, because the other three are similar to '*Lshift*' but simply operate in different directions.

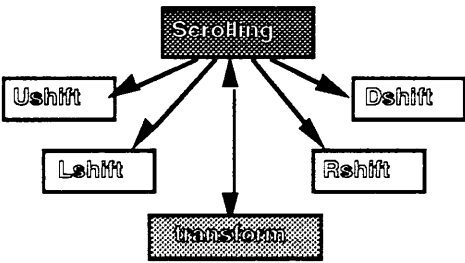


Fig. 8.54 The calls of procedure '*Scrolling*'

Procedure

Function

`Scrolling(picture, window.range)`

At the start of this procedure, a check is carried out to ascertain whether or not zooming has already been done. If it has not, then scrolling need not be implemented, and the procedure is aborted. On the other hand, if zooming has been done, '*Scrolling*' then takes the coordinate values of the display window (given in map coordinates) and displays the scrolling menu which allows the user to specify the direction of scrolling. Fig. 8.55 shows the menu used for scrolling.

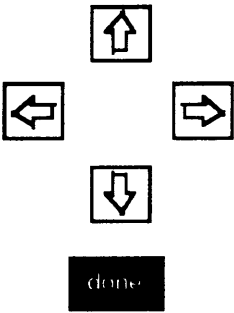


Fig. 8.55 The menu used for scrolling

The procedure then calls whichever one of the four matched procedures is needed to perform the scrolling operation in the required direction. Once the scrolling operations are completed, the map coordinate values at which this scrolling ends are then recorded in the global scope of the module.

transform(avalue, screen.range)

This procedure transforms a value entered in screen units into the equivalent map units.

Lshift(screen.range, window, picture)

This shifts the viewport over the map leftwards. The righthand part of the window is copied to the left end of the screen. The resulting gap on the right is then filled by drawing the picture within that specified area.

The other three possible shifts - Rshift, Ushift and Dshift - are implemented in a similar manner.

8.4 Quit & the Storage of Cartographic Symbols

The generation of all required cartographic symbols is carried out for all the features held in the database for the area in question in a sequential manner. Once a particular feature has been passed through this module, it will be assigned with a flag in its record stating that this feature has already been symbolised. So next time, the system will look for those other features to be included in the map which have not yet been defined in a symbolic form.

After each individual symbolising activity, each of these items will be flushed out of the screen and will be saved in the 'MDB' database as pictures in the record of their features. They will not be seen again until data retrieval is carried out.

Once all the data available for display in cartographic form in a particular type of data (points, lines, or polygons) have been consumed, the system will automatically quit the operation and allow the user to choose another type of data or to quit the module.

If the user chooses to quit the module, he has the capability of recording all the changes made to the data, or to discard all the changes that have been carried out during the whole session. In the case where the save command is issued, all the changes to the data will be recorded in the database, and these features need never be submitted to this module again.

8.5 *Summary*

In this chapter, the graphics procedures used for the cartographic representation of different types of features have been discussed. These procedures are performed for all those features entered into the system, whereby every feature, after having been passed to a suitable procedure, has its resulting cartographic representation stored as a picture in its record in the database.

This module is so organised that the process of cartographic representation may be interrupted at any stage and returned to in a subsequent run of the program.

CHAPTER 9

CHAPTER 9: DATA RETRIEVAL MODULE

9.1 *Introduction*

This module is concerned with the different ways of retrieving data by making full use of the data structures described earlier in Chapter 5, namely the hierarchical feature coding system and the four relational data structures of the features themselves. This chapter includes a description of the different procedures which handle the issuing of queries and which ensure the display and saving of the results of these queries.

This procedure is carried out in the first instance by placing the pointer in the box containing the appropriate menu entry and then clicking the mouse to retrieve the required information. Records matching the given qualification are then retrieved and displayed. If the query answer is to be saved, record identifiers are stored in a specific structure instead of creating new tables, as is the case with other systems based on SQL.

Retrievals are carried out relative to three different partitioning mechanisms. Firstly, the type of data required (Polygons, Lines or Points) may be selected. Secondly, a specific layer of the data, for instance Roads or Land Cover, can be selected. Thirdly, a particular kind of feature can be selected according to the hierarchy of the feature coding system (see Appendix A). In fact, any combination of these can be used to restrict the amount of data which has to be retrieved. For example, the user may select all the Polygons occurring in the Land Cover layer for display.

As stated above, retrievals are carried out by menu selection. For example, if the 'Type' menu is being activated, the user can then choose to retrieve features of any one of the three types, namely 'Polygon'; 'Line' or 'Point', or any combination of two of them or all three together. Should the user decided to retrieve data from the 'Layer' menu, this can be done by first activating the 'Layer' menu, then simply selecting the layers of interest by pointing at them and clicking the mouse as required. Furthermore, the object oriented implementation of the database is exploited at this stage as well by allowing the user to retrieve single entities from the databases, by using the 'Entity' menu. Using this menu, the user can retrieve features down the hierarchy of the feature classification. For example, the user can retrieve data according to a specified feature class, category and attribute.

Moreover, a combination of selections from two menus at a time is allowed. Thus a user can decide to retrieve items from both the 'Type' menu and the 'Layer' menu or from the 'Entity' menu and the 'Type' menu. It should be noted however that a combined selection from the 'Entity' menu and the 'Layer' menu is not provided because of the fact that an entity that belongs to a particular layer cannot be retrieved from another layer.

9.2 Module Architecture

The tasks carried out by this module involve various different operations which are concerned with (i) the layout of the screen to handle the various activities taking place; (ii) establishing access to the databases; and (iii) the various retrieval operations required. It also provides some other facilities such as zooming and saving the queries and the information retrieved by them. Fig. 9.1 illustrates these different activities.

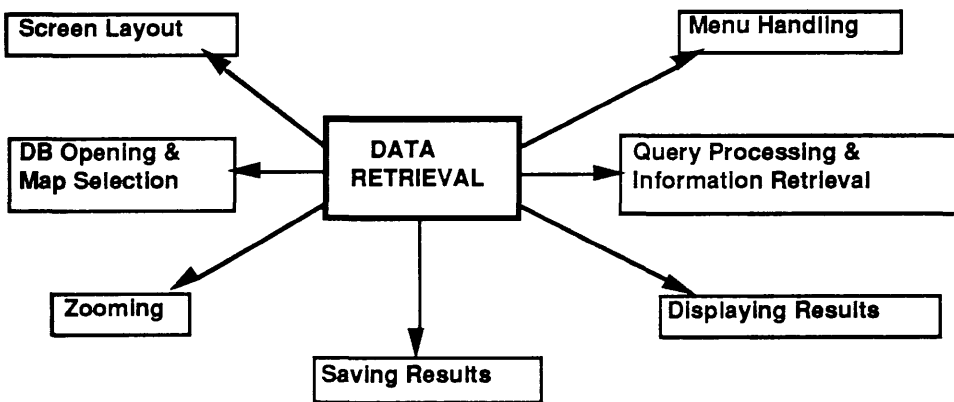


Fig. 9.1 The module's activities

Some of these activities are performed via a series of procedure calls at the module level while the rest are carried out by direct commands from the main body of the module. The procedures which can be called are:- '*prepform*'; '*text.write*'; '*zoomin*'; '*zoomout*'; '*checkm*'; '*activate*'; '*centity*'; '*clayer*'; '*retType*'; '*retLayer*'; '*selectEntity*'; '*retEntity*'; '*retT+L*' and '*retT+E*'. Fig. 9.2 shows the module's calls to these procedures, while Fig. 9.3 illustrates the flowchart of the module. The first four procedures have already been described in Chapter 6, Section 6.5.2; the rest (together with those procedures called by them) will be described in the following sections.

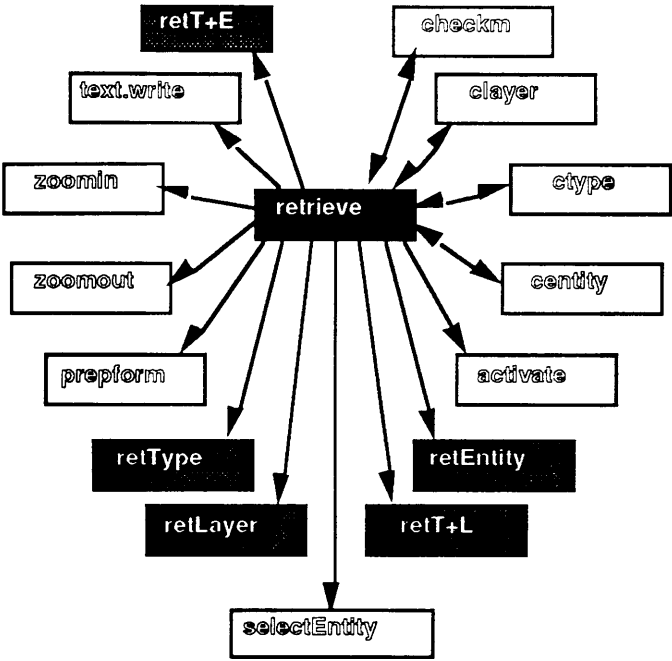
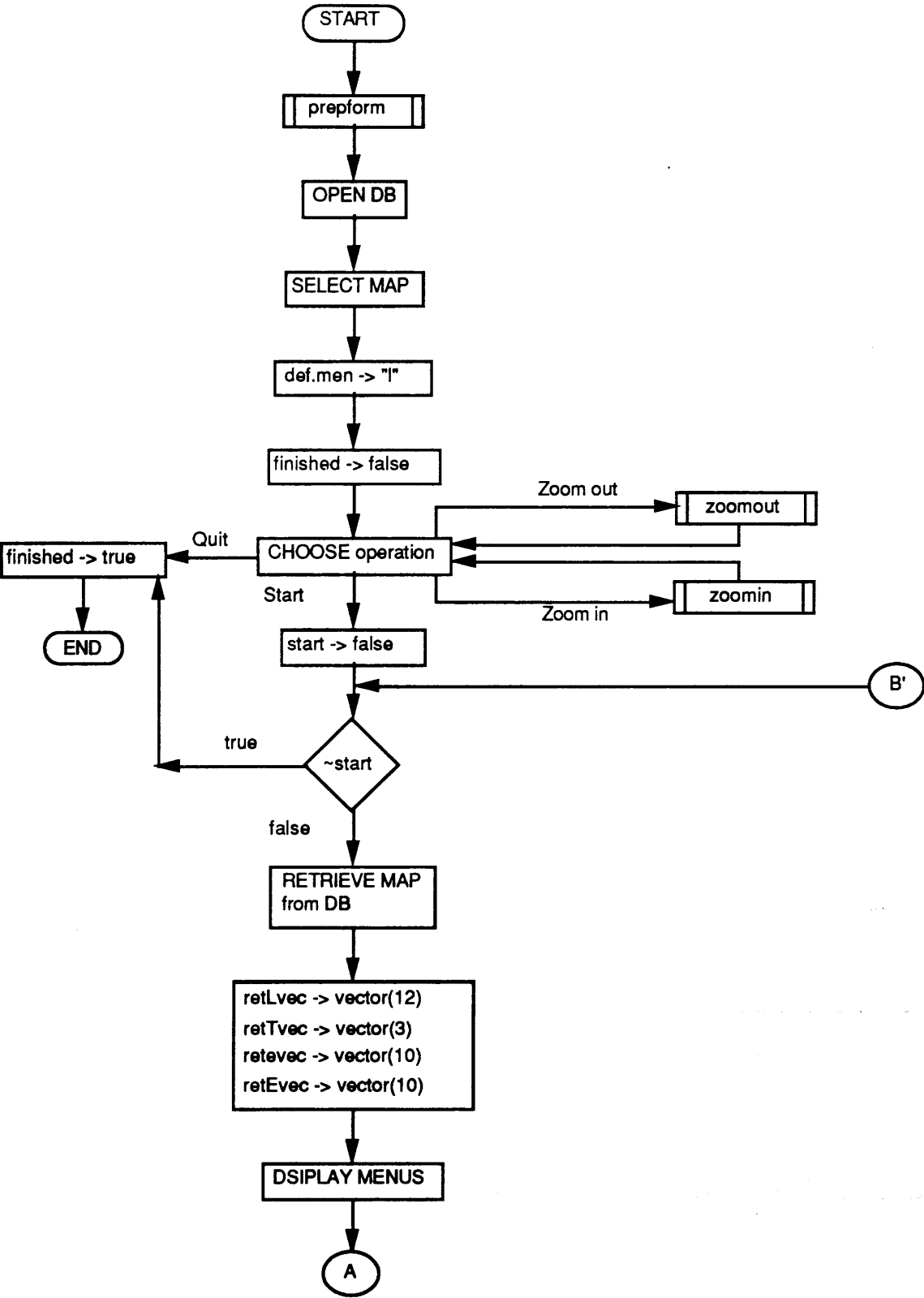


Fig. 9.2 The module's calls

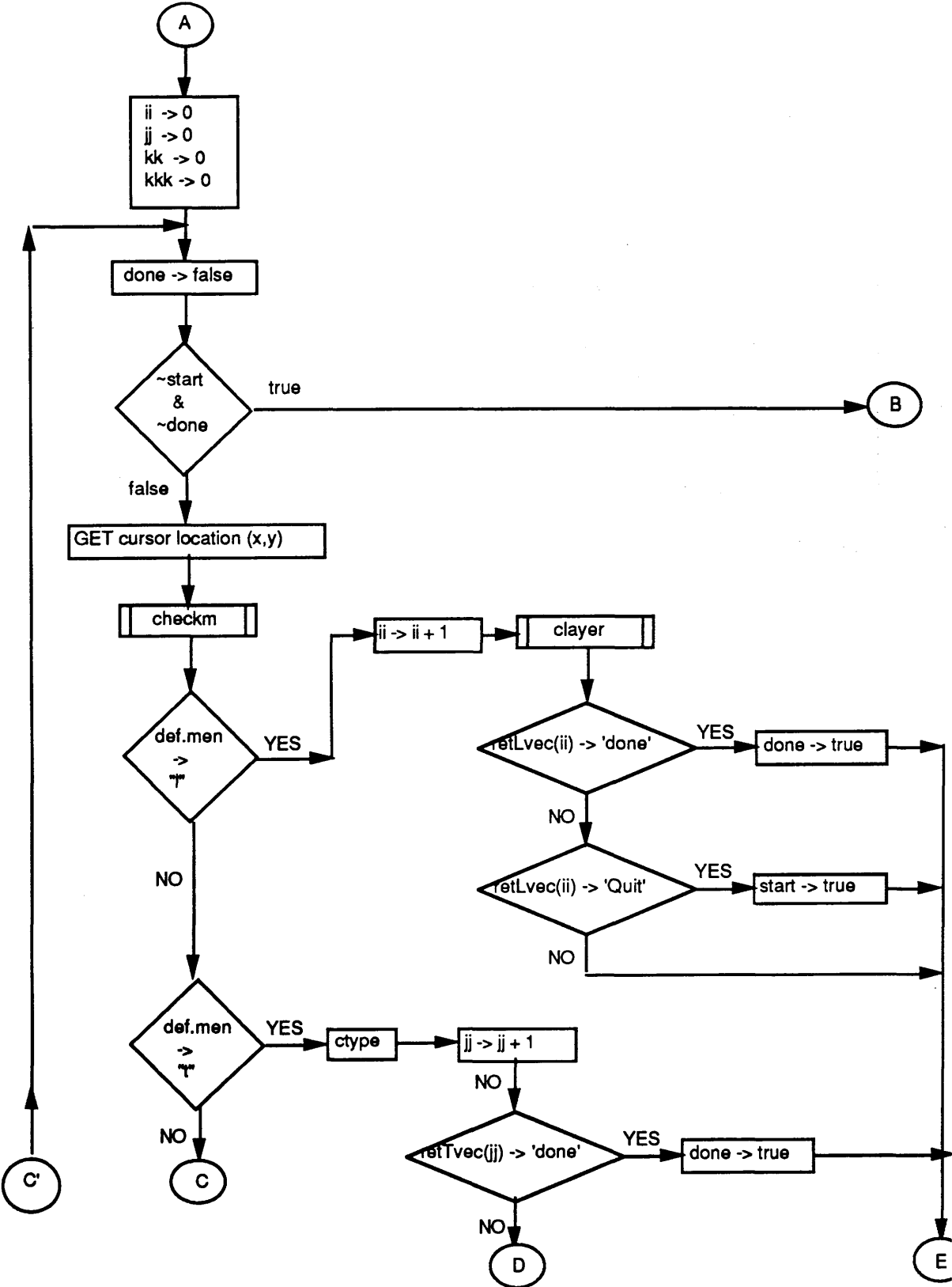
In the following sections, the description of each of these activities is given with an emphasis on their relevance to query processing and information retrieval.

9.3 Screen Layout

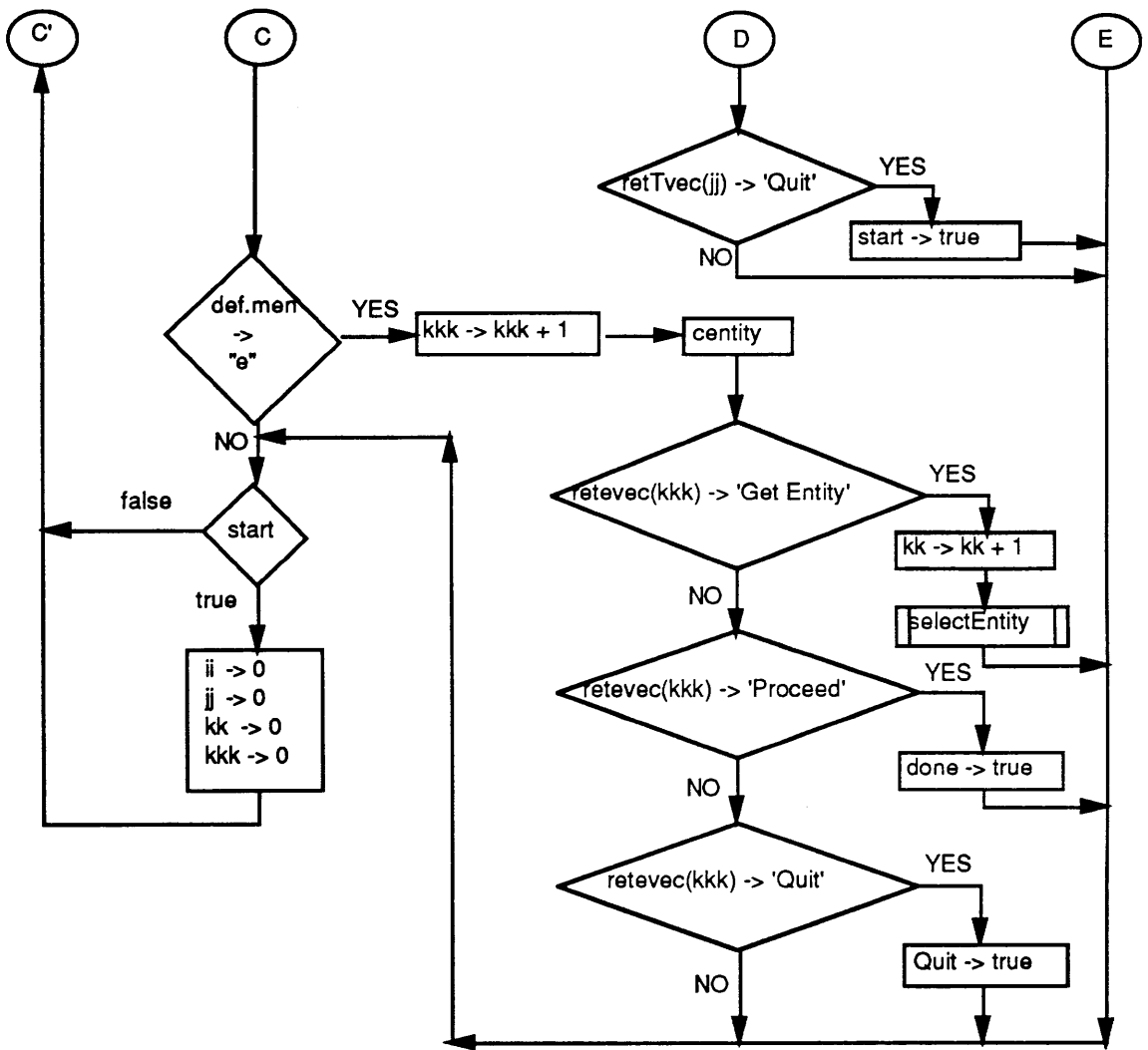
The screen layout is created so that the major part is allocated to the graphics display of the map which results from the transactions performed by the module. This is done using procedure '*preform*' described in Chapter 6, Section 6.5.2 and illustrated in Fig. 6.8.



Cont.



Cont.



Cont.

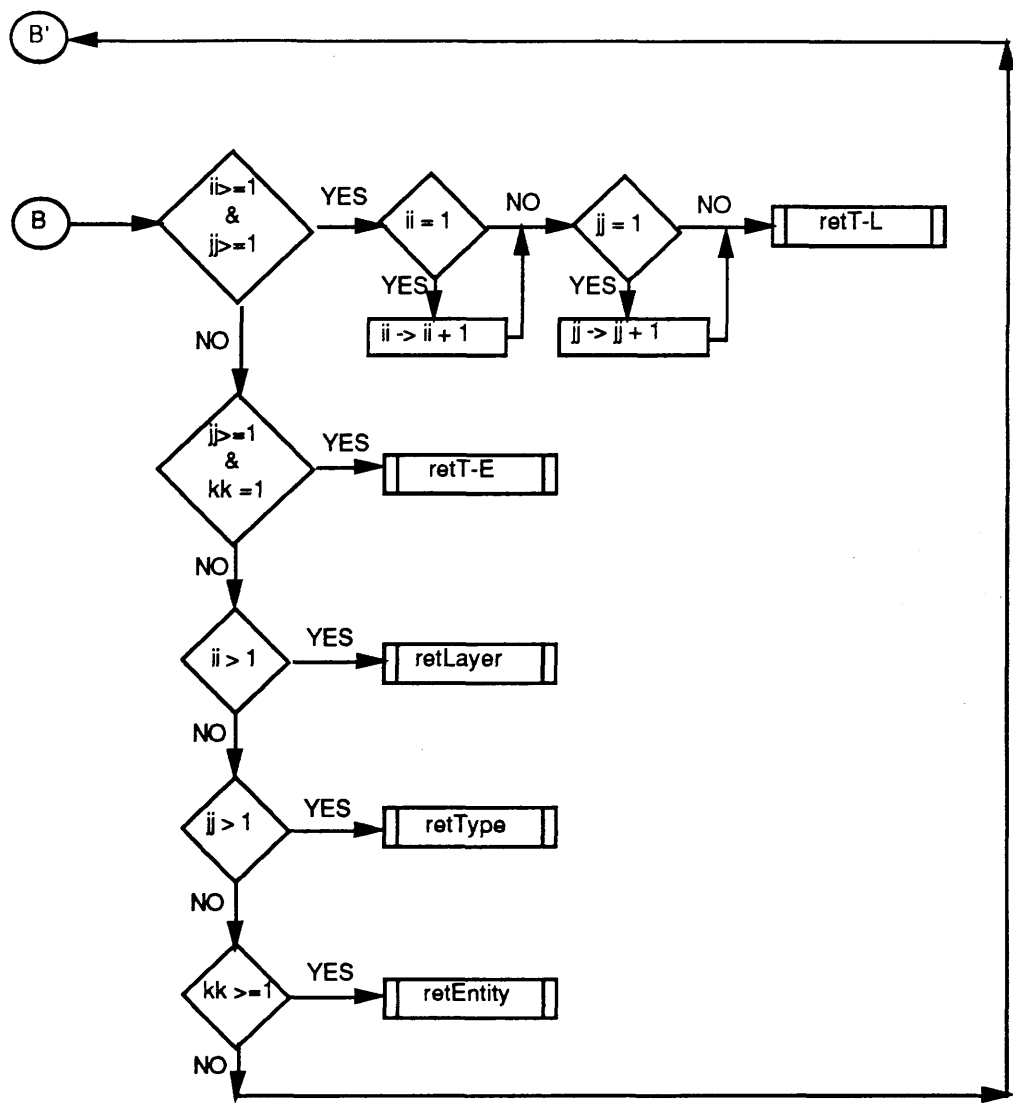


Fig. 9.3 The flowchart of the module 'retrieve'

9.4 Database Opening & Map Selection

The particular database which is the concern of this module is '*MDB*', where the data of the features are being held. In this module, database '*MDB*' is opened in the 'read' mode. The various maps which have been stored and are available within this database are then presented in a menu form, thus allowing the user to select the particular map which has to be retrieved from the database.

The map's general details (i.e. the scale of the map; the coordinate values of the edges, the grid interval, etc.) are then retrieved and its grid is displayed on the screen. No further retrieval from the database will be carried out until the user specifies the exact details of the

data to be retrieved. This is done in the next stage after the user has chosen the zone of interest to work on.

9.5 *Zooming*

Zooming in and out facilities are available throughout the period that this module remains activated. The description of these procedures can be found in Chapter 6, Section 6.5.2.

9.6 *Menu Handling*

Generally speaking, this module has been built around the concept of retrieving information from the database in three ways; by types; by layers and finally by individual entities.

As has been discussed earlier, queries are issued to the system using a set of menus, where each menu represents a different means of data retrieval. The module manages these menus and activates the one at the foreground of the screen. Menus are sent backward and forward by pointing at the header or title box showing the menu entry that is wanted and clicking the mouse. This will cause that particular menu to be brought forward to the foreground of the screen. This is done using procedures '*checkm*' and '*activate*'. The foreground menu in the default setting is the one which allows retrieval by layer. Fig. 9.4 shows these menus at the default setting. The module also keeps track of all the activities taking place during the process, such as which menu is active and which items of a particular menu are being selected.

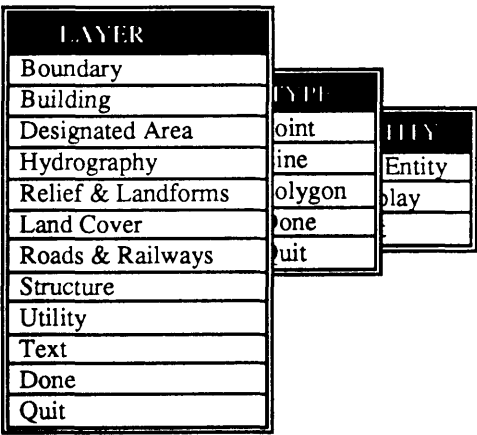


Fig. 9.4 The default setting of the retrieval menus

The various menus are handled by two procedures '*checkm*' and '*activate*'. Their jobs are

to distinguish which menu is selected and to make it active. These procedures are described below.

Procedure

Function

checkm(x, y)

This procedure takes the 'x' and 'y' values of the location of the pointer when the mouse is clicked and returns an identifier of the chosen menu.

activate(x, y)

This procedure takes the result of 'checkm' and arranges the display of the menus accordingly. Thus, any menu displayed in the foreground is the active one. Menus are sent backward and forward by clicking the mouse when the pointer points anywhere inside the box showing the title of the menu. Fig. 9.5 shows the new setting of these menus when the 'Type' menu is activated.

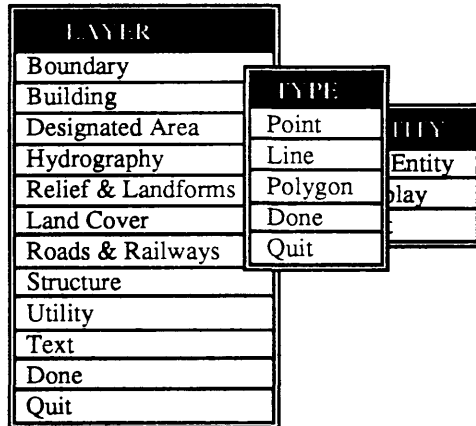


Fig. 9.5 The setting of the menus at the 'Type' selection

The user can, if he wants to, select from both the 'Layer' and 'Type' menus at any one time. Similarly, the 'Entity' and 'Type' menus can also be used in combination.

9.7 Query Processing & Information Retrieval

When queries are issued, a select command is issued to retrieve the records satisfying the required qualifications and when the selection is a combination of two menus (e.g. the

'Type' and 'Layer' menus), the system then searches for those features of the specified layer(s) in the structure(s) of the selected type(s); i.e. either one or a combination of A-holder; L-holder; P-holder and T-holder. Those features satisfying the layer and type conditions are then retrieved and displayed on the screen.

The procedures dealing with data retrieval are described in five subsections. Section 9.7.1 describes those procedures dealing with retrieval of data by type. Section 9.7.2 deals with the procedures retrieving data by layer. Section 9.7.3, deals with the retrieval of data by individual entities. Section 9.7.4 describes the retrieval of data by a combination of type and layer information. Finally, Section 9.7.5 details the retrieval of data by a combination of entity and type.

9.7.1 Retrieval By Type

Retrieval by type is carried out by first activating the menu which allows selection of the three types of data (Point, Line and Polygon) to be carried out. This is done by calling the procedure 'ctype'. Once the selection of one or all the types has been completed, the procedure responsible for the retrieval of data, 'retType', is called. It takes a vector of strings which contains the types required for the transaction. Fig. 9.6 illustrates the calls made at the module level to retrieve by Type. Then a check of types is made and depending on the results of this check, a selection from the three following procedures 'RetApic'; 'RetLpic' and 'RetPpic' is made and the appropriate ones are called. These procedures, together with 'ctype', are described below.

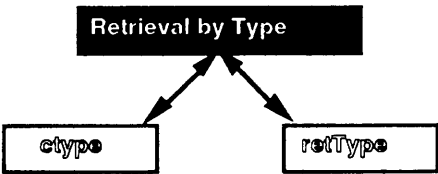


Fig. 9.6 The calls made for retrieval by Type at the module level

Procedure

ctype(x, y)

Function

The operation that this procedure carries out is to specify the type(s) of the data to be retrieved. When the 'Type' menu has been activated, then, given the 'x' and 'y' values of the position (or

positions) at which the pointer was located when the mouse was clicked, this procedure determines which type (or types) have been selected for data retrieval. Two other operations, 'Done' and 'Quit', are also available within the 'Type' menu. When 'Done' is selected, this means that the end of the type selection process has been reached. This will result in a message being sent to the system to start the next phase. On the other hand, if 'Quit' has been selected, this will abort the whole operation and return the user to the menu of modules. Fig. 9.7 shows the display of the 'Type' menu.

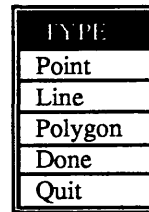


Fig. 9.7 The menu for 'Type' retrieval

The procedure returns the type(s) to be retrieved.

RetApic(string map.name; pnt A-structure)

This procedure takes a pointer to structure 'A-holder' and forms and draws the pictures of the features stored in it.

RetLpic(string map.name; pnt L-structure)

This procedure takes a pointer to structure 'L-holder' and forms and draws the pictures of the features stored in it.

RetPpic(string map.name; pnt P-structure)

This procedure takes a pointer to structure 'P-holder' and forms and draws the pictures of the features stored in it.

retType(*string type.vec, string database.name, pnt map.name)

This procedure takes a vector of strings indicating the selected type(s). It then checks the chosen types and calls any or all of the three procedures dealing immediately with data retrieval by type. These procedures return the graphic results which are then sent to the screen. Fig. 9.8 illustrates the calls made by this procedure.

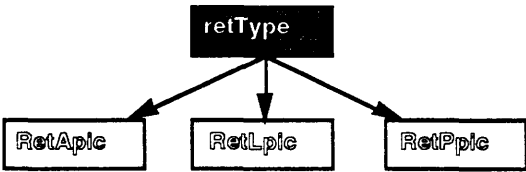


Fig. 9.8 'retType' procedure calls

9.7.2 Retrieval By Layer

Retrieval by layer is done by first activating the menu of layers (which is that displayed as the default setting) by placing the pointer in the box showing the title 'Layer' and then clicking the mouse. The menu shown in the foreground of Fig. 9.4 is the one which allows the selection of layers to be carried out. The required layer(s) can then be selected from the menu by simple pointing and clicking actions. Procedure '*clayer*' is designed to detect the chosen layer(s). The retrieval operation is then carried out by calling the procedure '*retLayer*' (see Fig. 9.9), which in turn calls procedure '*codestrip*'.

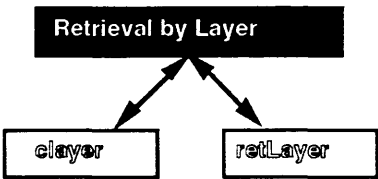


Fig. 9.9 The calls for retrieval by Layer made at the module level

| <u>Procedure</u> | <u>Function</u> |
|-------------------------------|---|
| <code>clayer(int x, y)</code> | <p>This procedure determines the layer (or layers) needed to be retrieved when the 'Layer' menu is activated. There are ten layers and two commands available from this menu, see Fig. 9.10. The two commands are the same as those contained in '<i>ctype</i>' used in the 'Type' menu and perform the same jobs. The ten layers are the nine individual classes in the classification described in Appendix A, while the tenth layer is text. The procedure returns a vector of strings of the selected layer(s) for retrieval.</p> |

| LAYER |
|--------------------|
| Boundary |
| Building |
| Designated Area |
| Hydrography |
| Relief & Landforms |
| Land Cover |
| Roads & Railways |
| Structure |
| Utility |
| Text |
| Done |
| Quit |

Fig. 9.10 The menu for 'Layer' retrieval

`codestrip(int code)`

This is a very simple procedure which takes a code number and determines the specific layer to which the feature holding this code belongs by returning a number between zero and nine (see Appendix A).

`retLayer(*string layer.vec, string database.name; pntr map.name)`

The procedure opens the specified database and then accesses the specific table corresponding to the selected map. It then analyses the vector of layers which contains the selected layers which need to be retrieved. Features satisfying the requirements (checked by calling procedure '*codestrip*') are then retrieved and displayed.

The calls made by procedure '*retLayer*' are shown in Fig. 9.11.

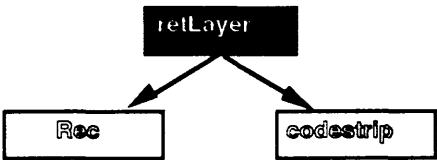


Fig. 9.11 '*retLayer*' Procedure Calls

9.7.3 Retrieval by Entity

Retrieval by entity is the most elaborate part of this module among the retrieval options since it enables the user to select and retrieve by class, category, features and finally (at the lowest level of classification) by the specific attributes of the features themselves (if these have been provided). This could be one of the greatest advantages of employing the hybrid

database system examined in Chapter 5.

In a manner similar to that used with the other options described earlier, retrieval by entity is carried out by first activating the menu with the title 'Entity' (shown in Fig. 9.12) and then choosing the box 'Get Entity'. The procedure responsible for doing so is '*centity*'. The selection of the entity code then follows using procedure '*select.Entity*' which in turn calls procedure '*obtain.code*'. Finally, the retrieval operation starts by using procedure '*retEntity*' which in turn calls four other lower level procedures. Fig. 9.13 illustrates the calls made at the module level.



Fig. 9.12 The menu for 'Entity' retrieval

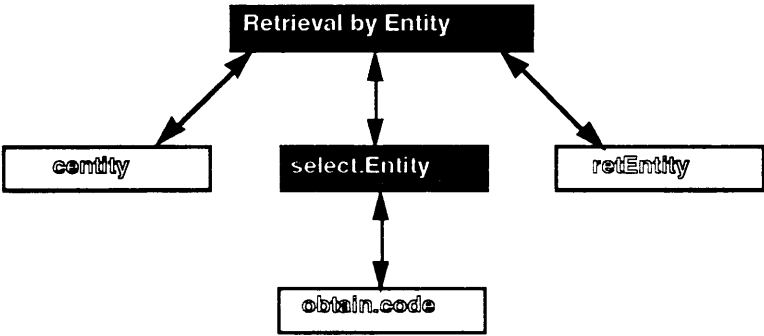


Fig. 9.13 The calls made to retrieve by 'Entity' at the module level

The other four procedures called by '*retEntity*' are: '*class.code*'; '*retfromA*'; '*retfromL*' and '*retfromP*'. Only the first two procedures ('*class.code*' and '*retfromA*') will be described since the other two procedures ('*retfromL*' and '*retfromP*') are similar to '*retfromA*' except that they retrieve the data from different structures of the database, i.e. from 'L-holder and P-holder instead of from A-holder.

| <u>Procedure</u> | <u>Function</u> |
|--------------------------------|--|
| <code>centity(int x, y)</code> | If the 'Entity' menu is activated, then this procedure determines which operation is required when the pointer is placed in a specific box inside the menu area and the mouse is clicked. These operations |

are:-

- i) 'Get Entity' which leads to four consecutive menus that allow the user to identify which entity is to be retrieved;
- ii) 'Display' which allows the display of the retrieved entities on the screen; and
- iii) 'Quit' which returns the user to the module menu. The procedure returns the name of the operation requested.

select.Entity()

The task of this procedure is to first call up procedure '*obtain.code*' and then to store the returned code with its level of classification in a vector of two elements.

obtain.code()

This procedure allows the user to select the level of classification at which the retrieval of data will be carried out. Thus retrieval can be made on the class level, which is equal to retrieval by layer. On a lower level, features can be retrieved according to categories. Furthermore, they can also be retrieved by stating the feature name itself and finally, retrieval can be carried out according to the attributes of the features (when available). This operation is carried out via four consecutive menus showing the classes; the categories within a class; the features within a category; and finally the possible attributes of a specific feature. Fig. 9.14 illustrates an example of the hierarchy of the menus generated by this procedure.

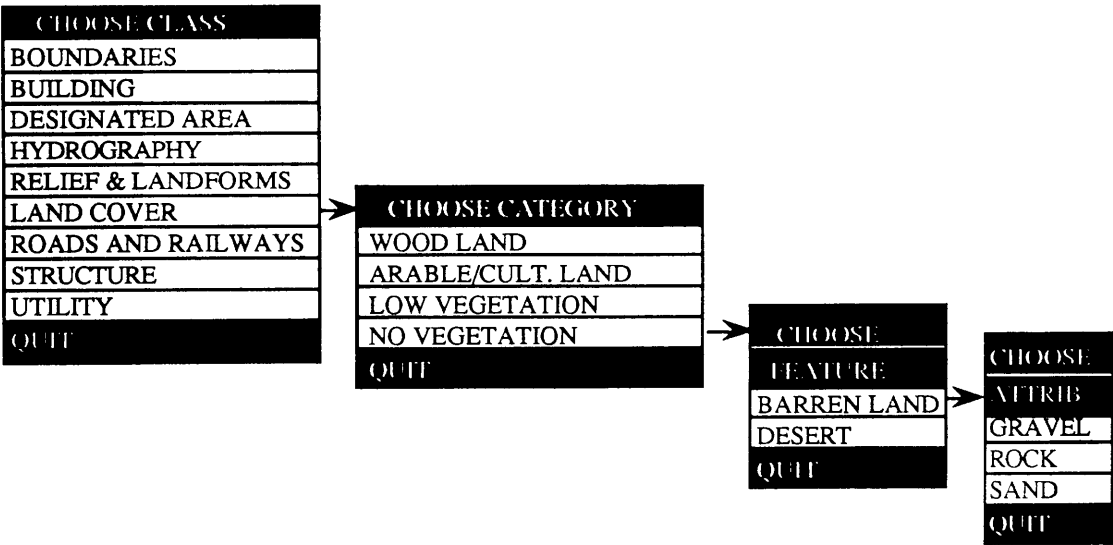


Fig. 9.14 The hierarchy of the menus to generate a code

If the user chooses a class from the menu and then chooses the 'Quit' command from the class menu, then all the features of the database having their code number indicating that class will be retrieved from the database and will be displayed. A similar approach has been adopted for categories, features and attributes.

Procedure '*obtain.code*' declares another procedure '*choose*' within its scope which allows the scan of a particular table in a database thus creating a text-based menu of the elements found in that table.

The procedure returns the code obtained and the level at which the code was generated, i.e. if the user had chosen to retrieve by class, then the level is '1'; if the selection was by category, then the level is '2'; and so on. Fig. 9.15 illustrates the calls made by this procedure.

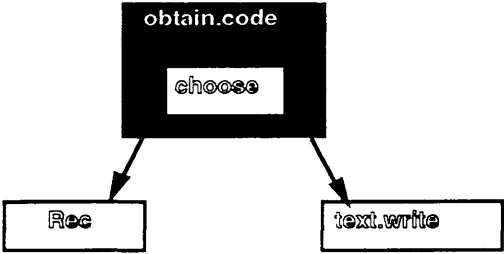


Fig. 9.15 The calls made by procedure '*obtain.code*'

class.code(*int chosen.code.level; int stored.code)

Whenever the level of searching has also been defined, this procedure determines whether or not a chosen code matches those codes passed to the procedure during the database search operation. It returns a boolean value, 'true', if it matches, or, the value 'false' if it does not.

retfromA(string database.name; pntnr map.name)

This procedure establishes access to the specified '*map.name*' table within the given '*database.name*'. It then scans through the codes of the features stored in the structure A-holder, checking for those features whose codes match the selected code. This is done by calling procedure '*class.code*'. Successful searches are then reported to the screen.

retEntity(*int code.level; string database.name; pntnr map.name)

The task of this procedure is to make the calls to the three procedures concerned with the search and retrieval of features by specifying the chosen code and the level at which it was selected - as described earlier in procedure *'obtain.code'*. It calls procedures *'retfromA'*; *'retfromL'* and *'retfromP'*. Fig. 9.16 shows these calls together with the lower level calls of these procedures to procedure *'class.code'*.

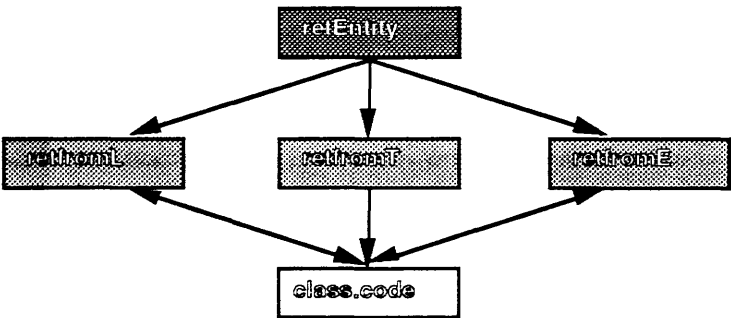


Fig. 9.16 The calls of procedure *'retEntity'*

9.7.4 Retrieval by a Combined Selection from the Type and Layer Menus

A layer can contain features of different types (Polygons, Lines and Points). For example, the layer 'Roads and Railways' contains the codes for the associated features which belong to type polygon, such as those used for filling embankments and cuttings. It also contains features such as road signs (which are of type point) and obviously contains those features such as roads and railways (which are of type line). So, sometimes it will be very desirable (or indeed required) to retrieve those particular features of a specific type from a particular layer. This is the task of procedure *'retT+L'*. Based on the values returned from procedures *'checkm'* and *'activate'*, the main body of the module calls procedure *'retT+L'* to retrieve the required information. The description of procedure *'retT+L'* is given below.

| <u>Procedure</u> | <u>Function</u> |
|--|---|
| <code>retT+L(*string vec.of.layers, vec.of.types; string database.name; pntnr map.name)</code> | Given the layers to be searched in a vector, the procedure first analyses this vector and discards the unwanted layers. Then it checks the selected types and starts the search through the 'Type' structure ignoring those features which do not meet both conditions of type and layer. On the other hand, those features which do meet |

the two conditions are then reported to the user on the screen. In turn, procedure 'retT+L' calls only one procedure, which is 'codestrip', (see Fig. 9.17).

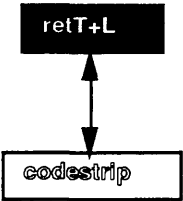


Fig. 9.17 The call made by procedure 'retT+L'

9.7.5 Retrieval by a Combined Selection from the Type and Entity Menus

This kind of retrieval is aimed mainly at speeding up the retrieval process. Rather than searching through the whole of the database, (in the case where the user is not sure of the possible types of features), the action of specifying a type will reduce the search time by a factor of 3. The selection of features is then carried out by calling procedure 'retT+E' which analyses the query and then passes the job of retrieval to one of the three procedures described earlier; either 'retfromA'; 'retfromL' or 'retfromP'. The description of procedure 'retT+E' is as follows.

| Procedure | Function |
|--|---|
| retT+E(string atype, database.name; int class.code; pntr map.name) | The task of this procedure is to identify the type passed to it from the main body of the module. Once this has been done, it then calls the appropriate retrieval procedure to carry out the operation. This could be any one of the following:- 'retfromA'; 'retfromL' or 'retfromP'. In any case, it will also call procedure 'class.code'. Fig. 9.18 illustrates these calls. |

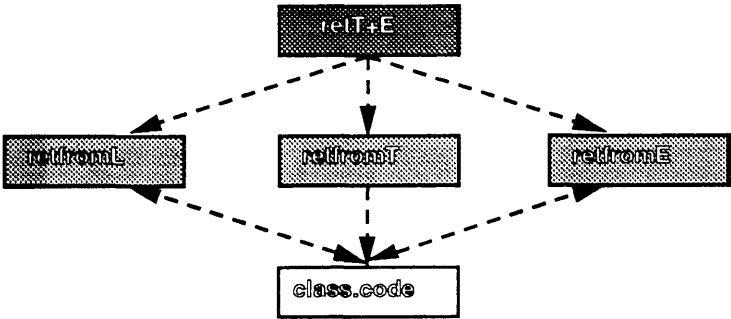


Fig. 9.18 The calls of procedure 'retT+E'

9.8 *Displaying the Results*

The results of the retrieval process are reported to the user at the time of search operation. However, the addresses of the retrieved data are also compiled in a vector so that they can be reached in a later process. This is done by storing the addresses (i.e. the feature identifiers) together with a snap-shot of the screen (in pixel format) in a database called '*MapImages*'. This is designed to help in sending the data to the output devices.

9.9 *Saving the Results*

As has been mentioned above in Section 9.8, the result of a retrieval session can be stored in the database '*MapImages*' for later usage. When the 'save' command is issued, the system replies by asking the user to supply a 'name' to that particular query session. The resulting information is then stored in the database. This database is structured so that it has, at its highest level, a table where the names of the queries are stored. Within this table, each name is supplied with a pointer to a structure holding information about the map for which the query was issued. This information may include the map name; the extent of the map; the grid interval; a snap-shot of the screen, including all the retrieved data; and finally four vectors of type 'integer' containing the identifiers of the retrieved features. Fig. 9.19 illustrates the design of the '*MapImages*' database. Since the retrieved data stored are only pointers, there is no duplication of the data.

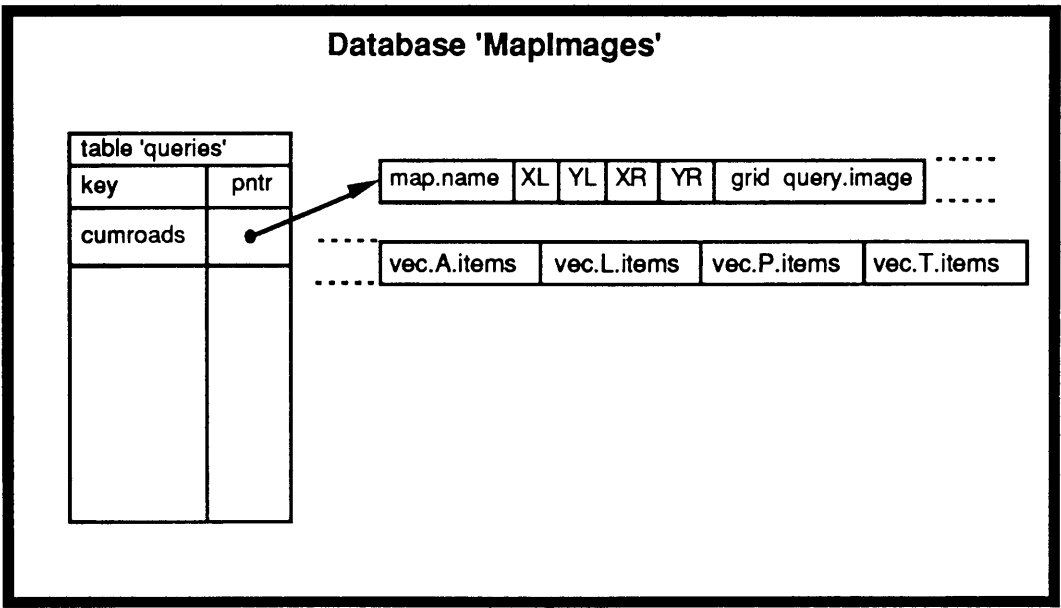


Fig. 9.19 The structure of the database '*MapImages*'

9.10 *Summary*

From the description of the Data Retrieval module, it is very clear that the adoption of the hybrid system of database structure described earlier in Chapter 5 is very useful for the generation of a wide scope of queries which is a particularly important feature for any information system. A further advantage has been the adoption of the object oriented provision that is supported by PS-algol which has made the implementation of these functions possible and practicable.

However, in this chapter, the different methods of data retrieval have been described with an emphasis on the ability of the system to reach those features which have very specific attributes. This has been done by providing the user with a variety of ways of identifying them.

CHAPTER 10

and given the fact that the two curves intersect at $x = 0$, we have $\frac{1}{2} = \frac{1}{2} + \frac{1}{2}$, which is not true. Therefore, the two curves do not intersect at $x = 0$. The only other possibility is that the two curves intersect at $x = 1$. In this case, we have $\frac{1}{2} = \frac{1}{2} + \frac{1}{2}$, which is true. Therefore, the two curves intersect at $x = 1$. The area between the two curves is the area of the region bounded by the two curves and the line $x = 1$. This area is given by the integral $\int_0^1 (1 - x) dx$, which evaluates to $\frac{1}{2}$.

PROBLEM 10

Let $f(x) = x^2 + 1$ and $g(x) = x^2 - 1$. The curves intersect at $x = -1$ and $x = 1$. The area between the two curves is the area of the region bounded by the two curves and the line $x = 1$. This area is given by the integral $\int_{-1}^1 (1 - x) dx$, which evaluates to $\frac{1}{2}$.

CHAPTER 10: HARD-COPY DATA OUTPUT

10.1 *Introduction*

This chapter is aimed at presenting the means by which the end result of queries can be presented in a hard-copy form. In this project, a raster-based laser printer has been used to produce the dumped images of the screen. Alternatively, a large-format vector-based central plotter (Calcomp 1039) linked to the ICL 3980 mainframe computer can be accessed via a graphic plotter driver. The graphics package used for this purpose is called 'Ghost 80' and has been developed by the Culham Laboratory of the United Kingdom Atomic Energy Authority (UKAEA). Ghost 80 is the latest version of the program suite which has been used as a standard graphics output package in the University since the early 1970s.

This chapter is divided so that Section 10.2 discusses the details of the retrieval of the map images which are to be sent to the output devices. Section 10.8 details the process of outputting to the raster-based laser printer. Section 10.8 describes the process of sending the results to the vector-based plotter. Finally, Section 10.9 is the summary.

10.2 *Module Organization*

The hard-copy output module is organized so that it deals first with the ways by which the user is able to select the map images from the database where they are stored. Secondly, the module deals with the retrieval of the images and the selection of the appropriate output devices. Finally it provides the means by which the hard-copy results may be obtained. Fig. 10.1 illustrates the functions carried out by the module.

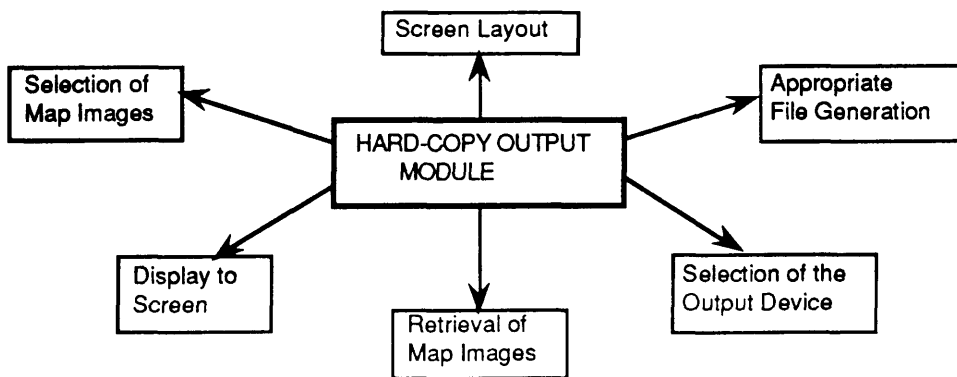


Fig. 10.1 The module organization

These functions are handled in either of two ways. The first involves calling module procedures, while the second is implemented by using direct commands from within the body of the module. The module makes use of eight procedures. These are:- '*drawline*'; '*rec*'; '*Rec*'; '*preperform*'; '*ShowSymbol*'; '*showOne*'; '*LaserDump*' and '*PlotDump*'. The first four procedures were described in Chapter 6, Section 6.5.2. The other procedures will be described later in this chapter. In the following sections, a description of each of the functions shown in Fig. 10.1 will be given.

10.3 *Screen Layout*

The screen layout in this module is similar to those described in the Cartographic Representation and Data Retrieval modules in Chapters 7 and 8. Hence use is made of procedure '*preperform*' (described earlier in Chapter 6, Section 6.5.2) and it is called from the global procedures 'utility' database.

10.4 *Selection of Map Images*

When the 'Output' module is selected from the menu of modules, the system will automatically open the '*MapImages*' database and scan the table entitled 'maps'. A menu of all the map images stored in the database is then generated from which the user can select the required map, Fig. 10.2 illustrates an example of this menu. This activity is taking place using the '*chooser*' procedure from the PS-algol utilities database.

| Choose Map Image |
|------------------|
| cum/Roads |
| cum/Grassland |
| cum/Lines |
| Quit |

Fig. 10.2 The selection of a Map Image from a menu

10.5 *Display to Screen*

Once a specific Map Image has been chosen, the stored image of the map (which is of a PS-algol image type) is then retrieved from the database '*MapImages*' and displayed on the screen.

10.6 *Retrieval of Map Images*

As has been discussed in Chapter 9, all the features comprising the map which has resulted from the selection process have their identifiers stored in the database '*MapImages*' as well. If the required map image is to be output to a plotter, then first of all, these specific identifiers are retrieved from the '*MapImages*' database. Then the features themselves are retrieved from the database '*MDB*' and have their pictures decomposed as will be seen later in Section 10.8.3.

On the other hand, outputting the map to laser printers only requires the retrieval of the image of the map from the database '*MapImages*'. This will be described in Section 10.8.1.

10.7 *Selection of Output Devices*

At this stage, the user is given the choice to send the output either to a text file (by implication to a plotter) or to an image file (which in practice means to a laser printer). This is done through a text-based menu with two options: (i) a text file output; and (ii) an image file output. Fig. 10.3 shows the menu from which the selection of the appropriate output file can be carried out.

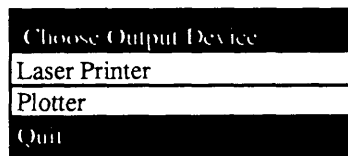


Fig. 10.3 The output selection menu

10.8 *File Generation*

Depending on the means of output, either the stored image of the resulting query result will be sent to a graphics file (in the case of laser printers) which will be described below in Section 10.8.1, or the pictures of the features concerned will be retrieved and be sent to two different files in the form of the sets of coordinate values and commands required for the cartographic representation of the data to be drawn by the central vector plotter. This will be described in more detail in Section 10.8.2.

Initially the result of the selection will be in the form of a PS-algol picture. In order to produce a hard-copy version of this picture, it has first to be transformed into a form which is acceptable to each of the output devices. The software which uses the laser printer requires the query to be transformed into a PS-algol image. To use the graphic plotter, on the other hand, the data must be stored in text files. Therefore Section 10.8.1 describes how an image is sent to a laser printer, while Section 10.8.2 describes the process of picking apart the data structure underlying a picture and outputting the data as a series of text files. Section 10.8.3 describes the PS-algol picture data structure. Section 10.8.4 details a general overview of the use of the Ghost package and Section 10.8.5 describes the program which calls the Ghost package.

10.8.1 *Output to Raster-Based Laser Printers*

When a laser printer is used to print out the resulting graphics, the stored map image is then processed by a program written by Dr. Cooper of the Department of Computing Science which has been incorporated into the GIS package. This program will transform the format of an image produced on the Sun work station into the Postscript format acceptable to the laser printer. It takes the whole of the current window and prints it out. Therefore the process of outputting the map image consists of drawing the resulting picture on the screen and then simply calling the printing procedure and dumping the screen image out to the laser printer. In fact, an Apple Laser Writer laser printer has been used in the present project, but in principle any Postscript compatible laser printer can be used for the purpose. The procedure responsible for doing this is called '*LaserDump*'.

Procedure

Function

LaserDump()

When the user selects the output to a laser printer, the module calls this procedure which copies the screen into an image file. Then it issues a system command to run program '*laserdump*' (which has been developed outside the system environment by Dr. Cooper of the Department of Computing Science). Fig. 10.4 illustrates the result of this procedure.

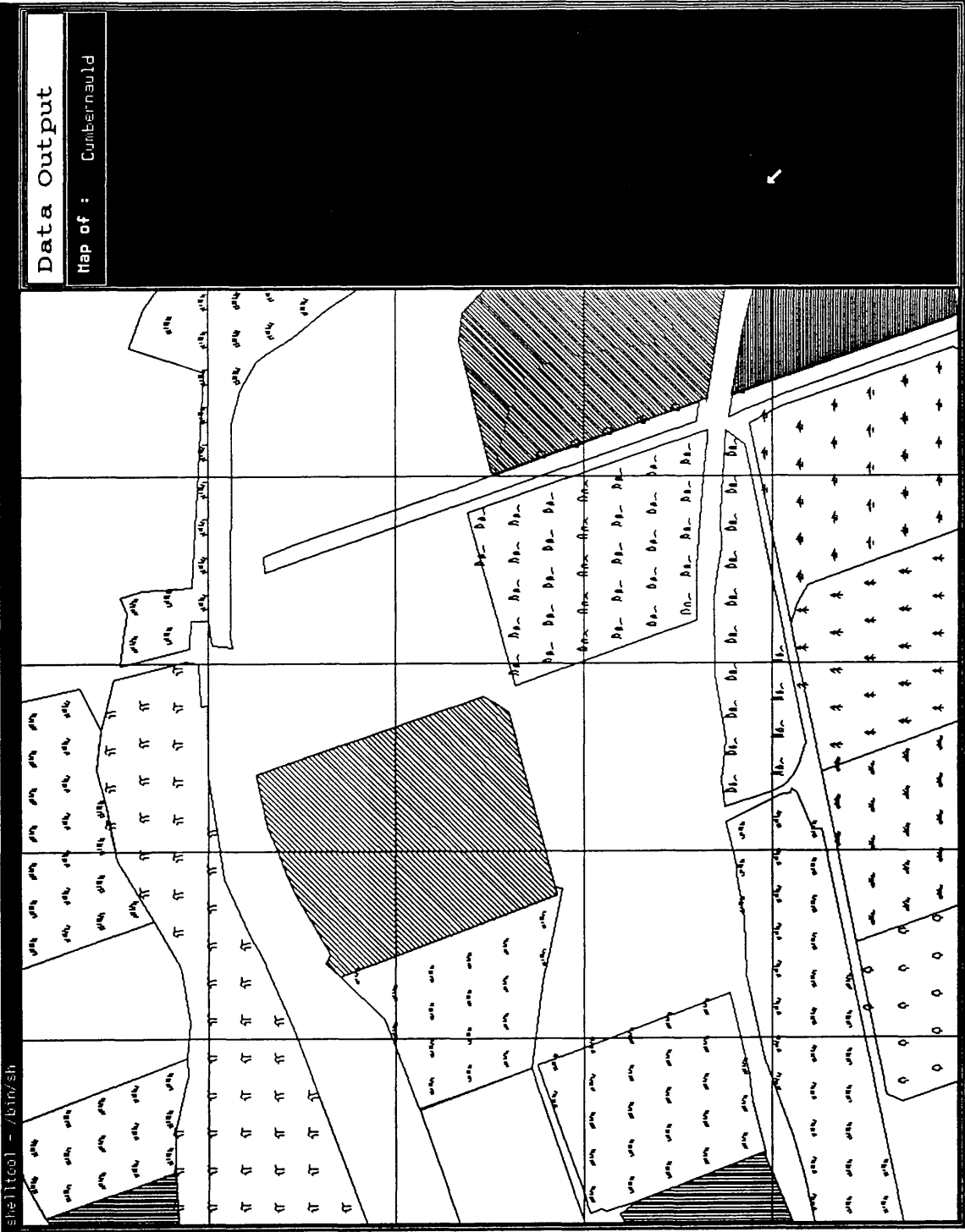


Fig. 10.4 The output from a laser printer

10.8.2 Output to a Vector-Based Plotter

As has been mentioned earlier in Section 10.6 (Retrieval of Map Images), the coordinate values of the map image are retrieved from the 'MDB' database and are sent to two different files. The first contains the coordinate values of all those features of type Polygon, Line and Point. Since polygons may require a specific cartographic representation (for instance, the style of filling) as well as requiring the lines defining the polygon itself (the perimeter) to be plotted, these two distinct pictures for polygons should be included in the output file as well. However, Line and Point features do not necessarily require their original data to be produced. For example, if a road has to be represented, the original data could be the digitized central line of the road, while the cartographic representation of the road is a double line. For this reason, the cartographic representations of Line and Point features are stored as series of x- and y- coordinate values of each component of the line or the point (for instance, the end points of the dashes in a dashed line). The second file holds the text data including the coordinate positions, font style, size and orientation of the text, etc. Tables 10.1 and 10.2 illustrate an example of the files produced to obtain hard-copy output using vector-based plotters.

Table 10.1: The features file

| | | | |
|-------------|-----------|-------|-------|
| Cumbernauld | | | |
| 775000 | 775000 | 80000 | 80000 |
| 3 | | | |
| 77438.040 | 78147.860 | | |
| 77437.460 | 78156.000 | | |
| 77447.100 | 78157.600 | | |
| 2 | | | |
| 77587.600 | 78931.520 | | |
| 77604.600 | 78933.700 | | |

Table 10.2: The text file

| | | | |
|-------------|-----------|---------------|-------|
| Cumbernauld | | | |
| 775000 | 775000 | 80000 | 80000 |
| 77123.060 | 78080.240 | well cou20 | 10 |
| 77688.940 | 78196.100 | Tr Mark cou20 | 10 |
| 77759.160 | 78556.320 | Rd Sign cou20 | 10 |

A program written in FORTRAN 77 is then run to call the graphics package Ghost 80 and to send these two files to the central plotter at the Computer Centre. This program will be described later in this chapter.

10.8.3 PS-algol Picture Data Structures

It is worthwhile mentioning the method by which pictures are stored in the persistent store. Pictures are stored as groupings of points. Points forming a picture can be related together

in either of two ways: i) either they are linked by lines (see Fig. 10.5) or ii) they form a complementary part of the picture but they are not linked all together (see Fig. 10.6).

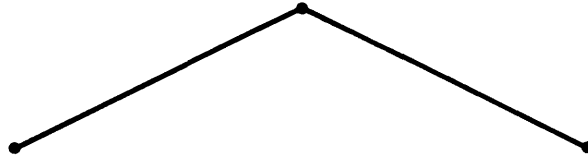


Fig. 10.5 This picture is formed by three points and two lines joining them

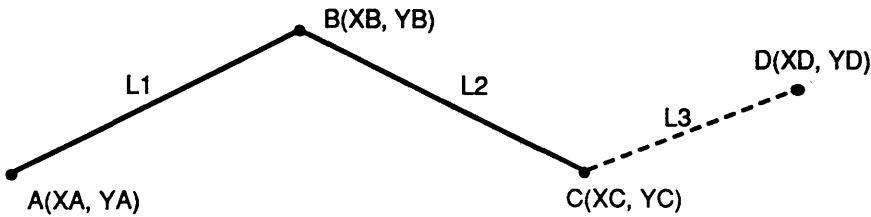


Fig. 10.6 The picture in this diagram is formed of four points 'A', 'B', 'C' and 'D'.

The first three are joined by lines 'L1' and 'L2' while 'D' just exists as an individual point

Points are stored in a structure called 'poin.strc' which holds the x- and y-coordinate values, while the linkages between the different points are kept in another structure called 'oprtn.strc', Fig. 10.7(a) illustrates structure 'poin.strc' and Fig. 10.7(b) illustrates that of 'oprtn.strc'.



Fig. 10.7(a) An illustration of structure 'poin.strc'

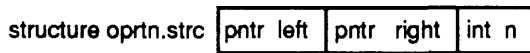


Fig. 10.7(b) An illustration of structure 'oprtn.strc'

A picture is then composed of a hierarchy of these structures. At the highest level, structure 'oprtn.strc' describes the picture in hand. This structure is composed of three fields:- i) a left pointer; ii) a right pointer; and iii) an integer (zero or one) to indicate whether or not this structure points to a lower level structure. ('0' means no and '1' means yes).

This structure will point left and right (where appropriate) to lower level structures. These could be either of type 'oprtn.strc' or 'poin.strc'. This process goes on until all the points of the picture have been consumed. Fig. 10.8 shows the hierarchy of the structures used for the picture shown in Fig. 10.6.

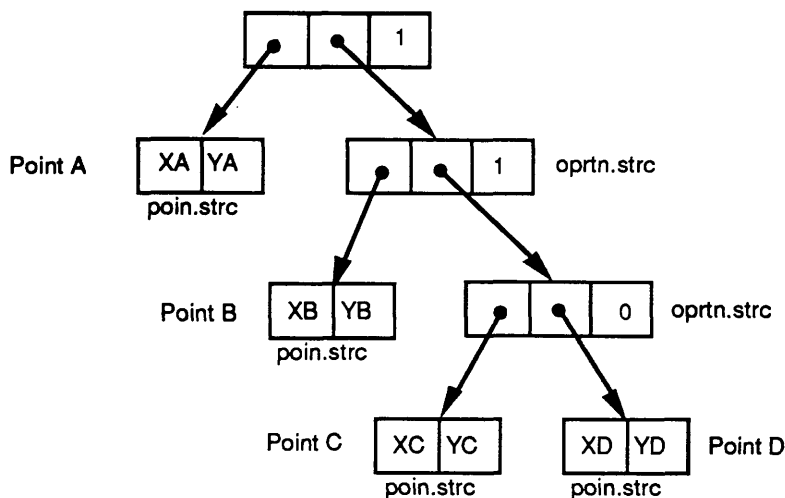


Fig. 10.8 The hierarchy of the building of structures to represent a picture

Two procedures are provided to allow the retrieval of the coordinate values of the graphic representation of pictures used from the persistent store. These are '*showOne*' and '*showSymbol*'. However, these two procedures are controlled by another called '*PlotDump*' which, in fact, manages the calls to be made to these two procedures and generates the output file for the coordinate values. These procedures are described below.

| Procedure | Function |
|---|---|
| <code>showSymbol(pntr P; string I; int W)</code> | This is a recursive procedure which, given a pointer to the location where a picture is stored in the persistent store, starts retrieving the coordinate values of the different points encountered while going through the hierarchy of the picture in either direction (left and right). At the same time, these coordinate values are then loaded into two vectors, one for the x-values and the other for the y-values. |
| <code>showOne(string S; pntr V)</code> | This procedure takes a pointer to a picture in the database and then calls procedure ' <i>showSymbol</i> ' to extract the coordinate values of the different points forming that picture. |
| <code>PlotDump(string OriginalMapName; pntr X-structure)</code> | This procedure generates two files. The first is to hold the coordinate values of the decomposed pictures retrieved from the database, and the second is to hold the text to be displayed with the |

graphics images. The procedure then retrieves a vector of the features' identifiers which were selected during a selection session and stored in the database '*MapImages*' and starts calling procedure '*showOne*' which in turn calls procedure '*ShowSymbol*' for each item of the vector retrieved. Once a feature picture has been through '*showOne*' and '*ShowSymbol*', its coordinate values are then obtained, '*PlotDump*' will then copy them into the generated file which by default will have the same name as the map image plus an extension. So if the map image name is for example 'cum/Rd', the coordinate file name will be 'cum/Rd.coo' and the text file will be 'cum/Rd.txt'.

10.8.4 Use of Ghost

By definition, Ghost is a comprehensive graphical output system, which means that it is a software package which can provide any program written in a high-level language with the ability to create graphic plots or images on any available graphics device. It is supplied in the form of a library of graphics subroutines which can be loaded along with the calling program [Prior, 1985].

Ghost 88 is the heavily revised version of the previous edition of the system (Ghost 77) which has been implemented on the ICL 3980 mainframe computer located in the Computer Centre. It is therefore accessible to any user of the ICL mainframe.

The use of the Ghost subroutines is very much like the use of other FORTRAN subroutines which are made using the command 'CALL'. For example, the statement - CALL POSITN(X, Y) - will cause the pen (in the case of a plotter) to move to the position defined by the X- and Y- coordinate values. Using Ghost, graphics files can be directed to either of the output devices available, namely a graphics display unit (screen) or any other attached plotter for hard-copy output, for instance the central plotter in the Computing Centre. The instruction with which the output device is specified is issued at run time after the program containing the call commands has been compiled. This instruction is as follows:-

G80(program.name, device.name)

where 'program.name' is the name of the program within which the calls are being made; and 'device.name' is the name of the output device to which the drawings should be sent, for instance 'ADM' is used for the graphics display unit and 'Gplot' is used for the central plotter at the Computing Centre.

10.8.5 *The Plotting Program*

The plotting program is composed mainly of three subroutines written in FORTRAN 77. Given the name of the file which has been mailed to the VME environment (without any extension) from the Sun graphics work station on which the GIS sits at present, the program will read the data and call the Ghost package to obtain the output.

i) The first subroutine is aimed at reading the heading of the file which is composed of two lines, the map name and the extent of coverage of the map. The values shown in the second line will help in producing a box within which all the graphics of the features will be drawn.

ii) Since every line is formed of a number of segments, this number is given at the beginning of any line. Thus, the second subroutine will read the number of points comprising that line and will call the Ghost package to position the pen at the first point specified by the X- and Y- coordinates. Then after having read all the points which follow, the line defining the feature is drawn. This will be carried out for all the features contained in the file.

iii) The third subroutine reads its data from the text file. It reads the X- and Y- coordinate values of a point, then the text and the size and finally calls the Ghost package to display the text in the specified location. This is done for all the text contained in the file. Fig. 10.9 illustrates an example of the output obtained from this program.

10.9 *Summary*

In this chapter, the hard-copy output of the result of queries has been discussed. The supported output devices are laser printers and the central plotter at the Computer Centre. However, there should have been a program to allow direct access to the plotters from the

Sun work station but this could not be done due to the lack of availability of plotters at the Department of Computing Science. For this reason, the resulted data files had to be 'mailed' to the VME environment and then the 'Ghost' package was used to obtain high-quality large format maps.

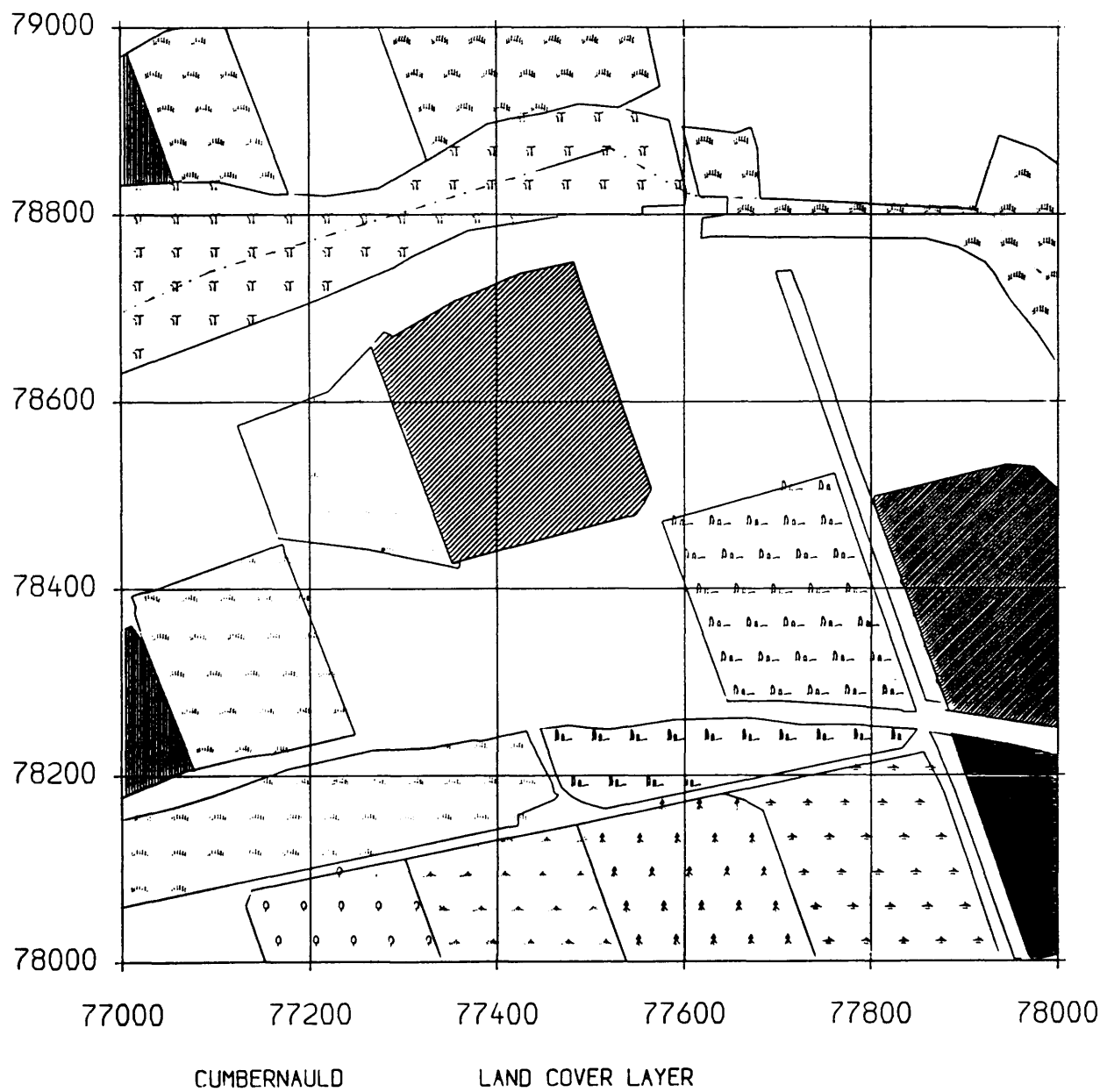


Fig. 10.9 The output from a plotter

CHAPTER 11

CHAPTER 11: CONCLUSION & RECOMMENDATIONS

PART I: Conclusion

11.1 *Introduction*

Having covered the different aspects of the project and the various components of the new GIS system, the experience gained with PS-algol in terms of learning the language, modelling the problem, and storing and retrieving the data will be discussed in this chapter. Also various aspects of the implementation of a prototype GIS based on the novel architecture permitted by the PS-algol language will be reviewed and discussed, including comparisons with existing GIS systems.

11.2 *Learning the Language*

PS-algol is presented to users in a 40 page reference manual [PPRR 12], which is also supported by 'An Introduction to PS-algol' [PPRR 31] documented in 107 pages. Graphics operations are also explained explicitly in two other pieces of documentation entitled 'Implementation Issues in Persistent Graphics' [PPRR 23] and 'An Integrated Graphics Programming Environment' [PPRR 14]. For advanced programmers, there are two other references: 'User Interface Tools in PS-algol' [PPRR 56] and 'Applications Programming in PS-algol' [Cooper, 87].

It should be said here that, in spite of the apparently voluminous reference material, these documents are not really designed to enable a beginner to learn the language easily nor to assist him in exploring its potential application to a problem. During the development of the system, most of the difficulties which arose were only resolved with the help and advice of those staff members in the Department of Computing Science who have been actively involved in the development of the language and have had extensive experience with it. It should also be mentioned here that, at certain stages, without the direct assistance of these staff members (especially Dr. Cooper) in explaining many aspects of the language which were not well documented, it would have taken a very much longer time to achieve the results presented in this report. While the present author has had good access to this advice, obviously other users who might not have this close relationship with the developers will experience many difficulties.

It is obvious therefore that further explanatory documentation of a professional standard covering the various aspects of the PS-algol language should be produced in order that the programmer does not need to consult with or refer to the developers of the language. This is the case with other commonly used languages and certainly this is an urgent matter if the language is to move from its present rather experimental status to become a widely used language. Since the language has so many attractive and useful features, it would be a great pity if potential users were put off or frustrated using it because of lack of professionally produced documentation.

11.3 *Properties of the Language*

The principal properties of the language discussed earlier in Chapter 4 are:-

- a) the principle of data type completeness;
- b) the principle of abstraction;
- c) the principle of correspondence;
- d) the inclusion of pictures and images as data type objects;
- e) and, most importantly, the principle of data persistence, which makes the language relatively easy to work with.

Each of these properties is discussed in some detail below:-

- a) The main advantage of applying the principle of data type completeness is that procedures are first class types, which means that there is a simple and well understood mechanism for modular system construction using procedure calls. The safety of the system is provided by the type mechanism of the language which guarantees the type checking for data of any persistence. A further advantage of data type completeness is that there is no need to remember arbitrary restrictions placed on the handling of different data types.
- b) On the other hand, the power and facilities of abstract data types can be obtained from encapsulating data in a procedure which yields as its result a record containing other procedures to manipulate this data. Separate compilation of modules may be achieved by storing the procedures in the persistent store.
- c) The principle of correspondence applied in PS-algol allows the programmer to declare

and use names in the same way anywhere in a program, thus allowing him to make his declarations to be as close as possible to the use of the variable and to allow logically related code to be grouped into modules. This also means that all aspects of the program - computations, data storage and retrieval and the user interface - are all written in a consistent style.

d) Another excellent feature of PS-algol is the availability of 'picture' and 'image' constructs which help in the creation of graphical systems. When pictures and images are given the same rights as any other data type of the language (complying with the principle of 'data type completeness'), this allows pictures and images to be assigned, passed, stored in and consequently retrieved from databases. Also, most importantly, they can be concatenated to form complex pictures or images which can then be manipulated as an individual feature. Clearly, the storage of pictures and images in a database is a fundamental problem when implementing a GIS. Therefore having explicit operations to do so makes PS-algol a suitable vehicle for the implementation of a GIS.

e) The provision of data persistence as an implemented feature of PS-algol provides programmers with the facility to manipulate data whose lifetime does not extend beyond the program activation (as is the case with other traditional languages), but also to be able to manipulate persistent data with the same ease. Again, this is a most important attribute in the context of cartographic data where the base material is definitely persistent, while the results of an analysis carried out by the GIS may be quite transient.

Introducing the idea of persistence to languages such as PS-algol, also reduces the amount of source code considerably, particularly that part of the code concerned with transferring the data to and from the storage system. In traditional languages, this mechanism is performed in three phases: (i) considering the real world application area and modelling it in data structures inside a program; (ii) the modelling of the database and devising the types of structures needed for long term storage; and (iii) the mapping between these two models. The maintenance of this mapping is quite a difficult task and is costly in terms of time and effort. A persistent language uses a single model to represent data in the program and in the backing store.

Another advantage of the provision of persistence is the solid type protection which is offered by PS-algol, whereas that offered by other programming languages is often lost

across the mapping [Atkinson & Morrison, 1986]. Therefore, the programming environment provides protection against the misuse of data by handling it as it were of a different type.

All these properties implemented in PS-algol help to make the language easier to learn and understand. Thus the ability to build the model in hand is a simpler task to carry out.

11.4 Modelling the Real World

PS-algol offers a very good environment in which to build and develop models of the real world. This is because of the built-in database management system which allows the programmer direct access to the database rather than needing other foreign means or another system to carry out the task and which, more often than not, set obstacles between the programmer's visualization and the final realization of the model. This results in less written source code and in turn implies a better control (from the programmer's point of view) over the progress of the development of the program.

11.5 Data Storage and Retrieval in PS-algol

Integrating a Database Management System within a programming language (which is in effect what PS-algol achieves) makes the task of building large programs easier. Since programmers can use the language commands to define their schemas, records, data attributes, etc., there is no need for the inclusion of any foreign routines from outside the language domain.

The process of manipulating data is well supported in PS-algol. As has already been said, the same data structures are used in the program as in the long-term store. Storage of a complex composite data object consists merely of entering a reference to the 'top' of the object into the persistent store and 'committing' this transaction.

In PS-algol, the transaction mechanism makes the concurrent revision of data values safe. The effects of a transaction are not visible to other transactions until the transaction has been committed. Programs starting after this stage will use the new version of the data for the whole program execution.

Data retrieval mechanisms are also supplied. Retrieving a complex composite object is a single operation. The object is then traversed by the operations of the program which would have been required even if the data are not persistent.

11.5 *Modular Programming*

With any large and complex system, as is the case with a GIS which is composed of several inter-related applications, it is desirable and often necessary to construct the system from separately compiled pieces of code. Therefore, a modular approach to its construction is essential, so it is necessary to be able to compile separate 'modules' and then link them together.

In PS-algol, this is quite possible since procedures are first class data objects which means that procedures can be assigned or they can be passed from one procedure to another. A procedure may also be the result of expressions or other procedures. Yet again, they may be the elements of structures or vectors. Finally and most importantly, they can be compiled and stored in databases. Thus the process of writing a program in PS-algol consists of writing small procedures and storing them in the database for use by lower level procedures. Management support for the process of handling these many procedures could be extremely valuable and is in the course of being developed.

11.7 *Speed of Processing*

It is worth mentioning here that the relatively slow speed of processing the code contained in PS-algol programs, is one of the main factors against the language, since it tends to be rather slow especially when dealing with graphical data. This could be the result of the type checking procedure or the binding mechanism described earlier in Chapter 4. On the other hand, the language is still experimental, and there is no reason why optimization cannot be applied to it to improve the speed of processing, given industrial quality backing.

It should also be mentioned here that the speed can be improved by using faster hardware. For example, at the beginning of the research carried out for this project, the author used the ICL Perq machines which, at that time, were installed in the Department of Computing Science. With these machines, the compilation and running of programs was extremely slow. For example, to run the program 'incode' which loads the feature coding system into

the database, took about twenty four hours, but when this same program was run on the Sun 3/50 machines in the same Department, the time spent was about 2 1/2 mins. The Department of Computing Science has just installed another generation of the Sun work station - a RISC-based Sun 4/80. These computers can run at a still higher speed which will inevitably reduce greatly the problem of speed mentioned above.

11.8 *The Prototype GIS*

Having surveyed a number of the systems available and marketed in the field of GIS, as reported in Chapter 3, the obvious point that can be made about all of these systems is that they have two quite different types of database:-

- i) a database that holds the textual data, the numerical data, etc., and
- ii) a graphical database.

These two databases are then linked together via different mechanisms, such as the bucket in the case of Intergraph. Undoubtedly the need for such linkages will make the development and maintenance of the system a substantial piece of work to carry out.

On the other hand, those systems which require other foreign systems, such as an existing RDBMS in the case of most current relational GISs, need a considerable effort to achieve the necessary degree of integration between the component systems, so that the databases can be queried. Also, some other systems (like for instance EMIS) need the information part of another foreign system to be integrated with their own system to produce the final system.

Thus it can be seen that normally the task of producing a GIS is not a simple one. However, using PS-algol, the author has managed to build the prototype GIS described in this thesis so that it provides a unique approach in which the graphical constructs of the features are stored in the system side by side with the corresponding attributes of the features. This has resulted in far less use of pointers and identifiers and consequently there is a much smaller programming burden to be carried on the shoulders of the programmer.

Also using a separate feature coding system, which is then linked to the features held in the

database, is a significant help in structuring the data. As a result, this has improved immensely the quality and the way in which queries are analyzed and the results are retrieved. It has also helped the storage aspects of the data by reducing the length of the records used in the features database.

PART II: Recommendations

11.9 Introduction

Within the scope of the time and resources available for the project, the system may be regarded as a satisfactory development. However, as it stands at the moment, it should be regarded as a thematic mapping information system rather than a full GIS. However, it should form a very good basis for further developments and enhancements to be carried out and implemented upon it.

11.10 General Recommendations

Some of the recommendations suggested by the author are of a more general nature while others are concerned with specific modules. The general recommendations for developing the system include:-

- i) the inclusion of an indexing facility to reference entities of the real world to some known locations so that these features can be accessed either by location or by feature code. This means that features, for example, can be indexed according to their grid squares within a relation in the database where their identifiers are also stored.
- ii) the inclusion of help menus to assist the user at the different stages of the running of the program;
- iii) the introduction of the facility which would allow the user to enter commands from the keyboard as well as using menus, the pointer and the mouse; and
- iv) the ability to allow users to define their own symbols and introduce them to the database.

11.11 Modules Recommendations

On the other hand, further developments at the level of the modules could be carried out as

follows:-

11.11.1 *Data Entry*

Data entry to the system should be made available from different sources and different methods of data collection. In particular, the system should be able to read in data in different formats such as OSTF used by the Ordnance Survey; SIF used by Intergraph; and the DXF format used by Autocad and other CAD systems. Also the introduction of a module which allows the user to enter data directly from a range of commonly used digitizing tablets (GTCO, Summagraphics, Calcomp, etc.) could be of great benefit to the system.

11.11.2 *Cartographic Representation*

Although the cartographic representation has taken up a major part of the development of the system, it should be enhanced further by introducing more symbologies for the three types of features, points, lines and polygons. For example, buildings could be denoted by their activities, such as hospitals or swimming pools; roads could have arrows to indicate traffic directions; etc.

11.11.3 *Data Retrieval*

As it stands at the moment, data retrieval deals only with the graphic data. However this is definitely not the only kind of data that the user may need, so other types of data should also be made available to the user. For example, by providing a menu of the fields of the three types of data when the selection of features has taken place, the user may then select the fields that he wants to look at or retrieve.

11.11.4 *Data Output*

Data output goes hand in hand with data retrieval. The system is capable of producing output to laser printers as well as plotters. However, the system should also be able to produce textual data in the form of tables, reports, etc. This should be carried out as well by developing a report generation module that can be linked to both the data retrieval and the data output modules.

11.11.5 Applications Module

A completely new module should be included into the system to allow for further processing of the data and to upgrade the current system fully into the GIS domain. This module should contain different applications such as polygon overlay and intersections; line intersection with polygons; point in polygon searches; network operations, etc. Other analysis features such as attribute analysis; interpolation; and map projection and edge matching could also be provided to help transform the present prototype system into a fully featured and operational GIS.

11.12 Summary

An immediate conclusion that can be drawn from this thesis is that the use of PS-algol as the sole language to develop the system described in this thesis took place over the comparative limited time of three years. This was the case in spite of the fact that most of the first year of the author's research period was spent in learning the language and attending database design and management courses in the Department of Computing Science. It can be seen clearly that the language reduces greatly the time and the amount of source code needed to develop the system. Thus it reduces the effort required to be spent on the database design and interfaces. In particular, it eliminates the need for the construction of a different database for the graphical data, as was the case with all the systems described in Chapter 3. All of which decreases the burden on the shoulders of the system designer.

Employing the hybrid database system discussed in Chapter 5, has also succeeded in reducing the search time procedure by an average factor of three, since, when entity retrievals are required, the search will be carried out over one third of the data contained in the database and not the entire database.

Using the separate but linked hierarchical database for the feature coding system made the selection of items to be retrieved systematic and the actual retrieval procedure simpler. Furthermore, the use of record identifiers rather than copying the retrieved data to another structure eliminates any duplication of data.

The user interface developed in the system has been done through menus and icons, which

reduces the possibility of entering wrong data in a wrong format, and thus can be considered as friendly.

Finally, the modular implementation of the system allows the future expansion of any of the modules described earlier or the introduction of new ones.

11.13 *Epilogue*

In the author's opinion, this project is potentially of great importance in the field of GIS since it approaches the topic from a different point of view, and attempts to deal with a number of the problems existing at the core of the GIS rather than those occurring at the periphery. It tackles in a new way the design and the construction of the GIS database upon which the end product will be judged for failure or success.

On the other hand, this project has also been of some importance to the Department of Computing Science and, in particular to the Persistent Programming Research Group, since the project has provided a direct and real-life application which allows the PS-algol language to be fully exercised.

Finally and personally, the project has extended greatly my knowledge of several different yet related fields. A considerable knowledge of the field of Digital Mapping and GIS is a prerequisite to carry out such a project, and this has gradually been acquired over the period of this research. Also, extensive programming experience and a knowledge of the design of databases and database management systems were needed. Again the project has been the vehicle through which these have been acquired albeit somewhat painfully.

In summary, the experience gained through the different stages of the project has included:-

- i) the knowledge of the different data capture procedures, operations, instrumentation, and data formats;
- ii) knowledge of GIS system design, functions, analysis, and manipulations.
- iii) a broad view of most of the better known GIS systems currently available on the market, together with some insight into their respective approaches to the design of their databases, functions, etc.
- iv) structured programming in Pascal;
- v) database design;

- vi) database management system functions and operations;
- vii) structured programming in PS-algol; and last but not least
- viii) some idea as to how to carry out research in the fields of Topographic and Computing Science, integrating concepts from both disciplines.

Hopefully, all of these will be of great value in my future professional career.

BIBLIOGRAPHY

BIBLIOGRAPHY

- Abel D.J., 1988. *Relational Data Management Facilities for Spatial Information Systems*, Proceedings, Third International Symposium on Spatial Data Handling. Sydney: 9-18.
- Anon, 1985. *Lab-Log/Software*. Laboratory for Computer Graphics and Spatial Analysis, Harvard University. Cambridge, Mass.: 23 - 27.
- Atkinson M.P. & Buneman O.P., 1987. *Types and Persistence in Database Programming Languages*, ACM Computing Surveys, 19, 2: 105-190.
- Atkinson M.P. & Morrison R., 1986. [PPRR-19] *Integrated Persistent Programming Systems*, International Conference on System Sciences, Hawaii.
- Atkinson M.P. & Morrison R., 1989. *Persistence - Where Next?* in J. Rosenberg (Ed.) Proceedings, 3rd International Workshop on Persistent Object Systems - Their Design, Implementation and Use, Newcastle, N.S.W.
- Atkinson M.P., 1978. *Programming Languages and Databases*, Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (ed. S.P. Yao), IEEE: 408-419.
- Atkinson M.P., Bailey P., Cockshott W.P., Chisholm K.J. & Morrison R., 1984. [PPR-8-84] *Progress with Persistent Programming, Database-Role and Structure*, Cambridge University Press.
- Atkinson M.P., Morrison R. & Pratten G.D., 1986. [PPRR-21] *A Persistent Information Space Architecture*, IFIP'86 Conference, Dublin.
- Blakeman D.A., 1987. *Some Thoughts About GIS Data Entry*. ASPRS, GIS' 87 - San Fransisco, I: 226 - 233.
- Bouille F., 1978. *Structuring Cartographic Data and Spatial Processing with the HBDS*. Harvard Papers on Geographic Information Systems, Cambridge, Mass.
- Brown A.L. & Dearle A., 1986. *Implementation Issues in Persistent Graphics*, Department of Computational Science, University of St. Andrews.
- Burrough P.A., 1985. *Principles of Geographical Information Systems*. Clarendon Press, Oxford.

- Byrne S.T. & Neil L., 1983. *Application of the WildMap System in a Production Environment*, Photogrammetric Record, XI, 61: 47 - 52.
- Byrne S.T., 1986. *Digital Mapping: Some Commercial Experiences*, Photogrammetric Record, XII, 68: 143 - 154.
- Canadian Council on Surveying and Mapping, 1984. *National Standards for the Exchange of Digital Topographic Data*, Draft Reports Published by the Topographic Survey Division, Surveys and Mapping Branch, Ottawa, Ontario.
- Carrick R., Cole J. & Morrison R., 1986. [PPRR-31] *An Introduction to PS-algol Programming*, Dept. of Computational Science, University of St. Andrews.
- Charlwood G., Moon G. & Tulip J., 1987. *Developing a DBMS for Geographic Information- A Review*, Wild Heerbrugg, Switzerland.
- Cimon N. & Quigley T., 1986. *Evolution of a Geographic Information System: Integration into the Oregon Range Evaluation Computing Facility*. Proceedings of ASPRS "Geographic Information System Workshop": 99 - 109.
- CODASYL, 1971. *Codasyl Data Base Task Group Report*, Codasyl Committee on Data System Languages, Technical Report, ACM.
- Codd E.F., 1982. *Relational Database: A Practical Foundation for Productivity*. Communications of the ACM, 24: 109 - 117.
- Coe P.K. & Quigley T.M., 1986. *Application of a Geographic Information System for the Oregon Range Evaluation Project*. Proceedings of ASPRS "Geographic Information System Workshop": 88 - 98.
- Cooper R., 1987. *Applications Programming in PS-algol*, Department of Computing Science, University of Glasgow.
- Cooper R., 1988. [PPRR-56] *A Utility Library for PS-algol*, in User Interface Tools in PS-algol, Department of Computing Science, University of Glasgow.
- Cooper R., 1989. *On the Utilization of Persistent Programming Environment*, Published Ph. D. thesis, Department of Computing Science, University of Glasgow.
- Dangermond J., 1985. *A Review of Digital Data Commonly Available & Some of the Practical Problems of Entering them into a GIS*, ESRI, Redlands, CA.
- Dangermond J., 1986. *The Software Toolbox Approach To Meeting The User's Needs For GIS Analysis*. Proceedings of ASPRS "Geographic Information System

Workshop": 66 - 75.

- Date C.J., 1981(a). *An Introduction to DataBase Systems*, 3th ed., Addison-Wesley, Reading, Mass.
- Date C.J., 1981(b). *Referential Integrity*, in the Seventh International Conference on Very Large Data Bases, Cannes, France.
- Dueker K., 1985. *Geographic Information Systems: Toward a Geo-relational Structure*, Proceedings of AutoCarto 7: 172-177.
- Eastman J.R., 1987. *Mapping Out a Plan of Action*, Digital Review August 3, 1987, Davis Publishing Co.: 1 - 7.
- Egenhofer M.J. & Frank A.U., 1988. *Designing Object-Oriented Query Languages for GIS: Human Interface Aspects*, Proceedings, Third International Symposium on Spatial Data Handling. Sydney: 79-96.
- Elgarf T.M., 1986. *The Role Of A Mapping Company In (GIS) Field*, Proceedings of Geographic Information Systems Workshop, Georgia.
- Exler R.D., 1987. *Appropriate Uses of Topology in Geographic Information Management Systems*. ASPRS, GIS' 87 - San Fransisco, I: 234 - 238.
- Gatrell A.C. & Charlton M., 1987. *ODYSSEY: A Low Cost Package for Geographical Information Systems Research*, Northern Regional Research Laboratory, Research Report No 6.
- Gittins D., 1986. *Query Language Systems*, Edward Arnold (Publishers) Ltd.
- Goh P.C., 1989. *A Graphic Query Language for Cartographic & Land Information Systems*, International Journal of Geographical Information Systems, 3, 3: 245-255.
- Haralick R.M., 1980. *A Spatial Data Structure For Geographic Information Systems*. in Map Data Processing, editors Freeman H., and Pieroni G.G.: 63 - 99.
- Harrington S., 1987. *Computer Graphics, A Programming Approach*, 2nd ed. McGraw-Hill.
- Horowitz E. & Sahni S., 1984. *Fundamentals of Data Structures in PASCAL*, Computer Science Press.

- IBM, 1982. *IMS/VS General Information Manual*. Form No. GH20-1260. IBM. White Plains. N.Y.
- Interview, 1987 *Spatial Editor, Spatial Analysis: Topology in the Vax Environment*, Interview J., Third Quarter, 6, 3: 28-29.
- ISO, 1987. *Information Processing Systems - Database Language SQL*, ISO Standard 9075.
- Jones A.K. & Liskove B.H., 1978. *A Language Extension for Expressing Constraints on Data Access*, Communications of the ACM, 21(5): 358-367.
- Klein D.H., 1987. *Combining Both GIS and CADD Capabilities in a Single PC-Based Automated Mapping System for a Small Incorporated City*. ASPRS, GIS' 87 - San Francisco, II: 730 - 738.
- Levinsohn A., Langford G., Rayner M., Rintoul J. & Eccles R., 1987. *A Microcomputer-Based GIS for Assessing Recreation Suitability*. ASPRS, GIS' 87 - San Francisco, II: 739 - 747.
- M & S Computing, Inc., 1978. *Interactive Graphic Design System (IGDS)- Municipal & Utility Applications*, Report No. 78-110.
- M & S Computing, Inc., 1979. *Introduction- Data Management & Retrieval System (DMRS)*, Report No. 78-057.
- Macro A. & Buxton J., 1987. *The Craft of Software Engineering*, Addison-Wesley.
- Maggio R.C. & Wunneburger D.F., 1986. *A Microcomputer-based Geographic Information System for Natural Resource Managers*. Proceedings of ASPRS "Geographic Information System Workshop": 296 - 300.
- McFadden F.R. & Hoffer J.A., 1988. *Data Base Management*. 2nd ed., Benjamin/Cummings Publishing Company, Inc.
- McGregor J. & Watt A., 1986. *The Art of Graphics for the IBM PC*, University of Sheffield, Addison-Wesley.
- Menon S. & Smith T.R., 1989. *A Declarative Spatial Query Processor for Geographic Information Systems*. J. Photogrammetric Engineering and Remote Sensing. LV, 11: 1593 - 1600.
- Morrison R., Brown A.L., Dearle A. & Atkinson M.P., 1986. [PPRP-14] *An Integrated Graphics Programming Environment*, EUROGRAPHICS UK Conference,

University of Glasgow.

- Morrison R., Dearle A., Bailey P.J., Brown A.L. & Atkinson M.P., 1985. [PPRR-15] *The Persistent Store as an Enabling Technology for Integrated Project Support Environment*, International Conference on Software Engineering, Imperial College, University of London.
- Nijssen G.M., 1980. *Database Semantics*, in Atkinson, M. (ed.), Infotech State of the Art Report on Database, Infotech.
- Olle T.W., 1978. *The Codasyl Approach to Data Base Management*, John Wiley & Sons.
- Oracle Corporation, 1984. *SQL/UGI Reference Guide*, Oracle Corporation, Menlo Park, California.
- Oracle Corporation, 1985. *Introduction to SQL*, Oracle Corporation, Belmont, California.
- Oracle Corporation, 1987. *Oracle, The SQL Development Method*, Oracle Corporation, Belmont, California.
- Organick E.I., 1972. *The MULTICS System*, MIT Press, Cambridge, Massachusetts.
- Palimaka J., Halustchack O. & Walker W., 1986. *Integration of a Spatial and Relational Database Within a Geographic Information System*, ACSM Annual Spring Meeting, Washington, D.C.
- Parker D., 1990. *Land Information Databases*, in 'Engineering Surveying Technology', eds. T.J.M. Kennie & G. Petrie,; 427-477.
- Parker H.D., 1987. *What is a Geographic Information System?* ASPRS, GIS' 87 - San Fransisco, I: 72 - 80.
- Parker H.D., 1988. *The Unique Qualities of a Geographic Information System: A Commentary*. J. Photogrammetric Engineering and Remote Sensing, LIV, 11: 1547 - 1549.
- Parker H.D., 1989. *GIS Software 1989: A Survey and Commentary*. J. Photogrammetric Engineering and Remote Sensing, LV, 11: 1585 - 1591.
- Parks B.O. & Simmons G.A., 1987. *Managing Cartographic Modeling and Linneage Problems Using a Microcomputer*. ASPRS, GIS' 87 - San Fransisco, II: 748 - 756.

- Petrie G., 1990. *Digital Mapping Technology: Procedures and Applications*, in Engineering Surveying Technology. eds T.J.M. Kennie & G. Petrie: 329-390.
- Peuquet D.J., 1984. *A Conceptual Framework and Comparison of Spatial Data Models*, Cartographica, 21, 4: 66-113.
- Peuquet D.J., 1987. *Data Models for Very Large Geographic Databases*, Proceedings of International Workshop on Geographical Information System, Beijing: 149-162.
- PPRR 12, 1987. *PS-algol Reference Manual*, 4th ed. University of Glasgow and University of St. Andrews.
- Prior W.A.J., 1985. *The Ghost-80 User Manual*, Release 7. UKAEA, Culham Laboratory, Abingdon. Oxon. England
- Radwan M.M., Kure J. & Al-Harthi M., 1988. *Data Structure in Topographic Data Bases*, ITC Journal, 1988-4: 327 - 331.
- Scheitlin T.E., 1986. *A Professional Geographic Management System*. Proceedings of ASPRS "Geographic Information System Workshop" : 31 - 36.
- Shapiro L.G. & Haralick R.A., 1980. *Spatial Data Structure*. Geoprocessing. 1: 313 - 337.
- Shapiro L.G., 1980. *Design of a Spatial Information System*, in Map Data Processing, eds. Freeman H., and Pieroni G.: 101 - 117.
- Simon H.A., 1987. *Data Organization in System 9*, Wild Heerbrugg, Switzerland.
- Smith J.M., 1971. *GPL/1 - APL/1 Extension for Computer Graphics*, AFIPS SJGC: 511-528.
- Synercom Technology Inc., 1984, *EMIS Software Information Kit*, Synercom, Sugar Land, Texas.
- SysScan, 1986. *GINIS BASIC*, Product Description Rerport PD-014, Release 4.1, SysScan, Kongsberg. Norway.
- SysScan, 1988. *DNMS, Map Information System for Utilities*, SysScan, Kongsberg. Norway.
- Ullman J., 1982. *Principles of Database Systems*, Computer Science Press, Rockville, Md.

- Van Roessel J.W. & Fornight E.A., 1985. *A Relational Approach to Vector Data Structure Conversion*, Proceedings of AutoCarto 7,: 541-551.
- Van Roessel J.W., 1986. *Design of a Spatial Data Structure Using the Relational Normal Forms*, Proceedings, Second International Symposium on Spatial Data Handling. Zurich: 251-272.
- Walker W., Palimaka J. & Halustchak O., 1987. *Designing a Commercial GIS - A Spatial Relational Database Approach*, Proceedings of International Workshop on Geographical Information System, Beijing: 342-351.
- Wild, 1980(a). *Informap: Interactive Graphic Mapping & Data Base System*, Wild Heerbrugg. Switzerland.
- Wild, 1980(b). *Wildmap: Interactive Photogrammetric Data Base Mapping System*, Wild Heerbrugg. Switzerland.
- Yoeli P., 1982. *Cartographic Drawing with Computer*, eds. McCullagh M.J. & Mather P.M., Computer Applications, 8, Nottingham, England.





UNIVERSITY OF GLASGOW

**THE DESIGN AND IMPLEMENTATION
OF
A PROTOTYPE GEOGRAPHIC INFORMATION SYSTEM
USING A NOVEL ARCHITECTURE
BASED ON PS-ALGOL**

BY

ABDULHAKIM A. ABDALLAH

B. Sc. (Civil Eng.),

P.G.Diploma (Photogrammetry & Remote Sensing)

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
March 1990

Table of Contents

VOLUME II

APPENDIX A: THE FEATURE CODING SYSTEM

- A.1 Introduction
- A.2 The Feature Coding System
- A.3 Special Codes for Feature Attributes
 - A.3.1 Unspecified
 - A.3.2 Other
 - A.3.3 Abandoned
- A.4 The Feature Coding System Listing

APPENDIX B: CREATION OF THE DIFFERENT DATABASES

APPENDIX C: GLOBAL PROCEDURES

APPENDIX D: DATA ENTRY MODULE

APPENDIX E: CARTOGRAPHIC REPRESENTATION MODULE

APPENDIX F: DATA RETRIEVAL MODULE

APPENDIX G: HARD-COPY DATA OUTPUT MODULE

APPENDIX H: OPERATIONAL MANAGEMENT SYSTEM PROGRAM

APPENDIX A

APPENDIX A: THE FEATURE CODING SYSTEM

A.1 Introduction

Knowing that topographic features are identifiable entities within the topographical environment, these features can be grouped together according to specified criteria. According to the classification diagram in Fig. A.1, topographic features are classified into nine classes forming Level I of the classification. Each class is then subdivided into several categories resulting in Level II. Then under each category, the features themselves are listed in Level III, and finally, the feature attributes are given in Level IV.

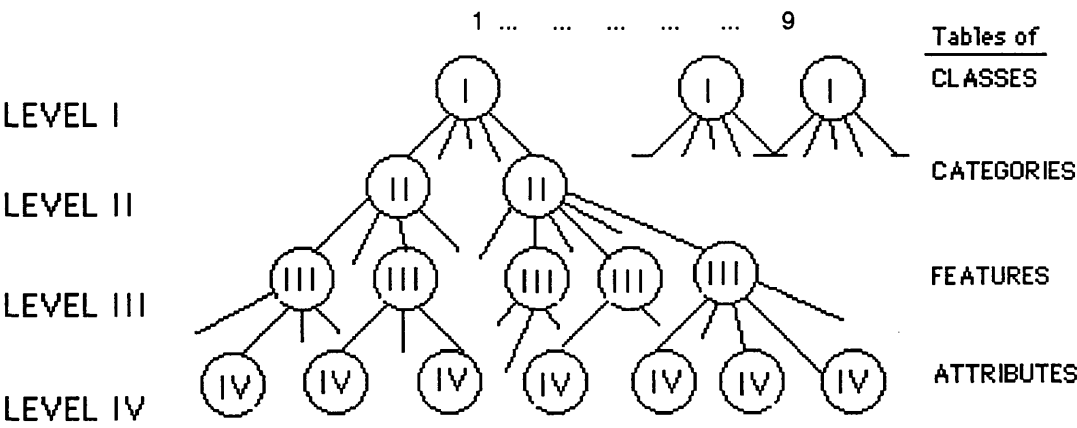


Fig. A.1 Diagrammatic representation of the feature coding system

A.2 The Feature Coding System

The feature coding system adopted in this project is derived from the "National Standards for the Exchange of Digital Topographic Data" published by the Canadian Council on Surveying and Mapping in 1984. The reason for using the classification on this system is that it has been devised in a very systematic way which helps tremendously in structuring the database which eventually will use it. Regarding the four levels of the classification system, the following coding scheme has been adopted :

***** is the format of any code and is of type integer.

1 2 3 4 5 6 7 8

In this scheme, the star number "1" may vary from 1 to 9 and represents the classification within Level I (i.e. 'classes'), that is one of the following :

1- Designated Areas

- 2- Buildings
- 3- Structures
- 4- Roads & Railways
- 5- Utilities
- 6- Boundaries
- 7- Hydrography
- 8- Relief & Landforms
- 9- Land Cover

Then stars numbered "2" and "3" may vary from 1 to 99 and describe Level II of the classification. They define the different categories. For example, Designated Area is the first class of the classification and the branch from it comprises the following categories:

- 00- Agriculture
- 01- Commercial
- 02- Conservation
- 03- Education
- 04- Religious
- 05- Residential
- 06- Sanitation or Waste Disposal
- 07- Government
- 08- Industrial
- 09- Medical
- 10- Recreational, Cultural, Historical or Ornamental
- 11- Transportation
- 12- Administrative, Political or Cadastral

The next two stars "4" and "5" may vary from 1 to 99 as well and are used to classify the features within the above categories. The remaining three stars may vary from 10 to 990 (by an increment of 10) and describe the attributes of these features. For example :

- "100 " means Designated Area - Agricul.
- "100 02 " means Designated Area - Agricul. - Farm
- "100 02 000 " means Designated Area - Agricul. - Farm - unspecified
- "100 02 010 " means Designated Area - Agricul. - Farm - Dairy

A clearance of 9 between codes is given to allow for the future expansion of the system. These codes are meant and are used for the system development. They are intended to speed up the processing of data only and the user does not need to see any of them. As has been discussed in Chapter 9, since the selection of data for retrieval is carried out using menus which in turn do not show any of these other codes, but rather the elements of the features themselves, there is no need for the user to use or see these codes.

A.3 Special Codes for Feature Attributes

On the other hand, there are some special feature attributes that have not been included in the code listing, which will be presented at the end of this Appendix. These are special cases and they are identified as follows :

A.3.1 Unspecified : 000

By default, a feature having an unspecified or indefinite attribute will be assigned the code for level IV of "000".

A.3.2 Other : 999

This attribute code is not included in the topographic feature listing, since it can be applied to practically all features. To give the attribute "Other", simply use the code "999" in Level IV. Automatically this will mean that the attributes which are available do not describe the feature concerned sufficiently well, and that the feature has a definite attribute which would not allow it to receive the default code "000" for being unspecified.

A.3.3 Abandoned : 001

Since a number of features could be abandoned (for instance a railway station may no longer be in use), the attribute "001" is being used as a standard code specifically for those features which have been abandoned. The attribute code "001" is to be assigned to all abandoned features, including abandoned features which already have attributes.

10- DESIGNATED AREA - Agriculture

| | |
|------------|--------------------------|
| 100 00 000 | Agriculture Land Reserve |
| 100 01 000 | Agriculture Region |
| 100 02 000 | Farm |
| 100 02 010 | Farm- Dairy |
| 100 02 020 | Farm- Experimental |
| 100 02 030 | Farm- Fruit |
| 100 02 040 | Farm- Fur |
| 100 02 050 | Farm- Mixed |
| 100 02 060 | Farm- Pig |
| 100 02 070 | Farm- Potato |
| 100 02 080 | Farm- Poultry |
| 100 02 090 | Farm- Seedcrop |
| 100 02 100 | Farm- Sheep |
| 100 02 110 | Farm- Sod |
| 100 02 120 | Farm- Tree |
| 100 03 000 | Feedlot |
| 100 04 000 | Ranch |
| 100 04 010 | Ranch- Cattle |
| 100 04 020 | Ranch- Horse |

11- DESIGNATED AREA - Commercial

| | |
|------------|----------------------------|
| 101 00 000 | Lot (Car) |
| 101 01 000 | Lot (Other Vehicle) |
| 101 05 000 | Shopping Centre Complex |
| 101 10 000 | Yard |
| 101 10 010 | Yard- Auto Wrecker |
| 101 10 020 | Yard- Coal |
| 101 10 030 | Yard- Junk/ Scrap/ Salvage |
| 101 10 040 | Yard- Lumber |
| 101 10 050 | Yard- Stock |
| 101 10 060 | Yard- Storage |

12- DESIGNATED AREA - Conservation

| | |
|------------|---------------------|
| 102 00 000 | Archaeological Area |
| 102 01 000 | Ecological Area |
| 102 02 000 | Forest Reserve |
| 102 03 000 | Sanctuary |
| 102 03 010 | Sanctuary- Bird |
| 102 03 020 | Sanctuary- Wildlife |
| 102 04 000 | Wilderness Area |

13- DESIGNATED AREA - Education

| | |
|------------|-------------------|
| 103 00 000 | College Campus |
| 103 01 000 | School Grounds |
| 103 02 000 | University Campus |

14- DESIGNATED AREA - Religious

| | |
|------------|-------------------|
| 104 00 000 | Cemetery |
| 104 01 000 | Religious Complex |

15- DESIGNATED AREA - Residential

| | |
|------------|------------------|
| 105 00 000 | Colony/Commune |
| 105 01 000 | Mobile Home Park |
| 105 02 000 | Trailer Park |

16- DESIGNATED AREA - Sanitation/Waste Disposal

| | |
|------------|------------------------------|
| 106 00 000 | Dump |
| 106 00 010 | Dump- Sanitary Landfill Site |
| 106 01 000 | Hazardous Waste Disposal |
| 106 02 000 | Liquid Waste Disposal Area |
| 106 03 000 | Oxidation Bed/Pond |
| 106 04 000 | Sewage Leaching Field |
| 106 05 000 | Slag Heap / Pile |
| 106 06 000 | Tailing Pile /Pond /Dump |

17- DESIGNATED AREA - Government

| | |
|------------|-----------------------------------|
| 107 00 000 | Fish Hatchery |
| 107 01 000 | Government Reserve |
| 107 01 010 | Government Reserve-Buffer Zone |
| 107 02 000 | Jail Complex |
| 107 02 010 | Jail Complex- County |
| 107 02 020 | Jail Complex- Provincial |
| 107 02 030 | Jail Complex- Territorial |
| 107 03 000 | Legislative Grounds |
| 107 04 000 | Penitentiary Complex |
| 107 04 010 | Penitentiary Complex-Max Security |
| 107 04 020 | Penitentiary Complex-Med Security |
| 107 04 030 | Penitentiary Complex-Min Security |

18- DESIGNATED AREA - Industrial

| | |
|------------|--------------------------------------|
| 108 00 000 | Cannery Complex |
| 108 00 010 | Cannery Complex- Fish Products |
| 108 00 020 | Cannery Complex- Fruit & Vegetables |
| 108 00 030 | Cannery Complex- Shellfish |
| 108 01 000 | Electric Power Generation Complex |
| 108 01 010 | Elect- Hydroelectric |
| 108 01 020 | Elect- Thermal(Coal) |
| 108 01 030 | Elect- Thermal(Nuclear) |
| 108 01 040 | Elect- Thermal(Oil) |
| 108 01 050 | Elect- Thermal(Unspecified) |
| 108 02 000 | Electric Substation Complex |
| 108 03 000 | Factory/ Plant Complex |
| 108 03 010 | Factory- Aircraft |
| 108 03 020 | Factory- Aircraft Parts |
| 108 03 030 | Factory- Baking |
| 108 03 040 | Factory- Boat Building |
| 108 03 050 | Factory- Brewery |
| 108 03 060 | Factory- Cement |
| 108 03 070 | Factory- Chemical |
| 108 03 080 | Factory- Communication Equipment |
| 108 03 090 | Factory- Dairy Products |
| 108 03 100 | Factory- Distillery |
| 108 03 110 | Factory- Electrical Equipment |
| 108 03 120 | Factory- Feed |
| 108 03 130 | Factory- Food Proc.(Non-Canning) |
| 108 03 140 | Factory- Furniture |
| 108 03 150 | Factory- Garment |
| 108 03 160 | Factory- Machinery |
| 108 03 170 | Factory- Major Appliances |
| 108 03 180 | Factory- Metal Stamping & Processing |
| 108 03 190 | Factory- Motor Vehicle |
| 108 03 200 | Factory- Motor Vehicle Parts |
| 108 03 210 | Factory- Paint |
| 108 03 220 | Factory- Petrochemical |
| 108 03 230 | Factory- Pharmaceutical & Medicine |
| 108 03 240 | Factory- Plastics Fabrication |
| 108 03 250 | Factory- Poultry Proc.(Non-Canning) |
| 108 03 260 | Factory- Publishing & Printing |
| 108 03 270 | Factory- Rubber Products |
| 108 03 280 | Factory- Ship Building |
| 108 03 290 | Factory- Slaughter & Meat Processing |
| 108 03 300 | Factory- Soft Drink |
| 108 03 310 | Factory- Winery |
| 108 04 000 | Gas Field |
| 108 05 000 | Industrial Complex |
| 108 06 000 | Lumber Camp |
| 108 07 000 | Mill Complex |
| 108 07 010 | Mill- Feed |
| 108 07 020 | Mill- Flour |
| 108 07 030 | Mill- Iron & Steel |
| 108 07 040 | Mill- Pulp & Paper |
| 108 07 050 | Mill- Saw |
| 108 07 060 | Mill- Shake/Shingle |
| 108 07 070 | Mill- Textile |
| 108 07 080 | Mill- Veneer & Plywood |
| 108 07 090 | Mill- Wire & Rope |
| 108 08 000 | Mine (Open Pit) |
| 108 08 010 | Mine- Asbestos |
| 108 08 020 | Mine- Calcite |
| 108 08 040 | Mine- Copper |

| | |
|------------|--------------------------------|
| 108 08 050 | Mine- Feldspar |
| 108 08 060 | Mine- Gold |
| 108 08 070 | Mine- Gypsum |
| 108 08 080 | Mine- Iron Ore |
| 108 08 090 | Mine- Lead & Zinc |
| 108 08 100 | Mine- Magnesium |
| 108 08 110 | Mine- Nephline Syenite |
| 108 08 120 | Mine- Nickel |
| 108 08 130 | Mine- Silica (Quartz) |
| 108 08 140 | Mine- Silver |
| 108 08 150 | Mine- Titanium |
| 108 08 160 | Mine- Tungsten |
| 108 08 170 | Mine- Uranium |
| 108 09 000 | Mine (Placer) Gold |
| 108 10 000 | Mine (Strip) |
| 108 10 010 | Mine- Bentonite |
| 108 10 020 | Mine- Coal |
| 108 10 030 | Mine- Oil Sands |
| 108 11 000 | Mine (Underground) |
| 108 11 010 | Mine- Antimony |
| 108 11 020 | Mine- Asbestos |
| 108 11 030 | Mine- Barite |
| 108 11 040 | Mine- Coal |
| 108 11 050 | Mine- Columbium |
| 108 11 060 | Mine- Copper |
| 108 11 070 | Mine- Feldspar |
| 108 11 080 | Mine- Fluorite |
| 108 11 090 | Mine- Gold |
| 108 11 100 | Mine- Iron Ore |
| 108 11 110 | Mine- Lead & Zinc |
| 108 11 120 | Mine- Lithium |
| 108 11 130 | Mine- Mercury |
| 108 11 140 | Mine- Molybdenum |
| 108 11 150 | Mine- Nickel |
| 108 11 160 | Mine- Platinum Group |
| 108 11 170 | Mine- Potash |
| 108 11 180 | Mine- Salt |
| 108 11 190 | Mine- Selenium |
| 108 11 200 | Mine- Silver |
| 108 11 210 | Mine- Sulphur |
| 108 11 220 | Mine- Talc |
| 108 11 230 | Mine- Titanium |
| 108 11 240 | Mine- Tungsten |
| 108 11 250 | Mine- Uranium |
| 108 12 000 | Oil & Gas Field |
| 108 13 000 | Oil Battery/ Tank Farm |
| 108 14 000 | Oil Field |
| 108 15 000 | Peat Cutting |
| 108 16 000 | Pit |
| 108 16 010 | Pit- Gravel |
| 108 16 020 | Pit- Sand |
| 108 16 030 | Pit- Shale |
| 108 17 000 | Quarry |
| 108 18 000 | Refinery Complex |
| 108 18 010 | Refinery- Gasoline/Oil |
| 108 18 020 | Refinery-Liquid Petrol Gas/Oil |
| 108 18 030 | Refinery- Metal |
| 108 18 040 | Refinery- Sugar |
| 108 19 000 | Research Centre Complex |
| 108 20 000 | Salt Evaporator |

19- DESIGNATED AREA - Medical

| | |
|------------|-------------------------|
| 109 00 000 | Hospital Complex |
| 109 00 010 | Hospital- Children |
| 109 00 020 | Hospital- General |
| 109 00 030 | Hospital- Mental Health |
| 109 00 040 | Hospital- Military |
| 109 00 050 | Hospital- University |
| 109 01 000 | Sanitarium Complex |

110- DESIGNATED AREA - Recreational,Cultural,Historical or Ornamental

| | |
|------------|------------------|
| 110 00 000 | Battlefield |
| 110 01 000 | Botanical Garden |

| | |
|------------|-----------------------------|
| 110 02 000 | Campground/Campsite |
| 110 02 010 | Campground- Federal |
| 110 02 020 | Campground- Municipal |
| 110 02 030 | Campground- Private |
| 110 02 040 | Campground- Provincial |
| 110 02 050 | Campground- Regional |
| 110 03 000 | Court |
| 110 03 010 | Court- Basketball |
| 110 03 020 | Court- Tennis |
| 110 04 000 | Drive-In Theatre |
| 110 05 000 | Exhibition Ground |
| 110 06 000 | Fairground |
| 110 07 000 | Flower Bed |
| 110 08 000 | Fort |
| 110 09 000 | Golf Course |
| 110 09 010 | Golf Course- Miniature |
| 110 10 000 | Historic Area |
| 110 11 000 | Lacrosse Box |
| 110 13 000 | Lawn/Bowling Green |
| 110 14 000 | Look-Out |
| 110 14 010 | Look-Out/Covered |
| 110 14 020 | Look-Out/Scenic |
| 110 15 000 | Off-Road Vehicle Test Area |
| 110 16 000 | Outdoor Theatre |
| 110 17 000 | Park |
| 110 17 010 | Park- Amusement |
| 110 17 020 | Park- Memorial |
| 110 17 030 | Park- Municipal |
| 110 17 040 | Park- National |
| 110 17 050 | Park- Provincial |
| 110 17 060 | Park- Regional |
| 110 18 000 | Patio |
| 110 19 000 | Picnic Site |
| 110 20 000 | Playground |
| 110 21 000 | Playing Field (Sports) |
| 110 22 000 | Race Track |
| 110 22 010 | Race Track- Athletic |
| 110 22 020 | Race Track- Automotive |
| 110 22 030 | Race Track- Bicycle |
| 110 22 040 | Race Track- Horse |
| 110 23 000 | Range (Civilian) |
| 110 23 010 | Range- Archery |
| 110 23 020 | Range- Golf Driving |
| 110 23 030 | Range- Pistol |
| 110 23 040 | Range- Rifle |
| 110 23 050 | Range- Skeet/Trap |
| 110 24 000 | Resort |
| 110 24 010 | Resort- Summer |
| 110 24 020 | Resort- Winter |
| 110 25 000 | Riding Academy |
| 110 26 000 | Rink (Outdoor) |
| 110 27 000 | Ski Area |
| 110 28 000 | Soccer Field |
| 110 29 000 | Sports/Recreational Complex |
| 110 30 000 | Zoo |

111- DESIGNATED AREA - Transportation

| | |
|------------|-------------------------|
| 111 00 000 | Airfield |
| 111 01 000 | Airport |
| 111 02 000 | Airstrip |
| 111 03 000 | Anchorage |
| 111 03 010 | Anchorage- Seaplane |
| 111 03 020 | Anchorage- Ship (Large) |
| 111 03 030 | Anchorage- Ship (Small) |
| 111 04 000 | Ferry Route |
| 111 05 000 | Harbour/Port |
| 111 06 000 | Helipad |
| 111 07 000 | Landing (Boat) |
| 111 08 000 | Parking Apron |
| 111 09 000 | Parking Lot |
| 111 09 010 | Parking Lot- Paved |
| 111 09 020 | Parking Lot- Unpaved |
| 111 10 000 | Railway Yard |
| 111 11 000 | Rest Yard |

| | |
|------------|---------------------------|
| 111 12 000 | Runway (Airport/Airfield) |
| 111 13 000 | Seaplane Base |
| 111 14 000 | Taxiway (Airport) |

112- DESIGNATED AREA - Admin, Political/Cadastral

| | |
|------------|--|
| 112 00 000 | Block |
| 112 01 000 | Block-face |
| 112 02 000 | Borough |
| 112 03 000 | Built-up Area |
| 112 04 000 | Census Agglomeration |
| 112 05 000 | Census Consolidated Subdivision |
| 112 06 000 | Census Division |
| 112 07 000 | Census Farm |
| 112 08 000 | Census Metropolitan Area (CMS) |
| 112 09 000 | (CMS)/Census Agglomerations |
| 112 10 000 | Census Subdivision |
| 112 10 010 | Census Sub- Borough |
| 112 10 020 | Census Sub- Canton |
| 112 10 040 | Census Sub- City |
| 112 10 050 | Census Sub- Community |
| 112 10 060 | Census Sub- County (Municipality) |
| 112 10 070 | Census Sub- District (Municipality) |
| 112 10 080 | Census Sub- Hamlet |
| 112 10 090 | Census Sub- Improvement District |
| 112 10 100 | Census Sub- Local Government District |
| 112 10 110 | Census Sub- Local Improvement District |
| 112 10 120 | Census Sub- Municipal Corporation |
| 112 10 130 | Census Sub- Municipal District |
| 112 10 140 | Census Sub- Parish |
| 112 10 160 | Census Sub- Resort Village |
| 112 10 170 | Census Sub- Rural Municipality |
| 112 10 190 | Census Sub- Settlement |
| 112 10 200 | Census Sub- Special Area |
| 112 10 210 | Census Sub- Subdivision/County Municipal |
| 112 10 220 | Census Sub- Subdivision/Region District |
| 112 10 230 | Census Sub- Summer Village |
| 112 10 240 | Census Sub- Town |
| 112 10 250 | Census Sub- Township |
| 112 10 260 | Census Sub- Township and Royalty |
| 112 10 270 | Census Sub- Unorganized |
| 112 10 280 | Census Sub- Village |
| 112 11 000 | Census Tract |
| 112 11 010 | Census Tract- Provincial |
| 112 12 000 | City |
| 112 13 000 | Componenet (Census) |
| 112 14 000 | Concession |
| 112 15 000 | County |
| 112 16 000 | District |
| 112 17 000 | District Municipality |
| 112 18 000 | Easement |
| 112 19 000 | Electoral District/Riding |
| 112 19 010 | Electoral District- Federal |
| 112 19 020 | Electoral District- Provincial |
| 112 20 000 | Electoral District/Ward |
| 112 20 010 | Electoral District- Municipal |
| 112 21 000 | Enumeration Area |
| 112 22 000 | Environment Region |
| 112 23 000 | Forest & Grazing District |
| 112 24 000 | Hamlet/Community/Locality |
| 112 25 000 | Highway/Transportation |
| 112 25 010 | Highway/Transportation- District |
| 112 25 020 | Highway/Transportation- Region |
| 112 27 000 | Human Resources Region |
| 112 28 000 | Improvement District |
| 112 29 000 | Land & Housing Region |
| 112 30 000 | Land Assessment District |
| 112 31 000 | Land District |
| 112 32 000 | Land Grant (DLS) |
| 112 33 000 | Land Management District |
| 112 34 000 | Land Recording District |
| 112 35 000 | Land Title District |
| 112 36 000 | Lease |
| 112 36 010 | Lease- Campsite |
| 112 36 020 | Lease- Coal |

| | |
|------------|----------------------------|
| 112 36 030 | Lease- Gas |
| 112 36 040 | Lease- Mineral |
| 112 36 050 | Lease- Oil |
| 112 36 060 | Lease- Placer Mining |
| 112 36 070 | Lease- Potash |
| 112 36 080 | Lease- Pulp |
| 112 37 000 | Legal Subdivision (DLS) |
| 112 38 000 | Licence/Permit |
| 112 38 010 | Licence/Permit- Coal |
| 112 38 020 | Licence/Permit- Gas |
| 112 38 030 | Licence/Permit- Mineral |
| 112 38 040 | Licence/Permit- Oil |
| 112 38 050 | Licence/Permit- Pulp |
| 112 38 060 | Licence/Permit- Timber |
| 112 38 070 | Licence/Permit- Tree Farm |
| 112 39 000 | Lot |
| 112 39 010 | Lot- District |
| 112 39 020 | Lot- River |
| 112 39 030 | Lot- Road |
| 112 39 040 | Lot- Township |
| 112 40 000 | Metropolitan Area |
| 112 41 000 | Mineral Claim |
| 112 42 000 | Mining District/Division |
| 112 43 000 | Parcel |
| 112 44 000 | Parish |
| 112 45 000 | Patented Land |
| 112 46 000 | Polling Division |
| 112 47 000 | Quarter Section |
| 112 49 000 | Range |
| 112 50 000 | Regional District |
| 112 51 000 | Regional Municipality |
| 112 52 000 | Resource Management Region |
| 112 53 000 | Right of Way |
| 112 54 000 | Road Allowance |
| 112 55 000 | Rural Municipality |
| 112 56 000 | School District |
| 112 57 000 | Section |
| 112 58 000 | Settlement |
| 112 59 000 | Subdivision |
| 112 60 000 | Sub-lot |
| 112 61 000 | Timber Berth |
| 112 62 000 | Town |
| 112 63 000 | Township |
| 112 63 010 | Township- DLS |
| 112 63 020 | Township- Municipal |
| 112 64 000 | Village |
| 112 65 000 | Water Management Region |

20- BUILDING - Agriculture

| | |
|------------|-------------------|
| 200 00 000 | Barn |
| 200 01 000 | Granary |
| 200 02 000 | Greenhouse |
| 200 03 000 | Maple Sugar Shack |
| 200 04 000 | Shed |
| 200 04 010 | Shed- Drying |
| 200 04 020 | Shed- Machinery |
| 200 05 000 | Stable |

21- BUILDING - Commercial

| | |
|------------|---------------------------|
| 201 00 000 | Bank |
| 201 01 000 | Bar/Beer-Parlour/Saloon |
| 201 02 000 | Booth/Snack Bar/Canteen |
| 201 03 000 | Cabin (Tourist) |
| 201 04 000 | Car Wash |
| 201 05 000 | Crematorium |
| 201 06 000 | Funeral Home |
| 201 07 000 | Grain Elevator |
| 201 08 000 | Hotel/Motel/Tourist Lodge |
| 201 09 000 | Kennel |
| 201 10 000 | Office |
| 201 10 010 | Office- Commercial |
| 201 11 000 | Restaurant/Cafe |
| 201 12 000 | Service Station |

| | |
|------------|--------------------------------------|
| 201 12 010 | Service Station- Refueling |
| 201 12 020 | Service Station- Repair (Automotive) |
| 201 13 000 | Shopping Centre |
| 201 14 000 | Store |
| 201 14 010 | Store- Personal Service |
| 201 14 020 | Store- Retail |
| 201 14 030 | Store- Wholesale |
| 201 15 000 | Trading Post |
| 201 16 000 | Warehouse |
| 201 16 010 | Warehouse- Storage |
| 201 16 020 | Warehouse- Wholesale |

22- BUILDING - Communication

| | |
|------------|-----------------------------|
| 202 00 000 | Station (Communication) |
| 202 00 010 | Station- Microwave |
| 202 00 020 | Station- Radio |
| 202 00 030 | Station- Radio - Government |
| 202 00 040 | Station- Radio - Military |
| 202 00 050 | Station- Radio - Private |
| 202 00 060 | Station- Radio - Telegraph |
| 202 00 070 | Station- Radio - Telephone |
| 202 00 080 | Station- Relay |
| 202 00 090 | Station- Satellite |
| 202 00 100 | Station- TV |
| 202 00 110 | Station- TV - Government |
| 202 00 120 | Station- TV - Private |
| 202 01 000 | Telegraph Office |
| 202 02 000 | Telephone Office |

23- BUILDING - Education

| | |
|------------|--------------------------------|
| 203 00 000 | College |
| 203 00 010 | College- Community |
| 203 00 020 | College- Military |
| 203 01 000 | Library |
| 203 02 000 | School |
| 203 02 010 | School- Art |
| 203 02 020 | School- Business |
| 203 02 030 | School- Community |
| 203 02 040 | School- Day Care |
| 203 02 050 | School- Day Care- Private |
| 203 02 060 | School- Day Care- Public |
| 203 02 070 | School- Day Care- Separate |
| 203 02 080 | School- Elementary |
| 203 02 090 | School- Elementary- Private |
| 203 02 100 | School- Elementary- Public |
| 203 02 110 | School- Elementary- Separate |
| 203 02 120 | School- Kindergarten |
| 203 02 130 | School- Kindergarten- Private |
| 203 02 140 | School- Kindergarten- Public |
| 203 02 150 | School- Kindergarten- Separate |
| 203 02 160 | School- Military |
| 203 02 170 | School- Nursery |
| 203 02 180 | School- Nersery- Private |
| 203 02 190 | School- Nersery- Public |
| 203 02 200 | School- Nersery- Separate |
| 203 02 210 | School- Primary |
| 203 02 220 | School- Primary- Private |
| 203 02 230 | School- Primary- Public |
| 203 02 240 | School- Primary- Separate |
| 203 02 250 | School- Retraining |
| 203 02 260 | School- Secondary |
| 203 02 270 | School- Secondary- Private |
| 203 02 280 | School- Secondary- Public |
| 203 02 290 | School- Secondary- Separate |
| 203 02 300 | School- Technical |
| 203 02 310 | School- Technical- Private |
| 203 02 320 | School- Technical- Public |
| 203 02 330 | School- Technical- Separate |
| 203 03 000 | University |

24- BUILDING - Government

| | |
|------------|-----------|
| 204 00 000 | City Hall |
|------------|-----------|

| | |
|------------|----------------------------|
| 204 01 000 | Courthouse |
| 204 01 010 | Courthouse- Federal |
| 204 01 020 | Courthouse- Provincial |
| 204 02 000 | Customs Office |
| 204 03 000 | Customs Post |
| 204 04 000 | Customs Warehouse |
| 204 05 000 | Detention Home/Centre |
| 204 05 010 | Detention- Juvenile |
| 204 05 020 | Detention- Pre-sentencing |
| 204 06 000 | Embassy |
| 204 07 000 | Filtration Plant |
| 204 08 000 | Fire Lookout - Building |
| 204 08 010 | Fire Lookout - Tower |
| 204 09 000 | Fire Station |
| 204 10 000 | Government Agent Office |
| 204 11 000 | Government Office Bld |
| 204 11 010 | Government- Federal |
| 204 11 020 | Government- Municipal |
| 204 11 030 | Government- Provincial |
| 204 12 000 | Jail |
| 204 12 010 | Jail- County |
| 204 12 020 | Jail- Provincial |
| 204 12 030 | Jail- Territorial |
| 204 13 000 | Legislative Building |
| 204 14 000 | Municipal Hall |
| 204 15 000 | Observatory (Astronomical) |
| 204 16 000 | Park Administration Bld |
| 204 17 000 | Parliament Building |
| 204 18 000 | Penitentiary |
| 204 18 010 | Penitentiary- Max Security |
| 204 18 020 | Penitentiary- Med Security |
| 204 18 030 | Penitentiary- Min Security |
| 204 19 000 | Police Station |
| 204 19 010 | Police Station- Municipal |
| 204 19 020 | Police Station- Provincial |
| 204 21 000 | Post Office |
| 204 22 000 | Ranger/Warden Station |
| 204 23 000 | Research Centre Government |
| 204 24 000 | Town Hall |
| 204 25 000 | Village Hall |
| 204 26 000 | Weigh Scale Office |

25- BUILDING - Industrial

| | |
|------------|--|
| 205 00 000 | Cannery |
| 205 00 010 | Cannery- Fish Products |
| 205 00 020 | Cannery- Fruit & Vegetables |
| 205 00 030 | Cannery- Shelfish |
| 205 01 000 | Electric Power Generating Station (EPGS) |
| 205 01 010 | EPGS- Hydroelectric |
| 205 01 020 | EPGS- Thermal (Coal) |
| 205 01 030 | EPGS- Thermal (Nuclear) |
| 205 01 040 | EPGS- Thermal (Oil) |
| 205 01 050 | EPGS- Thermal (Unspecified) |
| 205 02 000 | Electrical Substation |
| 205 03 000 | Factory/Plant |
| 205 03 010 | Factory- Aircraft |
| 205 03 020 | Factory- Aircraft Parts |
| 205 03 030 | Factory- Baking |
| 205 03 040 | Factory- Boat Building |
| 205 03 050 | Factory- Brewery |
| 205 03 060 | Factory- Cement |
| 205 03 070 | Factory- Chemical |
| 205 03 080 | Factory- Communication Equipment |
| 205 03 090 | Factory- Dairy Products |
| 205 03 100 | Factory- Distillery |
| 205 03 110 | Factory- Electrical Equipment |
| 205 03 120 | Factory- Feed |
| 205 03 130 | Factory- Food Processing (Non-Canning) |
| 205 03 140 | Factory- Furniture |
| 205 03 150 | Factory- Garment |
| 205 03 160 | Factory- Gas Processing |
| 205 03 170 | Factory- Machinery |
| 205 03 180 | Factory- Major Appliances |
| 205 03 190 | Factory- Metal Stamping & Processing |

| | |
|------------|--|
| 205 03 200 | Factory- Motor Vehicle |
| 205 03 210 | Factory- Motor Vehicle Parts |
| 205 03 220 | Factory- Paint |
| 205 03 230 | Factory- Petrochemical |
| 205 03 240 | Factory- Pharmaceutical & Medicine |
| 205 03 250 | Factory- Plastics Fabrication |
| 205 03 260 | Factory- Poultry Proc (NonFactory-Canning) |
| 205 03 270 | Factory- Publishing & Printing |
| 205 03 280 | Factory- Rubber Products |
| 205 03 290 | Factory- Sash & Door |
| 205 03 300 | Factory- Ship Building |
| 205 03 310 | Factory- Slaughter & Meat Processing |
| 205 03 320 | Factory- Soft Drink |
| 205 03 330 | Factory- Winery |
| 205 04 000 | Gas Compressor Station |
| 205 05 000 | Lumber Camp Building |
| 205 06 000 | Meter Station |
| 205 07 000 | Mill |
| 205 07 010 | Mill- Feed |
| 205 07 020 | Mill- Flour |
| 205 07 030 | Mill- Iron & Steel |
| 205 07 040 | Mill- Pulp and Paper |
| 205 07 050 | Mill- Saw |
| 205 07 060 | Mill- Shake/Shingle |
| 205 07 070 | Mill- Textile |
| 205 07 080 | Mill- Veneer and Plywood |
| 205 07 090 | Mill- Wire and Rope |
| 205 08 000 | Pipeline Satellite Stn. |
| 205 09 000 | Pumping Station (PS) |
| 205 09 010 | PS- Gasoline |
| 205 09 020 | PS- Liquid Petroleum Gas/Oil Products |
| 205 09 030 | PS- Water |
| 205 10 000 | Refinery |
| 205 10 010 | Refinery- Gasoline/Oil |
| 205 10 020 | Refinery- Liquid Petroleum Gas |
| 205 10 030 | Refinery- Metal |
| 205 10 040 | Refinery- Sugar |
| 205 11 000 | Research Centre-Industrial |

26- BUILDING - Medical

| | |
|------------|---------------------------|
| 206 00 000 | Clinic |
| 206 00 010 | Clinic- Convalescent |
| 206 00 030 | Clinic- Dental |
| 206 00 040 | Clinic- Handicapped |
| 206 00 050 | Clinic- Orthopaedic |
| 206 00 060 | Clinic- Rehabilitation |
| 206 00 070 | Clinic- Veterinary |
| 206 01 000 | Hospital |
| 206 01 010 | Hospital- Children |
| 206 01 020 | Hospital- Cottage |
| 206 01 030 | Hospital- General |
| 206 01 040 | Hospital- Geriatric |
| 206 01 050 | Hospital- Mental Health |
| 206 01 060 | Hospital- Military |
| 206 01 070 | Hospital- University |
| 206 02 000 | Research Centre - Medical |
| 206 03 000 | Sanitorium |

27- BUILDING - Waste Disposal

| | |
|------------|--------------------------|
| 207 00 000 | Outhouse (Sanitation) |
| 207 01 000 | Restroom/Toilet Facility |
| 207 02 000 | Sewage Pumping Station |
| 207 03 000 | Sewage Treatment Plant |

28- BUILDING - Recreational, Cultural, Historical/Ornamental

| | |
|------------|------------------------|
| 208 00 000 | Arena |
| 208 00 010 | Arena- Boxing |
| 208 00 020 | Arena- Curling |
| 208 00 030 | Arena- Hockey |
| 208 00 040 | Arena- Track and Field |
| 208 00 050 | Arena- Velodrome |
| 208 01 000 | Art Centre |

| | |
|------------|--------------------------------|
| 208 02 000 | Art Gallery |
| 208 03 000 | Auditorium |
| 208 04 000 | Boathouse |
| 208 05 000 | Bowling Alley |
| 208 06 000 | Change House |
| 208 07 000 | Club House |
| 208 07 010 | Club House- Canoeing |
| 208 07 020 | Club House- Cricket |
| 208 07 030 | Club House- Flying |
| 208 07 040 | Club House- Fraternal |
| 208 07 050 | Club House- Golf |
| 208 07 060 | Club House- Rowing |
| 208 07 070 | Club House- Rugby |
| 208 07 080 | Club House- Skeet Shooting |
| 208 07 090 | Club House- Skiing |
| 208 07 100 | Club House- Speedboat Racing |
| 208 07 110 | Club House- Tennis |
| 208 07 120 | Club House- Yachting |
| 208 08 000 | Fitness/Athletic/Sport complex |
| 208 09 000 | Hall |
| 208 09 010 | Hall- Amusement |
| 208 09 020 | Hall- Billiard |
| 208 09 030 | Hall- Community |
| 208 09 040 | Hall- Dance |
| 208 09 050 | Hall- Exhibition |
| 208 09 060 | Hall- Music |
| 208 10 000 | Historical Building |
| 208 11 000 | Marina |
| 208 12 000 | Museum |
| 208 12 010 | Museum- Aeronautical |
| 208 12 020 | Museum- Military |
| 208 12 030 | Museum- Natural History |
| 208 12 040 | Museum- Nautical |
| 208 12 050 | Museum- Science |
| 208 13 000 | Planetarium |
| 208 14 000 | Ruin |
| 208 15 000 | Swimming Pool (Indoors) |
| 208 16 000 | Theatre |
| 208 16 010 | Theatre- Live Performance |
| 208 16 020 | Theatre- Moving Picture |

209- BUILDING - Residential

| | |
|------------|------------------------------|
| 209 00 000 | Apartment |
| 209 01 000 | Cabin/Hut/Shack |
| 209 02 000 | Cottage/Chalet |
| 209 03 000 | Dwelling |
| 209 04 000 | Dwelling (Institutional) |
| 209 04 010 | Dwelling- Club (Residential) |
| 209 04 020 | Dwelling- Hostel |
| 209 04 030 | Dwelling- Nursing |
| 209 04 040 | Dwelling- Old-age |
| 209 04 050 | Dwelling- Orphanage |
| 209 04 060 | Dwelling- Student Residence |
| 209 04 070 | Dwelling- YMCA/YWCA |
| 209 05 000 | Garage (Non-commercial) |
| 209 05 010 | Garage- Double - Attached |
| 209 05 020 | Garage- Double - Detached |
| 209 05 030 | Garage- Multiple - Attached |
| 209 05 040 | Garage- Multiple - Detached |
| 209 05 050 | Garage- Single - Attached |
| 209 05 060 | Garage- Single - Detached |
| 209 06 000 | House |
| 209 06 010 | House- Detached (Single) |
| 209 06 030 | House - Floating |
| 209 06 040 | House- Row |
| 209 06 050 | House- Semi-detached |
| 209 07 000 | Manse/Rectory/Presbytery |
| 209 08 000 | Mobile Home |
| 209 09 000 | Shed (Garden) |

210- BUILDING - Religious

| | |
|------------|--------------------|
| 210 00 000 | Building Religious |
| 210 01 000 | Cathedral |

| | | | |
|-----|----|-----|--------------|
| 210 | 02 | 000 | Chapel |
| 210 | 03 | 000 | Church |
| 210 | 04 | 000 | Church Hall |
| 210 | 05 | 000 | Convent |
| 210 | 06 | 000 | Meeting Hall |
| 210 | 07 | 000 | Monastery |
| 210 | 08 | 000 | Mosque |
| 210 | 09 | 000 | Pagoda |
| 210 | 10 | 000 | Seminary |
| 210 | 11 | 000 | Shrine |
| 210 | 12 | 000 | Synagogue |
| 210 | 13 | 000 | Temple |

212- BUILDING - Other

| | | | |
|-----|----|-----|---------------------|
| 211 | 00 | 000 | Building - Dome |
| 211 | 01 | 000 | Building - Multiuse |

30- STRUCTURE - Agriculture

| | | | |
|-----|----|-----|----------------|
| 300 | 00 | 000 | Bin (Storage) |
| 300 | 01 | 000 | Corn Crib |
| 300 | 02 | 000 | Feeder Station |
| 300 | 03 | 000 | Silo |
| 300 | 04 | 000 | Stock Pen |
| 300 | 05 | 000 | Wind Mill |

31- STRUCTURE - Commercial

| | | | |
|-----|----|-----|------------------------------|
| 301 | 00 | 000 | Bill Board |
| 301 | 01 | 000 | Fuel Pump |
| 301 | 01 | 010 | Fuel Pump- Diesel |
| 301 | 01 | 030 | Fuel Pump- Gasoline |
| 301 | 02 | 000 | Fuel Pump Island |
| 301 | 03 | 000 | Parking Garage |
| 301 | 03 | 010 | Parking Garage- Multi-Level |
| 301 | 03 | 020 | Parking Garage- Single-Level |
| 301 | 04 | 000 | Stand/Kiosk |
| 301 | 05 | 000 | Storage Tank |
| 301 | 05 | 010 | Storage Tank- Fuel |

32- STRUCTURE - Communication

| | | | |
|-----|----|-----|-----------------------|
| 302 | 00 | 000 | Antenna |
| 302 | 00 | 010 | Antenna- Microwave |
| 302 | 00 | 020 | Antenna- Radio |
| 302 | 00 | 030 | Antenna- TV |
| 302 | 01 | 000 | Tower/Mast |
| 302 | 01 | 010 | Tower/Mast- Microwave |
| 302 | 01 | 020 | Tower/Mast- Radio |
| 302 | 01 | 030 | Tower/Mast- TV |

33- STRUCTURE - Government

| | | | |
|-----|----|-----|---------------------------|
| 303 | 00 | 000 | Station (Climate/Weather) |
| 303 | 02 | 000 | Station (Tide Monitoring) |
| 303 | 03 | 000 | Station (Water Quality) |

34- STRUCTURE - Industrial

| | | | |
|-----|----|-----|---------------------|
| 304 | 00 | 000 | Bunker |
| 304 | 01 | 000 | Burner |
| 304 | 01 | 010 | Burner- Garbage |
| 304 | 01 | 020 | Burner- Sawdust |
| 304 | 02 | 000 | Crane |
| 304 | 02 | 010 | Crane- Moveable |
| 304 | 02 | 020 | Crane- Stationary |
| 304 | 03 | 000 | Drydock |
| 304 | 03 | 010 | Drydock- Floating |
| 304 | 03 | 020 | Drydock- On Land |
| 304 | 04 | 000 | Gas Well |
| 304 | 05 | 000 | Grating |
| 304 | 06 | 000 | Hopper |
| 304 | 07 | 000 | Injection Well |
| 304 | 07 | 010 | Injection Well- Gas |

| | |
|------------|-----------------------------|
| 304 07 020 | Injection Well- Water/Steam |
| 304 08 000 | Kiln |
| 304 08 010 | Kiln- Brick |
| 304 08 020 | Kiln- Lumber |
| 304 09 000 | Mine entrance/Adit |
| 304 10 000 | Mine Headframe |
| 304 11 000 | Mine Shaft |
| 304 12 000 | Oil Well |
| 304 13 000 | Platform (Offshore Drill) |
| 304 14 000 | Slipway |
| 304 15 000 | Smoke Stack/Chimney |
| 304 16 000 | Weigh Scale |

35- STRUCTURE - Recreational,Cultural,Historical/Ornamental

| | |
|------------|-------------------------|
| 305 00 000 | Amphitheatre |
| 305 01 000 | Amusement Train |
| 305 02 000 | Bandstand |
| 305 03 000 | Bleachers |
| 305 04 000 | Fireplcae (Outdoor) |
| 305 05 000 | Flag Pole |
| 305 06 000 | Fountain |
| 305 08 000 | Grandstand |
| 305 09 000 | Landmark |
| 305 10 000 | Monument (Historical) |
| 305 10 010 | Monument- Memorial |
| 305 11 000 | Observation Tower |
| 305 12 000 | Park Entrance |
| 305 13 000 | Pavilion/Shelter |
| 305 14 000 | Picnic Table (Fixed) |
| 305 16 000 | Plaque (Historical) |
| 305 17 000 | Roller Coaster |
| 305 18 000 | Skateboard Bowl |
| 305 19 000 | Ski Jump |
| 305 20 000 | Ski Lift |
| 305 21 000 | Stadium |
| 305 21 020 | Stadium- Football |
| 305 22 000 | Statue |
| 305 23 000 | Sun Dial |
| 305 24 000 | Swimming Pool (Outdoor) |

36- STRUCTURE - Religious

| | |
|------------|-------------------------|
| 306 00 000 | Church Tower - Attached |
| 306 00 010 | Church Tower - Detached |
| 306 01 000 | Minaret - Attached |
| 306 01 010 | Minaret - Detached |
| 306 02 000 | Religious Monument |
| 306 03 000 | Spire/Steeple |

37- STRUCTURE - Residential

| | |
|------------|------------------------------|
| 307 00 000 | Carport |
| 307 00 010 | Carport- Double - Attached |
| 307 00 020 | Carport- Double - Detached |
| 307 00 030 | Carport- Multiple - Attached |
| 307 00 040 | Carport- Multiple - Detached |
| 307 00 050 | Carport- Single - Attached |
| 307 00 060 | Carport- Single - Detached |

38- STRUCTURE - Sanitation/Waste Disposal

| | |
|------------|-------------|
| 308 00 000 | Incinerator |
| 308 01 000 | Refuse Bin |
| 308 02 000 | Septic Tank |

39- STRUCTURE - Transportation

| | |
|------------|-----------------|
| 309 00 000 | Aerial Cableway |
| 309 01 000 | Beacon |
| 309 01 010 | Beacon- Air |
| 309 01 020 | Beacon- Marine |
| 309 02 000 | Bollard |
| 309 03 000 | Bus Shelter |
| 309 04 000 | Capstan |

| | |
|------------|---------------------------------|
| 309 05 000 | Conveyor |
| 309 06 000 | Dock |
| 309 06 010 | Dock- Cargo |
| 309 06 020 | Dock- Coal |
| 309 06 030 | Dock- Ferry |
| 309 06 040 | Dock- Floating |
| 309 06 050 | Dock- Ore |
| 309 07 000 | Dolphin |
| 309 08 000 | Fog Signal |
| 309 09 000 | Jetty |
| 309 10 000 | Launching Ramp |
| 309 11 000 | Light House |
| 309 12 000 | Lock |
| 309 13 000 | Pier |
| 309 14 000 | Platform |
| 309 14 010 | Platform- Cargo |
| 309 14 020 | Platform- Freight (Hydraulic) |
| 309 14 030 | Platform- Passenger (Covered) |
| 309 14 040 | Platform- Passenger (Uncovered) |
| 309 15 000 | Quay |
| 309 16 000 | Radar Dome |
| 309 16 010 | Radar Dome- Government |
| 309 16 020 | Radar Dome- Military |
| 309 17 000 | Ramp |
| 309 18 000 | Ranging Light/Marker |
| 309 19 000 | Siren |
| 309 20 000 | Slip |
| 309 21 000 | Warning Light |
| 309 22 000 | Wharf |
| 309 23 000 | Windcone |

310- STRUCTURE - Other

| | |
|------------|----------------|
| 310 00 000 | Barrier |
| 310 00 010 | Barrier - Berm |
| 310 01 000 | Fence |
| 310 02 000 | Gate |
| 310 03 000 | Noise Barrier |
| 310 04 000 | Steps |
| 310 05 000 | Structure |
| 310 06 000 | Wall |
| 310 07 000 | Wind Charger |
| 310 08 000 | Wind Pump |

40- ROADS OR RAILWAYS - Road

| | |
|------------|---------------------------------|
| 400 00 000 | Road |
| 400 00 010 | Road - Under Construction (U/C) |
| 400 01 000 | Road (Gravel Divided) |
| 400 01 010 | Road/GD - 1 Lane Each Way (LEW) |
| 400 01 020 | Road/GD - 2 LEW |
| 400 01 030 | Road/GD - 3 LEW |
| 400 01 040 | Road/GD - Proposed |
| 400 01 050 | Road/GD - U/C |
| 400 01 060 | Road/GD - U/C - 1 LEW |
| 400 01 070 | Road/GD - U/C - 2 LEW |
| 400 01 080 | Road/GD - U/C - 3 LEW |
| 400 02 000 | Road (Gravel Undivided) |
| 400 02 010 | Road/GU - 1 Lane |
| 400 02 020 | Road/GU - 2 Lanes |
| 400 02 030 | Road/GU - 3 Lanes |
| 400 02 040 | Road/GU - Proposed |
| 400 02 050 | Road/GU - U/C |
| 400 02 060 | Road/GU - UC-1 Lane |
| 400 02 070 | Road/GU - UC-2 Lanes |
| 400 02 080 | Road/GU - UC-3 Lanes |
| 400 03 000 | Road (Paved Divided) |
| 400 03 010 | Road/PD - Elevated |
| 400 03 020 | Road/PD - E- 1 LEW |
| 400 03 030 | Road/PD - E- 2 LEW |
| 400 03 040 | Road/PD - E- 3 LEW |
| 400 03 050 | Road/PD - E- 4 LEW |
| 400 03 060 | Road/PD - E- > 4 LEW |
| 400 03 070 | Road/PD - Not Elevated |
| 400 03 080 | Road/PD - NE- 1 LEW |

| | |
|------------|------------------------------|
| 400 03 090 | Road/PD - NE- 2 LEW |
| 400 03 100 | Road/PD - NE- 3 LEW |
| 400 03 110 | Road/PD - NE- 4 LEW |
| 400 03 120 | Road/PD - NE- > 4 LEW |
| 400 03 130 | Road/PD - NE- Proposed |
| 400 03 140 | Road/PD - Under Construction |
| 400 03 010 | Road/PD - UC-Elevated |
| 400 03 020 | Road/PD - UC-E- 1 LEW |
| 400 03 030 | Road/PD - UC-E- 2 LEW |
| 400 03 040 | Road/PD - UC-E- 3 LEW |
| 400 03 050 | Road/PD - UC-E- 4 LEW |
| 400 03 060 | Road/PD - UC-E- > 4 LEW |
| 400 03 070 | Road/PD - UC-Not Elevated |
| 400 03 080 | Road/PD - UC-NE- 1 LEW |
| 400 03 090 | Road/PD - UC-NE- 2 LEW |
| 400 03 100 | Road/PD - UC-NE- 3 LEW |
| 400 03 110 | Road/PD - UC-NE- 4 LEW |
| 400 03 120 | Road/PD - UC-NE- > 4 LEW |
| 400 04 000 | Road (Paved Undivided) |
| 400 04 010 | Road/PU - Elevated |
| 400 04 020 | Road/PU - E- 1 Lane |
| 400 04 030 | Road/PU - E- 2 Lanes |
| 400 04 040 | Road/PU - E- 3 Lanes |
| 400 04 050 | Road/PU - E- 4 Lanes |
| 400 04 060 | Road/PU - E- > 4 Lanes |
| 400 04 070 | Road/PU - Not Elevated |
| 400 04 080 | Road/PU - NE - 1 Lane |
| 400 04 090 | Road/PU - NE - 2 Lanes |
| 400 04 100 | Road/PU - NE - 3 Lanes |
| 400 04 110 | Road/PU - NE - 4 Lanes |
| 400 04 120 | Road/PU - NE - > 4 Lanes |
| 400 04 130 | Road/PU - NE - Proposed |
| 400 04 140 | Road/PU - U/C |
| 400 04 150 | Road/PU - UC-Elevated |
| 400 04 160 | Road/PU - UC/E- 1 Lane |
| 400 04 170 | Road/PU - UC/E- 2 Lanes |
| 400 04 180 | Road/PU - UC/E- 3 Lanes |
| 400 04 190 | Road/PU - UC/E- 4 Lanes |
| 400 04 200 | Road/PU - UC/E- > 4 Lanes |
| 400 04 210 | Road/PU - UC Not Elevated |
| 400 04 220 | Road/PU - UC/NE- 1 Lane |
| 400 04 230 | Road/PU - UC/NE- 2 Lanes |
| 400 04 240 | Road/PU - UC/NE- 3 Lanes |
| 400 04 250 | Road/PU - UC/NE- 4 Lanes |
| 400 04 260 | Road/PU - UC/NE- > 4 Lanes |
| 400 05 000 | Road (Unimproved) |
| 400 05 010 | Road(Unimproved) - U/C |
| 400 06 000 | Road (Winter Road) |

41- ROADS OR RAILWAYS - Accessway

| | |
|------------|------------------------|
| 401 00 000 | Accessway |
| 401 00 010 | Accessway - Private |
| 401 00 020 | Accessway - Public |
| 401 01 000 | Accessway (Gravel) |
| 401 01 010 | Accessway/G - Private |
| 401 01 020 | Accessway/G - Public |
| 401 02 000 | Accessway (Paved) |
| 401 02 010 | Accessway/P - Private |
| 401 02 020 | Accessway/P - Public |
| 401 03 000 | Accessway (Unimproved) |
| 401 03 010 | Accessway/U - Private |
| 401 03 020 | Accessway/U - Public |

42- ROADS OR RAILWAYS - Track or Trail

| | |
|------------|----------------------------|
| 402 00 000 | Track |
| 402 00 010 | Track - Cart/Tractor |
| 402 00 020 | Track - Winter |
| 402 01 000 | Trail |
| 402 01 010 | Trail - Portage |
| 402 01 020 | Trail - Proposed |
| 402 01 030 | Trail - Ski |
| 402 01 040 | Trail - Snowmobile |
| 402 01 050 | Trail - Under Construction |

| | | | |
|-----|----|-----|-----------------------------|
| 402 | 02 | 000 | Trail (Broadwalk) |
| 402 | 03 | 000 | Trail (Improved) |
| 402 | 03 | 010 | Trail/I - Bicycle |
| 402 | 03 | 020 | Trail/I - Equestrian |
| 402 | 03 | 030 | Trail/I - Pedestrian/Hiking |
| 402 | 04 | 000 | Trail (Paved) |
| 402 | 04 | 010 | Trail/P - Bicycle |
| 402 | 04 | 020 | Trail/P - Pedestrian/Hiking |
| 402 | 05 | 000 | Trail (Unimproved) |
| 402 | 05 | 010 | Trail/U - Bicycle |
| 402 | 05 | 020 | Trail/U - Equestrian |
| 402 | 05 | 030 | Trail/U - Pedestrian/Hiking |

43- ROADS OR RAILWAYS - Associated Feature

| | | | |
|-----|----|-----|------------------------------|
| 403 | 00 | 000 | Bridge (Pedestrian/Cycle) |
| 403 | 01 | 000 | Bridge (Railway) |
| 403 | 01 | 010 | Bridge/Rl - Arch |
| 403 | 01 | 020 | Bridge/Rl - Covered |
| 403 | 01 | 030 | Bridge/Rl - Draw |
| 403 | 01 | 040 | Bridge/Rl - Lift |
| 403 | 01 | 050 | Bridge/Rl - Swing |
| 403 | 01 | 060 | Bridge/Rl - Trestle |
| 403 | 02 | 000 | Bridge(Road & Railway) |
| 403 | 02 | 010 | Bridge/Rd-Rl - Arch |
| 403 | 02 | 020 | Bridge/Rd-Rl - Covered |
| 403 | 02 | 030 | Bridge/Rd-Rl - Draw |
| 403 | 02 | 040 | Bridge/Rd-Rl - Lift |
| 403 | 02 | 050 | Bridge/Rd-Rl - Swing |
| 403 | 02 | 060 | Bridge/Rd-Rl - Trestle |
| 403 | 03 | 000 | Bridge (Road) |
| 403 | 03 | 010 | Bridge/Rd - Arch |
| 403 | 03 | 020 | Bridge/Rd - Covered |
| 403 | 03 | 030 | Bridge/Rd - Draw |
| 403 | 03 | 040 | Bridge/Rd - Floating |
| 403 | 03 | 050 | Bridge/Rd - Lift |
| 403 | 03 | 060 | Bridge/Rd - Suspension |
| 403 | 03 | 070 | Bridge/Rd - Swing |
| 403 | 04 | 000 | Bus Way |
| 403 | 05 | 000 | Cattle Gate |
| 403 | 06 | 000 | Causeway (Railway) |
| 403 | 07 | 000 | Causeway (Road & Rail) |
| 403 | 08 | 000 | Causeway (Road) |
| 403 | 09 | 000 | Crossing Gate (Railway) |
| 403 | 10 | 000 | Culvert (Railway) |
| 403 | 11 | 000 | Culvert (Road) |
| 403 | 12 | 000 | Curb |
| 403 | 13 | 000 | Cut (Railway) |
| 403 | 14 | 000 | Cut (Road) |
| 403 | 15 | 000 | Embankment/Fill (Railway) |
| 403 | 16 | 000 | Embankment/Fill (Road) |
| 403 | 17 | 000 | Guard Rail/Guide Rail |
| 403 | 18 | 000 | Gutter |
| 403 | 19 | 000 | Interchange |
| 403 | 19 | 000 | Interchange- Cloverleaf |
| 403 | 20 | 000 | Intersection |
| 403 | 20 | 010 | Intersection- Diamond |
| 403 | 20 | 020 | Intersection- Traffic Circle |
| 403 | 20 | 030 | Intersection- Trumpet |
| 403 | 20 | 040 | Intersection- Y/Fork |
| 403 | 21 | 000 | Median |
| 403 | 22 | 000 | Median Barrier |
| 403 | 23 | 000 | Overpass(Pedestrian/Cycle) |
| 403 | 24 | 000 | Parking Meter |
| 403 | 25 | 000 | Parking Rails |
| 403 | 26 | 000 | Pavement Marking |
| 403 | 27 | 000 | Post (Mile or Kilometre) |
| 403 | 28 | 000 | Retaining Wall |
| 403 | 29 | 000 | Revetment |
| 403 | 30 | 000 | Road Number Symbol |
| 403 | 31 | 000 | Runaway Preventer |
| 403 | 32 | 000 | Sidewalk |
| 403 | 32 | 010 | Sidewalk- Gravel |
| 403 | 32 | 020 | Sidewalk- Paved |
| 403 | 33 | 000 | Sign/Sign Post |

| | |
|------------|---------------------------|
| 403 34 000 | Snow Shed (Railway) |
| 403 35 000 | Snow Shed (Road) |
| 403 36 000 | Stop (Railway) |
| 403 37 000 | Subway Entrance |
| 403 38 000 | Switch (Railway) |
| 403 39 000 | Toll Gate |
| 403 40 000 | Traffic Island |
| 403 41 000 | Traffic Light |
| 403 42 000 | Traffic Light Control Box |
| 403 43 000 | Traffic Signal (Railway) |
| 403 44 000 | Tunnel |
| 403 45 000 | Tunnel (Railway) |
| 403 46 000 | Tunnel (Road) |
| 403 47 000 | Turntable (Railway) |
| 403 48 000 | Underpass |
| 403 48 010 | Underpass- Cattle |
| 403 48 020 | Underpass- Pedestrian |
| 403 49 000 | Viaduct |
| 403 50 000 | Warning Light (Railway) |

44- ROADS OR RAILWAYS - Through Rail Line

| | |
|------------|-------------------------------|
| 404 00 000 | Monorail |
| 404 01 000 | Rail Line |
| 404 01 010 | Rail Line - Narrow Gauge |
| 404 01 020 | Rail Line/NG - Multiple Track |
| 404 01 030 | Rail Line/NG - Single Track |
| 404 01 040 | Rail Line - Proposed |
| 404 01 050 | Rail Line - U/C |
| 404 02 000 | Rail Line (Double Track) |
| 404 02 010 | Rail Line/DT - Elevated |
| 404 02 020 | Rail Line/DT - Surface |
| 404 02 030 | Rail Line/DT - Underground |
| 404 02 040 | Rail Line/DT - Subway |
| 404 02 050 | Rail Line/DT - Train |
| 404 02 060 | Rail Line/DT - Tramway |
| 404 03 000 | Rail Line (Multiple Track) |
| 404 03 010 | Rail Line/MT - Elevated |
| 404 03 020 | Rail Line/MT - Surface |
| 404 03 030 | Rail Line/MT - Underground |
| 404 03 040 | Rail Line/MT - Subway |
| 404 03 050 | Rail Line/MT - Train |
| 404 03 060 | Rail Line/MT - Tramway |
| 404 04 000 | Rail Line (Single Track) |
| 404 04 010 | Rail Line/ST - Elevated |
| 404 04 020 | Rail Line/ST - Surface |
| 404 04 030 | Rail Line/ST - Underground |
| 404 04 040 | Rail Line/ST - Subway |
| 404 04 050 | Rail Line/ST - Train |
| 404 04 060 | Rail Line/ST - Tramway |

45- ROADS OR RAILWAYS - Subsidiary Rail Line

| | |
|------------|-----------------------------|
| 405 00 000 | Siding |
| 405 00 010 | Siding - Light Rail Transit |
| 405 00 020 | Siding - Subway |
| 405 00 030 | Siding - Train |
| 405 00 040 | Siding - Tramway |
| 405 01 000 | Siding (Narrow Gauge) |
| 405 02 000 | Spur |
| 405 02 010 | Spur - Light Rail Transit |
| 405 02 020 | Spur - Subway |
| 405 02 030 | Spur - Train |
| 405 02 040 | Spur - Tramway |
| 405 03 000 | Spur (Narrow Gauge) |

50- UTILITY - Utility

| | |
|------------|----------------------------|
| 500 00 000 | Cable |
| 500 00 010 | Cable - Above Ground (A.G) |
| 500 00 020 | Cable - Electrical |
| 500 00 030 | Cable - Electrical - A.G |
| 500 00 040 | Cable - Electrical - U.G |
| 500 00 050 | Cable - Electrical - U.W |
| 500 00 060 | Cable - Telegraph |

| | |
|------------|------------------------------------|
| 500 00 070 | Cable - Telegraph - A.G |
| 500 00 080 | Cable - Telegraph - U.G |
| 500 00 090 | Cable - Telegraph - U.W |
| 500 00 100 | Cable - Telephone |
| 500 00 110 | Cable - Telephone - A.G |
| 500 00 120 | Cable - Telephone - U.G |
| 500 00 130 | Cable - Telephone - U.W |
| 500 00 140 | Cable - Underground (U.G) |
| 500 00 150 | Cable - Underwater (U.W) |
| 500 01 000 | Cable TV |
| 500 01 010 | Cable - A.G |
| 500 01 020 | Cable - U.G |
| 500 02 000 | Canal/Ditch -Sewage/Storm |
| 500 03 000 | Catch Basin -Sewage/Storm |
| 500 04 000 | Collector - Sewage |
| 500 05 000 | Conduit |
| 500 05 010 | Conduit - Electrical |
| 500 05 020 | Conduit - Electrical - A.G |
| 500 05 030 | Conduit - Electrical - U.G |
| 500 05 040 | Conduit - Telephone |
| 500 05 050 | Conduit - Telephone - A.G |
| 500 05 060 | Conduit - Telephone - U.G |
| 500 06 000 | Disposal Bed - Sewage |
| 500 07 000 | Filtration Bed - Water |
| 500 08 000 | Hydrant |
| 500 09 000 | Junction Box - Electrical |
| 500 10 000 | Light Standard-Electrical |
| 500 11 000 | Line (Transmission) |
| 500 11 010 | Line/T - Electrical |
| 500 11 020 | Line/T - Electrical - Primary |
| 500 11 030 | Line/T - Electrical - Secondary |
| 500 11 040 | Line/T - Electrical - Tertiary |
| 500 11 050 | Line/T - Telegraph |
| 500 11 060 | Line/T - Telephone |
| 500 12 000 | Manhole |
| 500 12 010 | Manhole - Electrical |
| 500 12 020 | Manhole - Heating |
| 500 12 030 | Manhole - Sewage (Swg) |
| 500 12 040 | Manhole/Swg - Sanitation |
| 500 12 050 | Manhole/Swg - Storm |
| 500 12 060 | Manhole/Swg - Storm & Sanitation |
| 500 12 070 | Manhole - Telephone |
| 500 13 000 | Outlet |
| 500 13 010 | Outlet - Electrical |
| 500 13 020 | Outlet - Sanitation |
| 500 14 000 | Pedestal |
| 500 14 010 | Pedestal - Electrical |
| 500 14 020 | Pedestal - Multiple Use |
| 500 14 030 | Pedestal - Telephone |
| 500 15 000 | Pipeline |
| 500 15 010 | Pipeline - A.G |
| 500 15 020 | Pipeline - Crude Oil/Synthetic Oil |
| 500 15 030 | Pipeline - CO/SO/Flow/Gather Line |
| 500 15 040 | Pipeline - CO/SO/F-G L- A.G |
| 500 15 050 | Pipeline - CO/SO/F-G L -U.G |
| 500 15 060 | Pipeline - CO/SO/F-G L -U.W |
| 500 15 070 | Pipeline - CO/SO - Transmission |
| 500 15 080 | Pipeline - CO/SO/T - A.G |
| 500 15 090 | Pipeline - CO/SO/T - U.G |
| 500 15 100 | Pipeline - Heating |
| 500 15 110 | Pipeline - H - Hot Water |
| 500 15 120 | Pipeline - H/Hot Water -A.G |
| 500 15 130 | Pipeline - H/Hot Water -U.G |
| 500 15 140 | Pipeline - H - Steam |
| 500 15 150 | Pipeline - H/Steam - A.G |
| 500 15 160 | Pipeline - H/Steam - U.G |
| 500 15 170 | Pipeline - Natural Gas |
| 500 15 180 | Pipeline - NE- Distribution |
| 500 15 190 | Pipeline - NE/D - U.G |
| 500 15 200 | Pipeline - NE-Gather/Flow Line |
| 500 15 210 | Pipeline - NE/G-F L - A.G |
| 500 15 220 | Pipeline - NE/G-F L - U.G |
| 500 15 230 | Pipeline - NE/G-F L - U.W |
| 500 15 240 | Pipeline - NE-Service |
| 500 15 250 | Pipeline - NE-Service - U.G |

| | |
|------------|----------------------------------|
| 500 15 260 | Pipeline - NE-Transmission |
| 500 15 270 | Pipeline - NE/T - A.G |
| 500 15 280 | Pipeline - NE/T - U.G |
| 500 15 290 | Pipeline - Natural Gas Liquids |
| 500 15 300 | Pipeline - NEL - U.G |
| 500 15 310 | Pipeline - Non Potable Water-A.G |
| 500 15 320 | Pipeline - Non Potable Water-U.G |
| 500 15 330 | Pipeline - Petroleum Products |
| 500 15 340 | Pipeline - PP - U.G |
| 500 15 350 | Pipeline - Potable Water-A.G |
| 500 15 360 | Pipeline - PW - U.G |
| 500 15 370 | Pipeline - Sanitation |
| 500 15 380 | Pipeline - Solids/Slurry |
| 500 15 390 | Pipeline - SS - Coal |
| 500 15 400 | Pipeline - SS - Coal - A.G |
| 500 15 410 | Pipeline - SS - Coal - U.G |
| 500 15 420 | Pipeline - SS - Mine Tailings |
| 500 15 430 | Pipeline - SS - MT- A.G |
| 500 15 440 | Pipeline - SS - MT- U.G |
| 500 15 450 | Pipeline - SS - Sulphur |
| 500 15 460 | Pipeline - SS - S- A.G |
| 500 15 470 | Pipeline - SS - S- U.G |
| 500 15 480 | Pipeline - Storm |
| 500 15 490 | Pipeline - S- Sanitation |
| 500 15 500 | Pipeline - Underground |
| 500 16 000 | Pipeline - Heater-PP |
| 500 17 000 | Pole |
| 500 17 010 | Pole - Electrical |
| 500 17 020 | Pole - Telephone |
| 500 18 000 | Pump |
| 500 18 010 | Pump - Petroleum Products |
| 500 18 020 | Pump - Sewage |
| 500 18 030 | Pump - Water |
| 500 19 000 | Regulator |
| 500 19 010 | Regulator - Heating/Steam |
| 500 19 020 | Regulator - Petrol Products |
| 500 19 030 | Regulator - Water |
| 500 20 000 | Service Box - Electrical |
| 500 21 000 | Settling Bassin - Sewage |
| 500 22 000 | Sludge Well - Sewage |
| 500 23 000 | Sprinkler -Irrigation(Fixed) |
| 500 24 000 | Tank |
| 500 24 010 | Tank - Petroleum Products |
| 500 24 020 | Tank - PP- Horizontal |
| 500 24 030 | Tank - PP- Vertical |
| 500 24 040 | Tank - Sewage |
| 500 24 050 | Tank - Swg- Aeration |
| 500 24 060 | Tank - Swg- Digestion |
| 500 24 070 | Tank - Swg- Re-aeration |
| 500 24 080 | Tank - Swg- Sedimentation |
| 500 24 090 | Tank - Swg- Settling |
| 500 24 100 | Tank - Swg- Skimming |
| 500 24 110 | Tank - Water (Holding) |
| 500 25 000 | Tap - Water |
| 500 26 000 | Telephone (Pay) |
| 500 27 000 | Tower - Water (Holding) |
| 500 28 000 | Tower/Pylon - Electrical |
| 500 29 000 | Transformer - Electrical |
| 500 29 010 | Transformer/E - A.G |
| 500 29 020 | Transformer/E - On Pad |
| 500 29 030 | Transformer/E - On Pole |
| 500 29 040 | Transformer/E - U.G |
| 500 29 050 | Transformer - Substation |
| 500 30 000 | Utilidor |
| 500 31 000 | Valve |
| 500 31 010 | Valve - Heating |
| 500 31 020 | Valve - Petroleum Products |
| 500 31 030 | Valve - Water |

60- BOUNDARIES - Boundary/Associated Feature

| | |
|------------|--------------------|
| 600 00 000 | Boundary |
| 600 00 010 | Boundary - Borough |
| 600 00 020 | Boundary - City |
| 600 00 030 | Boundary - County |

| | |
|------------|---|
| 600 00 040 | Boundary - District |
| 600 00 050 | Boundary - District Municipality |
| 600 00 060 | Boundary - Forest & Grazing District |
| 600 00 070 | Boundary - Government Reserve |
| 600 00 080 | Boundary - Highways/Transportation District |
| 600 00 090 | Boundary - Improvement District |
| 600 00 100 | Boundary - Metropolitan Area |
| 600 00 110 | Boundary - Military Reserve |
| 600 00 120 | Boundary - Parish |
| 600 00 130 | Boundary - Park |
| 600 00 140 | Boundary - Park - Municipal |
| 600 00 150 | Boundary - Park - National |
| 600 00 160 | Boundary - Park - Provincial |
| 600 00 170 | Boundary - Park - Regional |
| 600 00 180 | Boundary - Regional Municipality |
| 600 00 190 | Boundary - Rural Municipality |
| 600 00 200 | Boundary - Township |
| 600 00 210 | Boundary - Township Municipality |
| 600 00 220 | Boundary - Water Management District |
| 600 01 000 | Boundary (Baseline) |
| 600 02 000 | Boundary (International) |
| 600 02 010 | Boundary - Unsurveyed |
| 600 03 000 | Boundary (Interprovincial) |
| 600 03 010 | Boundary/Ip - Unsurveyed |
| 600 04 000 | Boundary (Interstate) |
| 600 04 010 | Boundary/Is - Unsurveyed |
| 600 05 000 | Boundary (Interterritorial) |
| 600 05 010 | Boundary/It - Unsurveyed |
| 600 06 000 | Boundary (Meridian) |
| 600 07 000 | Boundary (Party Wall) |
| 600 08 000 | Boundary Sign |
| 600 09 000 | Corner |
| 600 10 000 | Corner (Block) |
| 600 11 000 | Corner (Easement) |
| 600 14 000 | Corner (Lot) |
| 600 14 010 | Corner - District |
| 600 14 020 | Corner - River |
| 600 14 030 | Corner - Road |
| 600 14 040 | Corner - Township |
| 600 15 000 | Corner (Mineral Claim) |
| 600 16 000 | Corner (Parcel) |
| 600 17 000 | Corner (Quarter Section) |
| 600 18 000 | Corner (Range) |
| 600 19 000 | Corner (Right of Way) |
| 600 20 000 | Corner (Section) |
| 600 21 000 | Corner (Sub-lot) |
| 600 22 000 | Corner (Timber Berth) |
| 600 23 000 | Corner (Township) |
| 600 23 040 | Corner - Municipal |
| 600 23 050 | Corner - M- Surveyed |
| 600 23 060 | Corner - M- Unsurveyed |
| 600 24 000 | Correction Line |
| 600 25 000 | Surveyed Line |

61- BOUNDARIES - Survey Monument

| | |
|------------|---------------------------------|
| 601 00 000 | Monument (Cadastral) |
| 601 00 010 | Monument - AP - Aluminium Post |
| 601 00 020 | Monument - CC - Cut Cross |
| 601 00 030 | Monument - CLS - Capped |
| 601 00 040 | Monument - DLS - Capped |
| 601 00 050 | Monument - IB - Iron Bar |
| 601 00 060 | Monument - IP - Iron Post |
| 601 00 070 | Monument - IP PITS-IP-FourPits |
| 601 00 080 | Monument - IPM - IP & Mound |
| 601 00 090 | Monument - SIB - Standard IB |
| 601 00 100 | Monument - SSIB - Short SIB |
| 601 00 110 | Monument - STONE M -Stone M |
| 601 00 120 | Monument - WIT IP-Witness IP |
| 601 00 130 | Monument - WP - Wooden Post |
| 601 01 000 | Monument |
| 601 01 010 | Monument - Calibration Baseline |
| 601 01 020 | Monument - Control Survey |
| 601 01 030 | Monument - CS- Horiz.Control |
| 601 01 040 | Monument - CS-HC- 1st Order |

| | |
|------------|--------------------------------------|
| 601 01 050 | Monument - CS-HC- 2nd Order |
| 601 01 060 | Monument - CS-HC- 3rd Order |
| 601 01 070 | Monument - CS-HC- Astronomic |
| 601 01 080 | Monument - CS-HC- Doppler |
| 601 01 090 | Monument - International Boundary |
| 601 01 100 | Monument - Interprovincial Boundary |
| 601 01 110 | Monument - Interterritorial Boundary |
| 601 01 120 | Monument - Vertical Control |
| 601 01 130 | Monument -VC- Federal |
| 601 01 140 | Monument -VC- Municipal |
| 601 01 150 | Monument -VC- Private |
| 601 01 160 | Monument -VC- Provincial |

62- BOUNDARIES - Unmonumented Control Point

| | |
|------------|--------------------------------|
| 602 00 000 | Horizontal & Vertical C.P |
| 602 00 010 | H & V CP - Photogrammetric |
| 602 00 020 | H & V CP - Surveyed |
| 602 01 000 | Horizontal Control Point (HCP) |
| 602 01 010 | HCP - Aerodist |
| 602 01 020 | HCP - Photogrammetric |
| 602 01 030 | HCP - Surveyed |
| 602 02 000 | Vertical Control Point (VCP) |
| 602 02 010 | VCP - APR |
| 602 02 020 | VCP - Photogrammetric |
| 602 02 030 | VCP - Surveyed |

63- BOUNDARIES - Other

| | |
|------------|------------------|
| 603 00 000 | Chainage Station |
| 603 01 000 | Photo Centre |

70- HYDROGRAPHY - Water Course/Associat Feature

| | |
|------------|-------------------------------------|
| 700 01 000 | Aqueduct |
| 700 02 000 | Canal |
| 700 02 010 | Canal - Irrigation |
| 700 02 020 | Canal - Navigable |
| 700 02 030 | Canal - Non-navigable |
| 700 02 040 | Canal - Route Through Water |
| 700 03 000 | Channel |
| 700 03 010 | Channel - Navigable |
| 700 03 020 | Channel - Non-navigable |
| 700 04 000 | Confluence |
| 700 05 000 | Dam |
| 700 05 020 | Dam - Concrete |
| 700 05 030 | Dam - Earth |
| 700 05 040 | Dam - Masonry |
| 700 05 050 | Dam - Rockfill |
| 700 06 000 | Ditch |
| 700 06 010 | Ditch - Drainage |
| 700 06 020 | Ditch - Irrigation |
| 700 07 000 | Falls |
| 700 08 000 | Fish Ladder |
| 700 09 000 | Flow Arrow |
| 700 10 000 | Flume |
| 700 11 000 | Ford |
| 700 12 000 | Penstock |
| 700 13 000 | Rapids |
| 700 13 010 | Rapids - On Narrow River/Stream |
| 700 13 020 | Rapids - On River/Stream-Continuous |
| 700 14 000 | River/Stream |
| 700 14 010 | River/Stream - Braided |
| 700 14 020 | River/Stream - Disappearing |
| 700 14 030 | River/Stream - Dry |
| 700 14 040 | River/Stream - Indefinite |
| 700 14 050 | River/Stream - Intermittent |
| 700 14 060 | River/Stream - Perennial |
| 700 14 070 | River/Stream/P -Mean Water Level |
| 700 14 080 | River/Stream/P -Time of Photo Level |
| 700 15 000 | Shoreline |
| 700 15 010 | Shoreline - Indefinite |
| 700 15 020 | Shoreline - Wooded |
| 700 16 000 | Slide (Log) |
| 700 17 000 | Sluice |

| | |
|------------|-------------|
| 700 18 000 | Sluice Gate |
| 700 19 000 | Spillway |
| 700 20 000 | Tailrace |
| 700 21 000 | Weir |

71- HYDROGRAPHY - Inland Water Body

| | |
|------------|------------------------------|
| 701 00 000 | Dugout |
| 701 01 000 | Flooded Land |
| 701 01 010 | Flooded Land- Inundated |
| 701 02 000 | Lagoon |
| 701 03 000 | Lake |
| 701 03 010 | Lake - Alkali |
| 701 03 020 | Lake - Dry |
| 701 03 030 | Lake - Indefinite |
| 701 03 040 | Lake - Intermittent |
| 701 03 050 | Lake - Marshy |
| 701 03 060 | Lake - Perennial |
| 701 03 060 | Lake/P - Mean Water Level |
| 701 03 070 | Lake/P - Time of Photo Level |
| 701 03 080 | Lake - Salt |
| 701 03 090 | Lake - Shallow |
| 701 03 100 | Lake - Tundra |
| 701 04 000 | Oxbow |
| 701 05 000 | Pond |
| 701 05 010 | Pond - Tailing |
| 701 05 020 | Pond - Tundra |
| 701 06 000 | Quarry (Water-filled) |
| 701 07 000 | Reservoir |
| 701 07 010 | Reservoir - Underground |
| 701 08 000 | Slough |

72- HYDROGRAPHY - Wetlands

| | |
|------------|------------------|
| 702 00 000 | Bog |
| 702 00 010 | Bog - Cranberry |
| 702 00 020 | Bog - Floating |
| 702 00 030 | Bog - Palsa |
| 702 00 040 | Bog - Peat |
| 702 00 050 | Bog - String |
| 702 01 000 | Fen |
| 702 01 010 | Fen - Floating |
| 702 01 020 | Fen - Palsa |
| 702 01 030 | Fen - String |
| 702 02 000 | Marsh |
| 702 02 010 | Marsh - In Water |
| 702 02 020 | Marsh - Salt |
| 702 04 000 | Swamp |

73- HYDROGRAPHY - Permanent Frozen Hydrography Feature

| | |
|------------|-----------------|
| 703 00 000 | Crevasse |
| 703 01 000 | Glacial Ice |
| 703 02 000 | Glacier |
| 703 03 000 | Ice Cap |
| 703 04 000 | Ice Shelf Limit |
| 703 05 000 | Icefield |
| 703 06 000 | Snowfield |

74- HYDROGRAPHY - Related Hydrographic Feature

| | |
|------------|------------------------------|
| 704 00 000 | Beach |
| 704 00 010 | Beach - Mud |
| 704 00 020 | Beach - Pebble |
| 704 00 030 | Beach - Rock |
| 704 00 040 | Beach - Sand |
| 704 01 000 | Breakwall/Breakwater |
| 704 01 010 | Breakwall/Breakwater - Large |
| 704 01 020 | Breakwall/Breakwater - Small |
| 704 02 000 | Crib |
| 704 03 000 | Dyke/Levee |
| 704 04 000 | Groyne |
| 704 05 000 | Island |
| 704 05 010 | Island - Marshy |
| 704 05 020 | Island - Wooded |

| | |
|------------|------------------------------|
| 704 06 000 | Ledge |
| 704 07 000 | Piling/Pile |
| 704 08 000 | Reef |
| 704 09 000 | Rock |
| 704 09 010 | Rock - Exposed |
| 704 09 020 | Rock - Submerged |
| 704 10 000 | Sand Bar |
| 704 11 000 | Sea Wall |
| 704 12 000 | Shoal |
| 704 13 000 | Tidal Flats |
| 704 13 010 | Tidal Flats - Limits Unknown |
| 704 14 000 | Tide Limit |
| 704 15 000 | Water Mark |
| 704 15 010 | Water Mark - High |
| 704 15 020 | Water Mark - Low |
| 704 16 000 | Wreck |
| 704 16 010 | Wreck - Awash |
| 704 16 020 | Wreck - Exposed |
| 704 16 030 | Wreck - Submerged |

75- HYDROGRAPHY - Ground Water Feature

| | |
|------------|-----------------------|
| 705 00 000 | Spring |
| 705 00 010 | Spring - Fresh |
| 705 00 020 | Spring - Hot |
| 705 00 030 | Spring - Intermittent |
| 705 00 040 | Spring - Mineral |
| 705 01 000 | Well (Water) |
| 705 01 010 | Well/W - Artesian |
| 705 01 020 | Well/W - Brine |
| 705 01 030 | Well/W - Drilled |
| 705 01 040 | Well/W - Dug |

76- HYDROGRAPHY - Coastal Feature

| | |
|------------|---------------------------------|
| 706 00 000 | Coastline |
| 706 00 010 | Coastline - Mean Sea Level |
| 706 00 020 | Coastline - Time of Photo Level |
| 706 01 000 | Kelp Bed |

80- RELIEF & LANDFORM - Relief

| | |
|------------|------------------------------|
| 800 00 000 | Contour (Bathymetric) |
| 800 00 010 | Contour/B - Approximate |
| 800 00 020 | Contour/B - Auxiliary |
| 800 00 030 | Contour/B - Depression |
| 800 01 000 | Contour (Glacier, Ice, Snow) |
| 800 01 010 | Contour/G-I-S - Approximate |
| 800 01 020 | Contour/G-I-S - Auxiliary |
| 800 01 030 | Contour/G-I-S - Depression |
| 800 01 040 | Contour/G-I-S - Form Line |
| 800 02 000 | Contour (Land) |
| 800 02 010 | Contour/L - Approximate |
| 800 02 020 | Contour/L - Auxiliary |
| 800 02 030 | Contour/L - Depression |
| 800 02 040 | Contour/L - Form Line |
| 800 03 000 | Hachured Area |
| 800 04 000 | Relief Shaded Area |
| 800 05 000 | Spot Height |
| 800 06 000 | Water Level |

81- RELIEF & LANDFORM - Landform

| | |
|------------|--------------------|
| 801 00 000 | Alluvial Fan |
| 801 01 000 | Bench (Landform) |
| 801 02 000 | Bluff |
| 801 03 000 | Cave |
| 801 04 000 | Cinder Cone |
| 801 05 000 | Cliff |
| 801 06 000 | Crater |
| 801 06 010 | Crater - Explosion |
| 801 06 020 | Crater - Impact |
| 801 06 030 | Crater - Volcanic |
| 801 07 000 | Distorted Surface |
| 801 08 000 | Drumlin |

| | |
|------------|---------------------------|
| 801 09 000 | Dune |
| 801 10 000 | Dyke |
| 801 11 000 | Escarpment |
| 801 12 000 | Esker |
| 801 13 000 | Fan Delta |
| 801 14 000 | Hill |
| 801 15 000 | Lava Flow |
| 801 16 000 | Moraine |
| 801 16 010 | Moraine - Lateral |
| 801 16 020 | Moraine - Medial |
| 801 16 030 | Moraine - Terminal |
| 801 17 000 | Mountain Peak |
| 801 19 000 | Pass (Mountain) |
| 801 20 000 | Pingo |
| 801 21 000 | Raised Beach/Strand Lines |
| 801 22 000 | Re-entrant |
| 801 23 000 | Rock Outcrop |
| 801 24 000 | Scree/Talus |
| 801 25 000 | Sinkhole |
| 801 26 000 | Slide |
| 801 26 010 | Slide - Mud |
| 801 26 020 | Slide - Rock |
| 801 27 000 | Spur (Landform) |
| 801 28 000 | Tundra Polygons |
| 801 29 000 | Volcano |

90- LAND COVER - Woodland

| | |
|------------|---------------------------|
| 900 00 000 | Burn |
| 900 01 000 | Clear Area (Natural) |
| 900 02 000 | Cut Line |
| 900 03 000 | Firebreak/Fireguard (Cut) |
| 900 04 000 | Grove |
| 900 05 000 | Harvested Area |
| 900 06 000 | Reforested Area |
| 900 07 000 | Row of Trees |
| 900 08 000 | Stand |
| 900 09 000 | Tree |
| 900 10 000 | Wooded Area |
| 900 10 010 | Wooded Area- Coniferous |
| 900 10 020 | Wooded Area- Deciduous |
| 900 10 030 | Wooded Area- Mixed |

91- LAND COVER - Arable/Cultivated Land

| | |
|------------|---------------------------------|
| 901 00 000 | Cropland |
| 901 00 010 | Cropland- Grain |
| 901 00 020 | Cropland-Pasture & Forage Crops |
| 901 00 030 | Cropland- Root Crops |
| 901 01 000 | Hopfield |
| 901 02 000 | Market Garden |
| 901 03 000 | Nursery |
| 901 03 010 | Nursery - Horticulture |
| 901 03 020 | Nursery - Mixed |
| 901 03 030 | Nursery - Silviculture |
| 901 04 000 | Orchard |
| 901 05 000 | Vineyard |

92- LAND COVER - Low Vegetation

| | |
|------------|----------------|
| 902 00 000 | Grassland |
| 902 01 000 | Hedge/Hedgerow |
| 902 02 000 | Scrub/Brush |
| 902 03 000 | Shrub |
| 902 04 000 | Tundra |

93- LAND COVER - No Vegetation

| | |
|------------|----------------------|
| 903 00 000 | Barren Land |
| 903 00 010 | Barren Land - Gravel |
| 903 00 020 | Barren Land - Rock |
| 903 00 030 | Barren Land - Sand |
| 903 01 000 | Desert |

APPENDIX B

APPENDIX B: CREATION OF THE DIFFERENT DATABASES

This Appendix lists all the programs needed to generate the menus used in the main program; to create the required databases; and to store these menus in the appropriate databases. These menus include the point and area symbols menu, the line patterns menu, the fonts menu, the scaling menu, the line slope menu and the default menus.

Program 'incode' which loads the feature coding system into the database 'Code' is also listed at the end of this Appendix.

All these programs should be run beforehand in order to create the different databases needed by the main program, otherwise run-time errors will be encountered. This can be done by writing a batch file which will execute these programs automatically when the system is first installed.

```

let FONTsdb:=open.database("FONTS","friend","read")
let fix13 = s.lookup( "fix13", FONTsdb)
let bold = s.lookup("fixb13",FONTsdb)
let big = s.lookup("met22",FONTsdb)
let large = s.lookup("hci45i",FONTsdb)
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
  do {write "No utilities database - do prcdbmaker first'n"; abort}
let prcget=
begin
  structure procpak(proc(string -> pntr) xproc)
  s.lookup("prcget",procsdb) (xproc)
end
let seditor={structure procpak(proc(string,string,int,int,int,int-> string)
  xproc)
  prcget("seditor") (xproc) }
let error.message={structure procpak(proc(string,int,int) xproc)
  prcget("error.message") (xproc) }
let more={structure procpak(proc(*string,int,int) xproc)
  prcget("more") (xproc) }
let form.generate={structure procpak(proc( -> pntr ) xproc)
  prcget("form.generate") (xproc)}
let form.null={structure procpak(proc( string,int,int,int,int,pntr ) xproc)
  prcget("form.null") (xproc) }
structure form.package( proc( pntr) Form.show;
  proc( ) Form.all.show;
  proc( string,int,int,int,int,bool,proc(),pntr ->
    pntr ) Form.add;
  proc( pntr ) Form.remove;
  proc( string,pntr ) Form.update;
  proc( ) Form.clear;
  proc( -> pntr ) Form.mouse;
  proc( ) Fender;
  proc( ) Form.monitor )
let set.up.choose = {structure procpak(proc(*string -> pntr) xproc)
  prcget("set.up.choose") (xproc)}
structure chooser.pack( proc( string, int, int -> string ) do.choose;
  proc( string ) add.choose;
  proc( string ) remove.choose;
  proc( int, int ) list.choose )
let table.to.text = {structure procpak(proc(pntr -> *string) xproc)
  prcget("table.to.text") (xproc)}
let polygon = proc(real Xcentre, Ycentre, Radian;string resolution -> pic)
begin
  let Pic1 := [Xcentre + Radian, Ycentre]
  let AngleInRadian := 0.0
  let X := 0.0
  let Y := 0.0
  let interval := 0
  let startang := 0
  let endang := 0
  case true of
    resolution = "tr" : { interval := 120; startang := 90; endang := 450 }
    resolution = "sq" : { interval := 90; startang := 45; endang := 405 }
    resolution = "hx" : { interval := 60; startang := 0; endang := 420 }
    resolution = "vl" : { interval := 30; startang := 0; endang := 390 }
    resolution = "low" : { interval := 15; startang := 0; endang := 375 }
    resolution = "hi" : { interval := 10; startang := 0; endang := 370 }
    default: { interval := 5; startang := 0; endang := 365 }
  for angle = startang to endang by interval do
    begin
      AngleInRadian := angle * pi / 180
      X := Radian * cos(AngleInRadian) + Xcentre
      Y := Radian * sin(AngleInRadian) + Ycentre
      Pic1 := if angle = startang then [X,Y]
        else Pic1 ^ [X,Y]
    end
  end
  Pic1
end
let iradian := 6
let oradian := 10
let win1 = limit screen to 40 by 40 at 0,0
let win2 = limit screen to 40 by 40 at 40,40
let win3 = limit screen to 40 by 40 at 80,80
let win4 = limit screen to 40 by 40 at 120,120

```

```

let hcircle := [0,0]
let scircle := [0,0]
let ccircle := [0,0]
let thcircle := [0,0]
let bcircle := polygon(20, 20, oradian, "vh")
for radian = iradian to 1 by -1 do
    ccircle := if radian = iradian then bcircle & polygon(20, 20, radian, "vh")
                else ccircle & polygon(20, 20, radian, "vh")
draw(win1, ccircle, 0, X.dim(win1), 0, Y.dim(win1) )
for radian = iradian to iradian -1 by -1 do
    thcircle := if radian = iradian then bcircle & polygon(20, 20, radian, "vh")
                 else thcircle & polygon(20, 20, radian, "vh")
draw(win2, thcircle, 0, X.dim(win2), 0, Y.dim(win2) )
!draw(screen, circle, 0, X.dim(screen), 0, Y.dim(screen) )
for radian = oradian to 1 by -1 do
    scircle := if radian = oradian then bcircle & polygon(20, 20, radian, "vh")
                else scircle & polygon(20, 20, radian, "vh")
draw(win3, scircle, 0, X.dim(win3), 0, Y.dim(win3) )
hcircle := bcircle & polygon(20, 20, 4, "vh" )
draw(win4, hcircle, 0, X.dim(win4), 0, Y.dim(win4) )
let ssquare := [0,0]
let csquare := [0,0]
let ttrimark := [0,0]
let bsquare := polygon(20, 20, oradian, "sq")
xor win1 onto win1
xor win2 onto win2
xor win3 onto win3
xor win4 onto win4
for radian = iradian-1 to 1 by -1 do
    csquare := if radian = iradian-1 then bsquare & polygon(20, 20, radian, "sq")
                else csquare & polygon(20, 20, radian, "sq" )
draw(win1, csquare, 0, X.dim(win1), 0, Y.dim(win1) )
for radian = oradian to 1 by -1 do
    ssquare := if radian = oradian then bsquare & polygon(20, 20, radian, "sq" )
                else ssquare & polygon(20, 20, radian, "sq" )
draw(win3, ssquare, 0, X.dim(win3), 0, Y.dim(win3) )
let triangle := polygon(20, 20, oradian, "tr" )
draw(win2, triangle, 0, X.dim(win2), 0, Y.dim(win2) )
let trimark := triangle & polygon(20, 20, 4, "vh" )
for i=1 to 4 do
    ttrimark := if i=1 then trimark & polygon(20, 20, i, "vh" )
                 else ttrimark & polygon(20, 20, i, "vh" )
draw(win4, trimark, 0, X.dim(win4), 0, Y.dim(win4) )
xor win1 onto win1
draw(win1, ttrimark, 0, X.dim(win1), 0, Y.dim(win1) )
let hexagon = polygon(20, 20, oradian, "hx")
xor win4 onto win4
draw(win4, hexagon, 0, X.dim(win4), 0, Y.dim(win4) )
let DBsym := open.database("Symbols", "mys", "write" )
if DBsym is error.record do DBsym := create.database("Symbols", "mys" )
if DBsym is error.record do write"cannot open database'n"
let symbols := s.lookup("%$Geosym", DBsym )
if symbols = nil do {
    symbols := table()
    s.enter("%$Geosym", DBsym, symbols ) }
structure geosym( pic a.pic)
s.enter("plain circle", symbols, geosym( bcircle) )
s.enter("solid circle", symbols, geosym( scircle) )
s.enter("hollow circle", symbols, geosym( hcircle) )
s.enter("thick hollow circle", symbols, geosym( thcircle) )
s.enter("city circle", symbols, geosym( ccircle) )
s.enter("plain square", symbols, geosym( bsquare) )
s.enter("solid square", symbols, geosym( ssquare) )
s.enter("city square", symbols, geosym( csquare) )
s.enter("plain triangle", symbols, geosym( triangle) )
s.enter("tri.mark", symbols, geosym( trimark) )
s.enter("thktri.mark", symbols, geosym( ttrimark) )
s.enter("hexagon", symbols, geosym( hexagon) )
if commit() ~= nil do write "could not store symbols'n"

```

```

let FONTsdb:=open.database("FONTS","friend","read")
let fix13 = s.lookup( "fix13", FONTsdb)
let bold = s.lookup("fixb13",FONTsdb)
let big = s.lookup("met22",FONTsdb)
let large = s.lookup("hcl45i",FONTsdb)
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
  do {write "No utilities database - do prcdbmaker first'n"; abort}
let prcget=
begin
  structure procpak(proc(string -> pntr) xproc)
  s.lookup("prcget",procsdb) (xproc)
end
let form.generate={structure procpak(proc( -> pntr ) xproc)
  prcget("form.generate") (xproc)}
let form.null={structure procpak(proc( string,int,int,int,int,pntr ) xproc)
  prcget("form.null") (xproc) }
structure form.package( proc( pntr) Form.show;
  proc( ) Form.all.show;
  proc( string,int,int,int,int,bool,proc(),pntr ->
    pntr ) Form.add;
  proc( pntr ) Form.remove;
  proc( string,pntr ) Form.update;
  proc( ) Form.clear;
  proc( -> pntr ) Form.mouse;
  proc( ) Fender;
  proc( ) Form.monitor )
let message.proc = proc(string txt, txt1, txt2; int xpos, ypos, xdim, ydim ->
  bool)
begin
  let BoxDim.x := 0
  let lenth1.x := length(txt1)
  let lenth2.x := length(txt2)
  if lenth1.x >= lenth2.x then BoxDim.x := lenth1.x
  else BoxDim.x := lenth2.x
  BoxDim.x := BoxDim.x * 10 + 10
  let Box1x := xpos + 10
  let Box1y := ypos + 15
  let Box2x := xpos + xdim div 2 + 5
  let Box2y := Box1y
  let TxtBox.x := xpos + 10
  let TxtDim.x := xdim - 10
  let TxtBox.y := ypos + ydim - 40
  let TxtDim.y := ydim div 3 - 5
  let msgeBox = limit screen to xdim + 10 by ydim + 10 at xpos, ypos
  let hmsgeBox = limit screen to xdim by ydim at xpos + 5, ypos + 5
  let msgeSave = image xdim + 10 by ydim + 10 of off
  copy msgeBox onto msgeSave
  xnor msgeBox onto msgeBox
  xor hmsgeBox onto hmsgeBox
  let F = form.generate()
  let Fadd = F( Form.add )
  let the.bool := false
  form.null( txt, TxtBox.x, TxtBox.y, TxtDim.x, TxtDim.y, fix13)
  let Yesproc = proc()
    { the.bool := true; F(Fender)() }
  let Noproc = proc()
    { the.bool := false; F(Fender)() }
  let dummy := Fadd( txt1, Box1x, Box1y, BoxDim.x, 30, false, Yesproc,
    fix13 )
  dummy := Fadd( txt2, Box2x, Box2y, BoxDim.x, 30, false, Noproc, fix13 )
  F(Form.monitor)()
  let count := 1
  while count ~= 2000 do count := count +1
  copy msgeSave onto msgeBox
  the.bool
end

let StringToInt = proc(string S -> int)
begin
  let X := 0
  let tsign := 1
  let start := 1
  if S(1|1) = "-" do
    begin

```

```

        tsign := -1
        start := 2
    end
    for i = start to length(S) do
        X := (10 * X + decode(S(i|1)) - 48)
    X := X * tsign
    X
end

let DBsym := open.database("Symbols","mys","write")
if DBsym is error.record do DBsym := create.database( "Symbols", "mys" )
let agrsyms := s.lookup( "%$Agrsym" , DBsym )
if agrsyms = nil do
    begin
        agrsyms := table()
        s.enter("%$Agrsym",DBsym,agsyms)
    end
structure agrsym( pic f.pic )
let to.continue := true
while to.continue do
    begin
        write "Enter datafile name :"
        let datafile := read.a.line()
        let fd = open (datafile,0)
        if fd = nullfile do
            begin
                write "The file ",datafile," cannot be opened'n"
                abort
            end
        let vecx = vector 1::10,1::30 of 0
        let vecy = vector 1::10,1::30 of 0
        let vecm = vector 1::30 of nil
        let I := 0
        let count := 0
        let PIC := [ 0.0, 0.0 ]
        let PIC1 := [ 0.0, 0.0 ]
        let aline := read.a.line(fd)
        let fc := aline(1|1)
        while ~eoi(fd) or fc ~= "e" do
            begin
                while fc ~= "/" and fc ~= "e" do
                    begin
                        let le := length(aline)
                        I := I + 1
                        let p := 1
                        let q := 0
                        let fstring := ""
                        let sstring := ""
                        while aline(p|1) ~= " " do
                            begin
                                fstring := fstring ++ aline(p|1)
                                p := p + 1
                            end
                        while aline(p|1) = " " do p := p + 1
                        q := p-1
                        sstring := aline(p|( le - q))
                        vecx(count,I) := StringToInt(fstring)
                        vecy(count,I) := StringToInt(sstring)
                        PIC := if I = 1 then [ vecx(count,I) , vecy(count,I) ]
                                else PIC ^ [ vecx(count,I) , vecy(count,I) ]
                        aline := read.a.line(fd)
                        fc := aline(1|1)
                    end
                end
                if ~eoi(fd) do
                    begin
                        aline := read.a.line(fd)
                        fc := aline(1|1)
                    end
                end
                I := 0
                count := count + 1
                if count = 1 then PIC1 := PIC
                    else PIC1 := PIC1 & PIC
                s.enter(datafile, agrsyms, agrsym( PIC1) )
            end
        close(fd)

```

```
    if commit() = nil do write"This symbol has been entered successfully'n"  
      to.continue := message.proc("Store Another Symbol ?", "Yes", "No", 200, 200,  
                                  250, 140)  
    xor screen onto screen  
end
```



```

let FONTsdb:=open.database("FONTS","friend","read")
let fix13 = s.lookup( "fix13", FONTsdb)
let bold = s.lookup("fixb13",FONTsdb)
let big = s.lookup("met22",FONTsdb)
let large = s.lookup("hcl45i",FONTsdb)
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
  do {write "No utilities database - do prcdbmaker first'n"; abort}
let prcget=
begin
  structure procpak(proc(string -> pntr) xproc)
  s.lookup("prcget",procsdb) (xproc)
end
let seditor={structure procpak(proc(string,string,int,int,int,int->
  string) xproc)
  prcget("seditor") (xproc) }
let error.message={structure procpak(proc(string,int,int) xproc)
  prcget("error.message") (xproc) }
let more={structure procpak(proc(*string,int,int) xproc)
  prcget("more") (xproc) }
let form.generate={structure procpak(proc( -> pntr ) xproc)
  prcget("form.generate") (xproc)}
let form.null={structure procpak(proc( string,int,int,int,int,pntr ) xproc)
  prcget("form.null") (xproc) }
structure form.package( proc( pntr) Form.show;
  proc( ) Form.all.show;
  proc( string,int,int,int,int,bool,proc(),pntr ->
  pntr ) Form.add;
  proc( pntr ) Form.remove;
  proc( string,pntr ) Form.update;
  proc( ) Form.clear;
  proc( -> pntr ) Form.mouse;
  proc( ) Fender;
  proc( ) Form.monitor )
let set.up.choose = {structure procpak(proc(*string -> pntr) xproc)
  prcget("set.up.choose") (xproc)}
structure chooser.pack( proc( string, int, int -> string ) do.choose;
  proc( string ) add.choose;
  proc( string ) remove.choose;
  proc( int, int ) list.choose )
let table.to.text = {structure procpak(proc(pntr -> *string) xproc)
  prcget("table.to.text") (xproc)}

!let xmstart := X.dim(screen)-250
let text.write = proc(int xpos,ypos;string name;font;#pixel anyimage)
begin
  if font = "cou20" then
    copy string.to.tile(name,"cou20") onto limit anyimage at xpos,ypos
  else if font = "fixb13" then
    copy string.to.tile(name,"fixb13") onto limit anyimage at xpos,ypos
  else copy string.to.tile(name,"fix13") onto limit anyimage at xpos,ypos
end
let drawline = proc(real x1,y1,x2,y2 -> pic )
begin
  let figure := [x1,y1]^[x2,y2]
  figure
end
let Box = proc(int x,y,side)
begin
  let a.pic := drawline(x,y,x+side,y)
  a.pic := a.pic & drawline(x+side,y,x+side,y+side)
  a.pic := a.pic & drawline(x+side,y+side,x,y+side)
  a.pic := a.pic & drawline(x,y+side,x,y)
  draw( screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
end
let rec = proc(int x,y,length,width)
begin
  let a.pic := drawline(x,y,x+length,y)
  a.pic := a.pic & drawline(x+length,y,x+length,y+width)
  a.pic := a.pic & drawline(x+length,y+width,x,y+width)
  a.pic := a.pic & drawline(x,y+width,x,y)
  draw( screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
end
let Rec = proc(int x,y,thelength,width; string anything, font,position;
  bool highlight)

```

```

begin
  let chsize := 0
  let ypos := 0
  if font = "cou20" then
    begin
      chsize := 14
      ypos := 7
    end
  else {chsize:= 8;ypos := 12}
  rec(x,y,thelength,width)
  let textlength := length(anything) * chsize
  let rest := 0
  if position = "begining" then rest := 10
  else if position = "end" then rest := x + thelength - textlength
  else rest := (thelength - textlength) div 2
  let textbox := limit screen to thelength-4 by 33 at x+2,y+width-35
  if font = "cou20" then
    copy string.to.tile(anything,"cou20") onto limit textbox at rest,ypos
  else if font = "fixb13" then
    copy string.to.tile(anything,"fixb13") onto limit textbox at rest,ypos
  else copy string.to.tile(anything,"fixl3") onto limit textbox at rest,ypos
  if highlight do nor textbox onto textbox
end
let DBsym := open.database("Symbols", "mys", "read" )
if DBsym is error.record do { write "could not open database'n"; abort }
let agrsyms := s.lookup("%$Agrsym", DBsym )
let geosyms := s.lookup("%$Geosym", DBsym )
structure agrsym(pic f.pic)
structure geosym(pic a.pic)
let fvec := table.to.text(agsyms)
let svec := table.to.text(geosyms)
let upper1 := upb(fvec)
let upper2 := upb(svec)
let thetop := upper1 + upper2
let symbols := vector 1::thetop of ""
for i=1 to upper2 do symbols(i) := svec(i)
let count := 0
for i=upper2+1 to thetop do { count := count + 1; symbols(i) := fvec(count) }
let xmstart := X.dim(screen) - 160
let xbox := xmstart + 10; let ypos := 0; let xplotn := 0
let ybox := 130; let xpos := 0; let xplotx := 0; let h := 0
let PIC := [0.0, 0.0]
let a.win := limit screen to 40 by 40
Rec(xmstart,120, 140, 340, "SYMBOLS", "cou20","middle", true)
for i=1 to 7 do {
  ypos := ybox + (i-1) * 40
  for j=1 to 3 do {
    xpos := xbox + (j-1) * 40
    let a.win := limit screen to 40 by 40 at xpos, ypos
    Box( xpos, ypos, 40 )
    h := (i-1) * 3 + j
    case true of
      h <= upper2 : { PIC := s.lookup(symbols(h), geosyms)(a.pic);
                     xplotn := 0; xplotx := 40 }
      default      : { PIC := s.lookup(symbols(h), agrsyms)(f.pic);
                     xplotn := -20; xplotx := 20 }
    draw(a.win, PIC, xplotn, xplotx, 0, 40 ) } }
let totalwin = limit screen to 150 by 350 at xmstart-5, 115
let symimage = image 150 by 350 of off
copy totalwin onto symimage
let DBvar := open.database("global","variables","write")
structure gimage(#pixel menuimage)
let theimage := s.lookup("Images", DBvar)
s.enter("Symbols Menu", theimage, gimage(symimage) )
s.enter("Images", DBvar, theimage)
if commit() ~= nil do write "Image is not stored"

```

```

let FONTsdb:=open.database("FONTS","friend","read")
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
  do {write "No utilities database - do procdmaker first'n"; abort}
let prcget=
begin
  structure procpak(proc(string -> pntr) xproc)
  s.lookup("prcget",procsdb)(xproc)
end
let seditor={structure procpak(proc(string,string,int,int,int,int->
  string) xproc)
  prcget("seditor")(xproc) }
let icon = proc(int x, y; string atext -> int)
begin
  let text.length := length(atext)
  let box.length := text.length * 8 + 20
  let xcoors = @1 of int [ x+5, x, x, x+5, x+box.length-5, x+box.length,
    x+box.length, x+box.length-5, x+5]
  let ycoors = @1 of int [ y, y+5, y+25, y+30, y+30, y+25, y+5, y, y]
  let xil = @1 of int [ x+7, x+2, x+2, x+7, x+box.length-7, x+box.length-2,
    x+box.length-2, x+box.length-7, x+7]
  let yil = @1 of int [ y+2, y+7, y+23, y+28, y+28, y+23, y+7, y+2, y+2 ]
  let PIC := [0,0]
  let PIC1 := [0,0]
  for i=1 to 9 do
    PIC := if i=1 then [ xcoors(i) , ycoors(i) ]
      else PIC ^ [ xcoors(i) , ycoors(i) ]
  for i=1 to 9 do
    PIC1 := if i=1 then [ xil(i) , yil(i) ]
      else PIC1 ^ [ xil(i) , yil(i) ]
  PIC := PIC & PIC1
  let icon.image = limit screen to box.length by 30 at x,y
  draw(screen, PIC,0,X.dim(screen),0,Y.dim(screen))
  copy string.to.tile( atext, "fixb13" ) onto limit icon.image at 8,10
  box.length
end
let text.write = proc(int xpos,ypos;string name,font;#pixel anyimage)
begin
  let Font := ""
  case true of
    font = "1" : Font := "fix09"
    font = "2" : Font := "ngr13"
    font = "3" : Font := "fix13"
    font = "4" : Font := "fixb13"
    font = "5" : Font := "gac16n"
    font = "6" : Font := "gacha16"
    font = "7" : Font := "met22"
    font = "8" : Font := "ngi20"
    font = "9" : Font := "non22"
    font = "10" : Font := "olde25"
    font = "11" : Font := "hci45i"
    default : Font := "cou20"
  copy string.to.tile(name,Font) onto limit anyimage at xpos,ypos
end
let drawline = proc(real x1,y1,x2,y2 -> pic )
begin
  let figure := [x1,y1]^[x2,y2]
  figure
end
let Box = proc(int x,y,side)
begin
  let a.pic := drawline(x,y,x+side,y)
  a.pic := a.pic & drawline(x+side,y,x+side,y+side)
  a.pic := a.pic & drawline(x+side,y+side,x,y+side)
  a.pic := a.pic & drawline(x,y+side,x,y)
  draw( screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
end
let rec = proc(int x,y,length,width)
begin
  let a.pic := drawline(x,y,x+length,y)
  a.pic := a.pic & drawline(x+length,y,x+length,y+width)
  a.pic := a.pic & drawline(x+length,y+width,x,y+width)
  a.pic := a.pic & drawline(x,y+width,x,y)
  draw( screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
end

```

```

let xstart := 200; let ystart := 250
let twinx := 500; let twiny := 400
let Fwin = limit screen to 45 by 303 at xstart+369, ystart+20
let fvec1 = @1 of string [ "2", "6", "5", "4", "3", "2", "1" ]
let fvec2 = @1 of string [ "11", "10", "7", "14", "9", "8", "2" ]
let fwin = limit screen to 47 by 316 at 19, 19
let fim1 = image 45 by 303 of off
let fim2 = image 45 by 303 of off
for i=1 to 7 do
  if i > 1 then {
    Box(20, 20+(i-1)*43, 43)
    text.write(25, 23+(i-1)*43, "A", fvec1(i), screen) }
  else { Box(20, 20+(i-1)*43, 43)
    text.write(25, 23, "More", "4", screen) }
    copy fwin onto fim1
    xor fwin onto fwin
  for i=1 to 7 do
    if i < 7 then {
      Box(20, 20+(i-1)*43, 43)
      text.write(25, 23+(i-1)*43, "A", fvec2(i), screen) }
    else { Box(20, 20+(i-1)*43, 43)
      text.write(25, 283, "Back", "4", screen) }
      copy fwin onto fim2
      xor fwin onto fwin
      let twin := limit screen to twinx by twiny at xstart, ystart
      let text.win = limit screen to 350 by 80 at 47+xstart, 147+ystart
    rec(xstart+370, ystart+323, 44, 30)
    rec(xstart+372, ystart+325, 40, 26)
    text.write(375, 326, "Font", "4", twin )
    let Fbox = limit screen to 47 by 33 at xstart+368, ystart+321
    let Fim := image 47 by 33 of off
    copy Fbox onto Fim
    xor Fbox onto Fbox
    let DBvar := open.database("global", "variables", "write")
    structure gimage(#pixel menuimage)
    let theimage := s.lookup("Images", DBvar)
    s.enter("Font Menu1", theimage, gimage(fim1) )
    s.enter("Font Menu2", theimage, gimage(fim2) )
    s.enter("Fone Title", theimage, gimage(Fim) )
    s.enter("Images", DBvar, theimage)
    if commit() ~= nil do write "Image is not stored"

```

```

let DBvar := open.database("global","variables","write")
if DBvar is error.record do { write "Error creating Database'n" }
structure gimage(#pixel menuimage)
let theimage := s.lookup("Images", DBvar)
if theimage = nil do
    theimage := table()
let Sht := X.dim(screen) - 540
let sorting = proc(*real k, r; int upper)
begin
    let swap = proc(*real v; cint i,j)
    begin
        let temp = v(i)
        v(i) := v(j)
        v(j) := temp
    end
    let sort = proc(*real x, y; int ubd -> int)
    begin
        let v := 0; let u := 0; let count := 0
        for i=1 to ubd -1 do
            begin
                v := i; u := i+1
                if x(v) > x(u) do
                    begin
                        swap(x,v,u)
                        swap(y,v,u)
                        count := count + 1
                    end
                end
            end
        end
        count
    end
    let howmany := sort(k,r,upper)
    while howmany ~= 0 do
        howmany := sort(k,r,upper)
    end
let linepara = proc(real x1, y1, x2, y2 -> *real)
begin
    let parameter = vector 1::2 of 0.0
    if x2-x1 = 0 then {parameter(1) := 0.0; parameter(2) := x2}
    else {
        parameter(1) := (y2-y1)/(x2-x1)
        parameter(2) := (y1*x2 - y2*x1)/(x2-x1)}
    parameter
end
let lineint = proc(real a,b,c,d -> *real)
begin
    let parameter = vector 1::2 of 0.0
    if a = 0 and d = 0 then {
        parameter(1) := c; parameter(2) := b}
    else if b=0 and c=0 then {
        parameter(1) := a; parameter(2) := d}
    else if c = 0 and b ~= 0 then {
        parameter(1) := -99999
        parameter(2) := -99999}
    else {parameter(1) := (d-b)/(a-c)
        parameter(2) := (a*d - b*c)/(a-c)}
    parameter
end
let perpenline = proc(real x,y,m -> *real)
begin
    let parameter = vector 1::2 of 0.0
    if m=0 then
        begin
            parameter(1) := x
            parameter(2) := 0
        end
    else {
        parameter(1) := -1/m
        parameter(2) := y - parameter(1)*x}
    parameter
end
let Lparaline = proc(real a,b,w; int K -> *real)
begin
    let const = vector 1::2 of 0.0
    const(1) := a
    const(2) := b + K * w * sqrt(1 + a*a)
end

```

```

const
end
let Rparaline = proc(real a,b,w; int K -> *real)
begin
  let const = vector 1::2 of 0.0
  const(1) := a
  const(2) := b - K * w * sqrt(1 + a*a)
  const
end
let drawline = proc(real x1,y1,x2,y2 -> pic )
begin
  let figure := [x1,y1]^[x2,y2]
  figure
end
let dashing = proc(real x1,y1,x2,y2,dash,gap -> pic )
begin
  let alpha := 0.0
  let a.pic := [ 0.0, 0.0 ]
  let period := dash + gap
  let dy := y2 - y1
  let dx := x2 - x1
  let lengthoffline := sqrt( dx * dx + dy * dy )
  if dx = 0 then
    if dy > 0 then alpha := pi/2
    else alpha := pi
  else
    begin
      let k := 1; let c := 1
      let NoOfPeriods := 0; let thegap := 0.0; let theperiod := 0.0
      alpha := atan(rabs(dy/dx))
      if dx < 0 do k := -k
      if dy < 0 do c := -c
      if lengthoffline > period then
        begin
          NoOfPeriods := truncate( lengthoffline / period )
          let rest := lengthoffline - NoOfPeriods * period
          thegap := rest / NoOfPeriods + gap
          theperiod := thegap + dash
          let X1 := vector 1::(NoOfPeriods+1) of 0.0
          let Y1 := vector 1::(NoOfPeriods+1) of 0.0
          let X2 := vector 1::(NoOfPeriods+1) of 0.0
          let Y2 := vector 1::(NoOfPeriods+1) of 0.0
          let thecos := cos(alpha)
          let thesin := sin(alpha)
          let dxgap := thecos * thegap
          let dxdash := thecos * dash
          let dxhdash := 0.5 * dxdash
          let dygap := thesin * thegap
          let dydash := thesin * dash
          let dyhdash := 0.5 * dydash
          X1(1) := x1
          Y1(1) := y1
          X2(1) := x1 + dxhdash * k
          Y2(1) := y1 + dyhdash * c
          a.pic := drawline( X1(1), Y1(1), X2(1), Y2(1) )
          for i = 2 to NoOfPeriods do
            begin
              X1(i) := X2(i-1) + dxgap * k
              Y1(i) := Y2(i-1) + dygap * c
              X2(i) := X1(i) + dxdash * k
              Y2(i) := Y1(i) + dydash * c
              a.pic := a.pic & drawline( X1(i), Y1(i), X2(i), Y2(i) )
            end
          end
          X1(NoOfPeriods+1) := X2(NoOfPeriods) + dxgap * k
          Y1(NoOfPeriods+1) := Y2(NoOfPeriods) + dygap * c
          X2(NoOfPeriods+1) := x2
          Y2(NoOfPeriods+1) := y2
          a.pic := a.pic & drawline( X1(NoOfPeriods+1),
                                   Y1(NoOfPeriods+1),
                                   X2(NoOfPeriods+1),
                                   Y2(NoOfPeriods+1) )
        end
      else a.pic := drawline(x1,y1,x2,y2)
    end
  end
a.pic
end

```

```

end
let Box = proc(int x,y,side)
begin
  let a.pic := drawline(x,y,x+side,y)
  a.pic := a.pic & drawline(x+side,y,x+side,y+side)
  a.pic := a.pic & drawline(x+side,y+side,x,y+side)
  a.pic := a.pic & drawline(x,y+side,x,y)
  draw( screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
end
let rec = proc(int x,y,length,width)
begin
  let a.pic := drawline(x,y,x+length,y)
  a.pic := a.pic & drawline(x+length,y,x+length,y+width)
  a.pic := a.pic & drawline(x+length,y+width,x,y+width)
  a.pic := a.pic & drawline(x,y+width,x,y)
  draw( screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
end
let Rec = proc(int x,y,thelength,width; string anything, font,position;
               bool highlight)
begin
  let chsize := 0
  let ypos := 0
  if font = "cou20" then
    begin
      chsize := 14
      ypos := 7
    end
  else {chsize:= 8;ypos := 12}
  rec(x,y,thelength,width)
  let textlength := length(anything) * chsize
  let rest := 0
  if position = "begining" then rest := 10
  else if position = "end" then rest := x + thelength - textlength
  else rest := (thelength - textlength) div 2
  let textbox := limit screen to thelength-4 by 33 at x+2,y+width-35
  if font = "cou20" then
    copy string.to.tile(anything,"cou20") onto limit textbox at rest,ypos
  else if font = "fixbl3" then
    copy string.to.tile(anything,"fixbl3") onto limit textbox at rest,ypos
  else copy string.to.tile(anything,"fixl3") onto limit textbox at rest,ypos
  if highlight do nor textbox onto textbox
end
let hatchpoly = proc(*real Xs,Ys; int size, enclaves; real pitch, angle, dash,
                    gap; *string names -> pic )
begin
  if angle = 0 do angle := 180
  angle := angle * pi /180
  let PIC := [ 0.0, 0.0 ]
  let xx := 0.0
  let yy := 0.0
  for i=1 to size do !transformation
    begin
      xx := Xs(i)
      yy := Ys(i)
      Xs(i) := xx * cos(angle) + yy * sin(angle)
      Ys(i) := -xx * sin(angle) + yy * cos(angle)
    end
  let Ymin := 900000.0; let Ymax := 0.0
  for i=1 to size do
    begin
      if Ys(i) > Ymax do Ymax := Ys(i)
      if Ys(i) < Ymin do Ymin := Ys(i)
    end
  let xdatarray := vector 1::10,1::50 of 0.0
  let ydatarray := vector 1::10,1::50 of 0.0
  let thesize := vector 1::10 of 0
  if enclaves ~= 0 do
    for count = 1 to enclaves do
      begin
        let filename := names(count)
        let df = open(filename,0)
        if df = nullfile do
          begin
            write "The file "," coords ", "cannot be opened'n"
            abort
          end
        end
      end
    end
  end
end

```

```

        end
        let num := readr(df)
        thesize(count) := truncate(num)
        for i=1 to thesize(count) do
            begin
                xdataarray(count)(i) := readr(df)
                ydataarray(count)(i) := readr(df)
            end
        end
        for i=1 to thesize(count) do
            begin
                xx := xdataarray(count)(i)
                yy := ydataarray(count)(i)
                xdataarray(count)(i) := xx * cos(angle) + yy * sin(angle)
                ydataarray(count)(i) := -xx * sin(angle) + yy * cos(angle)
            end
        end
        end
        let tempara := vector 1::2 of 0.0
        let coords := vector 1::2 of 0.0
        let ymin := 0.0; let ymax := 0.0
        let span := Ymax - Ymin
        let theXs := vector 1::50 of 0.0
        let theYs := vector 1::50 of 0.0
        let NoOfLines := truncate(span / pitch)
        let Y := vector 1::NoOfLines of 0.0
        let starty := 0.0
        if pitch > 1 do
            begin
                let rest := span - NoOfLines * pitch
                pitch := pitch + rest/NoOfLines
                starty := Ymin - pitch/2
            end
        end
        if pitch = 1 do starty := Ymin
        let NoOfInt := 0
        for i=1 to NoOfLines do
            begin
                Y(i) := starty + pitch * i
                for j=1 to size-1 do
                    begin
                        if Ys(j) < Ys(j+1) then
                            begin
                                ymin := Ys(j)
                                ymax := Ys(j+1)
                            end
                        else
                            begin
                                ymin := Ys(j+1)
                                ymax := Ys(j)
                            end
                        end
                        tempara := linepara( Xs(j), Ys(j), Xs(j+1), Ys(j+1) )
                        coords := lineint( 0.0, Y(i), tempara(1), tempara(2) )
                        if coords(2) > ymin and coords(2) < ymax do
                            begin
                                NoOfInt := NoOfInt + 1
                                theXs(NoOfInt) := coords(1)
                                theYs(NoOfInt) := coords(2)
                            end
                        end
                        if enclaves ~= 0 do
                            for count = 1 to enclaves do
                                for l=1 to thesize(count)-1 do
                                    begin
                                        if ydataarray(count)(l) < ydataarray(count)(l+1) then
                                            begin
                                                ymin := ydataarray(count)(l)
                                                ymax := ydataarray(count)(l+1)
                                            end
                                        else
                                            begin
                                                ymin := ydataarray(count)(l+1)
                                                ymax := ydataarray(count)(l)
                                            end
                                        end
                                        tempara := linepara( xdataarray(count)(l),
                                                                ydataarray(count)(l),
                                                                xdataarray(count)(l+1),
                                                                ydataarray(count)(l+1) )
                                        coords := lineint( 0.0, Y(i), tempara(1), tempara(2) )

```



```

        if coords(2) > ymin and coords(2) < ymax do
            begin
                NoOfInt := NoOfInt + 1
                theXs(NoOfInt) := coords(1)
                theYs(NoOfInt) := coords(2)
            end
        end
    end
    sorting(theXs,theYs,NoOfInt)
    for i=1 to NoOfInt do
        begin
            xx := theXs(i)
            yy := theYs(i)
            theXs(i) := xx * cos(angle) - yy * sin(angle)
            theYs(i) := xx * sin(angle) + yy * cos(angle)
        end
    if dash = -1 then
        for l=1 to NoOfInt by 2 do
            PIC := PIC & drawline(theXs(l),theYs(l),theXs(l+1),theYs(l+1) )
        else
            for l=1 to NoOfInt by 2 do
                PIC := PIC & dashing(theXs(l),theYs(l),theXs(l+1),
                    theYs(l+1), dash, gap)
            !draw(win, PIC, 0, X.dim(screen), 0, Y.dim(screen) )
            NoOfInt := 0
        end
    end
    PIC
end
let doblseg = proc(real X1, Y1, X2, Y2, width,type,dash,gap -> pic )
begin
    let a.pic := [ 0.0, 0.0 ]
    let Para := vector 1::2 of 0.0
    let Para1 := vector 1::2 of 0.0
    let Para10 := vector 1::2 of 0.0
    let Para2 := vector 1::2 of 0.0
    let Para3 := vector 1::2 of 0.0
    let Para30 := vector 1::2 of 0.0
    let Para4 := vector 1::2 of 0.0
    let Para5 := vector 1::2 of 0.0
    let Para50 := vector 1::2 of 0.0
    let k := 1
    Para := linepara(X1,Y1,X2,Y2)
    Para1 := perpenline(X1,Y1,Para(1))
    Para10 := perpenline(X2,Y2,Para(1))
    if X2 < X1 do k := -1 * k
    Para2 := Rparaline(Para(1),Para(2),width/2,k)
    Para4 := Lparaline(Para(1),Para(2),width/2,k)
    Para3 := lineint(Para1(1),Para1(2),Para2(1),Para2(2))
    Para30 := lineint(Para10(1),Para10(2),Para2(1),Para2(2))
    Para5 := lineint(Para1(1),Para1(2),Para4(1),Para4(2))
    Para50 := lineint(Para10(1),Para10(2),Para4(1),Para4(2))
    let xintersecR1 := Para3(1)
    let yintersecR1 := Para3(2)
    let xintersecL1 := Para5(1)
    let yintersecL1 := Para5(2)
    let xintersecR2 := Para30(1)
    let yintersecR2 := Para30(2)
    let xintersecL2 := Para50(1)
    let yintersecL2 := Para50(2)
    if type = -1 then {
        a.pic := drawline(xintersecR1,yintersecR1,xintersecR2,yintersecR2)
        a.pic := a.pic & drawline(xintersecL1,yintersecL1,xintersecL2,yintersecL2)}
    else if type = -2 then {
        a.pic := dashing(xintersecR1,yintersecR1,xintersecR2,yintersecR2,
            dash,gap)
        a.pic := a.pic & dashing(xintersecL1,yintersecL1,xintersecL2,
            yintersecL2,dash,gap)}
    else if type = -3 then {
        a.pic := dashing(xintersecR1,yintersecR1,xintersecR2,yintersecR2,
            dash,gap)
        a.pic := a.pic & drawline(xintersecL1,yintersecL1,xintersecL2,
            yintersecL2)}
    else if type = -4 do {
        a.pic := drawline(xintersecR1,yintersecR1,xintersecR2,yintersecR2)
        a.pic := a.pic & dashing(xintersecL1,yintersecL1,xintersecL2,

```

yintersecL2,dash,gap) }

```

a.pic
end
let doblines = proc(*real Xs, Ys; real width,type; int themax,dash,gap  -> pic )
begin
  let xintersecR := vector 1::100 of 0.0
  let yintersecR := vector 1::100 of 0.0
  let xintersecL := vector 1::100 of 0.0
  let yintersecL := vector 1::100 of 0.0
  let a.pic1 := [ 0.0, 0.0 ]
  let a.picr := [ 0.0, 0.0 ]
  let a.pic := [ 0.0, 0.0 ]
  let Para := vector 1::2 of 0.0
  let Para1 := vector 1::2 of 0.0
  let Para2 := vector 1::2 of 0.0
  let Para3 := vector 1::2 of 0.0
  let Para4 := vector 1::2 of 0.0
  let Para5 := vector 1::2 of 0.0
  let Para6 := vector 1::2 of 0.0
  let Para7 := vector 1::2 of 0.0
  let k := 1
  for i=1 to themax do
    begin
      if i < themax then
        begin
          k:= 1
          Para := linepara(Xs(i),Ys(i),Xs(i+1),Ys(i+1))
          Para1 := perpenline(Xs(i),Ys(i),Para(1))
          if Xs(i+1) < Xs(i) do k := -1 * k
          Para2 := Rparaline(Para(1),Para(2),width/2,k)
          Para4 := Lparaline(Para(1),Para(2),width/2,k)
          if i=1 then
            begin
              Para3 := lineint(Para1(1),Para1(2),Para2(1),Para2(2))
              Para5 := lineint(Para1(1),Para1(2),Para4(1),Para4(2))
              xintersecR(i) := Para3(1)
              yintersecR(i) := Para3(2)
              xintersecL(i) := Para5(1)
              yintersecL(i) := Para5(2)
              Para6 := Para2
              Para7 := Para4
            end
          else
            begin
              Para3 := lineint(Para2(1),Para2(2),Para6(1),Para6(2))
              Para5 := lineint(Para4(1),Para4(2),Para7(1),Para7(2))
              Para6 := Para2
              Para7 := Para4
              xintersecR(i) := Para3(1)
              yintersecR(i) := Para3(2)
              xintersecL(i) := Para5(1)
              yintersecL(i) := Para5(2)
            end
          end
        end
      else
        if i=themax do
          begin
            Para1 := perpenline(Xs(i),Ys(i),Para6(1))
            Para3 := lineint(Para1(1),Para1(2),Para6(1),Para6(2))
            Para5 := lineint(Para1(1),Para1(2),Para7(1),Para7(2))
            xintersecR(i) := Para3(1)
            yintersecR(i) := Para3(2)
            xintersecL(i) := Para5(1)
            yintersecL(i) := Para5(2)
          end
        end
      end
    end
  for i=1 to themax-1 do
    begin
      if type = -1 then {
        a.picr := drawline(xintersecR(i),yintersecR(i),xintersecR(i+1),
          yintersecR(i+1))
        a.picl := drawline(xintersecL(i),yintersecL(i),xintersecL(i+1),
          yintersecL(i+1))
        a.pic := if i=1 then a.picr & a.picl
          else a.pic & a.picr & a.picl}
      }
    end
  end
end

```

```

else if type = -2 then {
    a.picr := dashing(xintersecR(i),yintersecR(i),
                     xintersecR(i+1),yintersecR(i+1),dash,gap)
    a.picl := dashing(xintersecL(i),yintersecL(i),
                     xintersecL(i+1),yintersecL(i+1),dash,gap)
    a.pic := if i=1 then a.picr & a.picl
             else a.pic & a.picr & a.picl}
else if type = -3 then {
    a.picr := dashing(xintersecR(i),yintersecR(i),
                     xintersecR(i+1),yintersecR(i+1),dash,gap)
    a.picl := drawline(xintersecL(i),yintersecL(i),
                     xintersecL(i+1),yintersecL(i+1))
    a.pic := if i=1 then a.picr & a.picl
             else a.pic & a.picr & a.picl}
else if type = -4 do {
    a.picr := drawline(xintersecR(i),yintersecR(i),
                     xintersecR(i+1),yintersecR(i+1))
    a.picl := dashing(xintersecL(i),yintersecL(i),
                     xintersecL(i+1),yintersecL(i+1),dash,gap)
    a.pic := if i=1 then a.picr & a.picl
             else a.pic & a.picr & a.picl}
end
a.pic
end
let ddline = proc(*real Xs, Ys; real width; int themax,dash,gap -> pic )
begin
    let k := width
    let a.pic := [ 0.0, 0.0 ]
    let a.picl := [ 0.0, 0.0 ]
    for i=1 to themax-1 do
        begin
            while k ~= 0 do
                begin
                    a.picl := if k= width then dblseg(Xs(i),Ys(i),Xs(i+1),
                                                         Ys(i+1),k,-2,dash,gap)
                             else a.picl & dblseg(Xs(i),Ys(i),Xs(i+1),
                                                         Ys(i+1),k,-2,dash,gap)
                    k := k - 0.5
                end
                a.pic := if i=1 then a.picl
                         else a.pic & a.picl
            k := width
        end
    end
a.pic
end
let thkline = proc(*real Xarray, Yarray; int breadth, thetop -> pic )
begin
    let a.picl := [ 0.0, 0.0 ]
    let a.pic2 := [ 0.0, 0.0 ]
    let a.pic := [ 0.0, 0.0 ]
    for j=1 to thetop-1 do
        a.picl := if j=1 then drawline(Xarray(j),Yarray(j),Xarray(j+1),
                                         Yarray(j+1))
                 else a.pic & drawline(Xarray(j),Yarray(j),Xarray(j+1),
                                         Yarray(j+1))
    let half := breadth / 2
    for i=1 to thetop-1 do {
        let k := half
        while k > 0 do
            {a.pic2 := if k = half then dblseg(Xarray(i),Yarray(i),
                                                Xarray(i+1),Yarray(i+1),k, -1,-1,-1)
              else a.pic2 & dblseg(Xarray(i),Yarray(i),
                                    Xarray(i+1),Yarray(i+1),k, -1,-1,-1)
            k := k - 0.25}
            a.pic := if i=1 then a.picl & a.pic2
                     else a.pic & a.picl & a.pic2 }
        a.pic
    end
let railine = proc(*real Xarray, Yarray; int breadth, thetop -> pic )
begin
    let a.pic := [ 0.0, 0.0 ]
    let half := breadth div 2
    for i= 1 to half do
        a.pic := ddline(Xarray,Yarray,i, thetop,6,6)
    a.pic := a.pic & doline(Xarray,Yarray,breadth, -1, thetop, -1, -1)
end

```

```

a.pic
end
let bordering = proc(real x1,y1,x2,y2,dash,gap -> pic )
begin
  let alpha := 0.0
  let a.pic := [ 0.0, 0.0 ]
  let period := dash + gap
  let dy := y2 - y1
  let dx := x2 - x1
  let lengthofline := sqrt( dx * dx + dy * dy )
  if dx = 0 then
    if dy > 0 then alpha := pi/2
    else alpha := pi
  else
    begin
      let k := 1; let c := 1
      let NoOfPeriods := 0; let thegap := 0.0; let theperiod := 0.0
      alpha := atan(rabs(dy/dx))
      if dx < 0 do k := -k
      if dy < 0 do c := -c
      if lengthofline > period then
        begin
          let Periods := truncate( lengthofline / period )
          NoOfPeriods := Periods - truncate(Periods/3)
          let rest := lengthofline - NoOfPeriods * period
          thegap := rest / NoOfPeriods + gap
          theperiod := thegap + dash
          let X1 := vector 1::(NoOfPeriods+2) of 0.0
          let Y1 := vector 1::(NoOfPeriods+2) of 0.0
          let X2 := vector 1::(NoOfPeriods+2) of 0.0
          let Y2 := vector 1::(NoOfPeriods+2) of 0.0
          let thecos := cos(alpha)
          let thesin := sin(alpha)
          let dxgap := thecos * thegap
          let dxdash := thecos * dash
          let dxdashp := thecos
          let dxhdash := 0.5 * dxdash
          let dygap := thesin * thegap
          let dydash := thesin * dash
          let dydashp := thesin
          let dyhdash := 0.5 * dydash
          X1(1) := x1
          Y1(1) := y1
          X2(1) := x1 + dxhdash * k
          Y2(1) := y1 + dyhdash * c
          a.pic := drawline( X1(1), Y1(1), X2(1), Y2(1) )
          for i = 2 to NoOfPeriods do
            begin
              if i rem 2 = 0 then {
                X1(i) := X2(i-1) + dxgap * k
                Y1(i) := Y2(i-1) + dygap * c
                X2(i) := X1(i) + dxdashp * k
                Y2(i) := Y1(i) + dydashp * c
                a.pic := a.pic & drawline( X1(i), Y1(i), X2(i), Y2(i) ) }
              else {
                X1(i) := X2(i-1) + dxgap * k
                Y1(i) := Y2(i-1) + dygap * c
                X2(i) := X1(i) + dxdash * k
                Y2(i) := Y1(i) + dydash * c
                a.pic := a.pic & drawline( X1(i), Y1(i), X2(i), Y2(i) ) }
            end
          X1(NoOfPeriods+1) := X2(NoOfPeriods) + dxgap * k
          Y1(NoOfPeriods+1) := Y2(NoOfPeriods) + dygap * c
          X2(NoOfPeriods+1) := X1(NoOfPeriods+1) + dxdashp * k
          Y2(NoOfPeriods+1) := Y1(NoOfPeriods+1) + dydashp * c
          a.pic := a.pic & drawline( X1(NoOfPeriods+1),
            Y1(NoOfPeriods+1), X2(NoOfPeriods+1), Y2(NoOfPeriods+1) )
          X1(NoOfPeriods+2) := X2(NoOfPeriods+1) + dxgap * k
          Y1(NoOfPeriods+2) := Y2(NoOfPeriods+1) + dygap * c
          X2(NoOfPeriods+2) := x2
          Y2(NoOfPeriods+2) := y2
          a.pic := a.pic & drawline( X1(NoOfPeriods+2),
            Y1(NoOfPeriods+2), X2(NoOfPeriods+2), Y2(NoOfPeriods+2) )
        end
      end
    else a.pic := drawline(x1,y1,x2,y2)
  end
end

```

```

        end
        a.pic
    end
    let typemenu = proc()
    begin
        let typeimage = image 110 by 355 of off
        let themenu = limit screen to 100 by 355 at X.dim(screen)-110,120
        let xpos := X.dim(screen)-95
        let xloc := xpos + 10
        let xend := X.dim(screen)-25
        Rec(X.dim(screen)-105,130,95,280,"TYPES","cou20", "middle", true)
        let a.pic := drawline(xloc,345,xend,345)
        a.pic := a.pic & dashing(xloc,315,xend,315,10,5)
        a.pic := a.pic & dashing(xloc,285,xend,285,4,4)
        a.pic := a.pic & dashing(xloc,255,xend,255,12,2)
        a.pic := a.pic & dashing(xloc,225,xend,225,2,4)
        a.pic := a.pic & dashing(xloc,195,xend,195,1,2)
        a.pic := a.pic & dashing(xloc,165,xend,165,1,4)
        draw(screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
        copy themenu onto typeimage
        s.enter("Hlinetype", theimage, gimage(typeimage) )
        s.enter("Images", DBvar, theimage)
        if commit() ~= nil do write "Image is not stored"
    end
    let angmenu = proc()
    begin
        let angimage = image 155 by 270 of off
        let themenu = limit screen to 155 by 275 at 385+Sht,115
        Rec (390+Sht,340,140,38,"SLOPE ", "cou20", "middle",true)
        Box(390+Sht,290,40)
        Box(390+Sht,170,40)
        Box (490+Sht,290,40)
        Box (490+Sht,170,40)
        rec (420+Sht,220,80,60)
        Rec (420+Sht,120,80,30,"done","fix13", "middle",false)
        let picbox = limit screen to 116 by 56 at 392+Sht,332
        let picbox1 = limit screen to 40 by 40 at 390+Sht,290
        let picbox2 = limit screen to 40 by 40 at 390+Sht,170
        let picbox3 = limit screen to 40 by 40 at 490+Sht,290
        let picbox4 = limit screen to 40 by 40 at 490+Sht,170
        let picbox5 = limit screen to 74 by 54 at 423+Sht,223
        let picbox6 = limit screen to 80 by 30 at 420+Sht,120
        let thesize := 8
        let vecX = @1 of real [30,30,10,10,0,10,10,30]
        let vecY = @1 of real [10,22,22,32,16,0,10,10]
        let PIC := [ 0.0, 0.0 ]
        let PIC1 := [ 0.0, 0.0 ]
        for i=1 to thesize do
            begin
                PIC := if i = 1 then [ vecX(i) , vecY(i) ]
                        else PIC ^ [ vecX(i) , vecY(i) ]
                PIC1 := if i = 1 then PIC
                        else PIC1 & PIC
            end
            let PIC2 := rotate PIC1 by -90
            let PIC3 := rotate PIC1 by 90
            let PIC4 := scale PIC2 by 0.5,0.5
            let PIC5 := scale PIC3 by 0.5,0.5
            draw (picbox2, PIC2, -35,5,-5,35 )
            draw (picbox1, PIC3, -5,35,-35,5 )
            draw (picbox4, PIC4, -27,13,-13,27 )
            draw (picbox3, PIC5, -13,27,-27,13 )
            copy themenu onto angimage
            s.enter("Hangle", theimage, gimage(angimage) )
            s.enter("Images", DBvar, theimage)
            if commit() ~= nil do write "Image is not stored"
        end
    end
    let scalmenu = proc()
    begin
        let scalimage = image 150 by 320 of off
        let themenu = limit screen to 150 by 320 at 385+Sht, 95
        let picbox1 = limit screen to 40 by 40 at 390+Sht,200
        let picbox3 = limit screen to 40 by 40 at 490+Sht,200
        let picbox4 = limit screen to 40 by 40 at 440+Sht,150
        let picbox5 = limit screen to 40 by 40 at 440+Sht,250
    end

```

```

let picbox6 = limit screen to 80 by 30 at 420+Sht,100
Box (390+Sht,200,40)
Box (440+Sht,200,40)
Box (490+Sht,200,40)
Box (440+Sht,150,40)
Box (440+Sht,250,40)
Rec (420+Sht,100,80,30,"done","fix13", "middle",false)
Box (430+Sht,350,60)
let thesize := 8
let vecX = @1 of real [30,30,10,10,0,10,10,30]
let vecY = @1 of real [10,22,22,32,16,0,10,10]
let PIC := [ 0.0, 0.0 ]
let PIC1 := [ 0.0, 0.0 ]
for i=1 to thesize do
  begin
    PIC := if i = 1 then [ vecX(i) , vecY(i) ]
              else PIC ^ [ vecX(i) , vecY(i) ]
    PIC1 := if i = 1 then PIC
              else PIC1 & PIC
  end
let PIC2 := rotate PIC1 by -90
let PIC3 := rotate PIC2 by -90
let PIC4 := rotate PIC3 by -90
draw (picbox1, PIC1, -5,35,-5,35 )
draw (picbox4, PIC2, -35,5,-5,35 )
draw (picbox3, PIC3, -35,5,-35,5 )
draw (picbox5, PIC4, -5,35,-35,5 )
copy themenu onto scalimage
s.enter("Scaling", theimage, gimage(scalimage) )
s.enter("Images", DBvar, theimage)
if commit() ~= nil do write "Image is not stored"
end
let hatchspace = proc()
begin
  let themenu = limit screen to 100 by 350 at X.dim(screen)-105,120
  let spaceimage = image 105 by 355 of off
  Rec(X.dim(screen)-105,120,95,340,"space","fixb13", "middle",true)
  let xpos := X.dim(screen)-95
  let names = @1 of string[""]
  let vecx := vector 1::5 of 0.0
  let vecy := vector 1::5 of 0.0
  let xloc := xpos + 10
  let xend := X.dim(screen)-25
  let cons := 0.0
  let PIC := [ 0.0, 0.0 ]
  let APIC := [ 0.0, 0.0 ]
  for i=140 to 380 by 40 do
    begin
      cons := cons + 1
      vecy(1) := i; vecx(1) := xpos
      vecy(2) := i+30; vecx(2) := xpos
      vecy(3) := i+30; vecx(3) := xpos+80
      vecy(4) := i; vecx(4) := xpos+80
      vecy(5) := i; vecx(5) := xpos
      for i=1 to 4 do
        APIC := if i=1 then drawline( vecx(i), vecy(i), vecx(i+1),
                                      vecy(i+1) )
                  else APIC & drawline( vecx(i), vecy(i), vecx(i+1),
                                      vecy(i+1) )
        draw(screen, APIC, 0, X.dim(screen), 0, Y.dim(screen) )
        PIC := hatchpoly( vecx, vecy, 5, 0, cons, 90, -1, -1, names)
        draw(screen, PIC, 0, X.dim(screen), 0, Y.dim(screen) )
      end
      copy themenu onto spaceimage
      s.enter("Hspacing", theimage, gimage(spaceimage) )
      s.enter("Images", DBvar, theimage)
      if commit() ~= nil do write "Image is not stored"
    end
  end
let default.menu = proc()
begin
  let PIC := [ 0.0, 0.0 ]
  let xstart := X.dim(screen)-190
  let menuwin = limit screen to 195 by 400 at xstart-5,115
  let msgeSave = image 195 by 400 of off
  Rec(xstart,160,184,250,"Default","cou20", "middle",false)

```

```

Rec(xstart+ 9, 180, 85,40,"Spacing","fix13", "middle",false)
let abox1 = limit screen to 85 by 40 at xstart+ 9, 180
Rec(xstart+ 9, 240, 85,40,"Angle","fix13", "middle",false)
let abox2 = limit screen to 85 by 40 at xstart+ 9, 240
Rec(xstart+ 9, 300, 85,40,"Style","fix13", "middle",false)
let abox3 = limit screen to 85 by 40 at xstart+ 9, 300
Rec(xstart,120,92,40,"Accepted","fix13", "middle",false)
let picbox1 = limit screen to 88 by 36 at xstart+ 2, 122
Rec(xstart+92,120,92,40,"Change","fix13", "middle",false)
let picbox2 = limit screen to 88 by 36 at xstart+ 94, 122
let picbox3 = limit screen to 185 by 40 at xstart-1, 120
let bxstart := xstart+115
rec(bxstart,240,60,40)
nor abox1 onto abox1
nor abox2 onto abox2
nor abox3 onto abox3
let procddata := vector 1::4 of 0.0
let theperiod := vector 1::2 of 0.0
let x = @1 of real [bxstart, bxstart, bxstart+ 60, bxstart+ 60,bxstart]
let y = @1 of real [240,280,280,240,240]
let names = @1 of string [" "]
let thespacing := 2.0
let theangle := 0.0
let thedash := -1.0
let thegap := -1.0
PIC := hatchpoly(x,y,5,0,thespacing,theangle,thedash,thegap,names)
draw(screen, PIC, 0, X.dim(screen), 0, Y.dim(screen) )
copy menuwin onto msgeSave
s.enter("Hdefault", theimage, gimage(msgeSave) )
s.enter("Images", DBvar, theimage)
if commit() ~= nil do write "Image is not stored"
end
let linemenu = proc()
begin
let xstart := X.dim(screen)-190
let a.pic := [ 0.0, 0.0 ]
let menuwin = limit screen to 195 by 400 at xstart-5,115
let lineimage = image 200 by 330 of off
let xloc := X.dim(screen)-85
let xloc1 := xstart + 10
let xend := X.dim(screen)-25
let xend1 := xloc1 + 70
let xs := @1 of real [xloc1,xend1]
let ys := vector 1::2 of 0.0
Rec(xstart,120,184,310,"LINE TYPE","cou20", "middle", true)
a.pic := drawline(xloc,365,xend,365)
a.pic := a.pic & doblseg(xloc1,365,xend1,365,4,-1,-1,-1)
a.pic := a.pic & dashing(xloc,335,xend,335,10,5)
a.pic := a.pic & doblseg(xloc1,335,xend1,335,4,-2,10,5)
a.pic := a.pic & dashing(xloc,305,xend,305,4,4)
a.pic := a.pic & doblseg(xloc1,305,xend1,305,4,-3,10,5)
a.pic := a.pic & dashing(xloc,275,xend,275,12,2)
a.pic := a.pic & doblseg(xloc1,275,xend1,275,4,-4,10,5)
a.pic := a.pic & dashing(xloc,245,xend,245,2,4)
ys(1) := 245;ys(2) := 245
a.pic := a.pic & thkline(xs,ys,4,2)!thick line
a.pic := a.pic & dashing(xloc,215,xend,215,1,2)!dashed line
a.pic := a.pic & doblseg(xloc1,215,xend1,215,4,-2,4,3)!double dashed line
a.pic := a.pic & dashing(xloc,185,xend,185,1,4)!dashed line
ys(1) := 185;ys(2) := 185
a.pic := a.pic & ddline(xs,ys,4,2,6,6)!thick dashed line
a.pic := a.pic & bordering(xloc,155,xend,155,10,4)!border line
ys(1) := 155;ys(2) := 155
a.pic := a.pic & railine(xs,ys,4,2)!rail line
draw(screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
copy menuwin onto lineimage
s.enter("LineMenu", theimage, gimage(lineimage) )
s.enter("Images", DBvar, theimage)
if commit() ~= nil do write "Image is not stored"
end
typemenu()
xor screen onto screen
angmenu()
xor screen onto screen
scalmenu()

```

```
xor screen onto screen  
hatchspace()  
xor screen onto screen  
default.menu()  
xor screen onto screen  
linemenu()
```



```

let DB := open.database("code2","code2","write")
if DB is error.record do DB := create.database( "code2", "code2" )
if DB is error.record do { write "Error creating Database'n" }

! procedure to transfer string number to integers
let StringToInt = proc(string S -> int)
begin
    let X := 0
    for i = 1 to length(S) do
        X := 10 * X + decode(S(i|1)) - 48
    X
end

write "Give file name: "

let filename = read.a.line()
let fd = open(filename,0)

let digit1 := 0; let Name1 := ""
let digit2 := 0; let Name2 := ""
let digit3 := 0; let Name3 := ""
let digit4 := 0; let Name4 := ""
let L := ""
let Q := 0
let ch := ""
let P := 0
let Temp := ""

let Code2Table := table()
let level2table := table()
let level3table := table()
let level4table := table()

structure StringPack (string StringValue)
structure Code2Node (string identifier; pntr subtree)

write "Started'n"
xor screen onto screen
let window = limit screen to X.dim(screen)-100 by 100 at 100, 100

while ~eoi(fd) do
begin
    L := read.a.line(fd)
    while L = "" and ~eoi(fd) do L := read.a.line(fd)
    if digit(L(5|1))
    then begin
        digit3 := StringToInt(L(5|2))
        digit4 := StringToInt(L(8|3))
        if digit4 = 0
        then begin
            Temp := L(15 | length(L) - 14)
            Name4 := ""
        end
        else begin
            P := 15
            while L (P | 1) ~= "-" do P := P+1
            Temp := L( 15 | P - 16)
            Name4 := L( (P+2) | (length(L) - P - 1) )
        end
        if Temp ~= Name3
        do begin
            Name3 := Temp
            level4table := table()
            i.enter (digit3,level3table,Code2Node(Name3,level4table))
        end
        i.enter (digit4,level4table,Code2Node(Name4,StringPack(L(1|10))))
    end

    else begin
        P := 2
        digit2 := 0
        while digit(L(P|1)) do
            begin
                digit2 := 10 * digit2 + decode (L(P|1)) - 48
            end
        end
    end
end

```

```

        P := P+1
    end
    while L(P|1) ~= "-" do P := P + 1
    P := P+1
    while L(P|1) = " " do P := P + 1
    Q := P+1
    while L(Q|1) ~= "-" do Q := Q + 1
    Temp := L(P|Q-P-1)
    if Name1 ~= Temp do
        begin
            Name1 := Temp
            digit1 := decode (L(1|1)) - 48
            level2table := table()
            i.enter (digit1,Code2Table,Code2Node(Name1,level2table))
        end
        Name2 := L(Q+2|length(L)-Q-1)
        level3table := table()
        i.enter (digit2,level2table,Code2Node(Name2,level3table))
    end
    xor window onto window
    print digit1,digit2,digit3,digit4 at 100, 100
end
close (fd)
s.enter ("Code2",DB,Code2Table)
if commit() = nil then write "Code2 stored OK"
    else write "Code2 not stored"
?

```

APPENDIX C

APPENDIX C: GLOBAL PROCEDURES

This Appendix lists the program called 'utility.S' which contains all the global procedures discussed in Chapter 6. The program is constructed so that it first creates the database 'Global.Proc', then a listing of the twenty two procedures forming the Global Procedures are included. These are then packaged into a structure and stored in the database.

```

let FONTsdb:=open.database("FONTS","friend","read")
let fix13 = s.lookup( "fix13", FONTsdb)
let bold = s.lookup("fixb13",FONTsdb)
let big = s.lookup("met22",FONTsdb)
let large = s.lookup("hcl45i",FONTsdb)
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
  do {write "No utilities database - do prcdbmaker first'n"; abort}
let prcget=
  begin
    structure procpak(proc(string -> pntr) xproc)
      s.lookup("prcget",procsdb) (xproc)
    end
let seditor={structure procpak(proc(string,string,int,int,int,int->
  string) xproc)
  prcget("seditor") (xproc) }
let error.message={structure procpak(proc(string,int,int) xproc)
  prcget("error.message") (xproc) }
let more={structure procpak(proc(*string,int,int) xproc)
  prcget("more") (xproc) }
let form.generate={structure procpak(proc( -> pntr ) xproc)
  prcget("form.generate") (xproc)}
let form.null={structure procpak(proc( string,int,int,int,int,int,pntr ) xproc)
  prcget("form.null") (xproc) }
structure form.package( proc( pntr) Form.show;
  proc( ) Form.all.show;
  proc( string,int,int,int,int,int,bool,proc(),pntr ->
    pntr ) Form.add;
  proc( pntr ) Form.remove;
  proc( string,pntr ) Form.update;
  proc( ) Form.clear;
  proc( -> pntr ) Form.mouse;
  proc( ) Fender;
  proc( ) Form.monitor )

let xmstart := X.dim(screen)-250
let XOL := 0.0; let XOR := 0.0
let YOL := 0.0; let YOR := 0.0
let XZL := 0.0; let XZR := 0.0
let YZL := 0.0; let YZR := 0.0
let xstart := 10; let ystart := 10
let xend := X.dim(screen)- 10; let yend := Y.dim(screen) - 53
let xspan := xend - xstart; let yspan := yend - ystart
let TotalYmenWin := yend-ystart-97
let Screen = limit screen to xspan by yspan at xstart, ystart
let xlcorner := xstart + 4; let ylcorner := ystart+101
let X.G := xspan -244
let Y.G := TotalYmenWin-2
if X.G > Y.G then X.G := Y.G
  else Y.G := X.G
let GW = limit screen to X.G by Y.G at xlcorner, ylcorner
let xmenustart := X.dim(screen) - 250
let menuwin = limit screen to 236 by TotalYmenWin-3 at xmenustart,ylcorner
let menusaved = image 217 by 224 of off
let out.range := X.dim(screen) - Y.dim(screen)
let range := X.dim(screen) - out.range
let gw = limit screen to range by range at 0, 0
let message.proc = proc(string txt, txt1, txt2; int xpos, ypos, xdim, ydim ->
  bool)
  begin
    let BoxDim.x := 0
    let lenth1.x := length(txt1)
    let lenth2.x := length(txt2)
    if lenth1.x >= lenth2.x then BoxDim.x := lenth1.x
      else BoxDim.x := lenth2.x
    BoxDim.x := BoxDim.x * 8 + 20
    let Boxlx := xpos + 10
    let Boxly := ypos + 15
    let Box2x := xpos + xdim div 2 + 5
    let Box2y := Boxly
    let TxtBox.x := xpos + 10
    let TxtDim.x := xdim - 10
    let TxtBox.y := ypos + ydim - 40
    let TxtDim.y := ydim div 3 - 5
    let msgeBox = limit screen to xdim + 10 by ydim + 10 at xpos, ypos

```

```

let hmsgeBox = limit screen to xdim by ydim at xpos + 5, ypos + 5
let msgeSave = image xdim + 10 by ydim + 10 of off
copy msgeBox onto msgeSave
xnor msgeBox onto msgeBox
xor hmsgeBox onto hmsgeBox
let F = form.generate()
let Fadd = F( Form.add )
let the.bool := false
form.null( txt, TxtBox.x, TxtBox.y, TxtDim.x, TxtDim.y, fix13)
let Yesproc = proc()
  { the.bool := true; F(Fender)() }
let Noproc = proc()
  { the.bool := false; F(Fender)() }
let dummy := Fadd( txt1, Box1x, Box1y, BoxDim.x, 30, false, Yesproc,
  fix13 )
dummy := Fadd( txt2, Box2x, Box2y, BoxDim.x, 30, false, Noproc, fix13 )
F(Form.monitor)()
let count := 1
while count ~= 2000 do count := count + 1
copy msgeSave onto msgeBox
the.bool
end
let text.write = proc(int xpos,ypos;string name;font;#pixel anyimage)
begin
  if font = "cou20" then
    copy string.to.tile(name,"cou20") onto limit anyimage at xpos,ypos
  else if font = "fixb13" then
    copy string.to.tile(name,"fixb13") onto limit anyimage at xpos,ypos
  else copy string.to.tile(name,"fix13") onto limit anyimage at xpos,ypos
end
let stringtoreal = proc (string S -> real)
begin
  let p := 1; let x := 0.0
  let s1 := 0; let s2 := 0
  while ~digit(S(p|1)) do p := p + 1
  while S(p|1) ~= "." do
    begin
      s1 := s1 * 10 + decode(S(p|1)) - 48
      p := p + 1
    end
  let num := 1
  p := p + 1
  while p < length(S) do
    begin
      s2 := s2 * 10 + decode(S(p|1)) - 48
      p := p + 1
      num := num*10
    end
  x := s1 + s2/num
  x
end
let stringtoint = proc(string S -> int)
begin
  let X := 0
  let tsign := 1
  let start := 1
  if S(1|1) = "-" do
    begin
      tsign := -1
      start := 2
    end
  for i = start to length(S) do
    X := (10 * X + decode(S(i|1)) - 48)
  X := X * tsign
  X
end
let minmax = proc(*real avector; int size -> *real)
begin
  let values = vector 1::2 of 0.0
  values(1) := avector(1)
  values(2) := avector(1)
  for i=2 to size do
    begin
      if values(1) > avector(i) do values(1) := avector(i)!the min
      if values(2) < avector(i) do values(2) := avector(i)!the max
    end
  end

```

```

        end
    values
end
let icon = proc(int x, y; string atext -> int)
begin
    let text.length := length(atext)
    let box.length := text.length * 8 + 20
    let xcoors = @1 of int [ x+5, x, x, x+5, x+box.length-5, x+box.length,
                             x+box.length, x+box.length-5, x+5]
    let ycoors = @1 of int [ y, y+5, y+25, y+30, y+30, y+25, y+5, y, y]
    let xil = @1 of int [ x+7, x+2, x+2, x+7, x+box.length-7, x+box.length-2,
                          x+box.length-2, x+box.length-7, x+7]
    let yil = @1 of int [ y+2, y+7, y+23, y+28, y+28, y+23, y+7, y+2, y+2 ]
    let PIC := [0,0]
    let PIC1 := [0,0]
    for i=1 to 9 do
        PIC := if i=1 then [ xcoors(i) , ycoors(i) ]
                     else PIC ^ [ xcoors(i) , ycoors(i) ]
    for i=1 to 9 do
        PIC1 := if i=1 then [ xil(i) , yil(i) ]
                  else PIC1 ^ [ xil(i) , yil(i) ]
    PIC := PIC & PIC1
    let icon.image = limit screen to box.length by 30 at x,y
    draw(screen, PIC,0,X.dim(screen),0,Y.dim(screen))
    copy string.to.tile( atext, "fixb13" ) onto limit icon.image at 8,10
    box.length
end
let polygon = proc(real Xcentre, Ycentre, Radian;string resolution -> pic)
begin
    let Pic1 := [Xcentre + Radian, Ycentre]
    let AngleInRadian := 0.0
    let X := 0.0
    let Y := 0.0
    let interval = case true of
                     resolution = "tr" : 120
                     resolution = "rec" : 90
                     resolution = "hx" : 60
                     resolution = "vl" : 30
                     resolution = "low" : 15
                     resolution = "hi" : 10
                     default:5
    for angle =1 to 420 by interval do
        begin
            AngleInRadian := angle * pi / 180
            X := Radian * cos(AngleInRadian) + Xcentre
            Y := Radian * sin(AngleInRadian) + Ycentre
            Pic1 := Pic1^[X,Y]
        end
    Pic1
end
let Highlight = proc(*real x, y -> pic)
begin
    let a.pic := [ 0.0, 0.0 ]
    for i=lwb(x) to upb(x) do
        a.pic := if i=1 then polygon( x(i), y(i), 3, "hx" )
                     else a.pic & polygon( x(i), y(i), 3, "hx" )
    a.pic
end
let drawline = proc(real x1,y1,x2,y2 -> pic )
begin
    let figure := [x1,y1]^[x2,y2]
    figure
end
let Box = proc(int x,y,side)
begin
    let a.pic := drawline(x,y,x+side,y)
    a.pic := a.pic & drawline(x+side,y,x+side,y+side)
    a.pic := a.pic & drawline(x+side,y+side,x,y+side)
    a.pic := a.pic & drawline(x,y+side,x,y)
    draw( screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
end
let rec = proc(int x,y,length,width)
begin
    let a.pic := drawline(x,y,x+length,y)
    a.pic := a.pic & drawline(x+length,y,x+length,y+width)

```

```

    a.pic := a.pic & drawline(x+length,y+width,x,y+width)
    a.pic := a.pic & drawline(x,y+width,x,y)
    draw( screen, a.pic, 0, X.dim(screen), 0, Y.dim(screen) )
end
let Rec = proc(int x,y,thelength,width; string anything, font,position;
               bool highlight)
begin
    let chsize := 0
    let ypos := 0
    if font = "cou20" then
        begin
            chsize := 14
            ypos := 7
        end
    else {chsize:= 8;ypos := 12}
    rec(x,y,thelength,width)
    let textlength := length(anything) * chsize
    let rest := 0
    if position = "begining" then rest := 10
    else if position = "end" then rest := x + thelength - textlength
    else rest := (thelength - textlength) div 2
    let textbox := limit screen to thelength-4 by 33 at x+2,y+width-35
    if font = "cou20" then
        copy string.to.tile(anything,"cou20") onto limit textbox at rest,ypos
    else if font = "fixb13" then
        copy string.to.tile(anything,"fixb13") onto limit textbox at rest,ypos
    else copy string.to.tile(anything,"fixl3") onto limit textbox at rest,ypos
    if highlight do nor textbox onto textbox
end
let north.dir = proc(int angle; #pixel any.window )
begin
    let vecX = @1 of int [0,-12,-3,-10,0,10,3,12,0]
    let vecY = @1 of int [-10,-20,10,5,20,5,10,-20,-10]
    let PIC := [0.0,0.0]
    let PIC2 := [0.0,0.0]
    for i=1 to 9 do
        begin
            PIC := if i = 1 then [ vecX(i) , vecY(i) ]
                    else PIC ^ [ vecX(i) , vecY(i) ]
        end
    if angle = 0 then
        PIC2 := PIC
    else PIC2 := rotate PIC by angle
    let PIC3 := polygon(0,-2,22,"hi")
    PIC3 := PIC3 & drawline( -25, -2, 25, -2 )
    PIC3 := PIC3 & drawline( 0, -27, 0, 23 )
    PIC3 := PIC3 & PIC2
    let xfactor := sin(angle * pi / 180)
    let yfactor := cos(angle * pi / 180)
    let Nxplace := truncate(33 * xfactor) + 50
    let Nyplace := truncate(33 * yfactor) + 50
    let Sxplace := truncate(-43 * xfactor) + 50
    let Syplace := truncate(-43 * yfactor) + 50
    draw(any.window,PIC3,-50,60,-50,60)
    text.write(Nxplace, Nyplace, "N", "cou20", any.window)
    text.write(Sxplace, Syplace, "S", "cou20", any.window)
end
let feature.type = proc(int this.counter -> string)
begin
    let xloc := xmstart + 20
    let xend := xloc + 110
    let box1 = limit screen to 100 by 30 at xloc,220
    let box2 = limit screen to 100 by 30 at xloc,180
    let box3 = limit screen to 100 by 30 at xloc,140
    let thismenu = limit screen to 217 by 224 at xmstart+9,119
    if this.counter = 1 then {
        Rec(xmstart+10,305,215,37,"CHOOSE","cou20", "begining", true)
        Rec(xmstart+10,120,215,185,"FEATURE TYPE","cou20", "begining", true)
        Rec(xloc,140,100,30,"polygon","fixl3", "begining", false)
        rec(xloc+2,140+2,96,26)
        rec(xloc+3,140+3,95,25)
        Rec(xloc,180,100,30,"line","fixl3", "begining", false)
        rec(xloc+2,180+2,96,26)
        rec(xloc+3,180+3,95,25)
        Rec(xloc,220,100,30,"point","fixl3", "begining", false)
    }
end

```



```

rec(xloc+2,220+2,96,26)
rec(xloc+3,220+3,95,25)
copy thismenu onto menused
else copy menused onto thismenu
let thetype := ""
let xo := 0
let yo := 0
let done := false
while ~done do
begin
let lo := locator()
while ~lo(the.buttons)(1) do lo := locator()
xo := lo(X.pos)
yo := lo(Y.pos)
while lo(the.buttons)(1) do lo := locator()
if xo > xloc and xo < xend and yo > 220 and yo < 250 do
begin
nor box1 onto box1
thetype := "point"
done := true
end
if xo > xloc and xo < xend and yo > 180 and yo < 210 do
begin
nor box2 onto box2
thetype := "line"
done := true
end
if xo > xloc and xo < xend and yo > 140 and yo < 170 do
begin
nor box3 onto box3
thetype := "polygon"
done := true
end
end
xor thismenu onto thismenu
thetype
end
let zoomout = proc(pic PIC, thegrid; int Range, xshft, yshft; #pixel the.win )
begin
if XZL = 0 and XZR = 0 then
error.message(" Nothing to Zoom Out ", -1, -1 )
else
begin
xor the.win onto the.win
Box( xshft, yshft, Range)
Box( xshft+1, yshft+1, Range-2)
draw(the.win, thegrid, XOL, XOR, YOL, YOR )
draw(the.win, PIC, XOL, XOR, YOL, YOR )
XZL := 0.0; XZR := 0.0
YZL := 0.0; YZR := 0.0
end
end
let zoomin = proc(pic PIC, thegrid;int Range, count, xshft, yshft;
#pixel the.win )
begin
let xl := 0.0; let xr := 0.0
let yl := 0.0; let yr := 0.0
if count < 2 then {
xl := XOL; xr := XOR
yl := YOL; yr := YOR }
else {
xl := XZL; xr := XZR
yl := YZL; yr := YZR }
let xo := 0; let xe := 0
let yo := 0; let ye := 0
let lo := locator()
while ~lo(the.buttons)(1) do lo := locator()
xo := lo(X.pos) - xshft
yo := lo(Y.pos) - yshft
while lo(the.buttons)(1) do lo := locator()
xe := lo(X.pos) - xshft
ye := lo(Y.pos) - yshft
if xo > xe do {
let t := xo
xo := xe

```

```

    xe := t }
    if yo < ye do {
        let t := yo
        yo := ye
        ye := t }
    let xdif := rabs( xe - xo )
    let ydif := rabs( ye - yo )
    let adif := truncate( ( xdif + ydif ) / 2 )
    xe := xo + adif
    ye := yo - adif
    XZL := xl + (xr - xl) * xo / Range
    XZR := xl + (xr - xl) * xe / Range
    YZL := yl + (yr - yl) * ye / Range
    YZR := yl + (yr - yl) * yo / Range
    xor the.win onto the.win
    Box( xshft, yshft, Range)
    Box( xshft+1, yshft+1, Range-2)
    draw(the.win, thegrid, XZL, XZR, YZL, YZR )
    draw( the.win, PIC, XZL, XZR, YZL, YZR )
end
let checkin = proc(*real Xvec,Yvec; real Xmin,Xmax,Ymin,Ymax; int Size -> bool)
begin
    let PIC1 := [ 0.0, 0.0 ]
    let savedwin = image X.G by Y.G of off
    copy GW onto savedwin
    let Can := false
    PIC1 := Highlight(Xvec,Yvec)
    draw(GW,PIC1,Xmin,Xmax,Ymin,Ymax)
    let counter := 0; let limits := truncate(2*Size/3)
    for i=1 to Size do {
        if Xvec(i) < Xmax and Xvec(i) > Xmin and
            Yvec(i) < Ymax and Yvec(i) > Ymin do
            counter := counter + 1 }
    if counter ~= 0 and counter > limits then
        Can := message.proc("Can Identify Object","Yes","No",
            X.dim(screen)-215,120,200,140)
    else Can := false
    if ~Can do copy savedwin onto GW
    Can
end
let prepform = proc(string atitle)
begin
    xor gw onto gw
    Box( 0, 0, range)
    Box( 1, 1, range-2)
    rec(range, 0, out.range, Y.dim(screen) )
    rec(range+2, 2, out.range-4, Y.dim(screen)-4 )
    Rec(range+4, Y.dim(screen)-44, out.range-8, 38,atitle,"cou20",
        "begining", true)
    Rec(range+4, Y.dim(screen)-80, out.range-8, 36,"Map of :", "fixbl3",
        "begining", false)
end
structure global.proc( proc(string, string, string, int, int, int, int ->
    bool)MessageProc;
    proc(int, int, string, string, #pixel)Text.Write;
    proc(string -> real)StringToReal;
    proc(string -> int)StringToInt;
    proc(*real, int -> *real)MinMax;
    proc(int, int, string -> int)Icon;
    proc(real, real, string -> pic)Polygon;
    proc(*real, *real -> pic)highlight;
    proc(real, real,real, real -> pic)Drawline;
    proc(int, int, int)box;
    proc(int, int, int, int)rectangle;
    proc(int, int, int, int, string, string, string,
        bool)Rectangle;
    proc(int, #pixel)North.Dir;
    proc(int -> string)Feature.Type;
    proc(pic, pic, int, int, int, #pixel)Zoomout;
    proc(pic, pic, int, int, int, int, #pixel)Zoomin;
    proc(*real, *real, real, real, real, real, int ->
        bool)Checkin;
    proc(string)Prepform )
let Global.Pack = global.proc(message.proc, text.write, stringtoreal,
    stringtoint, minmax, icon, polygon, Highlight,

```

```
                drawline, Box, rec, Rec, north.dir, feature.type,  
                zoomout, zoomin, checkin, prepform)  
let ProcDB := open.database("Proc.Lib", "proc", "write")  
if ProcDB is error.record do ProcDB := create.database("Proc.Lib", "proc")  
let GLOBALS := s.lookup("Procedures", ProcDB)  
if GLOBALS = nil do  
  begin  
    GLOBALS := table()  
    s.enter("Procedures", ProcDB, GLOBALS)  
  end  
s.enter("Global.Proc", GLOBALS, Global.Pack)  
if commit() = nil do write "Global Procedures Stored Successfully'n"  
?  
?
```

APPENDIX D

APPENDIX D: DATA ENTRY MODULE

Appendix D contains the listing of the program concerned with Data Entry. It retrieves the required Global Procedures from the database 'Global.Proc' and then lists the module procedures needed and described in Chapter 7. The module is then stored in the database '%\$Modules'.

```

let DataEntry = proc()
begin
let FONTsdb:=open.database("FONTS","friend","read")
let fix13 = s.lookup( "fix13", FONTsdb)
let bold = s.lookup("fixb13",FONTsdb)
let big = s.lookup("met22",FONTsdb)
let large = s.lookup("hci45i",FONTsdb)
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
do {write "No utilities database - do prcddbmaker first'n"; abort}
let prcget=
begin
structure procpak(proc(string -> pntr) xproc)
s.lookup("prcget",procsdb) (xproc)
end
let seditor={structure procpak(proc(string,string,int,int,int,int->
string) xproc)
prcget("seditor") (xproc) }
let error.message={structure procpak(proc(string,int,int) xproc)
prcget("error.message") (xproc) }
let more={structure procpak(proc(*string,int,int) xproc)
prcget("more") (xproc) }
let form.generate={structure procpak(proc( -> pntr ) xproc)
prcget("form.generate") (xproc)}
let form.null={structure procpak(proc( string,int,int,int,int,pntr ) xproc)
prcget("form.null") (xproc) }
structure form.package( proc( pntr) Form.show;
proc( ) Form.all.show;
proc( string,int,int,int,int,bool,proc(),pntr ->
pntr ) Form.add;
proc( pntr ) Form.remove;
proc( string,pntr ) Form.update;
proc( ) Form.clear;
proc( -> pntr ) Form.mouse;
proc( ) Fender;
proc( ) Form.monitor )
let set.up.choose = {structure procpak(proc(*string -> pntr) xproc)
prcget("set.up.choose") (xproc)}
structure chooser.pack( proc( string, int, int -> string ) do.choose;
proc( string ) add.choose;
proc( string ) remove.choose;
proc( int, int ) list.choose )
let table.to.text = {structure procpak(proc(pntr -> *string) xproc)
prcget("table.to.text") (xproc)}
let DB = open.database("code2","code2","write")
if DB is error.record do { write"cannot open DB "; abort }
structure Code2Node(string identifier; pntr subtree)
structure menuPack( proc( string, int, int -> string) menuProc; c*string Lname;
c*int Lnum )
let mainDB := open.database("MDB","data","write")
if mainDB is error.record do mainDB := create.database( "MDB", "data" )
if mainDB is error.record do { error.message("Cannot open database'n",-1,-1);
abort }
let mapstable := s.lookup( "Maps", mainDB )
if mapstable = nil do
mapstable := table()
structure maps( string Package, SerialNo, Scale, North;real XL, YL, XR, YR, Grid;
*pntr AST, LST, PST, TST )
structure Aholder(int ATN, AID; bool AP; real Xc, Yc; string AFN, AC; *real AX,
AY;pic anypic; real AD1, AD2; int NOI; *int Inc; pic FA.pic )
structure Lholder(int LTN, LID; bool LP; string LFN, LC; *real LX, LY; pic L.pic;
real LD1, LD2; pic FL.pic )
structure Pholder(int PTN, PID; bool PP; string PFN, PC, PT; *real PX, PY;
pic P.pic)
structure Tholder(int Layer, Location; real xpos, ypos; string thetext, Font)
let Ainfo = vector 1::10000 of nil
let Linfo = vector 1::5000 of nil
let Pinfo = vector 1::3000 of nil
let DBvar := open.database("global","variables","write")
if DBvar is error.record do DBvar := create.database( "global", "variables" )
if DBvar is error.record do { write "Error creating Database'n" }
structure globals(int varval)
structure gpic(pic thepic)
let DBsym := open.database("Symbols","mys","read")
let agrsyms := s.lookup( "%$Agrsym" , DBsym )

```

```

let geosyms := s.lookup("%$Geosym", DBsym )
structure agrsym(pic f.pic )
structure geosym(pic a.pic)
structure gimage(#pixel menuimage)
let theimage := s.lookup("Images", DBvar)
let Hlinetype := s.lookup("Hlinetype",theimage) (menuimage)
let Hangle := s.lookup("Hangle",theimage) (menuimage)
let Scaling := s.lookup("Scaling",theimage) (menuimage)
let Hspacing := s.lookup("Hspacing",theimage) (menuimage)
let Hdefault := s.lookup("Hdefault",theimage) (menuimage)
let LineMenu := s.lookup("LineMenu",theimage) (menuimage)
let symenu := s.lookup("Symbols Menu", theimage) (menuimage)
let typeimage = image 110 by 355 of off
let angimage = image 155 by 270 of off
let scalimage = image 150 by 320 of off
let symtype := ""
let xmstart := X.dim(screen)-250
let Sht := X.dim(screen) - 540
structure global.proc(proc(string, string, string, int, int, int, int ->
    bool)MessageProc;
    proc(int, int, string, string, #pixel)Text.Write;
    proc(string -> real)StringToReal;
    proc(string -> int)StringToInt;
    proc(*real, int -> *real)MinMax;
    proc(int, int, string -> int)Icon;
    proc(real, real, real, string -> pic)Polygon;
    proc(*real, *real -> pic)highlight;
    proc(real, real,real, real -> pic)Drawline;
    proc(int, int, int)box;
    proc(int, int, int, int)rectangle;
    proc(int, int, int, int, string, string, string, bool)Rectan
    proc(int, #pixel)North.Dir;
    proc(int -> string)Feature.Type;
    proc(pic, pic, int, int, int, #pixel)Zoomout;
    proc(pic, pic, int, int, int, int, #pixel)Zoomin;
    proc(*real, *real, real, real, real, real, int -> bool)Check
    proc(string)Prepform )
let ProcDB := open.database("Proc.Lib", "proc", "write")
let GLOBALS := s.lookup("Procedures", ProcDB)
let message.proc = GLOBALS(MessageProc)
let text.write = GLOBALS(Text.Write)
let stringtoreal = GLOBALS(StringToReal)
let stringtoint = GLOBALS(StringToInt)
let minmax = GLOBALS(MinMax)
let icon = GLOBALS(Icon)
let polygon = GLOBALS(Polygon)
let Highlight = GLOBALS(highlight)
let drawline = GLOBALS(Drawline)
let Box = GLOBALS(box)
let rec = GLOBALS(rectangle)
let Rec = GLOBALS(Rectangle)
let north.dir = GLOBALS(North.Dir)
let feature.type = GLOBALS(Feature.Type)
let checkin = GLOBALS(Checkin)
let prepform = GLOBALS(Prepform)
let area = proc(*real xs, ys; int sizes -> real)
begin
    let tempx := vector 1::2 of 0.0
    let tempy := vector 1::2 of 0.0
    tempy := minmax(ys, sizes)
    let anarea := 0.0; let dx := 0.0
    let dy := 0.0; let ya := 0.0
    for i=1 to sizes-1 do
        begin
            let j := 1 + i
            dx := xs(j) - xs(i)
            dy := ys(j) - ys(i)
            ya := ( ys(j) + ys(i) )/2 - tempy(1)
            anarea := anarea + dx * ya
        end
        if anarea < 0 do anarea := rabs(anarea)
        anarea
    end
let perimeter = proc(*real xs, ys; int sizes -> real)
begin

```

```

let dx := 0.0
let peri := 0.0; let dy := 0.0
for i=1 to sizes-1 do
  begin
    let j := 1 + i
    dx := xs(j) - xs(i)
    dy := ys(j) - ys(i)
    peri := peri + sqrt( dx * dx + dy * dy )
  end
end
peri
end
let distance = proc(real x1, y1, x2, y2 -> real)
begin
  let Distance := sqrt( rabs( (x2-x1) * (x2-x1) + (y2-y1) * (y2-y1) ) )
  Distance
end
let nabor = proc(*real X,OX -> bool)
begin
  let upper1 := upb(X)
  let upper2 := upb(OX)
  let continue := false
  let found := false
  while ~continue do
    for i=1 to upper1 do
      begin
        for j=1 to upper2 do
          if OX(j) = X(i) do {
            found := true
            continue := true }
          if i=upper1 and ~found do
            begin
              found := false
              continue := true
            end
          end
        end
      end
    end
    found
  end
end
let answer = vector 1::12 of ""
let questions = @1 of string ["Name of Package used in Digitization",
  "Map of (Volume Set Name)",
  "Serial Number of Volume Set",
  "X,Y of the Lower Corner",
  "X,Y of the Opposite Corner",
  "Grid Interval",
  "North Bearing",
  "Total Physical Number of Files",
  "Physical File Number",
  "Source File Name",
  "Scale of the Map",
  "Number of Features' coordinates (2 or 3) "]

let xstart := 10; let ystart := 10
let xend := X.dim(screen)- 10; let yend := Y.dim(screen) - 53
let xspan := xend - xstart; let yspan := yend - ystart
let TotalYmenWin := yend-ystart-97
let Screen = limit screen to xspan by yspan at xstart, ystart
let xlcorner := xstart + 4; let ylcorner := ystart+101
let X.G := xspan -244
let Y.G := TotalYmenWin-2
if X.G > Y.G then X.G := Y.G
  else Y.G := X.G
let GW = limit screen to X.G by Y.G at xlcorner, ylcorner
let xmenustart := X.dim(screen) - 250
let menuwin = limit screen to 236 by TotalYmenWin-3 at xmenustart,ylcorner
let menusaved = image 217 by 224 of off
let out.range := X.dim(screen) - Y.dim(screen)
let range := X.dim(screen) - out.range
let gw = limit screen to range by range at 0, 0
let x.coord = vector 1::100 of 0.0
let y.coord = vector 1::100 of 0.0
let dummy.vec := vector 1::1000 of ""
let winsave := image X.G by Y.G of off
let XOL := 0.0; let XOR := 0.0
let YOL := 0.0; let YOR := 0.0
let XZL := 0.0; let XZR := 0.0
let YZL := 0.0; let YZR := 0.0

```



```

let TheGrid := 0.0
let thegrid := [ 0.0, 0.0 ]
let xt := range + 100; let yt := 271!Global
let tmenu = limit screen to 102 by 192 at xt-1, yt-1!Global
let xe := xt + 40; let ye := yt + 46!Global
let emenu = limit screen to 112 by 102 at xe-1, ye-1!Global
let xl := xt - 90; let yl := 100!Global
let lmenu = limit screen to 142 by 402 at xl-1, yl-1!Global
let eimage := s.lookup("Entity Menu", theimage)(menuimage)
let timage := s.lookup("Type Menu", theimage)(menuimage)
let limage := s.lookup("Layer Menu", theimage)(menuimage)
let def.menu := "1"
let zoomcounter := 0
structure feature(pic fea.pic )
let first.screen = proc()
begin
  Rec(0,0,X.dim(screen),Y.dim(screen),"Ps - GIS","cou20",
    "middle", true)
  let border1 = limit screen to X.dim(screen)-10 by Y.dim(screen)-43 at 5,5
  let border2 = limit screen to X.dim(screen)-20 by Y.dim(screen)-53 at 10,10
  nor border1 onto border1
  nor border2 onto border2
end
let loc.print = proc(int count,xloc,Phase;string phrase)
begin
  let cput := (11 - count) * Phase + 120
  copy string.to.tile(phrase, "fix13") onto limit Screen at xloc + 10,
    cput + 90
end
let text.window = proc()
begin
  let fixtext = limit screen to 140 by 95 at 13,13
  let yposition := 75
  copy string.to.tile("FEATURE      :", "fixb13") onto limit fixtext at 10,
    yposition
  copy string.to.tile("SEGMENTS      :", "fixb13") onto limit fixtext at 10,
    yposition - 17
  copy string.to.tile("CODE          :", "fixb13") onto limit fixtext at 10,
    yposition - 34
  copy string.to.tile("LEFT COORDS   :", "fixb13") onto limit fixtext at 10,
    yposition - 51
  copy string.to.tile("RIGHT COORDS  :", "fixb13") onto limit fixtext at 10,
    yposition - 68
  nor fixtext onto fixtext
end
let second.screen = proc(-> *string)
begin
  xor Screen onto Screen
  let phase := yspan div 15
  let interval := 1
  while interval < 13 do
    begin
      loc.print(interval,25,phase,questions(interval))
      interval := interval + 1
    end
    let continue := false
    while ~continue do
      begin
        for loop = 1 to 12 do
          begin
            loc.print(loop,400,phase,": ")
            repeat answer(loop) := seditor( questions(loop),
              answer( loop ), 100, 85, 500, 50 )
            while answer(loop) = "" do
              loc.print( loop, 410, phase,answer(loop) )
              loc.print( loop, 410, phase,answer(loop) )
            end
            continue := message.proc("Proceed or Retry ", "Done", "Cancel",
              xend-215, 120,
              200, 140)
          end
        end
        let dummy := system( "touch config" )
        let fod = open("config",2)
        for loop = 1 to 12 do
          output fod,answer(loop),'n'
        end
      end
    end
  end
end

```

```

close(fod)
for i=1 to 2000 do { }
xor Screen onto Screen
answer
end
let third.screen = proc(*string details;*real minmax -> pic)
begin
  let ystartdetail := yend-200
  rec(xstart+2,ystart+2,xspan-3,98)
  rec(xstart+3,ystart+3,xspan-5,96)
  rec(xmstart,ystart+100,238,TotalYmenWin)
  Rec(xmstart,ystartdetail,238,200,"File Details","fixb13","middle",true)
  let detail = limit screen to 238 by 200 at xmstart,ystartdetail
  let detail.data = @ 1 of string ["Scale      ", "File Name ", "Map of      ",
                                   "Package    "]
  let detail.data1 = @ 1 of string [details(11),details(10),details(2),
                                   details(1)]
  for i=1 to 4 do {
    let ypos := if i=1 then 10
                 else if i=2 then 50
                 else if i=3 then 90
                 else 130
    text.write( 10, ypos,detail.data(i),"fixb13",detail)
    text.write( 119, ypos,detail.data1(i),"fixl3",detail)
  }
  rec(xstart+2,110,xspan-243,TotalYmenWin)
  rec(xstart+3,110,xspan-243,TotalYmenWin)
  let the.grid := stringtoint( details(6) )
  let x.grid := minmax(1) + the.grid
  let y.grid := minmax(2) + the.grid
  let PIC := [0.0,0.0]
  let i := 1
  while x.grid <= minmax(3) do
    begin
      PIC := if i = 1 then [ x.grid,minmax(2) ] ^ [ x.grid,minmax(4) ] &
                        [ minmax(1),y.grid ] ^ [ minmax(3),y.grid ]
              else PIC & [ x.grid,minmax(2) ] ^ [ x.grid,minmax(4) ] &
                        [ minmax(1),y.grid ] ^ [ minmax(3),y.grid ]
      draw (GW, PIC, minmax(1),minmax(3),minmax(2),minmax(4) )
      x.grid := x.grid + the.grid
      y.grid := y.grid + the.grid
      i := i + 1
    end
  end
  let the.angle := stringtoint( details(7) )
  let x.dir := xmenustart - 140
  let a.window = limit screen to 120 by 120 at x.dir,Y.dim(screen)-170
  north.dir(the.angle, a.window)
  thegrid := PIC
  PIC
end
let drawfile = proc(string datafile;real Xmin,Xmax,Ymin,Ymax,Grid;bool Draw ->
                    pic)
begin
  let thevar := s.lookup("Variables",DBvar)
  if thevar = nil do
    thevar := table()
  let dfn = open (datafile,0)
  if dfn = nullfile do
    begin
      write "The file ",datafile," cannot be opened'n"
      abort
    end
  let filename = read.a.line(dfn)
  close (dfn)
  let fd = open (filename,0)
  if fd = nullfile do
    begin
      write "The file ",filename," cannot be opened'n"
      abort
    end
  end
  let PIC := [ 0.0, 0.0 ]
  let PIC1 := [ 0.0, 0.0 ]
  let nfeatures := 0;let KL := 1
  let features = vector 1 :: 100 of nil
  let tempvec := vector 1::1000 of ""

```

```

let L := read.a.line(fd); tempvec(KL) := L; KL := KL + 1
L := read.a.line(fd); tempvec(KL) := L; KL := KL + 1
L := read.a.line(fd); tempvec(KL) := L; KL := KL + 1
let thepos := KL
let readlines = proc(string aline -> *string )
begin
  let stemp = vector 1::100 of ""
  let con := 1
  stemp(con) := aline
  let P := 0; let acounter := 0
  if ~eoi(fd) and L(13|1) = "*" do
    begin
      con := con + 1
      L := read.a.line(fd); stemp(con) := L; con := con + 1
      L := read.a.line(fd)
      if eoi(fd) then
        begin
          P := 1
          while L(P|1) ~= " " do P := P + 1
          acounter := acounter + 1
          x.coord(acounter) := stringtoreal( L(1|P) )
          y.coord(acounter) := stringtoreal( L(P+1|length(L) - P) )
          PIC := if acounter = 1 then [x.coord(acounter) ,
                                     y.coord(acounter) ]
                                     else PIC ^ [x.coord(acounter) ,
                                                  y.coord(acounter) ]
          stemp(con) := L
        end
      else {
        let i := 0
        while digit( L(1|1) ) and ~eoi(fd) do
          begin
            i := i + 1
            P := 1
            while L(P|1) ~= " " do P := P + 1
            x.coord(i) := stringtoreal( L(1|P) )
            y.coord(i) := stringtoreal( L(P+1 | length(L) - P ) )
            PIC := if i = 1 then [x.coord(i) , y.coord(i) ]
                        else PIC ^ [x.coord(i) , y.coord(i) ]
            stemp(con) := L; con := con + 1
            L := read.a.line(fd)
            acounter := i
          end
          if eoi(fd) do {
            P := 1
            while L(P|1) ~= " " do P := P + 1
            acounter := acounter + 1
            x.coord(acounter) := stringtoreal( L( 1| P ) )
            y.coord(acounter) := stringtoreal( L( P+1|length(L) -
                                                    P ) )
            PIC := PIC ^ [x.coord(acounter) , y.coord(acounter) ]
            stemp(con) := L } }
          nfeatures := nfeatures + 1
          features( nfeatures ) := feature( PIC )
        end
      let thisvector := vector 1::acounter+2 of ""
      for k=1 to acounter+2 do
        thisvector(k) := stemp(k)
      for i = 1 to nfeatures do {
        if Draw do
          draw (GW, features(i)( fea.pic ), Xmin, Xmax, Ymin, Ymax )
          PIC1 := if i=1 then features(i)( fea.pic )
                  else PIC1 & features(i)( fea.pic ) }
        !draw (GW, PIC1, Xmin, Xmax, Ymin, Ymax )
        thisvector
      end
      L := read.a.line(fd)
    while ~eoi(fd) do
      begin
        let temporal := readlines(L)
        let thesize := upb(temporal)
        let vecsize := 0
        for i=thepos to (thepos + thesize - 1) do
          begin
            KL := KL + 1

```

```

        tempvec(i) := temporal(i-thepos+1)
    end
    thepos := KL
end
close(fd)
KL := KL - 1
let a.dummy := system("touch temp1")
let td = open("temp1",1)
for i=1 to KL do {
    output td,tempvec(i),"n" }
close(td)
s.enter("dr",thevar,gp1c(PIC1))
s.enter("Variables",DBvar,thevar)
if commit() ~= nil do error.message( "Could not store Picture'n", -1, -1 )
PIC1
end
let getdetail = proc(*string filedetail -> *real)
begin
    let thedetails = vector 1::6 of 0.0
    let counter := 0
    let mindata := filedetail(4)
    let thestringlength := length(mindata)
    for m=1 to thestringlength do
        if mindata(m|1) = "," do counter := m
        thedetails(1) := stringtoreal( mindata(1|counter-1) ); XOL := thedetails(1)
        thedetails(2) := stringtoreal( mindata(counter+1|thestringlength-counter));
        YOL := thedetails(2)

        let maxdata := filedetail(5)
        thestringlength := length(maxdata)
        for m=1 to thestringlength do
            if mindata(m|1) = "," do counter := m
            thedetails(3) := stringtoreal( maxdata(1|counter-1) ); XOR := thedetails(3)
            thedetails(4) := stringtoreal( maxdata(counter+1|thestringlength-counter));
            YOR := thedetails(4)

            thedetails(5) := stringtoint( filedetail(8) )
            thedetails(6) := stringtoint( filedetail(6) ); TheGrid := thedetails(6)
        thedetails
    end
end
let firstdrawn = proc(*real borders;string filename;bool to.draw -> pic)
begin
    let the.pic := drawfile(filename,borders(1),borders(3),borders(2),borders(4)
        borders(6),to.draw)
    the.pic
end
let zoomout = proc(pic PIC, thegrid; int Range, xshft, yshft; #pixel the.win )
begin
    if XZL = 0 and XZR = 0 then
        error.message(" Nothing to Zoom Out ", -1, -1 )
    else
        begin
            xor the.win onto the.win
            Box( xshft, yshft, Range)
            Box( xshft+1, yshft+1, Range-2)
            draw(the.win, thegrid, XOL, XOR, YOL, YOR )
            draw(the.win, PIC, XOL, XOR, YOL, YOR )
            XZL := 0.0; XZR := 0.0
            YZL := 0.0; YZR := 0.0
        end
    end
end
let zoomin = proc(pic PIC, thegrid;int Range, count, xshft, yshft; #pixel
    the.win )
begin
    let xl := 0.0; let xr := 0.0
    let yl := 0.0; let yr := 0.0
    if count < 2 then {
        xl := XOL; xr := XOR
        yl := YOL; yr := YOR }
    else {
        xl := XZL; xr := XZR
        yl := YZL; yr := YZR }
    let xo := 0; let xe := 0
    let yo := 0; let ye := 0
    let lo := locator()
    while ~lo(the.buttons)(1) do lo := locator()
    xo := lo(X.pos) - xshft

```

```

yo := lo(Y.pos) - yshft
while lo(the.buttons)(1) do lo := locator()
xe := lo(X.pos) - xshft
ye := lo(Y.pos) - yshft
if xo > xe do {
  let t := xo
  xo := xe
  xe := t }
if yo < ye do {
  let t := yo
  yo := ye
  ye := t }
let xdif := rabs( xe - xo )
let ydif := rabs( ye - yo )
let adif := truncate( ( xdif + ydif ) / 2 )
xe := xo + adif
ye := yo - adif
XZL := xl + (xr - xl) * xo / Range
XZR := xl + (xr - xl) * xe / Range
YZL := yl + (yr - yl) * ye / Range
YZR := yl + (yr - yl) * yo / Range
xor the.win onto the.win
Box( xshft, yshft, Range)
Box( xshft+1, yshft+1, Range-2)
draw(the.win, thegrid, XZL, XZR, YZL, YZR )
draw( the.win, PIC, XZL, XZR, YZL, YZR )
end
let change.code = proc(-> int)
begin
  let thecode := 0
  let former.image = image 238 by TotalYmenWin-3 of off
  copy menuwin onto former.image
  xor menuwin onto menuwin
  let ystartdetail := yend-180
  let detail = limit screen to 238 by 180 at xmstart,ystartdetail
  rec(xmstart,ystart+100,238,TotalYmenWin)
  Rec(xmstart,ystartdetail,238,180,"Feature Details","fixbl3","middle",true)
  let classification = @ 1 of string["Class      :", "Category  :",
                                     "Feature   :",
                                     "Attribute  :", "The Code  :"]

  for i=1 to 5 do {
    let ypos := if i=1 then 110
                  else if i=2 then 85
                  else if i=3 then 60
                  else if i=4 then 35
                  else 10
    text.write( 10, ypos,classification(i),"fixbl3",detail) }
  let VecNames = vector 1::100 of ""
  let VecNums  = vector 1::100 of 0
  let N := 0
  let procl = proc(int I; pnttr V -> bool)
  begin
    N := N + 1
    VecNames(N) := V(identifier)
    VecNums(N) := I
    true
  end
  let stopchase := false
  let cat = @1 of string ["Class","Category","Feature","Attribute"]
  let choices = vector 1::4 of ""
  let catWidth = X.dim(screen) div 4
  let choose := proc(int L; pnttr T); nullproc
  choose := proc(int L; pnttr T)
  begin
    let theMenu := s.lookup( "menu",T)
    if theMenu = nil do
      begin
        N := 0
        let X := i.scan(T,procl)
        let S := ""
        let levelname = vector 1::N of ""
        let levelnum = vector 1::N of 0
        for i=1 to N do
          { levelname(i) := VecNames(i);
            levelnum(i) := VecNums(i) }
        end
      end
    end
  end
end

```

```

        for i=1 to N-1 do for j=i+1 to N do
            { X := levelnum(i); levelnum(i) := levelnum(j);
              levelnum(j) := X; S := levelname(i);
              levelname(i) := levelname(j); levelname(j) := S }
        for i=1 to N do
            begin
                if length(levelname(i)) > 25 do
                    levelname(i) := levelname(i) (1|25)
                if levelname(i) = "" do levelname(i) := "-----"
            end
            let ch = set.up.choose(levelname) (do.choose)
            theMenu := menuPack(ch, levelname, levelnum)
            s.enter ( "menu", T, theMenu)
            if commit() ~= nil do print "COMMIT FAILS" at 50,300
        end
        let ch = theMenu(menuProc)
        let levelname = theMenu(Lname)
        let levelnum = theMenu(Lnum)
        N := upb(levelname)
        let choice = ch("Choose " ++ cat(L), X.dim(screen)-240,120)
        if choice = "" do stopchase := true
        let chint := 0
        if choice ~= "" do {
            for i=1 to N do
                if choice = levelname(i) do chint := levelnum(i)
            choices(L) := choice
            thecode := if L=1 then chint * 10000000
                       else if L = 2 then thecode + chint * 100000
                       else if L = 3 then thecode + chint * 1000
                       else thecode + chint
            let ypos := if L=1 then 110
                       else if L=2 then 85
                       else if L=3 then 60
                       else 35
            text.write( 119, ypos,choices(L),"fixb13",detail) }
            while ~stopchase and L < 4 do choose( L+1, i.lookup(chint,T) (subtree)
        end
        let levell := s.lookup("Code2",DB)
        choose( 1, levell)
        print thecode at X.dim(screen)-120,ystartdetail+10
        copy former.image onto menuwin
        thecode
    end
end
let text.entry = proc(string typeoftext,previous.string -> string)
begin
    let awindow := limit screen to 740 by 150 at 40,140
    let savedimage = image 750 by 150 of off
    copy awindow onto savedimage
    xor awindow onto awindow
    rec (41,141,738,148)
    rec (42,142,736,146)
    copy string.to.tile ("Enter The ", "fixb13") onto limit awindow at 50,110
    copy string.to.tile (typeoftext, "fixb13") onto limit awindow at 150,110
    copy string.to.tile ("(not longer than 43 characters including spaces)",
                        "fix13") onto limit awindow at 50,80
    write code(27),"N",code(27),"I"
    let atext := ""
    atext := seditor(" ",previous.string,50, 150,710,50)
    if length(atext) > 43 do
        atext := atext(1|43)
    write code(27),"E",code(27),"W"
    copy savedimage onto awindow
    atext
end
let centroid = proc(*real avector1,avector2; int the.size;real Xmin,Xmax,Ymin,
                    Ymax -> *real)
begin
    let xloc := xmstart + 20
    let thismenu = limit screen to 217 by 224 at xmstart+9,119
    let msg.menu = limit screen to 100 by 40 at xmstart+9,159
    rec(xmstart+10,120,215,180)
    rec(xmstart+11,121,213,178)
    text.write(10,150,"Chose the centroid of","fixb13",thismenu)
    text.write(10,120,"this area by clicking","fixb13",thismenu)
    text.write(10,90,"the left button of the","fixb13",thismenu)

```

```

text.write(10,60,"mouse on the desired ","fixb13",thismenu)
text.write(10,30,"location of polygon","fixb13",thismenu)
let xminmax := minmax(avector1,the.size)
let xmin := xminmax(1); let xmax := xminmax(2)
let yminmax := minmax(avector2,the.size)
let ymin := yminmax(1); let ymax := yminmax(2)
let counter := 1
let lo := locator()
let coords = vector 1::2 of 0.0
let proceed := false
while ~proceed do
  begin
    if counter > 1 do
      begin
        rec(xmstart+10,160,98,38)
        text.write(10,15,"Try again","fix13",msg.menu)
      end
      while ~lo(the.buttons)(1) do lo := locator()
      let xo := lo(X.pos) - xlcorner
      let yo := lo(Y.pos) - ylcorner
      coords(1) := Xmin + (Xmax - Xmin) * xo / X.G
      coords(2) := Ymin + (Ymax - Ymin) * yo / Y.G
      while lo(the.buttons)(1) do lo := locator()
      if coords(1) < xmax and coords(1) > xmin do
        if coords(2) < ymax and coords(2) > ymin do
          { proceed := true
            counter := counter + 1 }
        xor msg.menu onto msg.menu
      end
      xor thismenu onto thismenu
      coords
    end
  end
let picking = proc(*string givens; int areacounter,linecounter, pointcounter)
begin
  let thevar := s.lookup("Variables",DBvar)
  if thevar = nil do
    thevar := table()
  text.window()
  let xmin := 0.0; let xmax := 0.0
  let ymin := 0.0; let ymax := 0.0
  if XZR = 0 and YZR = 0 then {
    xmin := XOL; xmax := XOR
    ymin := YOL; ymax := YOR }
  else {
    xmin := XZL; xmax := XZR
    ymin := YZL; ymax := YZR }
  let yposition := 75
  let PIC := [ 0.0 , 0.0 ]
  let PIC1 := [ 0.0 , 0.0 ]
  let dummy.pic := [ 0.0 , 0.0 ]
  let Dummy.vec := @1 of int[0]
  let feature.name := ""
  let s.code := ""; let featype := ""
  let feature.count := areacounter + linecounter + pointcounter
  let local.counter := 0
  if feature.count > 0 do
    begin
      let atable := s.lookup("Maps", mainDB )
      let Avec := s.lookup(givens(2),atable)(AST)
      let Lvec := s.lookup(givens(2),atable)(LST)
      let Pvec := s.lookup(givens(2),atable)(PST)
      let Atop := upb(Avec)
      let Ltop := upb(Lvec)
      let Ptop := upb(Pvec)
      let thetop := 0
      for i= 1 to Atop do
        Ainfo(i) := Avec(i)
      thetop := Atop
      for i= 1 to Ltop do
        Linfo(i) := Lvec(i)
      thetop := thetop + Ltop
      for i= 1 to Ptop do
        Pinfo(i) := Pvec(i)
      thetop := thetop + Ptop
    end
  end

```

```

let the.centroid := vector 1::2 of 0.0
let previous.text := ""
let xdim := X.dim(screen) - 340 - xlcorner
let detail.window = limit screen to xdim by 95 at 153,14
let ind := 1; let typecount := 1
let exist := true
let out.of.range := false
let fe = open("temp2",0)
if fe = nullfile do
    exist := false
if exist do
    begin
        let dummy := system("rm temp1")
        dummy := system("touch temp1")
        dummy := system("cp temp2 temp1")
        dummy := system("rm temp2")
        close (fe)
    end
let fd = open("temp1",0)
if fd = nullfile do
    { write "the file temp1 cannot be opened"; abort }
let tempx = vector 1::100 of 0.0
let tempy = vector 1::100 of 0.0
let nfeatures := 0
let KL := 1
let L := read.a.line(fd); dummy.vec(KL) := L; KL := KL + 1
L := read.a.line(fd); dummy.vec(KL) := L; KL := KL + 1
L := read.a.line(fd); dummy.vec(KL) := L; KL := KL + 1
let thepos := KL
let readlines = proc(string aline -> *string)
begin
    while ~eoi(fd) and aline(1|1) = "/" do
        begin
            aline := read.a.line(fd); aline := read.a.line(fd)
            while digit( aline(1|1) ) and ~eoi(fd) do
                aline := read.a.line(fd)
            end
let stemp = vector 1::100 of ""
let con := 1
stemp(con) := aline
L := aline
let P := 0
let acounter := 0
if ~eoi(fd) and ( L(13|1) = "*" or L(14|1) = "*" ) do
    begin
        con := con + 1
        let Q := 1
        while L(Q|1) ~= "*" do Q := Q + 1
        let t.length := length(L) - Q - 2
        feature.name := L(Q+2|t.length)
        L := read.a.line(fd); stemp(con) := L; con := con + 1
        s.code := L(5|length(L)-4)
        L := read.a.line(fd)
        if eoi(fd) then
            begin
                P := 1
                while L(P|1) ~= " " do P := P + 1
                acounter := acounter + 1
                tempx(acounter) := stringtoreal( L( 1 | P) )
                tempy(acounter) := stringtoreal( L( P+1 | length(L) - P )
                stemp(con) := L
            end
        else
            begin
                let i := 0
                while digit( L(1|1) ) and ~eoi(fd) do
                    begin
                        i := i + 1
                        P := 1
                        while L(P|1) ~= " " do P := P + 1
                        tempx(i) := stringtoreal( L( 1 | P) )
                        tempy(i) := stringtoreal( L( P+1 | length(L) - P )
                        stemp(con) := L; con := con + 1
                        L := read.a.line(fd)
                        acounter := i
                    end
                end
            end
        end
    end
end

```



```

        end
        if eoi(fd) do {
            P := 1
            while L(P|1) ~= " " do P := P + 1
            acounter := acounter + 1
            tempx(acounter) := stringtoreal( L( 1| P ))
            tempy(acounter) := stringtoreal(L( P+1|length(L) - P )
            stemp(con) := L }
        end
        end
        let thisvector = vector 1::acounter+2 of ""
        for k=1 to acounter+2 do
            thisvector(k) := stemp(k)
        end
        thisvector
    end
    let done := false
    L := read.a.line(fd)
    while ~eoi(fd) and ~done do
        begin
            let temporal := readlines(L)
            let thesize := upb(temporal)
            let isin := true
            let checked := false
            let vecsize := thesize - 2
            if vecsize <= 0 do done := true
            let tempscreen = image X.G by Y.G of off
            copy GW onto tempscreen
            let Xs := vector 1::vecsize of 0.0
            let Ys := vector 1::vecsize of 0.0
            for avar = 1 to vecsize do
                begin
                    Xs(avar) := tempx(avar)
                    Ys(avar) := tempy(avar)
                end
            end
            if vecsize = 1 then
                isin := if Xs(1) < xmax and Xs(1) > xmin and
                    Ys(1) < ymax and Ys(1) > ymin then true
                    else false
            else
                begin
                    let Xminmax := minmax(Xs,vecsize)
                    let Yminmax := minmax(Ys,vecsize)
                    isin := if Xminmax(1) > xmin and Xminmax(2) < xmax and
                        Yminmax(1) > ymin and Yminmax(2) < ymax then true
                        else {
                            checked := true
                            checkin(Xs,Ys,xmin,xmax,ymin,ymax,vecsize) }
                end
            end
            if isin then {
                local.counter := local.counter + 1
                temporal(1) := "/" ++ temporal(1)
                for i=thepos to (thepos + thesize - 1) do
                    begin
                        KL := KL + 1
                        dummy.vec(i) := temporal(i - thepos + 1)
                    end
                end
                if vecsize < 2 then
                    begin
                        let xposition := truncate( (Xs(1) - xmin) * X.G / (xmax -
                            xmin)) - 5
                        let yposition := truncate( (Ys(1) - ymin) * Y.G / (ymax -
                            ymin)) - 5
                        text.write( xposition, yposition, "!", "fixb13", GW )
                        PIC := [ Xs(1) , Ys(1) ]
                    end
                end
            else
                begin
                    if ~checked do {
                        PIC1 := Highlight(Xs, Ys) !thkline(Xs,Ys,6,vecsize)
                        draw(GW,PIC1,xmin,xmax,ymin,ymax) }
                    for i=1 to vecsize do
                        PIC := if i = 1 then [ Xs(i) , Ys(i) ]
                            else PIC ^ [ Xs(i) , Ys(i) ]
                    end
                end
            end
            xor detail.window onto detail.window
        end
    end

```

```

copy string.to.tile(feature.name,"fix13") onto limit
                                detail.window at 25, yposition
print (vecsize-1) at 180,yposition - 5
copy string.to.tile(s.code,"fix13") onto limit detail.window at 20
                                yposition - 34
print xmin,ymin at 163, yposition - 40
print xmax,ymax at 163, yposition - 57
let feature.kind := feature.type(typecount)
typecount := typecount + 1
let feature.code := change.code()
if feature.code = 0 do feature.code := change.code()
let thename := text.entry("Name","")
let thismenu = limit screen to 217 by 94 at xmstart+9,350
if feature.kind = "polygon" then
    { the.centroid := centroid(Xs,Ys,vecsize,xmin,xmax,ymin,ymax)
      let a.pic := polygon(the.centroid(1),the.centroid(2),8,"hx")
      draw(GW,a.pic,xmin,xmax,ymin,ymax)
      areacounter := areacounter + 1 }
else if feature.kind = "line" then
    linecounter := linecounter + 1
else if feature.kind = "point" do
    pointcounter := pointcounter + 1
let thetext := text.entry("Comment",previous.text)
previous.text := thetext
case true of
feature.kind = "polygon" : Ainfo(areacounter) := Aholder(areacount
    feature.code, false, the.centroid(1), the.centroid(2), thename
    thetext, Xs, Ys, PIC, 0, 0, 0, Dummy.vec, dummy.pic)
feature.kind = "line" : Linfo(linecounter) := Lholder(linecounter,
    feature.code, false, thename, thetext, Xs, Ys, PIC, 0, 0,
    dummy.pic)
default : Pinfo(pointcounter) := Pholder(pointcounter,feature.code
    false, thename,thetext,"", Xs, Ys,PIC)
xor thismenu onto thismenu
if ~eoi(fd) then
    done := message.proc("Chose what next","Break","Proceed",
        xend-215, 120, 200, 140)

    else done := true
xor GW onto GW
copy tempscreen onto GW
}
else {
    for i=thepos to (thepos + thesize - 1) do {
        KL := KL + 1
        dummy.vec(i) := temporal(i-thepos+1) }
    out.of.range := true }
thepos := KL
end
if eoi(fd) then
begin
    if out.of.range then
begin
        let vec.text = @1 of string["This is the end of the file,",
                                "but there are still some      ",
                                "features not yet identified.",
                                "Advice to change zooming.",
                                "click to proceed"]

        more(vec.text,50,100)
        let a.dummy := system("touch temp2")
        let td = open("temp2",1)
        let ind := 1
        while dummy.vec(ind) ~= "" do
            begin
                output td,dummy.vec(ind),"n"
                dummy.vec(ind) := ""
                ind := ind + 1
            end
        close(td)
        close(fd)
    end
else
begin
    error.message("Nothing out of range",X.dim(screen)-250,120)
    let a.dummy := system("rm temp1; rm config")
    for i=1 to 1000 do

```

```

        dummy.vec(i) := ""
    end
end
else {
    dummy.vec(thepos) := L
    let local.count := thepos + 1
    while ~eoi(fd) do
        begin
            dummy.vec(local.count) := read.a.line(fd)
            local.count := local.count + 1
        end
    close(fd)
    let a.dummy := system("touch temp2")
    let td = open("temp2",1)
    for i=1 to local.count-1 do {
        output td,dummy.vec(i),"n"
        dummy.vec(i) := "" }
    close(td)
}
let abase := 1
if areacounter = 0 do abase := 0
let avec := vector abase::areacounter of nil
for i=1 to areacounter do
    avec(i) := Ainfo(i)
let lbase := 1
if linecounter = 0 do lbase := 0
let lvec := vector lbase::linecounter of nil
for i=1 to linecounter do
    lvec(i) := Linfo(i)
let pbase := 1
if pointcounter = 0 do pbase := 0
let pvec := vector pbase::pointcounter of nil
for i=1 to pointcounter do
    pvec(i) := Pinfo(i)
let feat.text := vector 1::1000 of nil
s.enter("a",thevar,globals(areacounter))
s.enter("l",thevar,globals(linecounter))
s.enter("p",thevar,globals(pointcounter))
s.enter("Variables",DBvar,thevar)
if commit() ~= nil do error.message( "Could not store Variables'n", -1,-1
s.enter(givens(2),mapstable,maps( givens(1), givens(3), givens(11),
    givens(7), XOL, YOL, XOR, YOR, TheGrid, avec, lvec, pvec,
    feat.text ) )
s.enter( "Maps", mainDB, mapstable )
if commit() ~= nil do { error.message("data are not stored'n",-1,-1 ) ;
    abort }
end
let xtransform = proc(real anx; int range -> real)
begin
    let unx := XZL + (XZR - XZL) * anx / range
    unx
end
let ytransform = proc(real anx; int range -> real)
begin
    let unx := YZL + (YZR - YZL) * anx / range
    unx
end
let shiftL = proc(int range; #pixel Win; pic PIC)
begin
    let mrange := range - 100
    let commonpart = image mrange by range of off
    let commonpic = limit screen to mrange by range at 98, 0
    copy commonpic onto commonpart
    xor Win onto Win
    let newin = limit screen to mrange by range at 0, 0
    copy commonpart onto newin
    let xst := xtransform(mrange+2, range)
    let diff := XZR - xst
    XZL := XZL + diff
    XZR := XZR + diff
    draw(gw, PIC, XZL, XZR, YZL, YZR )
end
let shiftR = proc(int range; #pixel Win; pic PIC)
begin
    let mrange := range - 100

```

```

let commonpart = image mrange by range of off
let commonpic = limit screen to mrange by range at 2, 0
copy commonpic onto commonpart
xor Win onto Win
let newin = limit screen to mrange by range at 100, 0
copy commonpart onto newin
let xst := xtransform(mrange+2, range)
let diff := XZR - xst
XZL := XZL - diff
XZR := XZR - diff
draw(gw, PIC, XZL, XZR, YZL, YZR )
end
let shiftD = proc(int range; #pixel Win; pic PIC)
begin
let mrange := range - 34
let commonpart = image range by mrange of off
let commonpic = limit screen to range by mrange at 0, 32
copy commonpic onto commonpart
xor Win onto Win
let newin = limit screen to range by mrange at 0, 0
copy commonpart onto newin
let yst := ytransform(mrange+2, range)
let diff := YZR - yst
YZL := YZL + diff
YZR := YZR + diff
draw(gw, PIC, XZL, XZR, YZL, YZR )
end
let shiftU = proc(int range; #pixel Win; pic PIC)
begin
let mrange := range - 34
let commonpart = image mrange by mrange of off
let commonpic = limit screen to mrange by mrange at 0, 2
copy commonpic onto commonpart
xor Win onto Win
let newin = limit screen to mrange by mrange at 0, 34
copy commonpart onto newin
let yst := ytransform(mrange+2, range)
let diff := YZR - yst
YZL := YZL - diff
YZR := YZR - diff
draw(gw, PIC, XZL, XZR, YZL, YZR )
end
let Scroll = proc(pic the.pic; int Range; #pixel win )
begin
let bxl := XZL
let bxr := XZR
let byl := YZL
let byr := YZR
if XZL = 0 and XZR = 0 then
error.message("Nothing to Scroll", -1, -1 )
else {
!draw( win, the.pic, bxl, bxr, byl, byr )
let shft := X.dim(screen) - 230
let xstep = 100
let ystep = 100
let xdif := rabs( XZR - XZL )
let ydif := rabs( YZR - YZL )
for i=1 to 2 do {
Box( (shft + (i-1) * 100), 200, 40 )
Box( (shft + 50), (150 + (i-1) * 100), 40 ) }
let picbox1 = limit screen to 40 by 40 at shft,200
let picbox3 = limit screen to 40 by 40 at shft+100,200
let picbox2 = limit screen to 40 by 40 at shft+50,150
let picbox4 = limit screen to 40 by 40 at shft+50,250
let icon.length := icon(shft+45, 100, "Done" )
let picbox5 = limit screen to icon.length by 30 at shft+45, 100
let vecx = @1 of int [ 30, 30, 10, 10, 0, 10, 10, 30 ]
let vecy = @1 of int [ 10, 22, 22, 32, 16, 0, 10, 10 ]
let VPIC1 := [ 0,0 ]
for i=1 to 8 do
VPIC1 := if i=1 then [ vecx(i), vecy(i) ]
else VPIC1 ^ [ vecx(i), vecy(i) ]
let VPIC2 := rotate VPIC1 by -90
let VPIC3 := rotate VPIC2 by -90
let VPIC4 := rotate VPIC3 by -90

```

```

draw ( picbox1, VPIC1, -5, 35, -5, 35 )
draw ( picbox2, VPIC2, -35, 5, -5, 35 )
draw ( picbox3, VPIC3, -35, 5, -35, 5 )
draw ( picbox4, VPIC4, -5, 35, -35, 5 )
let xo := 0; let yo := 0
let done := false
while ~done do
  begin
    let lo := locator()
    while ~lo(the.buttons)(1) do lo := locator()
    xo := lo(X.pos)
    yo := lo(Y.pos)
    while lo(the.buttons)(1) do lo := locator()
    if xo > shft + 45 and xo < (shft+45+icon.length) and
      yo > 100 and yo < 130 then
      begin
        nor picbox5 onto picbox5
        done := true
      end
    else if xo > shft and xo < shft + 40 and yo > 200 and
      yo < 240 then
      begin
        nor picbox1 onto picbox1
        bxr := bxr + xstep
        if bxr > XOR do {
          !error.message( "Exceeded limit of map", -1, -1 )
          bxr := XOR }
        shiftL(Range,win, the.pic)
        nor picbox1 onto picbox1
      end
    else if xo > shft + 100 and xo < shft + 140 and
      yo > 200 and yo < 240 then
      begin
        nor picbox3 onto picbox3
        bxl := bxl - xstep
        if bxl < XOL do {
          !error.message( "Exceeded limit of map", -1, -1 )
          bxl := XOL }
        shiftR(Range,win, the.pic)
        nor picbox3 onto picbox3
      end
    else if xo > shft + 50 and xo < shft + 90 and yo > 150 and
      yo < 190 then
      begin
        nor picbox2 onto picbox2
        byl := byl - ystep
        if byl < YOL do {
          !error.message( "Exceeded limit of map", -1, -1 )
          byl := YOL }
        shiftD(Range,win, the.pic)
        nor picbox2 onto picbox2
      end
    else if xo > shft + 50 and xo < shft + 90 and yo > 250 and
      yo < 290 do
      begin
        nor picbox4 onto picbox4
        byr := byr + ystep
        if byr > YOR do {
          !error.message( "Exceeded limit of map", -1, -1 )
          byr := YOR }
        shiftU(Range,win, the.pic)
        nor picbox4 onto picbox4
      end
    if ~done do {
      !xor win onto win
      Box( 0, 0, Range)
      Box( 1, 1, Range-2)
      !draw( win, the.pic, bxl, bxr, byl, byr )
    }
    !XZL := bxl; XZR := bxr
    !YZL := byl; YZR := byr
  end
  let tempwin = limit screen to 215 by 200 at shft - 5, 95
  xor tempwin onto tempwin
end

```

end

```

let trans.code = proc()
begin
  let thevar := s.lookup("Variables",DBvar)
  let areacount := s.lookup("a",thevar)(varval)
  let linecount := s.lookup("l",thevar)(varval)
  let pointcount := s.lookup("p",thevar)(varval)
  let details := vector 1::12 of ""
  let borders := vector 1::4 of 0.0
  let MinMax := vector 1::6 of 0.0
  let datafile := ""
  let operations = @1 of string [ "Zoom In", "Zoom Out", "Start" ]
  let exist := true
  let fe = open("config",0)
  if fe = nullfile do
    exist := false
  if exist then
    begin
      let ind := 1
      while ~eoi(fe) do {
        details(ind) := read.a.line(fe)
        ind := ind + 1 }
      close (fe)
      xor screen onto screen
      first.screen()
      datafile := details(10)
      MinMax := getdetail(details)
      let the.grid := third.screen(details,MinMax)
      let datapic := s.lookup("dr",thevar)(thepic)
      draw(GW,datapic, XOL, XOR, YOL, YOR)
      let parwin := limit screen to 120 by 30 at range+90,Y.dim(screen)-315
      let finished := false
      while ~finished do {
        let chooseoperation = set.up.choose(operations)(do.choose)
        let operationchosen = chooseoperation( "OPTIONS      ", xmstart+50,120)
        if operationchosen = "Zoom In" then {
          xor parwin onto parwin
          rec(range+91, Y.dim(screen)-314, 100, 28 )
          text.write(20, 10,operationchosen,"fixbl3",parwin)
          zoomcounter := zoomcounter + 1
          !zoomin( the.grid, thegrid, X.G, zoomcounter, xlcorner, ylcorner, GW )
          zoomin( datapic, thegrid, X.G, zoomcounter, xlcorner, ylcorner, GW )
          xor parwin onto parwin }
        else if operationchosen = "Zoom Out" then {
          xor parwin onto parwin
          rec(range+91, Y.dim(screen)-314, 100, 28 )
          text.write(20, 10,operationchosen,"fixbl3",parwin)
          !zoomout( the.grid, X.G, xlcorner, ylcorner, GW)
          zoomout( datapic, thegrid, X.G, xlcorner, ylcorner, GW)
          xor parwin onto parwin
          zoomcounter := 0 }
        else if operationchosen = "Start" then {
          if XZR = 0 and YZR = 0 do
            XZL := XOL; YZL := YOL; XZR := XOR; YZR := YOR
            picking(details, areacount, linecount, pointcount) }
        else if operationchosen = "" do
          finished := true }
      end
    else {
      xor screen onto screen
      first.screen()
      details := second.screen()
      datafile := details(10)
      MinMax := getdetail(details)
      let the.grid := third.screen(details,MinMax)
      let datapic := firstdrawn(MinMax,datafile,true)
      draw(GW,datapic, XOL, XOR, YOL, YOR)
      let parwin := limit screen to 120 by 30 at range+90,Y.dim(screen)-315
      let finished := false
      while ~finished do {
        let chooseoperation = set.up.choose(operations)(do.choose)
        let operationchosen = chooseoperation( "OPTIONS      ", xmstart+50, 120 )
        if operationchosen = "Zoom In" then {
          xor parwin onto parwin
          rec(range+91, Y.dim(screen)-314, 100, 28 )
          text.write(20, 10,operationchosen,"fixbl3",parwin)

```

```

        zoomcounter := zoomcounter + 1
        zoomin( datapic, thegrid, X.G, zoomcounter, xllcorner, yllcorner, GW )
        xor parwin onto parwin }
    else if operationchosen = "Zoom Out" then {
        xor parwin onto parwin
        rec(range+91, Y.dim(screen)-314, 100, 28 )
        text.write(20, 10, operationchosen, "fixbl3", parwin)
        zoomout( datapic, thegrid, X.G, xllcorner, yllcorner, GW)
        xor parwin onto parwin
        zoomcounter := 0 }
    else if operationchosen = "Start" then {
        if XZR = 0 and YZR = 0 do
            XZL := XOL; YZL := YOL; XZR := XOR; YZR := YOR
            picking(details, areacount, linecount, pointcount) }
        else if operationchosen = "" do
            finished := true }
    }
    copy winsave onto GW
end
end
structure modules1( proc() dataentry)
let moduleDB := open.database("M", "m", "write")
if moduleDB is error.record do moduleDB := create.database("M", "m")
let the.module := s.lookup("Module1", moduleDB)
if the.module = nil do
    begin
        the.module := table()
        s.enter("Module1", moduleDB, the.module)
    end
end
s.enter("dataentry", the.module, modules1(DataEntry) )
?
```

APPENDIX E

APPENDIX E: CARTOGRAPHIC REPRESENTATION MODULE

This Appendix contains the listing of the program dealing with Data Cartographic Representation. The module first retrieves the Global Procedures from the database 'Global.Proc' and then lists the different module procedures needed and described in Chapter 8. The module is stored in the database '%\$Modules'.

```

let CartoRep = proc()
begin
let FONTsdb:=open.database("FONTS","friend","read")
let fix13 = s.lookup( "fix13", FONTsdb)
let bold = s.lookup("fixb13",FONTsdb)
let big = s.lookup("met22",FONTsdb)
let large = s.lookup("hci45i",FONTsdb)
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
do {write "No utilities database - do prcddbmaker first'n"; abort}
let prcget=
begin
structure procpak(proc(string -> pntr) xproc)
s.lookup("prcget",procsdb) (xproc)
end
let seditor={structure procpak(proc(string,string,int,int,int,int-> string) xproc
prcget("seditor") (xproc) )}
let error.message={structure procpak(proc(string,int,int) xproc)
prcget("error.message") (xproc) )}
let more={structure procpak(proc(*string,int,int) xproc)
prcget("more") (xproc) )}
let form.generate={structure procpak(proc( -> pntr ) xproc)
prcget("form.generate") (xproc)}
let form.null={structure procpak(proc( string,int,int,int,int,int,pntr ) xproc)
prcget("form.null") (xproc) )}
structure form.package( proc( pntr) Form.show;
proc( ) Form.all.show;
proc( string,int,int,int,int,int,bool,proc(),pntr ->
pntr ) Form.add;
proc( pntr ) Form.remove;
proc( string,pntr ) Form.update;
proc( ) Form.clear;
proc( -> pntr ) Form.mouse;
proc( ) Fender;
proc( ) Form.monitor )
let set.up.choose = {structure procpak(proc(*string -> pntr) xproc)
prcget("set.up.choose") (xproc)}
structure chooser.pack( proc( string, int, int -> string ) do.choose;
proc( string ) add.choose;
proc( string ) remove.choose;
proc( int, int ) list.choose )
let table.to.text = {structure procpak(proc(pntr -> *string) xproc)
prcget("table.to.text") (xproc)}
structure global.proc(proc(string, string, string, int, int, int, int ->
bool)MessageProc;
proc(int, int, string, string, #pixel)Text.Write;
proc(string -> real)StringToReal;
proc(string -> int)StringToInt;
proc(*real, int -> *real)MinMax;
proc(int, int, string -> int)Icon;
proc(real, real, real, string -> pic)Polygon;
proc(*real, *real -> pic)highlight;
proc(real, real, real, real -> pic)Drawline;
proc(int, int, int)box;
proc(int, int, int, int)rectangle;
proc(int, int, int, int, string, string, string, bool)Rectan
proc(int, #pixel)North.Dir;
proc(int -> string)Feature.Type;
proc(pic, pic, int, int, int, #pixel)Zoomout;
proc(pic, pic, int, int, int, int, #pixel)Zoomin;
proc(*real, *real, real, real, real, real, int -> bool)Check
proc(string)Prepform )
let ProcDB := open.database("Proc.Lib", "proc", "write")
let GLOBALS := s.lookup("Procedures", ProcDB)
!let Global.Pack = s.lookup ("Global.Proc", GLOBALS)
let message.proc = GLOBALS(MessageProc)
let text.write = GLOBALS(Text.Write)
let stringtoreal = GLOBALS(StringToReal)
let stringtoint = GLOBALS(StringToInt)
let minmax = GLOBALS(MinMax)
let icon = GLOBALS(Icon)
let polygon = GLOBALS(Polygon)
let Highlight = GLOBALS(highlight)
let drawline = GLOBALS(Drawline)
let Box = GLOBALS(box)

```

```

let rec = GLOBALS(rectangle)
let Rec = GLOBALS(Rectangle)
let north.dir = GLOBALS(North.Dir)
let feature.type = GLOBALS(Feature.Type)
let checkin = GLOBALS(Checkin)
let prepform = GLOBALS(Prepform)
let DB = open.database("code2","code2","write")
if DB is error.record do { write"cannot open DB "; abort }
structure Code2Node(string identifier; pnttr subtree)
structure menuPack( proc (string, int, int -> string) menuProc; c*string Lname;
                    c*int Lnum )
let mainDB := open.database("MDB","data","write")
if mainDB is error.record do mainDB := create.database( "MDB", "data" )
if mainDB is error.record do { error.message("Cannot open database'n",-1,-1);
                             abort }
let mapstable := s.lookup( "Maps", mainDB )
if mapstable = nil do
    mapstable := table()
structure maps( string Package, SerialNo, Scale, North;real XL, YL, XR, YR, Grid;
                *pnttr AST, LST, PST, TST )
structure Aholder(int ATN, AID; bool AP; real Xc, Yc; string AFN, AC; *real AX,
                  AY; pic anypic; real AD1, AD2; int NOI; *int Inc; pic FA.pic )
structure Lholder(int LTN, LID; bool LP; string LFN, LC; *real LX, LY; pic L.pic;
                  real LD1, LD2; pic FL.pic )
structure Pholder(int PTN, PID; bool PP; string PFN, PC, PT; *real PX, PY;
                  pic P.pic)
structure Tholder(int Layer, Location; real xpos, ypos; string thetext, Font)
let Ainfo = vector 1::10000 of nil
let Linfo = vector 1::5000 of nil
let Pinfo = vector 1::3000 of nil
let DBvar := open.database("global","variables","write")
if DBvar is error.record do DBvar := create.database( "global", "variables" )
if DBvar is error.record do { write "Error creating Database'n" }
structure globals(int varval)
structure gpic(pic thepic)
let DBsym := open.database("Symbols","mys","read")
let agrsyms := s.lookup( "%$Agrsym" , DBsym )
let geosyms := s.lookup("%$Geosym", DBsym )
structure agrsym(pic f.pic )
structure geosym(pic a.pic)
structure gimage(#pixel menuimage)
let theimage := s.lookup("Images", DBvar)
let Hlinetype := s.lookup("Hlinetype",theimage) (menuimage)
let Hangle := s.lookup("Hangle",theimage) (menuimage)
let Scaling := s.lookup("Scaling",theimage) (menuimage)
let Hspacing := s.lookup("Hspacing",theimage) (menuimage)
let Hdefault := s.lookup("Hdefault",theimage) (menuimage)
let LineMenu := s.lookup("LineMenu",theimage) (menuimage)
let symenu := s.lookup("Symbols Menu", theimage) (menuimage)
let typeimage = image 110 by 355 of off
let angimage = image 155 by 270 of off
let scalimage = image 150 by 320 of off
let symtype := ""
let xmstart := X.dim(screen)-250
let innerangle = proc(real x1,y1,x2,y2 -> real)
begin
    let angle := 0.0
    let dx := x2 - x1
    let dy := y2 - y1
    if dx ~= 0 then
        begin
            let ang := atan(rabs(dy/dx)) * 180 / pi
            let index = case true of
                dx > 0 and dy > 0 :90 - ang
                dx > 0 and dy < 0 :90 + ang
                dx < 0 and dy < 0 :270 - ang
                dx < 0 and dy > 0 :270 + ang
                dx > 0 and dy = 0 :90
                dx < 0 and dy = 0 :270
                default:{-999999}
            angle := index
            if angle = -999999 do abort
        end
    else
        begin

```

```

        if dx = 0 and dy > 0 then
            angle := 0
        else if dx = 0 and dy < 0 do angle := 180
        end
    angle
end
let area = proc(*real xs, ys; int sizes -> real)
begin
    let tempx := vector 1::2 of 0.0
    let tempy := vector 1::2 of 0.0
    tempy := minmax(ys, sizes)
    let anarea := 0.0; let dx := 0.0
    let dy := 0.0; let ya := 0.0
    for i=1 to sizes-1 do
        begin
            let j := 1 + i
            dx := xs(j) - xs(i)
            dy := ys(j) - ys(i)
            ya := ( ys(j) + ys(i) )/2 - tempy(1)
            anarea := anarea + dx * ya
        end
    end
    if anarea < 0 do anarea := rabs(anarea)
    anarea
end
let perimeter = proc(*real xs, ys; int sizes -> real)
begin
    let dx := 0.0
    let peri := 0.0; let dy := 0.0
    for i=1 to sizes-1 do
        begin
            let j := 1 + i
            dx := xs(j) - xs(i)
            dy := ys(j) - ys(i)
            peri := peri + sqrt( dx * dx + dy * dy )
        end
    end
    peri
end
let distance = proc(real x1, y1, x2, y2 -> real)
begin
    let Distance := sqrt( rabs( (x2-x1) * (x2-x1) + (y2-y1) * (y2-y1) ) )
    Distance
end
let nabor = proc(*real X,OX -> bool)
begin
    let upper1 := upb(X)
    let upper2 := upb(OX)
    let continue := false
    let found := false
    while ~continue do
        for i=1 to upper1 do
            begin
                for j=1 to upper2 do
                    if OX(j) = X(i) do {
                        found := true
                        continue := true }
                    if i=upper1 and ~found do
                        begin
                            found := false
                            continue := true
                        end
                    end
                end
            end
        end
        found
    end
end
let polychck = proc(*real xs, ys; int sizes -> bool)
begin
    let angsum := 0.0
    let calsum := 0.0
    let h := 0
    let j := 0
    let ap1 := 0.0
    let ap2 := 0.0
    let angle := 0.0
    let closed := false
    for i=1 to sizes-1 do
        begin

```

```

        h := i-1
        j := i+1
        if h = 0 do h := sizes-1
        ap1 := innerangle(xs(i), ys(i), xs(h), ys(h) )
        ap2 := innerangle(xs(i), ys(i), xs(j), ys(j) )
        angle := ap1 - ap2
        if angle < 0 do angle := angle + 360
        angsum := angsum + angle
    end
    calsum := 90 * (2 * (sizes - 1) - 4)
    let diff := rabs(angsum - calsum)
    if diff > (1/100) then
        closed := false
    else closed := true
if ~closed do
    begin
        angsum := 0.0
        for i=1 to sizes-1 do
            begin
                h := i-1
                j := i+1
                if h = 0 do h := sizes-1
                ap1 := innerangle(xs(i), ys(i), xs(h), ys(h) )
                ap2 := innerangle(xs(i), ys(i), xs(j), ys(j) )
                angle := ap1 - ap2
                if angle < 0 then angle := rabs(angle)
                else angle := 360 - angle
                angsum := angsum + angle
            end
            calsum := 90 * (2 * (sizes - 1) - 4)
            let diff := rabs(angsum - calsum)
            if diff > (1/100) then
                closed := false
            else closed := true
        end
    end
end
let linepara = proc(real x1, y1, x2, y2 -> *real)
begin
    let parameter = vector 1::2 of 0.0
    if x2-x1 = 0 then {parameter(1) := 0.0; parameter(2) := x2}
    else {
        parameter(1) := (y2-y1)/(x2-x1)
        parameter(2) := (y1*x2 - y2*x1)/(x2-x1)}
    parameter
end
let perpenline = proc(real x,y,m -> *real)
begin
    let parameter = vector 1::2 of 0.0
    if m=0 then
        begin
            parameter(1) := x
            parameter(2) := 0
        end
    else {
        parameter(1) := -1/m
        parameter(2) := y - parameter(1)*x}
    parameter
end
let Lparaline = proc(real a,b,w; int K -> *real)
begin
    let const = vector 1::2 of 0.0
    const(1) := a
    const(2) := b + K * w * sqrt(1 + a*a)
    const
end
let Rparaline = proc(real a,b,w; int K -> *real)
begin
    let const = vector 1::2 of 0.0
    const(1) := a
    const(2) := b - K * w * sqrt(1 + a*a)
    const
end
let lineint = proc(real a,b,c,d -> *real)
begin

```

```

let parameter = vector 1::2 of 0.0
if a = 0 and d = 0 then {
parameter(1) := c; parameter(2) := b}
else if b=0 and c=0 then{
parameter(1) := a; parameter(2) := d}
else if c = 0 and b ~= 0 then {
parameter(1) := -99999
parameter(2) := -99999}
else {parameter(1) := (d-b)/(a-c)
parameter(2) := (a*d - b*c)/(a-c)}
parameter
end
let dashing = proc(real x1,y1,x2,y2,dash,gap -> pic )
begin
let alpha := 0.0
let a.pic := [ 0.0, 0.0 ]
let period := dash + gap
let dy := y2 - y1
let dx := x2 - x1
let lengthoffline := sqrt( dx * dx + dy * dy )
if dx = 0 then
if dy > 0 then alpha := pi/2
else alpha := pi
else
begin
let k := 1; let c := 1
let NoOfPeriods := 0; let thegap := 0.0; let theperiod := 0.0
alpha := atan(rabs(dy/dx))
if dx < 0 do k := -k
if dy < 0 do c := -c
if lengthoffline > period then
begin
NoOfPeriods := truncate( lengthoffline / period )
let rest := lengthoffline - NoOfPeriods * period
thegap := rest / NoOfPeriods + gap
thepperiod := thegap + dash
let X1 := vector 1::(NoOfPeriods+1) of 0.0
let Y1 := vector 1::(NoOfPeriods+1) of 0.0
let X2 := vector 1::(NoOfPeriods+1) of 0.0
let Y2 := vector 1::(NoOfPeriods+1) of 0.0
let thecos := cos(alpha)
let thesin := sin(alpha)
let dxgap := thecos * thegap
let dxdash := thecos * dash
let dxhdash := 0.5 * dxdash
let dygap := thesin * thegap
let dydash := thesin * dash
let dyhdash := 0.5 * dydash
X1(1) := x1
Y1(1) := y1
X2(1) := x1 + dxhdash * k
Y2(1) := y1 + dyhdash * c
a.pic := drawline( X1(1), Y1(1), X2(1), Y2(1) )
for i = 2 to NoOfPeriods do
begin
X1(i) := X2(i-1) + dxgap * k
Y1(i) := Y2(i-1) + dygap * c
X2(i) := X1(i) + dxdash * k
Y2(i) := Y1(i) + dydash * c
a.pic := a.pic & drawline( X1(i), Y1(i), X2(i), Y2(i) )
end
X1(NoOfPeriods+1) := X2(NoOfPeriods) + dxgap * k
Y1(NoOfPeriods+1) := Y2(NoOfPeriods) + dygap * c
X2(NoOfPeriods+1) := x2
Y2(NoOfPeriods+1) := y2
a.pic := a.pic & drawline( X1(NoOfPeriods+1), Y1(NoOfPeriods+1),
X2(NoOfPeriods+1), Y2(NoOfPeriods+1) )
end
end
else a.pic := drawline(x1,y1,x2,y2)
end
a.pic
end
let doblseg = proc(real X1, Y1, X2, Y2, width,type,dash,gap -> pic )
begin
let a.pic := [ 0.0, 0.0 ]

```

```

let Para := vector 1::2 of 0.0
let Para1 := vector 1::2 of 0.0
let Para10 := vector 1::2 of 0.0
let Para2 := vector 1::2 of 0.0
let Para3 := vector 1::2 of 0.0
let Para30 := vector 1::2 of 0.0
let Para4 := vector 1::2 of 0.0
let Para5 := vector 1::2 of 0.0
let Para50 := vector 1::2 of 0.0
let k := 1
Para := linepara(X1,Y1,X2,Y2)
Para1 := perpenline(X1,Y1,Para(1))
Para10 := perpenline(X2,Y2,Para(1))
if X2 < X1 do k := -1 * k
Para2 := Rparaline(Para(1),Para(2),width/2,k)
Para4 := Lparaline(Para(1),Para(2),width/2,k)
Para3 := lineint(Para1(1),Para1(2),Para2(1),Para2(2))
Para30 := lineint(Para10(1),Para10(2),Para2(1),Para2(2))
Para5 := lineint(Para1(1),Para1(2),Para4(1),Para4(2))
Para50 := lineint(Para10(1),Para10(2),Para4(1),Para4(2))
let xintersecR1 := Para3(1)
let yintersecR1 := Para3(2)
let xintersecL1 := Para5(1)
let yintersecL1 := Para5(2)
let xintersecR2 := Para30(1)
let yintersecR2 := Para30(2)
let xintersecL2 := Para50(1)
let yintersecL2 := Para50(2)
if type = -1 then {
a.pic := drawline(xintersecR1,yintersecR1,xintersecR2,yintersecR2)
a.pic := a.pic & drawline(xintersecL1,yintersecL1,xintersecL2,yintersecL2)}
else if type = -2 then {
a.pic := dashing(xintersecR1,yintersecR1,xintersecR2,yintersecR2,
dash,gap)
a.pic := a.pic & dashing(xintersecL1,yintersecL1,xintersecL2,
yintersecL2,dash,gap)}
else if type = -3 then {
a.pic := dashing(xintersecR1,yintersecR1,xintersecR2,yintersecR2,
dash,gap)
a.pic := a.pic & drawline(xintersecL1,yintersecL1,xintersecL2,
yintersecL2)}
else if type = -4 do {
a.pic := drawline(xintersecR1,yintersecR1,xintersecR2,yintersecR2)
a.pic := a.pic & dashing(xintersecL1,yintersecL1,xintersecL2,
yintersecL2,dash,gap)}
a.pic
end
let doblines = proc(*real Xs, Ys; real width,type; int themax,dash,gap -> pic )
begin
let xintersecR := vector 1::100 of 0.0
let yintersecR := vector 1::100 of 0.0
let xintersecL := vector 1::100 of 0.0
let yintersecL := vector 1::100 of 0.0
let a.pic1 := [ 0.0, 0.0 ]
let a.picr := [ 0.0, 0.0 ]
let a.pic := [ 0.0, 0.0 ]
let Para := vector 1::2 of 0.0
let Para1 := vector 1::2 of 0.0
let Para2 := vector 1::2 of 0.0
let Para3 := vector 1::2 of 0.0
let Para4 := vector 1::2 of 0.0
let Para5 := vector 1::2 of 0.0
let Para6 := vector 1::2 of 0.0
let Para7 := vector 1::2 of 0.0
let k := 1
for i=1 to themax do
begin
if i < themax then
begin
k:= 1
Para := linepara(Xs(i),Ys(i),Xs(i+1),Ys(i+1))
Para1 := perpenline(Xs(i),Ys(i),Para(1))
if Xs(i+1) < Xs(i) do k := -1 * k
Para2 := Rparaline(Para(1),Para(2),width/2,k)
Para4 := Lparaline(Para(1),Para(2),width/2,k)

```

```

    if i=1 then
        begin
            Para3 := lineint(Para1(1),Para1(2),Para2(1),Para2(2))
            Para5 := lineint(Para1(1),Para1(2),Para4(1),Para4(2))
            xintersecR(i) := Para3(1)
            yintersecR(i) := Para3(2)
            xintersecL(i) := Para5(1)
            yintersecL(i) := Para5(2)
            Para6 := Para2
            Para7 := Para4
        end
    else
        begin
            Para3 := lineint(Para2(1),Para2(2),Para6(1),Para6(2))
            Para5 := lineint(Para4(1),Para4(2),Para7(1),Para7(2))
            Para6 := Para2
            Para7 := Para4
            xintersecR(i) := Para3(1)
            yintersecR(i) := Para3(2)
            xintersecL(i) := Para5(1)
            yintersecL(i) := Para5(2)
        end
    end
else
    if i=themax do
        begin
            Para1 := perpenline(Xs(i),Ys(i),Para6(1))
            Para3 := lineint(Para1(1),Para1(2),Para6(1),Para6(2))
            Para5 := lineint(Para1(1),Para1(2),Para7(1),Para7(2))
            xintersecR(i) := Para3(1)
            yintersecR(i) := Para3(2)
            xintersecL(i) := Para5(1)
            yintersecL(i) := Para5(2)
        end
    end
end
for i=1 to themax-1 do
    begin
        if type = -1 then {
            a.picr := drawline(xintersecR(i),yintersecR(i),xintersecR(i+1),
                               yintersecR(i+1))
            a.picl := drawline(xintersecL(i),yintersecL(i),xintersecL(i+1),
                               yintersecL(i+1))
            a.pic := if i=1 then a.picr & a.picl
                     else a.pic & a.picr & a.picl}
        else if type = -2 then {
            a.picr := dashing(xintersecR(i),yintersecR(i),xintersecR(i+1),
                               yintersecR(i+1),dash,gap)
            a.picl := dashing(xintersecL(i),yintersecL(i),xintersecL(i+1),
                               yintersecL(i+1),dash,gap)
            a.pic := if i=1 then a.picr & a.picl
                     else a.pic & a.picr & a.picl}
        else if type = -3 then {
            a.picr := dashing(xintersecR(i),yintersecR(i),xintersecR(i+1),
                               yintersecR(i+1),dash,gap)
            a.picl := drawline(xintersecL(i),yintersecL(i),xintersecL(i+1),
                               yintersecL(i+1))
            a.pic := if i=1 then a.picr & a.picl
                     else a.pic & a.picr & a.picl}
        else if type = -4 do {
            a.picr := drawline(xintersecR(i),yintersecR(i),xintersecR(i+1),
                               yintersecR(i+1))
            a.picl := dashing(xintersecL(i),yintersecL(i),xintersecL(i+1),
                               yintersecL(i+1),dash,gap)
            a.pic := if i=1 then a.picr & a.picl
                     else a.pic & a.picr & a.picl}
        end
    end
a.pic
end
let ddline = proc(*real Xs, Ys; real width; int themax,dash,gap -> pic )
begin
    let k := width
    let a.pic := [ 0.0, 0.0 ]
    let a.picl := [ 0.0, 0.0 ]
    for i=1 to themax-1 do
        begin

```



```

        while k ~= 0 do
            begin
                a.pic1 := if k= width then dblseg(Xs(i),Ys(i),Xs(i+1),Ys(i+1),
                                                    k,-2,dash,gap)
                        else a.pic1 & dblseg(Xs(i),Ys(i),Xs(i+1),Ys(i+1),
                                                    k,-2,dash,gap)
                k := k - 0.5
            end
            a.pic := if i=1 then a.pic1
                    else a.pic & a.pic1
            k := width
        end
    end
a.pic
end
let thkline = proc(*real Xarray, Yarray; int breadth, thetop -> pic )
begin
    let a.pic1 := [ 0.0, 0.0 ]
    let a.pic2 := [ 0.0, 0.0 ]
    let a.pic := [ 0.0, 0.0 ]
    for j=1 to thetop-1 do
        a.pic1 := if j=1 then drawline(Xarray(j),Yarray(j),Xarray(j+1),
                                        Yarray(j+1))
                else a.pic & drawline(Xarray(j),Yarray(j),Xarray(j+1),
                                        Yarray(j+1))
    for i=1 to thetop-1 do {
        let k := half
        while k > 0 do
            {a.pic2 := if k = half then dblseg(Xarray(i),Yarray(i),Xarray(i+1),
                                                Yarray(i+1),k, -1,-1,-1)
            else a.pic2 & dblseg(Xarray(i),Yarray(i),Xarray(i+1),
                                                Yarray(i+1),k, -1,-1,-1)
            k := k - 0.25}
            a.pic := if i=1 then a.pic1 & a.pic2
                    else a.pic & a.pic1 & a.pic2 }
        a.pic
    end
let railine = proc(*real Xarray, Yarray; int breadth, thetop -> pic )
begin
    let a.pic := [ 0.0, 0.0 ]
    let half := breadth div 2
    for i= 1 to half do
        a.pic := ddline(Xarray,Yarray,i, thetop,6,6)
    a.pic := a.pic & doblne(Xarray,Yarray,breadth, -1, thetop, -1, -1)
    a.pic
end
let bordering = proc(real x1,y1,x2,y2,dash,gap -> pic )
begin
    let alpha := 0.0
    let a.pic := [ 0.0, 0.0 ]
    let period := dash + gap
    let dy := y2 - y1
    let dx := x2 - x1
    let lengthoffline := sqrt( dx * dx + dy * dy )
    if dx = 0 then
        if dy > 0 then alpha := pi/2
        else alpha := pi
    else
        begin
            let k := 1; let c := 1
            let NoOfPeriods := 0; let thegap := 0.0; let theperiod := 0.0
            alpha := atan(rabs(dy/dx))
            if dx < 0 do k := -k
            if dy < 0 do c := -c
            if lengthoffline > period then
                begin
                    let Periods := truncate( lengthoffline / period )
                    NoOfPeriods := Periods - truncate(Periods/3)
                    let rest := lengthoffline - NoOfPeriods * period
                    thegap := rest / NoOfPeriods + gap
                    theperiod := thegap + dash
                    let X1 := vector 1::(NoOfPeriods+2) of 0.0
                    let Y1 := vector 1::(NoOfPeriods+2) of 0.0
                    let X2 := vector 1::(NoOfPeriods+2) of 0.0
                    let Y2 := vector 1::(NoOfPeriods+2) of 0.0

```

```

let thecos := cos(alpha)
let thesin := sin(alpha)
let dxgap := thecos * thegap
let dxdash := thecos * dash
let dxdashp := thecos
let dxhdash := 0.5 * dxdash
let dygap := thesin * thegap
let dydash := thesin * dash
let dydashp := thesin
let dyhdash := 0.5 * dydash
X1(1) := x1
Y1(1) := y1
X2(1) := x1 + dxhdash * k
Y2(1) := y1 + dyhdash * c
a.pic := drawline( X1(1), Y1(1), X2(1), Y2(1) )
for i = 2 to NoOfPeriods do
  begin
    if i rem 2 = 0 then {
      X1(i) := X2(i-1) + dxgap * k
      Y1(i) := Y2(i-1) + dygap * c
      X2(i) := X1(i) + dxdashp * k
      Y2(i) := Y1(i) + dydashp * c
      a.pic := a.pic & drawline( X1(i), Y1(i), X2(i), Y2(i) ) }
    else {
      X1(i) := X2(i-1) + dxgap * k
      Y1(i) := Y2(i-1) + dygap * c
      X2(i) := X1(i) + dxdash * k
      Y2(i) := Y1(i) + dydash * c
      a.pic := a.pic & drawline( X1(i), Y1(i), X2(i), Y2(i) ) }
    end
    X1(NoOfPeriods+1) := X2(NoOfPeriods) + dxgap * k
    Y1(NoOfPeriods+1) := Y2(NoOfPeriods) + dygap * c
    X2(NoOfPeriods+1) := X1(NoOfPeriods+1) + dxdashp * k
    Y2(NoOfPeriods+1) := Y1(NoOfPeriods+1) + dydashp * c
    a.pic := a.pic & drawline( X1(NoOfPeriods+1), Y1(NoOfPeriods+1),
      X2(NoOfPeriods+1), Y2(NoOfPeriods+1) )
    X1(NoOfPeriods+2) := X2(NoOfPeriods+1) + dxgap * k
    Y1(NoOfPeriods+2) := Y2(NoOfPeriods+1) + dygap * c
    X2(NoOfPeriods+2) := x2
    Y2(NoOfPeriods+2) := y2
    a.pic := a.pic & drawline( X1(NoOfPeriods+2), Y1(NoOfPeriods+2),
      X2(NoOfPeriods+2), Y2(NoOfPeriods+2) )
  end
else a.pic := drawline(x1,y1,x2,y2)
end
a.pic
end
let linetype = proc(*real theX, theY; int size -> pic )
begin
  let xstart := X.dim(screen)-210
  let menuwin = limit screen to 195 by 400 at xstart-5,115
  copy LineMenu onto menuwin
  let a.pic := [ 0.0, 0.0 ]
  let xpos := X.dim(screen)-115
  let xpos1 := X.dim(screen)-205
  let xloc := xpos + 10
  let xloc1 := xstart + 10
  let xend := X.dim(screen)-45
  let xend1 := xloc1 + 90
  let xs := @1 of real [xloc1,xend1]
  let ys := vector 1::2 of 0.0
  let picbox1 = limit screen to 80 by 30 at xpos,350
  let picbox11 = limit screen to 80 by 30 at xpos1,350
  let picbox2 = limit screen to 80 by 30 at xpos,320
  let picbox22 = limit screen to 80 by 30 at xpos1,320
  let picbox3 = limit screen to 80 by 30 at xpos,290
  let picbox33 = limit screen to 80 by 30 at xpos1,290
  let picbox4 = limit screen to 80 by 30 at xpos,260
  let picbox44 = limit screen to 80 by 30 at xpos1,260
  let picbox5 = limit screen to 80 by 30 at xpos,230
  let picbox55 = limit screen to 80 by 30 at xpos1,230
  let picbox6 = limit screen to 80 by 30 at xpos,200
  let picbox66 = limit screen to 80 by 30 at xpos1,200
  let picbox7 = limit screen to 80 by 30 at xpos,170
  let picbox77 = limit screen to 80 by 30 at xpos1,170

```

```

let picbox8 = limit screen to 80 by 30 at xpos,140
let picbox88 = limit screen to 80 by 30 at xpos1,140
let xo := 0
let yo := 0
let done := false
while ~done do
  begin
    let lo := locator()
    while ~lo(the.buttons)(1) do lo := locator()
    xo := lo(X.pos)
    yo := lo(Y.pos)
    while lo(the.buttons)(1) do lo := locator()
    if xo > xloc and xo < xend and yo > 350 and yo < 380 do
      begin
        nor picbox1 onto picbox1
        for i=1 to size-1 do
          a.pic := if i=1 then drawline(theX(i), theY(i), theX(i+1),
                                         theY(i+1) )
                    else a.pic & drawline(theX(i), theY(i), theX(i+1),
                                         theY(i+1) )
        done := true
      end
    if xo > xloc1 and xo < xend1 and yo > 350 and yo < 380 do
      begin
        nor picbox11 onto picbox11
        a.pic := doblin(theX,theY,4,-1,size,-1,-1)
        done := true
      end
    if xo > xloc and xo < xend and yo > 320 and yo < 350 do
      begin
        nor picbox2 onto picbox2
        for i=1 to size-1 do
          a.pic := if i=1 then dashing(theX(i), theY(i), theX(i+1),
                                         theY(i+1),10,5)
                    else a.pic & dashing(theX(i), theY(i), theX(i+1),
                                         theY(i+1),10,5)
        done := true
      end
    if xo > xloc1 and xo < xend1 and yo > 320 and yo < 350 do
      begin
        nor picbox22 onto picbox22
        a.pic := doblin(theX,theY,4,-2,size,10,5)
        done := true
      end
    if xo > xloc and xo < xend and yo > 290 and yo < 320 do
      begin
        nor picbox3 onto picbox3
        for i=1 to size-1 do
          a.pic := if i=1 then dashing(theX(i), theY(i), theX(i+1),
                                         theY(i+1),4,4)
                    else a.pic & dashing(theX(i), theY(i), theX(i+1),
                                         theY(i+1),4,4)
        done := true
      end
    if xo > xloc1 and xo < xend1 and yo > 290 and yo < 320 do
      begin
        nor picbox33 onto picbox33
        a.pic := doblin(theX,theY,4,-3,size,10,5)
        done := true
      end
    if xo > xloc and xo < xend and yo > 260 and yo < 290 do
      begin
        nor picbox4 onto picbox4
        for i=1 to size-1 do
          a.pic := if i=1 then dashing(theX(i), theY(i), theX(i+1),
                                         theY(i+1),12,2)
                    else a.pic & dashing(theX(i), theY(i), theX(i+1),
                                         theY(i+1),12,2)
        done := true
      end
    if xo > xloc1 and xo < xend1 and yo > 260 and yo < 290 do
      begin
        nor picbox44 onto picbox44
        a.pic := doblin(theX,theY,4,-4,size,10,5)
        done := true
      end

```

```

end
if xo > xloc and xo < xend and yo > 230 and yo < 260 do
begin
nor picbox5 onto picbox5
for i=1 to size-1 do
a.pic := if i=1 then dashing(theX(i), theY(i), theX(i+1),
theY(i+1),2,4)
else a.pic & dashing(theX(i), theY(i), theX(i+1),
theY(i+1),2,4)
done := true
end
if xo > xloc1 and xo < xend1 and yo > 230 and yo < 260 do
begin
nor picbox55 onto picbox55
a.pic := thkline(theX, theY,4,size)
done := true
end
if xo > xloc and xo < xend and yo > 200 and yo < 230 do
begin
nor picbox6 onto picbox6
for i=1 to size-1 do
a.pic := if i=1 then dashing(theX(i), theY(i), theX(i+1),
theY(i+1),1,2)
else a.pic & dashing(theX(i), theY(i), theX(i+1),
theY(i+1),1,2)
done := true
end
if xo > xloc1 and xo < xend1 and yo > 200 and yo < 230 do
begin
nor picbox66 onto picbox66
a.pic := dblline(theX,theY,4,-2,size,1,4)
done := true
end
if xo > xloc and xo < xend and yo > 170 and yo < 200 do
begin
nor picbox7 onto picbox7
for i=1 to size-1 do
a.pic := if i=1 then dashing(theX(i), theY(i), theX(i+1),
theY(i+1),1,4)
else a.pic & dashing(theX(i), theY(i), theX(i+1),
theY(i+1),1,4)
done := true
end
if xo > xloc1 and xo < xend1 and yo > 170 and yo < 200 do
begin
nor picbox77 onto picbox77
a.pic := ddline(theX, theY, 4, size,6,6)
done := true
end
if xo > xloc and xo < xend and yo > 140 and yo < 170 do
begin
nor picbox8 onto picbox8
for i=1 to size-1 do
a.pic := if i=1 then bordering(theX(i), theY(i), theX(i+1),
theY(i+1),10,4)
else a.pic & bordering(theX(i), theY(i), theX(i+1),
theY(i+1),10,4)
done := true
end
if xo > xloc1 and xo < xend1 and yo > 140 and yo < 170 do
begin
nor picbox88 onto picbox88
a.pic := railine(theX, theY, 4, size)
done := true
end
end
let clrwin = limit screen to 205 by 450 at X.dim(screen)-215,120
xor clrwin onto clrwin
a.pic
end
let Sht := X.dim(screen) - 540
let xstart := 10; let ystart := 10
let xend := X.dim(screen)- 10; let yend := Y.dim(screen) - 53
let xspan := xend - xstart; let yspan := yend - ystart
let TotalYmenWin := yend-ystart-97

```

```

let Screen = limit screen to xspan by yspan at xstart, ystart
let xlcorner := xstart + 4; let ylcorner := ystart+101
let X.G := xspan -244
let Y.G := TotalYmenWin-2
if X.G > Y.G then X.G := Y.G
                else Y.G := X.G
let GW = limit screen to X.G by Y.G at xlcorner, ylcorner
let xmenustart := X.dim(screen) - 250
let menuwin = limit screen to 236 by TotalYmenWin-3 at xmenustart,ylcorner
let menusaved = image 217 by 224 of off
let out.range := X.dim(screen) - Y.dim(screen)
let range := X.dim(screen) - out.range
let gw = limit screen to range by range at 0, 0
let x.coord = vector 1::100 of 0.0
let y.coord = vector 1::100 of 0.0
let dummy.vec := vector 1::1000 of ""
let winsave := image X.G by Y.G of off
let XOL := 0.0; let XOR := 0.0
let YOL := 0.0; let YOR := 0.0
let XZL := 0.0; let XZR := 0.0
let YZL := 0.0; let YZR := 0.0
let TheGrid := 0.0
let thegrid := [ 0.0, 0.0 ]
let xt := range + 100; let yt := 271!Global
let tmenu = limit screen to 102 by 192 at xt-1, yt-1!Global
let xe := xt + 40; let ye := yt + 46!Global
let emenu = limit screen to 112 by 102 at xe-1, ye-1!Global
let xl := xt - 90; let yl := 100!Global
let lmenu = limit screen to 142 by 402 at xl-1, yl-1!Global
let eimage := s.lookup("Entity Menu", theimage)(menuimage)
let timage := s.lookup("Type Menu", theimage)(menuimage)
let limage := s.lookup("Layer Menu", theimage)(menuimage)
let def.menu := "1"
let zoomcounter := 0
structure feature(pic fea.pic )
let Fonting = proc(int x, y -> string )
begin
    let xstart := 200; let ystart := 250
    let twinx := 500; let twiny := 400
    let Fwin = limit screen to 45 by 303 at xstart+369,ystart+20
    let fvec1 = @1 of string [ "2", "6", "5", "4", "3", "2", "1" ]
    let fvec2 = @1 of string [ "11", "10", "7", "14", "9", "8", "2" ]
    let fwin := limit screen to 47 by 316 at 19, 19
    let Fim1 := s.lookup("Font Menu1",theimage)(menuimage)
    let Fim2 := s.lookup("Font Menu2",theimage)(menuimage)
    let twin := limit screen to twinx by twiny at xstart, ystart
    let text.win = limit screen to 350 by 80 at 47+xstart,147+ystart
    let Fbox = limit screen to 47 by 33 at xstart+368, ystart+321
    let Fim3 := s.lookup("Fone Title", theimage)(menuimage)
    let afont := "14"
    let atext := ""
    let fontype := true
    let text.writing = proc(int xpos,ypos;string name,font;#pixel anyimage)
        begin
            let Font := ""
            case true of
                font = "1" : Font := "fix09"
                font = "2" : Font := "ngr13"
                font = "3" : Font := "fix13"
                font = "4" : Font := "fixb13"
                font = "5" : Font := "gac16n"
                font = "6" : Font := "gacha16"
                font = "7" : Font := "met22"
                font = "8" : Font := "ngi20"
                font = "9" : Font := "non22"
                font = "10" : Font := "olde25"
                font = "11" : Font := "hci45i"
                default : Font := "cou20"
            copy string.to.tile(name,Font) onto limit anyimage at xpos,ypos
        end
    let chosefont = proc(bool Type -> string)
        begin
            let Fonts := ""
            let done := false
            while ~done do {

```

```

if Type then {
  copy Fim1 onto Fwin
  let loc := locator()
  while ~loc(the.buttons)(1) do loc := locator()
  while loc(the.buttons)(1) do loc := locator()
  let AX := loc(X.pos)-(xstart+369); let AY := loc(Y.pos)-(
    ystart+20)
  if AX > 0 and AX < 44 and AY > 0 and AY < 301 do {
    let ylocation := truncate(AY / 43)
    let thebox := limit screen to 44 by 44 at xstart+369,
      ystart+20+ylocation*43
    nor thebox onto thebox
    Fonts := case true of
      ylocation = 0 : "More"
      ylocation = 1 : "6"
      ylocation = 2 : "5"
      ylocation = 3 : "4"
      ylocation = 4 : "3"
      ylocation = 5 : "2"
      default : "1"
    xor Fwin onto Fwin
    if Fonts ~= "More" do
      copy Fim3 onto Fbox
      done := true } }
else {
  copy Fim2 onto Fwin
  let loc := locator()
  while ~loc(the.buttons)(1) do loc := locator()
  while loc(the.buttons)(1) do loc := locator()
  let AX := loc(X.pos)-(xstart+369); let AY := loc(Y.pos)-(
    ystart+20)
  if AX > 0 and AX < 44 and AY > 0 and AY < 301 do {
    let ylocation := truncate(AY / 43)
    let thebox := limit screen to 44 by 44 at xstart+369,
      ystart+20+ylocation*43
    nor thebox onto thebox
    Fonts := case true of
      ylocation = 2 : "7"
      ylocation = 5 : "8"
      ylocation = 4 : "9"
      ylocation = 1 : "10"
      ylocation = 3 : "14"
      ylocation = 6 : "Back"
      default : "11"
    xor Fwin onto Fwin
    if Fonts ~= "Back" do
      copy Fim3 onto Fbox
      done := true }
  }
}
!write Fonts
Fonts
end
let fonting = proc( string anytext -> string)
begin
  afont := chosefont(fonttype)
  while afont = "More" or afont = "Back" do
    { fonttype := ~fonttype
      afont := chosefont(fonttype) }
  xor text.win onto text.win
  text.writing(50, 150, anytext, afont, twin)
  afont
end
let retrieve = proc(-> string)
begin
  let tim = image twinx by twiny of off
  copy twin onto tim
  xor twin onto twin
  rec(xstart+2, ystart+2, twinx-4, twiny-4)
  rec(xstart+3, ystart+3, twinx-6, twiny-6)
  rec(xstart+5, ystart+5, twinx-10, twiny-10)
  rec(xstart+6, ystart+6, twinx-12, twiny-12)
  let xdone := icon(xstart+50, ystart+50, "OK")
  let xcanc := icon(xstart+250, ystart+50, "Cancel")
  copy Fim3 onto Fbox

```

```

let fbox = limit screen to xdone by 30 at xstart+50, ystart+50
let sbox = limit screen to xdone by 30 at xstart+250, ystart+50
atext := seditor("Enter Text", "", xstart+50, ystart+150, 200, 50 )
text.writing(50, 150, atext, afont, twin)
let done := false
while ~done do {
  let lo := locator()
  while ~lo(the.buttons)(1) do lo := locator()
  let axn := lo(X.pos); let ayn := lo(Y.pos)
  !let showplace := limit screen to 80 by 30 at axn-40, ayn-15
  !nor showplace onto showplace
  while lo(the.buttons)(1) do lo := locator()
  let ax := lo(X.pos); let ay := lo(Y.pos)
  !nor showplace onto showplace
  if ax > xstart+50 and ax < xstart+50+xdone and ay > ystart+50 and
    ay < ystart+80 then
    { nor fbox onto fbox
      done := true }
  else
    if ax > xstart+250 and ax < xstart+250+xdone and ay > ystart+50 and
      ay < ystart+80 then
      { nor sbox onto sbox
        atext := seditor("Enter Text", "", xstart+50, ystart+150, 200, 50 )
        xor text.win onto text.win
        text.writing(50, 150, atext, "14", twin)
        nor sbox onto sbox
        rec(xstart+370, ystart+323, 44, 30) }
      else
        if ax > xstart+375 and ax < xstart+419 and ay > ystart+326 and
          ay < ystart+356 do
          { nor Fbox onto Fbox
            afont := fonting(atext)
            !nor Fbox onto Fbox
          }
        }
      !write
      afont
    end
    let thefont := retrieve()
    text.writing(x, y, atext, thefont, gw)
    thefont
  end
end
let first.screen = proc()
begin
  Rec(0,0,X.dim(screen),Y.dim(screen),"Ps - GIS","cou20",
    "middle", true)
  let border1 = limit screen to X.dim(screen)-10 by Y.dim(screen)-43 at 5,5
  let border2 = limit screen to X.dim(screen)-20 by Y.dim(screen)-53 at 10,10
  nor border1 onto border1
  nor border2 onto border2
end
let text.window = proc()
begin
  let fixtext = limit screen to 140 by 95 at 13,13
  let yposition := 75
  copy string.to.tile("FEATURE      :", "fixb13") onto limit fixtext at 10,
    yposition
  copy string.to.tile("SEGMENTS    :", "fixb13") onto limit fixtext at 10,
    yposition - 17
  copy string.to.tile("CODE        :", "fixb13") onto limit fixtext at 10,
    yposition - 34
  copy string.to.tile("LEFT COORDS  :", "fixb13") onto limit fixtext at 10,
    yposition - 51
  copy string.to.tile("RIGHT COORDS :", "fixb13") onto limit fixtext at 10,
    yposition - 68
  nor fixtext onto fixtext
end
let zoomout = proc(pic PIC, thegrid; int Range, xshft, yshft; #pixel the.win )
begin
  if XZL = 0 and XZR = 0 then
    error.message(" Nothing to Zoom Out ", -1, -1 )
  else
    begin
      xor the.win onto the.win
      Box( xshft, yshft, Range)
    end
  end
end

```

```

        Box( xshft+1, yshft+1, Range-2)
        draw(the.win, thegrid, XOL, XOR, YOL, YOR )
        draw(the.win, PIC, XOL, XOR, YOL, YOR )
        XZL := 0.0; XZR := 0.0
        YZL := 0.0; YZR := 0.0
    end
end
let zoomin = proc(pic PIC, thegrid;int Range, count,xshft,yshft;#pixel the.win )
begin
    let xl := 0.0; let xr := 0.0
    let yl := 0.0; let yr := 0.0
    if count < 2 then {
        xl := XOL; xr := XOR
        yl := YOL; yr := YOR }
    else {
        xl := XZL; xr := XZR
        yl := YZL; yr := YZR }
    let xo := 0; let xe := 0
    let yo := 0; let ye := 0
    let lo := locator()
    while ~lo(the.buttons)(1) do lo := locator()
    xo := lo(X.pos) - xshft
    yo := lo(Y.pos) - yshft
    while lo(the.buttons)(1) do lo := locator()
    xe := lo(X.pos) - xshft
    ye := lo(Y.pos) - yshft
    if xo > xe do {
        let t := xo
        xo := xe
        xe := t }
    if yo < ye do {
        let t := yo
        yo := ye
        ye := t }
    let xdif := rabs( xe - xo )
    let ydif := rabs( ye - yo )
    let adif := truncate( ( xdif + ydif ) / 2 )
    xe := xo + adif
    ye := yo - adif
    XZL := xl + (xr - xl) * xo / Range
    XZR := xl + (xr - xl) * xe / Range
    YZL := yl + (yr - yl) * ye / Range
    YZR := yl + (yr - yl) * yo / Range
    xor the.win onto the.win
    Box( xshft, yshft, Range)
    Box( xshft+1, yshft+1, Range-2)
    draw(the.win, thegrid, XZL, XZR, YZL, YZR )
    draw( the.win, PIC, XZL, XZR, YZL, YZR )
end
let change.code = proc(-> int)
begin
    let thecode := 0
    let former.image = image 238 by TotalYmenWin-3 of off
    copy menuwin onto former.image
    xor menuwin onto menuwin
    let ystartdetail := yend-180
    let detail = limit screen to 238 by 180 at xmstart,ystartdetail
    rec(xmstart,ystart+100,238,TotalYmenWin)
    Rec(xmstart,ystartdetail,238,180,"Feature Details","fixb13","middle",true)
    let classification = @ 1 of string["Class      :", "Category  :",
                                      "Feature   :",
                                      "Attribute :", "The Code  :"]

    for i=1 to 5 do {
        let ypos := if i=1 then 110
                     else if i=2 then 85
                     else if i=3 then 60
                     else if i=4 then 35
                     else 10
        text.write( 10, ypos,classification(i),"fixb13",detail) }
    let VecNames = vector 1::100 of ""
    let VecNums = vector 1::100 of 0
    let N := 0
    let procl = proc(int I; pntr V -> bool)
    begin
        N := N + 1

```



```

    VecNames(N) := V(identifier)
    VecNums(N) := I
    true
end
let stopchase := false
let cat = @1 of string ["Class","Category","Feature","Attribute"]
let choices = vector 1::4 of ""
let catWidth = X.dim(screen) div 4
let choose := proc(int L; pntr T); nullproc
choose := proc(int L; pntr T)
begin
    let theMenu := s.lookup( "menu",T)
    if theMenu = nil do
        begin
            N := 0
            let X := i.scan(T,procl)
            let S := ""
            let levelname = vector 1::N of ""
            let levelnum = vector 1::N of 0
            for i=1 to N do
                { levelname(i) := VecNames(i);
                  levelnum(i) := VecNums(i) }
            for i=1 to N-1 do for j=i+1 to N do
                { X := levelnum(i); levelnum(i) := levelnum(j);
                  levelnum(j) := X; S := levelname(i);
                  levelname(i) := levelname(j); levelname(j) := S }
            for i=1 to N do
                begin
                    if length(levelname(i)) > 25 do
                        levelname(i) := levelname(i) (1|25)
                    if levelname(i) = "" do levelname(i) := "-----"
                end
                let ch = set.up.choose(levelname) (do.choose)
                theMenu := menuPack(ch, levelname, levelnum)
                s.enter ( "menu", T, theMenu)
                if commit() ~= nil do print "COMMIT FAILS" at 50,300
            end
            let ch = theMenu(menuProc)
            let levelname = theMenu(Lname)
            let levelnum = theMenu(Lnum)
            N := upb(levelname)
            let choice = ch("Choose " ++ cat(L), X.dim(screen)-240,120)
            if choice = "" do stopchase := true
            let chint := 0
            if choice ~= "" do {
                for i=1 to N do
                    if choice = levelname(i) do chint := levelnum(i)
                choices(L) := choice
                thecode := if L = 1 then chint * 10000000
                           else if L = 2 then thecode + chint * 100000
                           else if L = 3 then thecode + chint * 1000
                           else thecode + chint
                let ypos := if L=1 then 110
                           else if L=2 then 85
                           else if L=3 then 60
                           else 35
                text.write( 119, ypos,choices(L),"fixb13",detail) }
                while ~stopchase and L < 4 do choose( L+1, i.lookup(chint,T) (subtree)
            end
            let levell := s.lookup("Code2",DB)
            choose( 1, levell)
            print thecode at X.dim(screen)-120,ystartdetail+10
            copy former.image onto menuwin
            thecode
        end
    end
let text.entry = proc(string typeoftext,previous.string -> string)
begin
    let awindow := limit screen to 740 by 150 at 40,140
    let savedimage = image 750 by 150 of off
    copy awindow onto savedimage
    xor awindow onto awindow
    rec (41,141,738,148)
    rec (42,142,736,146)
    copy string.to.tile ("Enter The ","fixb13") onto limit awindow at 50,110
    copy string.to.tile (typeoftext,"fixb13") onto limit awindow at 150,110

```

```

copy string.to.tile ("(not longer than 43 characters including spaces)",
                    "fix13") onto limit awindow at 50,80
write code(27),"N",code(27),"I"
let atext := ""
atext := seditor(" ",previous.string,50, 150,710,50)
if length(atext) > 43 do
    atext := atext(1|43)
write code(27),"E",code(27),"W"
copy savedimage onto awindow
atext
end
let centroid = proc(*real avector1,avector2; int the.size;real Xmin,Xmax,Ymin,
                    Ymax -> *real)
begin
    let xloc := xmstart + 20
    let thismenu = limit screen to 217 by 224 at xmstart+9,119
    let msg.menu = limit screen to 100 by 40 at xmstart+9,159
    rec(xmstart+10,120,215,180)
    rec(xmstart+11,121,213,178)
    text.write(10,150,"Chose the centroid of","fixb13",thismenu)
    text.write(10,120,"this area by clicking","fixb13",thismenu)
    text.write(10,90,"the left button of the","fixb13",thismenu)
    text.write(10,60,"mouse on the desired ","fixb13",thismenu)
    text.write(10,30,"location of polygon","fixb13",thismenu)
    let xminmax := minmax(avector1,the.size)
    let xmin := xminmax(1); let xmax := xminmax(2)
    let yminmax := minmax(avector2,the.size)
    let ymin := yminmax(1); let ymax := yminmax(2)
    let counter := 1
    let lo := locator()
    let coords = vector 1::2 of 0.0
    let proceed := false
    while ~proceed do
        begin
            if counter > 1 do
                begin
                    rec(xmstart+10,160,98,38)
                    text.write(10,15,"Try again","fix13",msg.menu)
                end
                while ~lo(the.buttons)(1) do lo := locator()
                let xo := lo(X.pos) - xlcorner
                let yo := lo(Y.pos) - ylcorner
                coords(1) := Xmin + (Xmax - Xmin) * xo / X.G
                coords(2) := Ymin + (Ymax - Ymin) * yo / Y.G
                while lo(the.buttons)(1) do lo := locator()
                if coords(1) < xmax and coords(1) > xmin do
                    if coords(2) < ymax and coords(2) > ymin do
                        { proceed := true
                          counter := counter + 1 }
                    xor msg.menu onto msg.menu
                end
                xor thismenu onto thismenu
                coords
            end
        end
    let picking = proc(*string givens; int areacounter,linecounter, pointcounter)
    begin
        let thevar := s.lookup("Variables",DBvar)
        if thevar = nil do
            thevar := table()
        text.window()
        let xmin := 0.0; let xmax := 0.0
        let ymin := 0.0; let ymax := 0.0
        if XZR = 0 and YZR = 0 then {
            xmin := XOL; xmax := XOR
            ymin := YOL; ymax := YOR }
        else {
            xmin := XZL; xmax := XZR
            ymin := YZL; ymax := YZR }
        let yposition := 75
        let PIC := [ 0.0 , 0.0 ]
        let PIC1 := [ 0.0 , 0.0 ]
        let dummy.pic := [ 0.0 , 0.0 ]
        let Dummy.vec := @1 of int[0]
        let feature.name := ""
        let s.code := ""; let featype := ""

```

```

let feature.count := areacounter + linecounter + pointcounter
let local.counter := 0
if feature.count > 0 do
  begin
    let atable := s.lookup("Maps", mainDB )
    let Avec := s.lookup(givens(2),atable) (AST)
    let Lvec := s.lookup(givens(2),atable) (LST)
    let Pvec := s.lookup(givens(2),atable) (PST)
    let Atop := upb(Avec)
    let Ltop := upb(Lvec)
    let Ptop := upb(Pvec)
    let thetop := 0
    for i= 1 to Atop do
      Ainfo(i) := Avec(i)
    thetop := Atop
    for i= 1 to Ltop do
      Linfo(i) := Lvec(i)
    thetop := thetop + Ltop
    for i= 1 to Ptop do
      Pinfo(i) := Pvec(i)
    thetop := thetop + Ptop
  end
let the.centroid := vector 1::2 of 0.0
let previous.text := ""
let xdim := X.dim(screen) - 340 - xlcorner
let detail.window = limit screen to xdim by 95 at 153,14
let ind := 1; let typecount := 1
let exist := true
let out.of.range := false
let fe = open("temp2",0)
if fe = nullfile do
  exist := false
if exist do
  begin
    let dummy := system("rm temp1")
    dummy := system("touch temp1")
    dummy := system("cp temp2 temp1")
    dummy := system("rm temp2")
    close (fe)
  end
let fd = open("temp1",0)
if fd = nullfile do
  { write "the file temp1 cannot be opened"; abort }
let tempx = vector 1::100 of 0.0
let tempy = vector 1::100 of 0.0
let nfeatures := 0
let KL := 1
let L := read.a.line(fd); dummy.vec(KL) := L; KL := KL + 1
L := read.a.line(fd); dummy.vec(KL) := L; KL := KL + 1
L := read.a.line(fd); dummy.vec(KL) := L; KL := KL + 1
let thepos := KL
let readlines = proc(string aline -> *string)
  begin
    while ~eoi(fd) and aline(1|1) = "/" do
      begin
        aline := read.a.line(fd); aline := read.a.line(fd)
        while digit( aline(1|1) ) and ~eoi(fd) do
          aline := read.a.line(fd)
        end
let stemp = vector 1::100 of ""
let con := 1
stemp(con) := aline
L := aline
let P := 0
let acounter := 0
if ~eoi(fd) and ( L(13|1) = "*" or L(14|1) = "*" ) do
  begin
    con := con + 1
    let Q := 1
    while L(Q|1) ~= "*" do Q := Q + 1
    let t.length := length(L) - Q - 2
    feature.name := L(Q+2|t.length)
    L := read.a.line(fd); stemp(con) := L; con := con + 1
    s.code := L(5|length(L)-4)
    L := read.a.line(fd)
  end

```

```

    if eoi(fd) then
        begin
            P := 1
            while L(P|1) ~= " " do P := P + 1
            acounter := acounter + 1
            tempx(acounter) := stringtoreal( L( 1 | P) )
            tempy(acounter) := stringtoreal(L( P+1 | length(L) - P))
            stemp(con) := L
        end
    else
        begin
            let i := 0
            while digit( L(1|1) ) and ~eoi(fd) do
                begin
                    i := i + 1
                    P := 1
                    while L(P|1) ~= " " do P := P + 1
                    tempx(i) := stringtoreal( L( 1 | P) )
                    tempy(i) := stringtoreal( L( P+1 | length(L) - P ))
                    stemp(con) := L; con := con + 1
                    L := read.a.line(fd)
                    acounter := i
                end
            if eoi(fd) do {
                P := 1
                while L(P|1) ~= " " do P := P + 1
                acounter := acounter + 1
                tempx(acounter) := stringtoreal( L( 1 | P) )
                tempy(acounter) := stringtoreal(L(P+1 | length(L) - P)
                stemp(con) := L }
            end
        end
        let thisvector = vector 1::acounter+2 of ""
        for k=1 to acounter+2 do
            thisvector(k) := stemp(k)
        thisvector
    end
let done := false
L := read.a.line(fd)
while ~eoi(fd) and ~done do
    begin
        let temporal := readlines(L)
        let thesize := upb(temporal)
        let isin := true
        let checked := false
        let vecsize := thesize - 2
        if vecsize <= 0 do done := true
        let tempscreen = image X.G by Y.G of off
        copy GW onto tempscreen
        let Xs := vector 1::vecsize of 0.0
        let Ys := vector 1::vecsize of 0.0
        for avar = 1 to vecsize do
            begin
                Xs(avar) := tempx(avar)
                Ys(avar) := tempy(avar)
            end
        if vecsize = 1 then
            isin := if Xs(1) < xmax and Xs(1) > xmin and
                    Ys(1) < ymax and Ys(1) > ymin then true
                    else false
        else
            begin
                let Xminmax := minmax(Xs,vecsize)
                let Yminmax := minmax(Ys,vecsize)
                isin := if Xminmax(1) > xmin and Xminmax(2) < xmax and
                        Yminmax(1) > ymin and Yminmax(2) < ymax then true
                        else {
                            checked := true
                            checkin(Xs,Ys,xmin,xmax,ymin,ymax,vecsize) }
            end
        if isin then {
            local.counter := local.counter + 1
            temporal(1) := "/" ++ temporal(1)
            for i=thepos to (thepos + thesize - 1) do
                begin

```

```

        KL := KL + 1
        dummy.vec(i) := temporal(i - thepos + 1)
    end
    if vecsize < 2 then
    begin
        let xposition := truncate( (Xs(1) - xmin) * X.G / (xmax -
                                                                    xmin)) - 5
        let yposition := truncate( (Ys(1) - ymin) * Y.G / (ymax -
                                                                    ymin)) - 5
        text.write( xposition, yposition, "*", "fixb13", GW )
        PIC := [ Xs(1) , Ys(1) ]
    end
    else
    begin
        if ~checked do {
            PIC1 := Highlight(Xs, Ys) !thkline(Xs,Ys,6,vecsize)
            draw(GW,PIC1,xmin,xmax,ymin,ymax) }
        for i=1 to vecsize do
            PIC := if i = 1 then [ Xs(i) , Ys(i) ]
                    else PIC ^ [ Xs(i) , Ys(i) ]
        end
        xor detail.window onto detail.window
        copy string.to.tile(feature.name,"fix13") onto limit
                                                                    detail.window at 25, yposition
        print (vecsize-1) at 180,yposition - 5
        copy string.to.tile(s.code,"fix13") onto limit detail.window at 20
                                                                    yposition - 34
        print xmin,ymin at 163, yposition - 40
        print xmax,ymax at 163, yposition - 57
        let feature.kind := feature.type(typecount)
        typecount := typecount + 1
        let feature.code := change.code()
        if feature.code = 0 do feature.code := change.code()
        let thename := text.entry("Name","")
        let thismenu = limit screen to 217 by 94 at xmstart+9,350
        if feature.kind = "polygon" then
            { the.centroid := centroid(Xs,Ys,vecsize,xmin,xmax,ymin,ymax)
              let a.pic := polygon(the.centroid(1),the.centroid(2),8,"hx")
              draw(GW,a.pic,xmin,xmax,ymin,ymax)
              areacounter := areacounter + 1 }
        else if feature.kind = "line" then
            linecounter := linecounter + 1
        else if feature.kind = "point" do
            pointcounter := pointcounter + 1
        let thetext := text.entry("Comment",previous.text)
        previous.text := thetext
        case true of
        feature.kind = "polygon" : Ainfo(areacounter) := Aholder(
            areacounter, feature.code, false, the.centroid(1),
            the.centroid(2), thename, thetext, Xs, Ys, PIC, 0, 0, 0,
            Dummy.vec, dummy.pic)
        feature.kind = "line" : Linfo(linecounter) := Lholder(linecounter,
            feature.code, false, thename, thetext, Xs, Ys, PIC, 0, 0,
            dummy.pic)
        default : Pinfo(pointcounter) := Pholder(pointcounter,feature.code
            false, thename,thetext,"", Xs, Ys,PIC)
        xor thismenu onto thismenu
        if ~eoi(fd) then
            done := message.proc("Chose what next","Break","Proceed",
                                                                    xend-215, 120, 200, 140)

            else done := true
        xor GW onto GW
        copy tempscreen onto GW
    }
    else {
        for i=thepos to (thepos + thesize - 1) do {
            KL := KL + 1
            dummy.vec(i) := temporal(i-thepos+1) }
        out.of.range := true }
        thepos := KL
    end
    if eoi(fd) then
    begin
        if out.of.range then
        begin

```

```

        let vec.text = @1 of string["This is the end of the file,",
                                     "but there are still some      ",
                                     "features not yet identified.",
                                     "Advice to change zooming.",
                                     "click to proceed"]

        more(vec.text,50,100)
        let a.dummy := system("touch temp2")
        let td = open("temp2",1)
        let ind := 1
        while dummy.vec(ind) ~= "" do
            begin
                output td,dummy.vec(ind),"n"
                dummy.vec(ind) := ""
                ind := ind + 1
            end
        close(td)
        close(fd)
    end
else
    begin
        error.message("Nothing out of range",X.dim(screen)-250,120)
        let a.dummy := system("rm templ; rm config")
        for i=1 to 1000 do
            dummy.vec(i) := ""
        end
    end
end
else {
    dummy.vec(thepos) := L
    let local.count := thepos + 1
    while ~eoi(fd) do
        begin
            dummy.vec(local.count) := read.a.line(fd)
            local.count := local.count + 1
        end
    close(fd)
    let a.dummy := system("touch temp2")
    let td = open("temp2",1)
    for i=1 to local.count-1 do {
        output td,dummy.vec(i),"n"
        dummy.vec(i) := "" }
    close(td)
}
let abase := 1
if areacounter = 0 do abase := 0
let avec := vector abase::areacounter of nil
for i=1 to areacounter do
    avec(i) := Ainfo(i)
let lbase := 1
if linecounter = 0 do lbase := 0
let lvec := vector lbase::linecounter of nil
for i=1 to linecounter do
    lvec(i) := Linfo(i)
let pbase := 1
if pointcounter = 0 do pbase := 0
let pvec := vector pbase::pointcounter of nil
for i=1 to pointcounter do
    pvec(i) := Pinfo(i)
let feat.text := vector 1::1000 of nil
s.enter("a",thevar,globals(areacounter))
s.enter("l",thevar,globals(linecounter))
s.enter("p",thevar,globals(pointcounter))
s.enter("Variables",DBvar,thevar)
if commit() ~= nil do error.message( "Could not store Variables'n", -1,-1)
s.enter(givens(2),mapstable,maps( givens(1), givens(3), givens(11),
    givens(7), XOL, YOL, XOR, YOR, TheGrid, avec, lvec, pvec,
    feat.text ) )
s.enter( "Maps", mainDB, mapstable )
if commit() ~= nil do { error.message("data are not stored'n",-1,-1 ) ;
    abort }
end
let transform = proc(real anx; int range -> real)
begin
    let unx := XZL + (XZR - XZL) * anx / range
    unx
end

```

```

let shiftL = proc(int range; #pixel Win; pic PIC)
begin
    let mrange := range - 102
    let commonpart = image mrange by range of off
    let commonpic = limit screen to mrange by range at 100, 0
    copy commonpic onto commonpart
    xor Win onto Win
    let newin = limit screen to mrange by range at 0, 0
    copy commonpart onto newin
    let xst := transform(mrange+2, range)
    !let exwin := limit screen to 100 by range at mrange, 0
    !nor exwin onto exwin
    let diff := XZR - xst
    XZL := XZL + diff
    XZR := XZR + diff
    draw(gw, PIC, XZL, XZR, YZL, YZR )
end

let shiftR = proc(int range; #pixel Win; pic PIC)
begin
    let mrange := range - 100
    let commonpart = image mrange by range of off
    let commonpic = limit screen to mrange by range at 2, 0
    copy commonpic onto commonpart
    xor Win onto Win
    let newin = limit screen to mrange by range at 100, 0
    copy commonpart onto newin
    let xst := transform(mrange+2, range)
    let diff := XZR - xst
    XZL := XZL - diff
    XZR := XZR - diff
    draw(gw, PIC, XZL, XZR, YZL, YZR )
end

let shiftD = proc(int range; #pixel Win; pic PIC)
begin
    let mrange := range - 34
    let commonpart = image mrange by mrange of off
    let commonpic = limit screen to mrange by mrange at 2, 32
    copy commonpic onto commonpart
    xor Win onto Win
    let newin = limit screen to mrange by mrange at 2, 2
    copy commonpart onto newin
    for i=1 to 2000 do { }
end

let shiftU = proc(int range; #pixel Win; pic PIC)
begin
    let mrange := range - 34
    let commonpart = image mrange by mrange of off
    let commonpic = limit screen to mrange by mrange at 2, 2
    copy commonpic onto commonpart
    xor Win onto Win
    let newin = limit screen to mrange by mrange at 2, 32
    copy commonpart onto newin
    for i=1 to 2000 do { }
end

let Scroll = proc(pic the.pic; int Range; #pixel win )
begin
    let bxl := XZL
    let bxr := XZR
    let byl := YZL
    let byr := YZR
    if XZL = 0 and XZR = 0 then
        error.message("Nothing to Scroll", -1, -1 )
    else {
        !draw( win, the.pic, bxl, bxr, byl, byr )
        let shft := X.dim(screen) - 230
        let xstep = 100
        let ystep = 100
        let xdif := rabs( XZR - XZL )
        let ydif := rabs( YZR - YZL )
        for i=1 to 2 do {
            Box( (shft + (i-1) * 100), 200, 40 )
            Box( (shft + 50), (150 + (i-1) * 100), 40 ) }
        let picbox1 = limit screen to 40 by 40 at shft,200
        let picbox3 = limit screen to 40 by 40 at shft+100,200
        let picbox2 = limit screen to 40 by 40 at shft+50,150
    }
end

```

```

let picbox4 = limit screen to 40 by 40 at shft+50,250
let icon.length := icon(shft+45, 100, "Done" )
let picbox5 = limit screen to icon.length by 30 at shft+45, 100
let vecx = @1 of int [ 30, 30, 10, 10, 0, 10, 10, 30 ]
let vecy = @1 of int [ 10, 22, 22, 32, 16, 0, 10, 10 ]
let VPIC1 := [ 0,0 ]
for i=1 to 8 do
    VPIC1 := if i=1 then [ vecx(i), vecy(i) ]
              else VPIC1 ^ [ vecx(i), vecy(i) ]
let VPIC2 := rotate VPIC1 by -90
let VPIC3 := rotate VPIC2 by -90
let VPIC4 := rotate VPIC3 by -90
draw ( picbox1, VPIC1, -5, 35, -5, 35 )
draw ( picbox2, VPIC2, -35, 5, -5, 35 )
draw ( picbox3, VPIC3, -35, 5, -35, 5 )
draw ( picbox4, VPIC4, -5, 35, -35, 5 )
let xo := 0; let yo := 0
let done := false
while ~done do
    begin
        let lo := locator()
        while ~lo(the.buttons)(1) do lo := locator()
        xo := lo(X.pos)
        yo := lo(Y.pos)
        while lo(the.buttons)(1) do lo := locator()
        if xo > shft + 45 and xo < (shft+45+icon.length) and
           yo > 100 and yo < 130 then
            begin
                nor picbox5 onto picbox5
                done := true
            end
        else if xo > shft and xo < shft + 40 and yo > 200 and
           yo < 240 then
            begin
                nor picbox1 onto picbox1
                bxr := bxr + xstep
                if bxr > XOR do {
                    error.message( "Exceeded limit of map", -1, -1 )
                    bxr := XOR }
                shiftL(Range,win, the.pic)
                !bxl := bxr + xdif
                nor picbox1 onto picbox1
            end
        else if xo > shft + 100 and xo < shft + 140 and
           yo > 200 and yo < 240 then
            begin
                nor picbox3 onto picbox3
                bxl := bxl - xstep
                if bxl < XOL do {
                    error.message( "Exceeded limit of map", -1, -1 )
                    bxl := XOL }
                shiftR(Range,win, the.pic)
                !bxl := bxr - xdif
                nor picbox3 onto picbox3
            end
        else if xo > shft + 50 and xo < shft + 90 and yo > 150 and
           yo < 190 then
            begin
                nor picbox2 onto picbox2
                byl := byl - ystep
                if byl < YOL do {
                    error.message( "Exceeded limit of map", -1, -1 )
                    byl := YOL }
                !byr := byl + ydif
                shiftD(Range,win, the.pic)
                nor picbox2 onto picbox2
            end
        else if xo > shft + 50 and xo < shft + 90 and yo > 250 and
           yo < 290 do
            begin
                nor picbox4 onto picbox4
                byr := byr + ystep
                if byr > YOR do {
                    error.message( "Exceeded limit of map", -1, -1 )
                    byr := YOR }
            end

```



```

        done := true
    end
    if xo > xloc and xo < xend and yo > 180 and yo < 210 do
    begin
        nor picbox6 onto picbox6
        for bb =1 to 2000 do count := count + 1
        dashinfo(1) := 1; dashinfo(2) := 2
        done := true
    end
    if xo > xloc and xo < xend and yo > 150 and yo < 180 do
    begin
        nor picbox7 onto picbox7
        for bb =1 to 2000 do count := count + 1
        dashinfo(1) := 1; dashinfo(2) := 4
        done := true
    end
    end
    let mnuwin = limit screen to 240 by 350 at X.dim(screen)-250,100
    xor mnuwin onto mnuwin
    dashinfo
end
let hatchangle = proc( ->real)
begin
    let Shft := X.dim(screen) - 560
    let themenu = limit screen to 155 by 275 at 385+Shft,115
    copy Hangle onto themenu
    let picbox = limit screen to 116 by 56 at 392+Shft,332
    let picbox1 = limit screen to 40 by 40 at 390+Shft,290
    let picbox2 = limit screen to 40 by 40 at 390+Shft,170
    let picbox3 = limit screen to 40 by 40 at 490+Shft,290
    let picbox4 = limit screen to 40 by 40 at 490+Shft,170
    let picbox5 = limit screen to 74 by 54 at 423+Shft,223
    let picbox6 = limit screen to 80 by 30 at 420+Shft,120
    let theangle := 0
    print theangle at 440+Shft,240
    let count := 0
    let xo := 0
    let yo := 0
    let done := false
    while ~done do
    begin
        let lo := locator()
        while ~lo(the.buttons)(1) do lo := locator()
        let xo := lo(X.pos)
        let yo := lo(Y.pos)
        while lo(the.buttons)(1) do lo := locator()
        if xo > 420+Shft and xo < 500+Shft and yo > 120 and yo < 150 do
        begin
            for bb =1 to 2000 do count := count + 1
            nor picbox6 onto picbox6
            done := true
        end
        if xo > 390+Shft and xo < 430+Shft and yo > 290 and yo < 330 do
        begin
            nor picbox1 onto picbox1
            theangle := theangle + 15
            for bb =1 to 2000 do count := count + 1
            nor picbox1 onto picbox1
        end
        if xo > 490+Shft and xo < 530+Shft and yo > 290 and yo < 330 do
        begin
            nor picbox3 onto picbox3
            theangle := theangle + 5
            for bb =1 to 2000 do count := count + 1
            nor picbox3 onto picbox3
        end
        if xo > 490+Shft and xo < 530+Shft and yo > 170 and yo < 210 do
        begin
            nor picbox4 onto picbox4
            theangle := theangle - 5
            for bb =1 to 2000 do count := count + 1
            nor picbox4 onto picbox4
        end
        end
        if xo > 390+Shft and xo < 430+Shft and yo > 170 and yo < 210 do
        begin

```

```

        nor picbox2 onto picbox2
        theangle := theangle - 15
        for bb = 1 to 2000 do count := count + 1
        nor picbox2 onto picbox2
    end
    xor picbox5 onto picbox5
    print theangle at 440+Shft,240
end
xor themenu onto themenu
theangle
end
let drsym = proc(string datafile; real x, y, Xscale, Yscale -> pic)
begin
    let fd = open (datafile,0)
    if fd = nullfile do
        begin
            write "The file ",datafile," cannot be opened'n"
            abort
        end
    let vecx = vector 1::10,1::30 of 0.0
    let vecy = vector 1::10,1::30 of 0.0
    let vecm = vector 1::30 of nil
    let I := 0
    let count := 0
    let PIC := [ 0.0, 0.0 ]
    let PIC1 := [ 0.0, 0.0 ]
    let aline := read.a.line(fd)
    let fc := aline(1|1)
    while ~eoi(fd) or fc ~= "e" do
        begin
            while fc ~= "/" and fc ~= "e" do
                begin
                    let le := length(aline)
                    I := I + 1
                    let p := 1
                    let q := 0
                    let fstring := ""
                    let sstring := ""
                    while aline(p|1) ~= " " do
                        begin
                            fstring := fstring ++ aline(p|1)
                            p := p + 1
                        end
                    while aline(p|1) = " " do p := p + 1
                    q := p-1
                    sstring := aline(p|( le - q))
                    let tempx := stringtoint(fstring)
                    vecx(count,I) := tempx * Xscale + x
                    let tempy := stringtoint(sstring)
                    vecy(count,I) := tempy * Yscale + y
                    PIC := if I = 1 then [ vecx(count,I) , vecy(count,I) ]
                        else PIC ^ [ vecx(count,I) , vecy(count,I) ]
                    aline := read.a.line(fd)
                    fc := aline(1|1)
                end
            if ~eoi(fd) do
                begin
                    aline := read.a.line(fd)
                    fc := aline(1|1)
                end
            I := 0
            count := count + 1
            if count = 1 then PIC1 := PIC
                else PIC1 := PIC1 & PIC
            end
        end
    close(fd)
    PIC1
end
let symscale = proc(string anysymbol -> *real)
begin
    let Shft := X.dim(screen) - 560
    let themenu = limit screen to 150 by 320 at 375+Shft, 95
    copy Scaling onto themenu
    let SYM = s.lookup(anysymbol,agrsyms)(f.pic)
    let picbox2 := limit screen to 40 by 40 at 440+Shft,200

```

```

let picbox7 := limit screen to 40 by 40 at 430+Shft,350
draw(picbox2, SYM, -20,20,0,30)
draw(picbox7, SYM, -20,20,0,30)
let xscale := 1.0
let yscale := 1.0
let the.pic := [ 0.0, 0.0 ]
let scales = vector 1::2 of 0.0
let picbox1 = limit screen to 40 by 40 at 390+Shft,200
let picbox3 = limit screen to 40 by 40 at 490+Shft,200
let picbox4 = limit screen to 40 by 40 at 440+Shft,150
let picbox5 = limit screen to 40 by 40 at 440+Shft,250
let picbox6 = limit screen to 80 by 30 at 420+Shft,100
let picbox8 := limit screen to 60 by 60 at 430+Shft, 350
let xo := 0
let yo := 0
let done := false
while ~done do
  begin
    let lo := locator()
    while ~lo(the.buttons)(1) do lo := locator()
    xo := lo(X.pos)
    yo := lo(Y.pos)
    while lo(the.buttons)(1) do lo := locator()
    if xo > 420+Shft and xo < 500+Shft and yo > 100 and yo < 130 then
      begin
        for bb =1 to 2000 do { }
          nor picbox6 onto picbox6
        done := true
      end
    else if xo > 390+Shft and xo < 430+Shft and yo > 200 and yo < 240 the
      begin
        nor picbox1 onto picbox1
        xscale := xscale - 0.1
        for bb =1 to 2000 do { }
          nor picbox1 onto picbox1
        end
      end
    else if xo > 490+Shft and xo < 530+Shft and yo > 200 and yo < 240 the
      begin
        nor picbox3 onto picbox3
        xscale := xscale + 0.1
        for bb =1 to 2000 do { }
          nor picbox3 onto picbox3
        end
      end
    else if xo > 440+Shft and xo < 480+Shft and yo > 150 and yo < 190 the
      begin
        nor picbox4 onto picbox4
        yscale := yscale - 0.1
        for bb =1 to 2000 do { }
          nor picbox4 onto picbox4
        end
      end
    else if xo > 440+Shft and xo < 480+Shft and yo > 250 and yo < 290 do
      begin
        nor picbox5 onto picbox5
        yscale := yscale + 0.1
        for bb =1 to 2000 do { }
          nor picbox5 onto picbox5
        end
      end
    scales(1) := xscale; scales(2) := yscale
    xor picbox8 onto picbox8
    Box (430+Shft,350,60)
    the.pic := drsym(anysymbol, 470+Shft, 350, xscale, yscale )
    draw(screen, the.pic, 0, X.dim(screen) , 0, Y.dim(screen) )
  end
  xor themenu onto themenu
  scales
end
let plotsym = proc(real X, Y, space, pitch, xscale, yscale;int marker ;
  string a.symbol -> pic )
begin
  let xplot := 0
  if marker rem 2 ~= 0 then xplot := truncate(X + pitch/2)
    else xplot := truncate(X + pitch )
  let yplot := truncate(Y)
  let SYM := [ 0.0, 0.0 ]
  let thecounter :=0

```

```

while xplot < (X + space) do
  begin
    thecounter := thecounter + 1
    if thecounter = 1 then SYM := drsym(a.symbol, xplot, yplot, xscale,
                                         yscale )
      else SYM := SYM & drsym(a.symbol, xplot, yplot, xscale,
                              yscale )
    xplot := xplot + truncate(pitch)
  end
end
SYM
end
let sorting = proc(*real k, r; int upper)
begin
  let swap = proc(*real v; cint i,j)
  begin
    let temp = v(i)
    v(i) := v(j)
    v(j) := temp
  end
  let sort = proc(*real x, y; int ubd -> int)
  begin
    let v := 0; let u := 0; let count := 0
    for i=1 to ubd -1 do
      begin
        v := i; u := i+1
        if x(v) > x(u) do
          begin
            swap(x,v,u)
            swap(y,v,u)
            count := count + 1
          end
        end
      end
    count
  end
  let howmany := sort(k,r,upper)
  while howmany ~= 0 do
    howmany := sort(k,r,upper)
  end
end
let filling = proc(*real Xs,Ys; int enclaves; real pitch; *int names;
                  *pntr RetVec -> pic )
begin
  let PIC := [ 0.0, 0.0 ]
  let TPIC := [ 0.0, 0.0 ]
  let Ymin := 900000.0; let Ymax := -900000.0
  let size := upb(Xs) - lwb(Xs) + 1
  for i=1 to size do
    begin
      if Ys(i) > Ymax do Ymax := Ys(i)
      if Ys(i) < Ymin do Ymin := Ys(i)
    end
  let xdataarray := vector 1::10,1::50 of 0.0
  let ydataarray := vector 1::10,1::50 of 0.0
  let symbolNames = table.to.text(agrsyms)
  let chooseSymbol = set.up.choose(symbolNames)(do.choose)
  let symbolChosen = chooseSymbol("Choose Symbol",X.dim(screen)-150,100)
  let thescales := symscale(symbolChosen)
  let picbox = limit screen to 175 by 340 at X.dim(screen) - 175,80
  xor picbox onto picbox
  let tempara := vector 1::2 of 0.0
  let coords := vector 1::2 of 0.0
  let ymin := 0.0; let ymax := 0.0
  let span := Ymax - Ymin
  let theXs := vector 1::(size*2) of 0.0
  let theYs := vector 1::(size*2) of 0.0
  let NoOfLines := truncate(span / pitch)
  let Y := vector 1::NoOfLines of 0.0
  let starty := 0.0
  if pitch > 1 do
    begin
      let rest := span - NoOfLines * pitch
      pitch := pitch + rest/NoOfLines
      starty := Ymin - pitch/2
    end
  if pitch =1 do starty := Ymin
  let NoOfInt := 0

```

```

for i=1 to NoOfLines do
  begin
    Y(i) := starty + pitch * i
    for j=1 to size-1 do
      begin
        if Ys(j) < Ys(j+1) then
          begin
            ymin := Ys(j)
            ymax := Ys(j+1)
          end
        else
          begin
            ymin := Ys(j+1)
            ymax := Ys(j)
          end
        tempara := linepara( Xs(j), Ys(j), Xs(j+1), Ys(j+1) )
        coords := lineint( 0.0, Y(i), tempara(1), tempara(2) )
        if coords(2) > ymin and coords(2) < ymax do
          begin
            NoOfInt := NoOfInt + 1
            theXs(NoOfInt) := coords(1)
            theYs(NoOfInt) := coords(2)
          end
        if enclaves ~= 0 do
          for count = 1 to enclaves do
            begin
              xdatarray(count) := RetVec(names(count))(AX)
              ydatarray(count) := RetVec(names(count))(AY)
              let thesize := upb(xdatarray(count)) -
                lwb(xdatarray(count)) + 1
              for l=1 to thesize-1 do {
                if ydatarray(count)(l) < ydatarray(count)(l+1) then
                  begin
                    ymin := ydatarray(count)(l)
                    ymax := ydatarray(count)(l+1)
                  end
                else
                  begin
                    ymin := ydatarray(count)(l+1)
                    ymax := ydatarray(count)(l)
                  end
                tempara := linepara( xdatarray(count)(l),
                  ydatarray(count)(l), xdatarray(count)(l+1),
                  ydatarray(count)(l+1) )
                coords := lineint( 0.0, Y(i), tempara(1), tempara(2) )
                if coords(2) > ymin and coords(2) < ymax do
                  begin
                    NoOfInt := NoOfInt + 1
                    theXs(NoOfInt) := coords(1)
                    theYs(NoOfInt) := coords(2)
                  end
                }
              end
            end
          sorting(theXs,theYs,NoOfInt)
          let kay := 0
          for l=1 to NoOfInt by 2 do
            begin
              kay := kay + 1
              let distance := rabs(theXs(l) - theXs(l+1))
              if l=1 then PIC := plotsym( theXs(l), Y(i), distance, 40,
                thescales(1), thescales(2) , kay, symbolChosen)
              else PIC := PIC & plotsym( theXs(l), Y(i), distance, 40,
                thescales(1), thescales(2) , kay, symbolChosen)
            end
          TPIC := if i=1 then PIC
            else TPIC & PIC
          !draw(screen, PIC, 0, X.dim(screen), 0, Y.dim(screen) )
          NoOfInt := 0
        end
      end
    TPIC
  end
end
let hatchpoly = proc(*real Xs,Ys; int enclaves; real pitch, angle, dash, gap;
  *int names; *pntr RetVec -> pic )
  begin

```

```

let size := upb(Xs) - lwb(Xs) + 1
if angle = 0 do angle := 180
angle := angle * pi / 180
let xx := 0.0
let yy := 0.0
let PIC := [ 0.0, 0.0 ]
for i=1 to size do !transformation
begin
xx := Xs(i)
yy := Ys(i)
Xs(i) := xx * cos(angle) + yy * sin(angle)
Ys(i) := -xx * sin(angle) + yy * cos(angle)
end
let Ymin := 900000.0; let Ymax := -900000.0
for i=1 to size do
begin
if Ys(i) > Ymax do Ymax := Ys(i)
if Ys(i) < Ymin do Ymin := Ys(i)
end
let xdatarray := vector 1::10,1::50 of 0.0
let ydatarray := vector 1::10,1::50 of 0.0
let thesize := vector 1::10 of 0
if enclaves ~= 0 do
for count = 1 to enclaves do
begin
xdatarray(count) := RetVec(names(count))(AX)
ydatarray(count) := RetVec(names(count))(AY)
for i=1 to thesize(count) do
begin
xx := xdatarray(count)(i)
yy := ydatarray(count)(i)
xdatarray(count)(i) := xx * cos(angle) + yy * sin(angle)
ydatarray(count)(i) := -xx * sin(angle) + yy * cos(angle)
end
end
end
let tempara := vector 1::2 of 0.0
let coords := vector 1::2 of 0.0
let ymin := 0.0; let ymax := 0.0
let span := Ymax - Ymin
let theXs := vector 1::50 of 0.0
let theYs := vector 1::50 of 0.0
let NoOfLines := truncate(span / pitch)
let Y := vector 1::NoOfLines of 0.0
let starty := 0.0
if pitch > 1 do
begin
let rest := span - NoOfLines * pitch
pitch := pitch + rest/NoOfLines
starty := Ymin - pitch/2
end
if pitch = 1 do starty := Ymin
let NoOfInt := 0
for i=1 to NoOfLines do
begin
Y(i) := starty + pitch * i
for j=1 to size-1 do
begin
if Ys(j) < Ys(j+1) then
begin
ymin := Ys(j)
ymax := Ys(j+1)
end
else
begin
ymin := Ys(j+1)
ymax := Ys(j)
end
end
tempara := linepara( Xs(j), Ys(j), Xs(j+1), Ys(j+1) )
coords := lineint( 0.0, Y(i), tempara(1), tempara(2) )
if coords(2) > ymin and coords(2) < ymax do
begin
NoOfInt := NoOfInt + 1
theXs(NoOfInt) := coords(1)
theYs(NoOfInt) := coords(2)
end
end

```

```

        if enclaves ~= 0 do
        for count = 1 to enclaves do
            for l=1 to thesize(count)-1 do
                begin
                    if ydatarray(count)(1) < ydatarray(count)(l+1) then
                        begin
                            ymin := ydatarray(count)(1)
                            ymax := ydatarray(count)(l+1)
                        end
                    else
                        begin
                            ymin := ydatarray(count)(l+1)
                            ymax := ydatarray(count)(1)
                        end
                    end
                    tempara := linepara( xdatarray(count)(1),
                                        ydatarray(count)(1), xdatarray(count)(l+1),
                                        ydatarray(count)(l+1) )
                    coords := lineint( 0.0, Y(i), tempara(1), tempara(2) )
                    if coords(2) > ymin and coords(2) < ymax do
                        begin
                            NoOfInt := NoOfInt + 1
                            theXs(NoOfInt) := coords(1)
                            theYs(NoOfInt) := coords(2)
                        end
                    end
                end
            end
        end
        sorting(theXs,theYs,NoOfInt)
        for k=1 to NoOfInt do
            begin
                xx := theXs(k)
                yy := theYs(k)
                theXs(k) := xx * cos(angle) - yy * sin(angle)
                theYs(k) := xx * sin(angle) + yy * cos(angle)
            end
        end
        let tpic := [0.0, 0.0]
        if dash = -1 then
            for l=1 to NoOfInt by 2 do
                tpic := if l=1 then drawline( theXs(l), theYs(l), theXs(l+1),
                                                theYs(l+1) )
                        else tpic & drawline( theXs(l), theYs(l), theXs(l+1),
                                                theYs(l+1) )
            end
        else
            for l=1 to NoOfInt by 2 do
                tpic := if l=1 then dashing( theXs(l), theYs(l), theXs(l+1),
                                                theYs(l+1), dash, gap)
                        else tpic & dashing( theXs(l), theYs(l), theXs(l+1), theYs(l+1), dash,
                                                gap)
            end
        end
        NoOfInt := 0
        PIC := if i=1 then tpic
               else PIC & tpic
    end
    PIC
end
let hatchspace = proc(real angle,info1,info2 -> real)
begin
    let themenu = limit screen to 100 by 345 at X.dim(screen)-145,120
    copy Hspacing onto themenu
    rec(X.dim(screen)-145,120,95,340)
    let xpos := X.dim(screen)-135
    let cons := 0.0
    let theypos := 0; cons := 0.0
    let space := 0.0
    let xo := 0
    let constant := 0
    let yo := 0
    let done := false
    let lower := 140
    while ~done do
        begin
            let lo := locator()
            while ~lo(the.buttons)(1) do lo := locator()
            xo := lo(X.pos)
            yo := lo(Y.pos)
            while lo(the.buttons)(1) do lo := locator()
            if xo > xpos or xo < xpos + 80 do

```



```

        if yo > 140 or yo < 410 do
            begin
                constant := truncate( (yo-140) div 40 )
                theypos := 140 + constant * 40
                space := constant
                done := true
            end
        end
    let thisbox := limit screen to 80 by 30 at xpos,theypos
    nor thisbox onto thisbox
    xor themenu onto themenu
    space
end
let default.menu = proc(-> *real)
begin
    let xstart := X.dim(screen)-210
    let menuwin = limit screen to 195 by 400 at xstart-5,115
    copy Hdefault onto menuwin
    let abox1 = limit screen to 85 by 40 at xstart+ 9, 180
    let abox2 = limit screen to 85 by 40 at xstart+ 9, 240
    let abox3 = limit screen to 85 by 40 at xstart+ 9, 300
    let picbox1 = limit screen to 88 by 36 at xstart+ 2, 122
    let picbox2 = limit screen to 88 by 36 at xstart+ 94, 122
    let picbox3 = limit screen to 185 by 40 at xstart-1, 120
    let bxstart := xstart+115
    nor abox1 onto abox1
    nor abox2 onto abox2
    nor abox3 onto abox3
    let thespacing := 2.0
    let theangle := 0.0
    let thedash := -1.0
    let thegap := -1.0
    let procdata := vector 1::4 of 0.0
    let theperiod := vector 1::2 of 0.0
    let xo := 0
    let yo := 0
    let done := false
    while ~done do
        begin
            let lo := locator()
            while ~lo(the.buttons)(1) do lo := locator()
            let xo := lo(X.pos)
            let yo := lo(Y.pos)
            while lo(the.buttons)(1) do lo := locator()
            if xo > xstart and xo < xstart+92 and yo > 120 and yo < 160 do
                begin
                    for bb =1 to 2000 do { }
                    nor picbox1 onto picbox1
                    done := true
                end
            end
            if xo > xstart+92 and xo < xstart+184 and yo > 120 and yo < 160 do
                begin
                    for bb =1 to 2000 do { }
                    nor picbox2 onto picbox2
                    xor menuwin onto menuwin
                    Rec(xstart,160,184,250,"Choose to change","fix13", "begining",
                        true)
                    Rec(xstart+ 9, 180, 85,40,"Spacing","fix13", "middle",false)
                    Rec(xstart+ 9, 240, 85,40,"Angle","fix13", "middle",false)
                    Rec(xstart+ 9, 300, 85,40,"Style","fix13", "middle",false)
                    let finished := false
                    while ~finished do
                        begin
                            let lo := locator()
                            while ~lo(the.buttons)(1) do lo := locator()
                            let xo := lo(X.pos)
                            let yo := lo(Y.pos)
                            while lo(the.buttons)(1) do lo := locator()
                            if xo > xstart+9 and xo < xstart+180 and yo > 180 and
                                yo < 220 do
                                begin
                                    nor abox1 onto abox1
                                    for bb =1 to 2000 do { }
                                    xor menuwin onto menuwin
                                    thespacing := hatchspace(theangle,thedash,thegap)
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end

```

```

        copy Hdefault onto menuwin
        nor picbox2 onto picbox2
        nor abox1 onto abox1
        nor abox2 onto abox2
        nor abox3 onto abox3
    end
    if xo > xstart+9 and xo < xstart+240 and yo > 240 and
    yo < 280 do
    begin
        nor abox2 onto abox2
        for bb =1 to 2000 do { }
        xor menuwin onto menuwin
        theangle := hatchangle()
        copy Hdefault onto menuwin
        nor picbox2 onto picbox2
        nor abox1 onto abox1
        nor abox2 onto abox2
        nor abox3 onto abox3
    end
    if xo > xstart+9 and xo < xstart+300 and yo > 300 and
    yo < 340 do
    begin
        nor abox3 onto abox3
        for bb =1 to 2000 do { }
        xor menuwin onto menuwin
        theperiod := hatchtype()
        thedash := theperiod(1)
        thegap := theperiod(2)
        copy Hdefault onto menuwin
        nor picbox2 onto picbox2
        nor abox1 onto abox1
        nor abox2 onto abox2
        nor abox3 onto abox3
    end
    if xo > xstart and xo < xstart+92 and yo > 120 and
    yo < 160 do
    begin
        for bb =1 to 2000 do { }
        nor picbox1 onto picbox1
        finished := true
    end
    end
    done := true
end
end
procdata(1) := thedash
procdata(2) := thegap
procdata(3) := thespacing
procdata(4) := theangle
procdata
end
let hatch = proc(* real vecX, vecY; *int vecI; *pntr retvector -> pic )
begin
    let win = limit screen to 200 by 350 at X.dim(screen)-220,100
    xor win onto win
    let thedata = default.menu()
    let thedash := thedata(1)
    let thegap := thedata(2)
    let thespace := thedata(3)
    let theangle := thedata(4)
    let IncNo := if upb(vecI) = 0 then 0
                    else upb(vecI) - lwb(vecI) + 1
    let HPIC := hatchpoly(vecX,vecY,IncNo,thespace,theangle,thedash,thegap,
                        vecI,retvector)
    if XZR = 0 then
        draw(gw, HPIC, XOL, XOR, YOL, YOR )
    else draw(gw, HPIC, XZL, XZR, YZL, YZR )
    xor win onto win
    HPIC
end
let fill = proc(* real vecX, vecY; *int vecI; *pntr retvector -> pic )
begin
    let win = limit screen to 200 by 350 at X.dim(screen)-220,100
    xor win onto win
    let IncNo := if upb(vecI) = 0 then 0

```

```

                                else upb(vecI) - lwb(vecI) + 1
let FPIC := filling(vecX, vecY, IncNo, 35.0, vecI, retvector)
if XZR = 0 then
    draw(gw, FPIC, XOL, XOR, YOL, YOR )
else draw(gw, FPIC, XZL, XZR, YZL, YZR )
xor win onto win
FPIC
end
let inclave = proc(*real X, Y, OX, OY; real oxc, oyc -> bool)
begin
    let upper1 = upb(X); let upper2 = upb(OX)
    let ymin := 0.0; let ymax := 0.0
    let tempara := vector 1::2 of 0.0
    let coords := vector 1::2 of 0.0
    let Xint1 := vector 1::20 of 0.0
    let Yint1 := vector 1::20 of 0.0
    let Xint2 := vector 1::20 of 0.0
    let Yint2 := vector 1::20 of 0.0
    let smallercount := 0
    let NoOfInt1 := 0; let NoOfInt2 := 0
    let theminmax1 = minmax(X, upper1); let theminmax2 = minmax(Y, upper1)
    let theminmax3 = minmax(OX, upper2); let theminmax4 = minmax(OY, upper2)
    let minX1 = theminmax1(1); let maxX1 = theminmax1(2)
    let minY1 = theminmax2(1); let maxY1 = theminmax2(2)
    let minX2 = theminmax3(1); let maxX2 = theminmax3(2)
    let minY2 = theminmax4(1); let maxY2 = theminmax4(2)
    let inside := false
    if minX2 > minX1 and maxX2 < maxX1 do
        if minY2 > minY1 and maxY2 < maxY1 do
            begin
                for i=1 to upper2-1 do
                    begin
                        if OY(i) < OY(i+1) then {
                            ymin := OY(i)
                            ymax := OY(i+1) }
                        else {
                            ymin := OY(i+1)
                            ymax := OY(i) }
                        tempara := linepara( OX(i), OY(i), OX(i+1), OY(i+1) )
                        coords := lineint( 0.0, oyc, tempara(1), tempara(2) )
                        if coords(2) > ymin and coords(2) < ymax do
                            begin
                                NoOfInt1 := NoOfInt1 + 1
                                Xint1(NoOfInt1) := coords(1)
                                Yint1(NoOfInt1) := coords(2)
                            end
                        end
                    end
                for i=1 to upper1-1 do
                    begin
                        if Y(i) < Y(i+1) then {
                            ymin := Y(i)
                            ymax := Y(i+1) }
                        else {
                            ymin := Y(i+1)
                            ymax := Y(i) }
                        tempara := linepara( X(i), Y(i), X(i+1), Y(i+1) )
                        coords := lineint( 0.0, oyc, tempara(1), tempara(2) )
                        if coords(2) > ymin and coords(2) < ymax do
                            begin
                                NoOfInt2 := NoOfInt2 + 1
                                Xint2(NoOfInt2) := coords(1)
                                Yint2(NoOfInt2) := coords(2)
                            end
                        end
                    end
                end
                let xint1 := vector 1::NoOfInt1 of 0.0
                let yint1 := vector 1::NoOfInt1 of 0.0
                let xint2 := vector 1::NoOfInt2 of 0.0
                let yint2 := vector 1::NoOfInt2 of 0.0
                for i=1 to NoOfInt1 do {
                    xint1(i) := Xint1(i)
                    yint1(i) := Yint1(i) }
                for i=1 to NoOfInt2 do {
                    xint2(i) := Xint2(i)
                    yint2(i) := Yint2(i) }
                let Intminmax1 := minmax(xint1, NoOfInt1)

```

```

        let Intminmax2 := minmax(xint2, NoOfInt2)
        let minInt1 := Intminmax1(1)
        let maxInt1 := Intminmax1(2)
        let minInt2 := Intminmax2(1)
        let maxInt2 := Intminmax2(2)
        inside := case true of
            minInt1 > minInt2 and maxInt1 < maxInt2 : true
            rabs(minInt1 - minInt2) < (1/1000) and maxInt1 < maxInt2 : true
            minInt1 > minInt2 and rabs(maxInt1 - maxInt2) < (1/1000) : true
            default : false
        for h=1 to NoOfInt2 do {
            if minInt1 < xint2(h) do smallercount := smallercount + 1
            if smallercount rem 2 = 0 do inside := false }
        end
    end
inside
end
let areamenu = proc(*real Xs, Ys; *int InclVec; *pntr vecret -> pic)
begin
    let finished := false
    let options = @1 of string [ "Fill", "Hatch" ]
    let choseoption = set.up.choose(options)(do.choose)
    let optionchosen = choseoption("OPTIONS", xmstart+50, 120 )
    let thepic := [ 0.0, 0.0 ]
    case true of
        optionchosen = "Fill" : { thepic := fill(Xs, Ys, InclVec, vecret);
                                finished := true }
        optionchosen = "Hatch" : { thepic := hatch(Xs, Ys, InclVec, vecret);
                                finished := true }
        default : finished := true
    thepic
end
let pikarea = proc(*pntr retvector)
begin
    let area.prepare = proc(*real Xs, Ys; *int Inclaves -> real )
    begin
        let thesize := upb(Xs)
        let Oarea := area(Xs, Ys, thesize)
        let Iarea := 0.0
        let Ninclaves := upb(Inclaves)
        for i=1 to Ninclaves do
            begin
                let loc := Inclaves(i)
                let xin := retvector(loc)(AX)
                let yin := retvector(loc)(AY)
                let upper := upb(xin)
                Iarea := Iarea - area(xin, yin, upper)
            end
        let TotalArea := Oarea + Iarea
        TotalArea
    end
    let is.nabor := false
    let is.inclave := false
    let nabors := vector 1::20 of 0
    let inclaves := vector 0::20 of 0
    let infodis = limit screen to 225 by 102 at xmstart+5, 345
    let naborcounter := 0
    let inclavecounter := 0
    let xlen := X.dim(gw)
    let ylen := Y.dim(gw)
    let imagesaved = image xlen by ylen of off
    copy gw onto imagesaved
    let TPIC := [ 0.0, 0.0 ]
    let APIC := [ 0.0, 0.0 ]
    let theupper := upb(retvector)
    let i := 1
    let finished := false
    while i <= theupper and ~finished do
        begin
            if ~retvector(i)(AP) then {
                let Acount := retvector(i)(ATN)
                let featcode := retvector(i)(AID)
                let xc := retvector(i)(Xc)
                let yc := retvector(i)(Yc)
                let xs := retvector(i)(AX)
                let ys := retvector(i)(AY)

```

```

naborcounter := 0
inclavecounter := 0
let thearea := 0.0
let theperi := 0.0
let upper := upb(xs)
TPIC := Highlight( xs, ys)
if XZR = 0 then { draw(gw, TPIC, XOL, XOR, YOL, YOR ) }
else { draw(gw, TPIC, XZL, XZR, YZL, YZR ) }
let xminmax := minmax(xs,upper)
let yminmax := minmax(ys,upper)
let the.radian := ( xminmax(2) - xminmax(1) + yminmax(2) -
                    yminmax(1) ) * 2
for j=1 to theupper do
  if j ~= i do
    begin
      let oxc := retvector(j)(Xc)
      let oyc := retvector(j)(Yc)
      let oxs := retvector(j)(AX)
      let oys := retvector(j)(AY)
      let the.dis := distance(xc,yc,oxc,oyc)
      if the.dis < the.radian do {
        is.nabor := nabor(xs,oxs)
        if is.nabor then {
          naborcounter := naborcounter + 1
          nabors(naborcounter) := retvector(j)(ATN) }
        else {
          is.inclave := inclave(xs, ys, oxs, oys, oxc, oyc)
          if is.inclave do {
            inclavecounter := inclavecounter + 1
            inclaves(inclavecounter) := retvector(j)(ATN) }
          }
        }
      end
      let check := polycheck(xs, ys, upper)
      if check do {
        if inclavecounter > 0 then
          thearea := area.prepare(xs, ys, inclaves)
        else thearea := area(xs, ys, upper)
        theperi := perimeter(xs, ys, upper)
        retvector(i)(AD1) := thearea
        retvector(i)(AD2) := theperi
        xor infodis onto infodis
        rec(xmstart+10,350,215,92)
        rec(xmstart+11,351,213,90)
        text.write(10,60,"Area      ", "fixbl3",infodis)
        text.write(10,30,"Perimiter", "fixbl3",infodis)
        print thearea at xmstart+100,410
        print theperi at xmstart+100,380 }
        let lowerb := 0; let upperb := 0
        if inclavecounter > 0 then {
          lowerb := 1; upperb := inclavecounter }
        else {
          lowerb := 0; upperb := 0 }
        let inclavec := vector lowerb::upperb of 0
        for i=lowerb to upperb do
          inclavec(i) := inclaves(i)
          retvector(i)(NOI) := inclavecounter
          retvector(i)(Inc) := inclavec
          retvector(i)(AP) := true
          xor infodis onto infodis
          retvector(i)(FA.pic) := areamenu(xs, ys, inclavec, retvector)
          if i <= theupper then
            finished := message.proc("Chose what next","Break","Proceed",
                                    xend-215, 120, 200, 140)
          else finished := true
          i := i + 1
          xor gw onto gw
          copy imagesaved onto gw }
        else i := i + 1
      end
      if commit() ~= nil do error.message("Could not Store Information", -1, -1)
    end
  end
let pikline := proc(*pntr retvector)
begin
  let xlen := X.dim(gw)

```

```

let ylen := Y.dim(gw)
let imagesaved = image xlen by ylen of off
copy gw onto imagesaved
let TPIC := [ 0.0, 0.0 ]
let LPIC := [ 0.0, 0.0 ]
let theupper := upb(retvector)
let i := 1
let finished := false
while i <= theupper and ~finished do
  begin
    if ~retvector(i)(LP) then {
      let lcount := retvector(i)(LTN)
      let featcode := retvector(i)(LID)
      let xs := retvector(i)(LX)
      let ys := retvector(i)(LY)
      let upper := upb(xs)
      TPIC := Highlight( xs, ys)
      if XZR = 0 then
        draw(gw, TPIC, XOL, XOR, YOL, YOR )
      else draw(gw, TPIC, XZL, XZR, YZL, YZR )
      retvector(i)(LP) := true
      let thispic := linetype(xs, ys, upper)
      retvector(i)(FL.pic) := thispic
      if XZR = 0 then
        draw(gw, thispic, XOL, XOR, YOL, YOR )
      else draw(gw, thispic, XZL, XZR, YZL, YZR )
      i := i + 1
      if i <= theupper then
        finished := message.proc("Chose what next","Break","Proceed",
                                xend-215, 120, 200, 140)
      else finished := true
      xor gw onto gw
      copy imagesaved onto gw }
    else i := i + 1
  end
end
if commit() ~= nil do error.message("Could not Store Information", -1, -1)
end
let chpoint = proc(real xio, yio -> pic )
begin
  let totalwin = limit screen to 180 by 350 at xmstart-5, 115
  copy symenu onto totalwin
  let xo := 0; let yo := 0
  let xb := xmstart + 10; let xf := xb + 3 * 40
  let yb := 130; let yf := yb + 8 * 40
  let Procedure := ""
  let continue := false
  while ~continue do {
    let xpos := 0; let ypos := 0
    let lo := locator()
    while ~lo(the.buttons)(1) do lo := locator()
    xo := lo(X.pos); yo := lo(Y.pos)
    xo := xo - xb; yo := yo - yb
    while lo(the.buttons)(1) do lo := locator()
    let column := yo div 40
    let row := xo div 40
    case row of
      0 : case column of
        0 : { xpos := row * 40 + xb; ypos := column * 40 + yb
              let win := limit screen to 40 by 40 at xpos, ypos
              nor win onto win
              Procedure := "city circle"
              nor win onto win; continue := true }
        1 : { xpos := row * 40 + xb; ypos := column * 40 + yb
              let win := limit screen to 40 by 40 at xpos, ypos
              nor win onto win
              Procedure := "hollow circle"
              nor win onto win; continue := true }
        2 : { xpos := row * 40 + xb; ypos := column * 40 + yb
              let win := limit screen to 40 by 40 at xpos, ypos
              nor win onto win
              Procedure := "plain triangle"
              nor win onto win; continue := true }
        3 : { xpos := row * 40 + xb; ypos := column * 40 + yb
              let win := limit screen to 40 by 40 at xpos, ypos
              nor win onto win

```

```

        Procedure := "thick hollow circle"
        nor win onto win; continue := true }
4 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "bracken"
    nor win onto win; continue := true }
5 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "heath"
    nor win onto win; continue := true }
6 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "orchard"
    nor win onto win; continue := true }
7 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "saltings" }
default : { }
1 : case column of
0 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "city square"
    nor win onto win; continue := true }
1 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "plain circle"
    nor win onto win; continue := true }
2 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "solid circle"
    nor win onto win; continue := true }
3 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "thktri.mark"
    nor win onto win; continue := true }
4 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "coppice"
    nor win onto win; continue := true }
5 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "marsh"
    nor win onto win; continue := true }
6 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "reeds"
    nor win onto win; continue := true }
7 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "scrub"
    nor win onto win; continue := true }
default : { }
2 : case column of
0 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "hexagon"
    nor win onto win; continue := true }
1 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "plain square"
    nor win onto win; continue := true }

```

```

2 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "solid square"
    nor win onto win; continue := true }
3 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "tri. mark"
    nor win onto win; continue := true }
4 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "conifer"
    nor win onto win; continue := true }
5 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "non-con"
    nor win onto win; continue := true }
6 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "rgrass"
    nor win onto win; continue := true }
7 : { xpos := row * 40 + xb; ypos := column * 40 + yb
    let win := limit screen to 40 by 40 at xpos, ypos
    nor win onto win
    Procedure := "TXT"
    nor win onto win; continue := true }
    default : { }
default : { }
}
xor totalwin onto totalwin
let apic := [ 0.0, 0.0 ]
let xposn := 0; let xposx := 0
case Procedure of
    "plain circle" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "solid circle" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "hollow circle" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "thick hollow circle" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "city circle" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "plain square" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "solid square" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "city square" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "plain triangle" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "tri. mark" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "thktri.mark" : { xio := xio - 20; yio := yio - 15; xposn := 0;
        xposx := 40; symtype := "geo"; apic := s.lookup(Procedure,
            geosyms)(a.pic) }
    "TXT" : { let x := truncate (xio); let y := truncate (yio);
        let font := Fonting(x, y); apic := [ 0.0, 0.0 ] }
    default : { xio := xio-20; yio := yio; xposn := -20;
        xposx := 20; symtype := "agr"; apic := s.lookup(Procedure,
            agrsyms)(f.pic) }

```



```

let Xpos := truncate(xio); let Ypos := truncate(yio)
let plotwin := limit screen to 40 by 40 at Xpos, Ypos
draw(plotwin, apic, xposn-10, xposx+10, 0, 60)
apic
end
let pikpoint := proc(*pntr retvector)
begin
let xlen := X.dim(gw)
let ylen := Y.dim(gw)
let xmin := 0.0; let xmax := 0.0
let ymin := 0.0; let ymax := 0.0
let imagesaved = image xlen by ylen of off
copy gw onto imagesaved
let LPIC := [ 0.0, 0.0 ]
let theupper := upb(retvector)
let i := 1
let finished := false
while i <= theupper and ~finished do
begin
if ~retvector(i) (PP) then {
let Pcount := retvector(i) (PTN)
let featcode := retvector(i) (PID)
let xs := retvector(i) (PX)
let ys := retvector(i) (PY)
let upper := upb(xs)
if XZR = 0 then {
xmin := XOL; xmax := XOR; ymin := YOL; ymax := YOR }
else { xmin := XZL; xmax := XZR; ymin := YZL; ymax := YZR }
let xposition := truncate( (xs(1) - xmin) * xlen / (xmax - xmin)) - 3
let yposition := truncate( (ys(1) - ymin) * ylen / (ymax - ymin))
text.write(xposition-3, yposition-3, "*", "fixb13", gw )
retvector(i) (PP) := true
retvector(i) (P.pic) := chpoint( xposition, yposition)
retvector(i) (PT) := symtype
i := i + 1
if i <= theupper then
finished := message.proc("Chose what next","Break","Proceed",
xend-215, 120, 200, 140)
else finished := true
xor gw onto gw
copy imagesaved onto gw }
else i := i + 1
end
if commit() ~= nil do error.message("Could not Store Information", -1, -1)
end
let what.to.pick = proc(string mapof; pntr anytable; int anycounter)
begin
let feature.kind := feature.type(anycounter)
let retvec := case feature.kind of
"polygon" : s.lookup(mapof, anytable) (AST)
"line" : s.lookup(mapof, anytable) (LST)
default : s.lookup(mapof, anytable) (PST)
let parwin := limit screen to 90 by 30 at range+110, Y.dim(screen)-152
xor parwin onto parwin
rec(range+104, Y.dim(screen)-152, 100, 36)
case feature.kind of
"polygon" : { text.write(10, 12, "polygon", "fix13", parwin);
pikarea(retvec) }
"line" : { text.write(10, 12, "line", "fix13", parwin); pikline(retvec)
default : { text.write(10, 12, "point", "fix13", parwin);
pikpoint(retvec) }
end
let data.prep = proc()
begin
xor screen onto screen
prepform("Enhancement")
let atable := s.lookup("Maps", mainDB )
let key.names = table.to.text(atable)
let the.top := upb(key.names)
let choosemap = set.up.choose(key.names) (do.choose)
let mapchosen = choosemap("Choose Map of:", xmstart+50, 120 )
XOL := s.lookup(mapchosen, atable) (XL)
YOL := s.lookup(mapchosen, atable) (YL)
XOR := s.lookup(mapchosen, atable) (XR)
YOR := s.lookup(mapchosen, atable) (YR)

```

```

let parwin := limit screen to length(mapchosen)*8 +30 by 36 at range+80,
               Y.dim(screen)-80
text.write(10, 12,mapchosen,"fix13",parwin)
Rec(range+4, Y.dim(screen)-116, out.range-8, 36,"Operation :", "fixb13",
    "begining", false)
let sangle := s.lookup(mapchosen,atable) (North)
let angle := stringtoint(sangle)
let thepic := [ 0.0, 0.0]
let a.exist := false
let l.exist := false
let p.exist := false
let Afeat := s.lookup(mapchosen,atable) (AST)
let uba := upb(Afeat)
if uba ~= 0 do a.exist := true
let Lfeat := s.lookup(mapchosen,atable) (LST)
let ubl := upb(Lfeat)
if ubl ~= 0 do l.exist := true
let Pfeat := s.lookup(mapchosen,atable) (PST)
let ubp := upb(Pfeat)
if ubp ~= 0 do p.exist := true
let Apic := [ 0.0, 0.0]
let Lpic := [ 0.0, 0.0]
let Ppic := [ 0.0, 0.0]
if a.exist do {
    for i=lwb(Afeat) to upb(Afeat) do
        Apic := if i=1 then Afeat(i) (anypic)
                else Apic & Afeat(i) (anypic)
        thepic := thepic & Apic }
if l.exist do {
    for i=lwb(Lfeat) to upb(Lfeat) do
        Lpic := if i=1 then Lfeat(i) (L.pic)
                else Lpic & Lfeat(i) (L.pic)
        thepic := thepic & Lpic }
if p.exist do {
    for i=lwb(Pfeat) to upb(Pfeat) do
        Ppic := if i=1 then Pfeat(i) (P.pic)
                else Ppic & Pfeat(i) (P.pic)
        thepic := thepic & Ppic }
!xor gw onto gw
let a.window = limit screen to 120 by 120 at xmstart+50,Y.dim(screen)-300
north.dir(angle, a.window)
let is.nabor := false
draw( gw, thepic, XOL, XOR, YOL, YOR )
let operations = @1 of string [ "Zoom In", "Zoom Out", "Scroll",
                                "Pick Feature" ]

zoomcounter := 0
let a.pik.count := 0
parwin := limit screen to 120 by 30 at range+110,Y.dim(screen)-115
let finished := false
while ~finished do {
let chooseoperation = set.up.choose(operations) (do.choose)
let operationchosen = chooseoperation( "OPTIONS      ", xmstart+50, 120 )
if operationchosen = "Zoom In" then {
    xor parwin onto parwin
    text.write(10, 12,operationchosen,"fix13",parwin)
    zoomcounter := zoomcounter + 1
    zoomin( thepic, thegrid, range, zoomcounter, 0, 0, gw )
    xor parwin onto parwin }
else if operationchosen = "Zoom Out" then {
    xor parwin onto parwin
    text.write(10, 12,operationchosen,"fix13",parwin)
    zoomout(thepic, thegrid, range, 0, 0, gw)
    xor parwin onto parwin
    zoomcounter := 0 }
else if operationchosen = "Scroll" then {
    xor parwin onto parwin
    text.write(10, 12,operationchosen,"fix13",parwin)
    Scroll( thepic, range, gw )
    xor parwin onto parwin }
else if operationchosen = "Pick Feature" then {
    xor parwin onto parwin
    text.write(10, 12,operationchosen,"fix13",parwin)
    a.pik.count := a.pik.count + 1
    what.to.pick(mapchosen, atable, a.pik.count)
    let porwin := limit screen to 105 by 36 at range+100,Y.dim(screen)-153

```

```

        xor porwin onto porwin
        xor parwin onto parwin }
    else if operationchosen = "" do
        finished := true }
end
end
structure modules2( proc()cartorep)
let moduleDB := open.database("M", "m", "write")
if moduleDB is error.record do moduleDB := create.database("M", "m")
let the.module := s.lookup("Module2", moduleDB)
if the.module = nil do
    begin
        the.module := table()
        s.enter("Module2", moduleDB, the.module)
    end
s.enter("cartorep", the.module, modules2(CartoRep) )
?

```

APPENDIX F

APPENDIX F: DATA RETRIEVAL MODULE

Appendix F contains the listing of the program handling the data retrieval operations. The program retrieves the Global Procedures needed from the database 'Global.Proc' and then lists all those module procedures described in Chapter 9. The module is then stored in the database '%\$Modules'.

```

let DataRet = proc()
begin
let FONTsdb:=open.database("FONTS","friend","read")
let fix13 = s.lookup( "fix13", FONTsdb)
let bold = s.lookup("fixb13",FONTsdb)
let big = s.lookup("met22",FONTsdb)
let large = s.lookup("hci45i",FONTsdb)
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
do (write "No utilities database - do procdmaker first'n"; abort)
let prcget=
begin
structure procpak(proc(string -> pntr) xproc)
s.lookup("prcget",procsdb) (xproc)
end
let seditor={structure procpak(proc(string,string,int,int,int,int-> string) xproc
prcget("seditor") (xproc) )
let error.message={structure procpak(proc(string,int,int) xproc)
prcget("error.message") (xproc) )
let more={structure procpak(proc(*string,int,int) xproc)
prcget("more") (xproc) )
let form.generate={structure procpak(proc( -> pntr ) xproc)
prcget("form.generate") (xproc) )
let form.null={structure procpak(proc( string,int,int,int,int,int,pntr ) xproc)
prcget("form.null") (xproc) )
structure form.package( proc( pntr) Form.show;
proc( ) Form.all.show;
proc( string,int,int,int,int,bool,proc(),pntr ->
pntr ) Form.add;
proc( pntr ) Form.remove;
proc( string,pntr ) Form.update;
proc( ) Form.clear;
proc( -> pntr ) Form.mouse;
proc( ) Fender;
proc( ) Form.monitor )
let set.up.choose = {structure procpak(proc(*string -> pntr) xproc)
prcget("set.up.choose") (xproc)}
structure chooser.pack( proc( string, int, int -> string ) do.choose;
proc( string ) add.choose;
proc( string ) remove.choose;
proc( int, int ) list.choose )
let table.to.text = {structure procpak(proc(pntr -> *string) xproc)
prcget("table.to.text") (xproc)}
structure global.proc(proc(string, string, string, int, int, int, int ->
bool)MessageProc;
proc(int, int, string, string, #pixel)Text.Write;
proc(string -> real)StringToReal;
proc(string -> int)StringToInt;
proc(*real, int -> *real)MinMax;
proc(int, int, string -> int)Icon;
proc(real, real, real, string -> pic)Polygon;
proc(*real, *real -> pic)highlight;
proc(real, real,real, real -> pic)Drawline;
proc(int, int, int)box;
proc(int, int, int, int)rectangle;
proc(int, int, int, int, int, string, string, string, bool)Rectan
proc(int, #pixel)North.Dir;
proc(int -> string)Feature.Type;
proc(pic, pic, int, int, int, #pixel)Zoomout;
proc(pic, pic, int, int, int, int, #pixel)Zoomin;
proc(*real, *real,*real, real, real, real, int -> bool)Check
proc(string)Prepform )
let ProcDB := open.database("Proc.Lib", "proc", "write")
let GLOBALS := s.lookup("Procedures", ProcDB)
let message.proc = GLOBALS(MessageProc)
let text.write = GLOBALS(Text.Write)
let stringtoreal = GLOBALS(StringToReal)
let stringtoint = GLOBALS(StringToInt)
let minmax = GLOBALS(MinMax)
let icon = GLOBALS(Icon)
let polygon = GLOBALS(Polygon)
let highlight = GLOBALS(highlight)
let drawline = GLOBALS(Drawline)
let Box = GLOBALS(box)
let rec = GLOBALS(rectangle)

```

```

let Rec = GLOBALS(Rectangle)
let north.dir = GLOBALS(North.Dir)
let feature.type = GLOBALS(Feature.Type)
let checkin = GLOBALS(Checkin)
let prepform = GLOBALS(Prepform)
let DB = open.database("code2","code2","write")
if DB is error.record do { write "cannot open DB "; abort }
structure Code2Node(string identifier; pnttr subtree)
structure menuPack( proc (string, int, int -> string) menuProc; c*string lname;
                    c*int lnum )
let mainDB := open.database("MDB","data","write")
if mainDB is error.record do mainDB := create.database( "MDB", "data" )
if mainDB is error.record do { error.message("Cannot open database'n",-1,-1);
                             abort }
let mapstable := s.lookup( "Maps", mainDB )
if mapstable = nil do
    mapstable := table()
structure maps( string Package, SerialNo, Scale, North; real XL, YL, XR, YR, Grid;
                *pnttr AST, LST, PST, TST )
structure Aholder(int ATN, AID; bool AP; real Xc, Yc; string AFN, AC; *real AX,
                  AY; pic anypic; real AD1, AD2; int NOI; *int Inc; pic FA.pic )
structure Lholder(int LTN, LID; bool LP; string LFN, LC; *real LX, LY; pic L.pic;
                  real LD1, LD2; pic FL.pic )
structure Pholder(int PTN, PID; bool PP; string PFN, PC, PT; *real PX, PY;
                  pic P.pic)
structure Tholder(int Layer, Location; real xpos, ypos; string thetext, Font)
let Ainfo = vector 1::10000 of nil
let Linfo = vector 1::5000 of nil
let Pinfo = vector 1::3000 of nil
let DBvar := open.database("global","variables","write")
if DBvar is error.record do DBvar := create.database( "global", "variables" )
if DBvar is error.record do { write "Error creating Database'n" }
structure globals(int varval)
structure gpic(pic thepic)
let DBsym := open.database("Symbols","mys","read")
let agrsyms := s.lookup( "%$Agrsym" , DBsym )
let geosyms := s.lookup("%$Geosym", DBsym )
structure agrsym(pic f.pic )
structure geosym(pic a.pic)
structure gimage(#pixel menuimage)
let theimage := s.lookup("Images", DBvar)
let Hlinetype := s.lookup("Hlinetype",theimage)(menuimage)
let Hangle := s.lookup("Hangle",theimage)(menuimage)
let Scaling := s.lookup("Scaling",theimage)(menuimage)
let Hspacing := s.lookup("Hspacing",theimage)(menuimage)
let Hdefault := s.lookup("Hdefault",theimage)(menuimage)
let LineMenu := s.lookup("LineMenu",theimage)(menuimage)
let symenu := s.lookup("Symbols Menu", theimage)(menuimage)
let typeimage = image 110 by 355 of off
let angimage = image 155 by 270 of off
let scalimage = image 150 by 320 of off
let symtype := ""
let xmstart := X.dim(screen)-250
let Sht := X.dim(screen) - 540
let xstart := 10; let ystart := 10
let xend := X.dim(screen) - 10; let yend := Y.dim(screen) - 53
let xspan := xend - xstart; let yspan := yend - ystart
let TotalYmenWin := yend-ystart-97
let Screen = limit screen to xspan by yspan at xstart, ystart
let xlcorner := xstart + 4; let ylcorner := ystart+101
let X.G := xspan -244
let Y.G := TotalYmenWin-2
if X.G > Y.G then X.G := Y.G
                else Y.G := X.G
let out.range := X.dim(screen) - Y.dim(screen)
let range := Y.dim(screen)
let GW = limit screen to X.G by Y.G at xlcorner, ylcorner
let xmenustart := X.dim(screen) - 250
let menuwin = limit screen to 236 by TotalYmenWin-3 at xmenustart,ylcorner
let menuwin1 = limit screen to 244 by TotalYmenWin-3 at xmenustart-6,50
let menusaved = image 217 by 224 of off
let ystartdetail := yend-280
let detail = limit screen to 244 by 180 at xmstart-4,ystartdetail
let gw = limit screen to range by range at 0, 0
let x.coord = vector 1::100 of 0.0

```

```

let y.coord = vector 1::100 of 0.0
let dummy.vec := vector 1::1000 of ""
let winsave := image X.G by Y.G of off
let XOL := 0.0; let XOR := 0.0
let YOL := 0.0; let YOR := 0.0
let XZL := 0.0; let XZR := 0.0
let YZL := 0.0; let YZR := 0.0
let TheGrid := 0.0
let thegrid := [ 0.0, 0.0 ]
let xt := range + 100; let yt := 271!Global
let tmenu = limit screen to 102 by 192 at xt-1, yt-1!Global
let xe := xt + 40; let ye := yt + 12!Global
let emenu = limit screen to 112 by 136 at xe-1, ye-1!Global
let xl := xt - 90; let yl := 100!Global
let lmenu = limit screen to 142 by 402 at xl-1, yl-1!Global
let eimage := s.lookup("Entity Menu", theimage)(menuimage)
let timage := s.lookup("Type Menu", theimage)(menuimage)
let limage := s.lookup("Layer Menu", theimage)(menuimage)
let def.menu := "1"
let zoomcounter := 0
structure feature(pic fea.pic )
let first.screen = proc()
begin
  Rec(0,0,X.dim(screen),Y.dim(screen),"Ps - GIS","cou20",
    "middle", true)
  let border1 = limit screen to X.dim(screen)-10 by Y.dim(screen)-43 at 5,5
  let border2 = limit screen to X.dim(screen)-20 by Y.dim(screen)-53 at 10,10
  nor border1 onto border1
  nor border2 onto border2
end
let zoomout = proc(pic PIC, thegrid; int Range, xshft, yshft; #pixel the.win )
begin
  if XZL = 0 and XZR = 0 then
    error.message(" Nothing to Zoom Out ", -1, -1 )
  else
    begin
      xor the.win onto the.win
      Box( xshft, yshft, Range)
      Box( xshft+1, yshft+1, Range-2)
      draw(the.win, thegrid, XOL, XOR, YOL, YOR )
      draw(the.win, PIC, XOL, XOR, YOL, YOR )
      XZL := 0.0; XZR := 0.0
      YZL := 0.0; YZR := 0.0
    end
  end
end
let zoomin = proc(pic PIC, thegrid;int Range, count, xshft, yshft;
  #pixel the.win )
begin
  let xl := 0.0; let xr := 0.0
  let yl := 0.0; let yr := 0.0
  if count < 2 then {
    xl := XOL; xr := XOR
    yl := YOL; yr := YOR }
  else {
    xl := XZL; xr := XZR
    yl := YZL; yr := YZR }
  let xo := 0; let xe := 0
  let yo := 0; let ye := 0
  let lo := locator()
  while ~lo(the.buttons)(1) do lo := locator()
  xo := lo(X.pos) - xshft
  yo := lo(Y.pos) - yshft
  while lo(the.buttons)(1) do lo := locator()
  xe := lo(X.pos) - xshft
  ye := lo(Y.pos) - yshft
  if xo > xe do {
    let t := xo
    xo := xe
    xe := t }
  if yo < ye do {
    let t := yo
    yo := ye
    ye := t }
  let xdif := rabs( xe - xo )
  let ydif := rabs( ye - yo )

```



```

let adif := truncate( ( xdif + ydif ) / 2 )
xe := xo + adif
ye := yo - adif
XZL := xl + (xr - xl) * xo / Range
XZR := xl + (xr - xl) * xe / Range
YZL := yl + (yr - yl) * ye / Range
YZR := yl + (yr - yl) * yo / Range
xor the.win onto the.win
Box( xshft, yshft, Range)
Box( xshft+1, yshft+1, Range-2)
draw(the.win, thegrid, XZL, XZR, YZL, YZR )
draw( the.win, PIC, XZL, XZR, YZL, YZR )
end
let obtain.code = proc(-> *int)
begin
let thecode := 0
let former.image = image 242 by TotalYmenWin-3 of off
copy menuwin1 onto former.image
xor menuwin1 onto menuwin1
!let ystartdetail := yend-250
!let detail = limit screen to 244 by 180 at xmstart-4, ystartdetail
let detail.image = image 244 by 180 of off
!rec(xmstart, ystart+100, 238, TotalYmenWin)
Rec(xmstart, ystartdetail, 238, 180, "Feature Details", "fixbl3", "middle", true)
let classification = @ 1 of string["Class      :", "Category  :",
                                   "Feature   :",
                                   "Attribute :", "The Code  :"]

for i=1 to 5 do {
let ypos := if i=1 then 110
             else if i=2 then 85
             else if i=3 then 60
             else if i=4 then 35
             else 10
text.write( 10, ypos, classification(i), "fixbl3", detail) }
let VecNames = vector 1::100 of ""
let VecNums  = vector 1::100 of 0
let N := 0
let procl = proc(int I; pntr V -> bool)
begin
N := N + 1
VecNames(N) := V(identifier)
VecNums(N) := I
true
end
let code.detail := vector 1::2 of 0
let stopchose := false
let cat = @1 of string ["Class", "Category", "Feature", "Attribute"]
let choices = vector 1::4 of ""
let catWidth = X.dim(screen) div 4
let choose := proc(int L; pntr T); nullproc
choose := proc(int L; pntr T)
begin
let theMenu := s.lookup( "menu", T)
if theMenu = nil do
begin
N := 0
let X := i.scan(T, procl)
let S := ""
let levelname = vector 1::N of ""
let levelnum = vector 1::N of 0
for i=1 to N do
{ levelname(i) := VecNames(i);
  levelnum(i) := VecNums(i) }
for i=1 to N-1 do for j=i+1 to N do
{ X := levelnum(i); levelnum(i) := levelnum(j);
  levelnum(j) := X; S := levelname(i);
  levelname(i) := levelname(j); levelname(j) := S }
for i=1 to N do
begin
if length(levelname(i)) > 25 do
levelname(i) := levelname(i) (1|25)
if levelname(i) = "" do levelname(i) := "-----"
end
let ch = set.up.choose(levelname) (do.choose)
theMenu := menuPack(ch, levelname, levelnum)

```

```

        s.enter ( "menu", T, theMenu)
        if commit() ~= nil do print "COMMIT FAILS" at 50,300
        end
        let ch = theMenu(menuProc)
        let levelname = theMenu(Lname)
        let levelnum = theMenu(Lnum)
        N := upb(levelname)
        let choice = ch("Choose " ++ cat(L), X.dim(screen)-240,120)
        if choice = "" do stopchase := true
        let chint := 0
        if choice ~= "" do {
        for i=1 to N do
            if choice = levelname(i) do chint := levelnum(i)
        choices(L) := choice
        thecode := if L=1 then chint * 10000000
                    else if L=2 then thecode + chint * 100000
                    else if L=3 then thecode + chint * 1000
                    else thecode + chint
        let ypos := if L=1 then 110
                    else if L=2 then 85
                    else if L=3 then 60
                    else 35
        code.detail(2) := L
        text.write( 119, ypos,choices(L),"fixb13",detail) }
        while ~stopchase and L < 4 do choose( L+1, i.lookup(chint,T)(subtree)
        end
        let levell := s.lookup("Code2",DB)
        choose( 1, levell)
        print thecode at X.dim(screen)-120,ystartdetail+10
        copy detail onto detail.image
        copy former.image onto menuwin1
        copy detail.image onto detail
        !write class.counter
        code.detail(1) := thecode
        code.detail
    end
let clayer = proc(int x, y -> string)
begin
    let Layer := ""
    let xr := xl + 2
    let yr := (y-102) div 30
    let localm := limit screen to 136 by 26 at xr, yr*30+104
    nor localm onto localm
    Layer := case true of
        yr = 11 : "Building"
        yr = 10 : "Delimiter"
        yr = 9 : "Desig. Area"
        yr = 8 : "Hydrography"
        yr = 7 : "Hypsography"
        yr = 6 : "Land Cover"
        yr = 5 : "Road & Rail"
        yr = 4 : "Structure"
        yr = 3 : "Text"
        yr = 2 : "Utility"
        yr = 1 : "Done"
        default : "Quit"
    Layer
end
let ctype = proc(int x, y -> string)
begin
    let Type := ""
    let xr := xt + 2
    let yr := (y-273) div 30
    let localm := limit screen to 96 by 26 at xr, yr*30+275
    nor localm onto localm
    Type := case true of
        yr = 4 : "Points"
        yr = 3 : "Lines"
        yr = 2 : "Polygons"
        yr = 1 : "Done"
        default : "Quit"
    Type
end
let centity = proc(int x, y -> string)
begin

```

```

let Entity := ""
let xr := xe + 2
let ys := ye + 8
let yr := (y-ys) div 30
let localm := limit screen to 106 by 26 at xr, (yr*30) + ys
nor localm onto localm
Entity := case true of
  yr = 2 : "Get Entity"
  yr = 1 : "Proceed"
  default : "Quit"
Entity
end
let checkm = proc(int x, y -> string )
begin
  let themenu := ""
  if def.menu = "l" then
    if x > xl and x < xl+140 and y > yl and y < yl+364 do themenu := "l"
  else if def.menu = "t" then
    if x > xt and x < xt+100 and y > yt and y < yt + 154 do themenu := "t"
  else if def.menu = "e" do
    if x > xe and x < xe+110 and y > ye and y < ye + 96 do themenu := "e"
  themenu
end
let activate = proc(int x, y)
begin
  if def.menu = "l" then
    if x > xl and x < xl+140 and y > yl+364 and y < yl+400 then
      begin
        def.menu := "l"
        copy limage onto lmenu
      end
    else
      if x > xl+141 and x < xt+100 and y > 425 and y < 461 then
        begin
          def.menu := "t"
          copy timage onto tmenu
        end
      else
        if x > xt+100 and x < xe+110 and y > 381 and y < 417 do
          begin
            def.menu := "e"
            copy eimage onto emenu
          end
        end
      else
        if def.menu = "t" then
          if x > xl and x < xl+140 and y > yl+364 and y < yl+400 then
            begin
              def.menu := "l"
              copy limage onto lmenu
            end
          else
            if x > xt and x < xt+100 and y > 425 and y < 461 then
              begin
                def.menu := "t"
                copy timage onto tmenu
              end
            else
              if x > xt+100 and x < xe+110 and y > 381 and y < 417 do
                begin
                  def.menu := "e"
                  copy eimage onto emenu
                end
              end
            else if def.menu = "e" do
              if x > xl and x < xl+140 and y > yl+364 and y < yl+400 then
                begin
                  def.menu := "l"
                  copy limage onto lmenu
                end
              else
                if x > xt and x < xt+100 and y > 425 and y < 461 then
                  begin
                    def.menu := "t"
                    copy timage onto tmenu
                  end
                else

```

```

        if x > xt+40 and x < xe+110 and y > 317 and y < 417 do
            begin
                def.menu := "e"
                copy eimage onto emenu
            end
        end
    end
let codestrip = proc(int acode -> int)
    begin
        let layerid := truncate(acode/10000000)
        layerid
    end
structure Lstr(string thelayer; int featID; pic pic1, pic2)
structure Tstr1(string thetype1; int t1featID; pic t1pic1, t1pic2)
structure Tstr2(string thetype2; int t2featID; pic t2pic1, t2pic2)
structure Tstr3(string thetype3; int t3featID; pic t3pic1, t3pic2)
structure Estr(string theentity; int entityID; pic epic1, epic2)
let polygon.vec = vector 1::1000 of 0; let pol.count := 0
let line.vec = vector 1::1000 of 0; let line.count := 0
let point.vec = vector 1::1000 of 0; let point.count := 0
let text.vec = vector 1::1000 of 0; let text.count := 0
let retLayer = proc(*string layers.vec; string MapChosen; pntr Atable )
    begin
        let size := upb(layers.vec) - lwb(layers.vec) + 1
        let layerid := vector 1::size of 0
        for i=1 to size do
            layerid(i) := case true of
                layers.vec(i) = "Building"      : 2
                layers.vec(i) = "Delimiter"      : 6
                layers.vec(i) = "Desig. Area"    : 1
                layers.vec(i) = "Hydrography"    : 7
                layers.vec(i) = "Hypsography"    : 8
                layers.vec(i) = "Land Cover"     : 9
                layers.vec(i) = "Road & Rail"    : 4
                layers.vec(i) = "Structure"      : 3
                layers.vec(i) = "Utility"        : 5
                default                          : 0
            let Afeat := s.lookup(MapChosen, Atable) (AST)
            let Lfeat := s.lookup(MapChosen, Atable) (LST)
            let Pfeat := s.lookup(MapChosen, Atable) (PST)
            !let Text := s.lookup(MapChosen, Atable) (TST)
            Rec(range+4, Y.dim(screen)-116, out.range-8, 36, "Operation :", "fixbl3",
                "begining", false)
            let thepic := [ 0.0, 0.0 ]
            let Enhpic := [ 0.0, 0.0 ]
            let Asize := if upb(Afeat) > 0 then upb(Afeat) - lwb(Afeat) + 1
                        else upb(Afeat) - lwb(Afeat)
            let Lsize := if upb(Lfeat) > 0 then upb(Lfeat) - lwb(Lfeat) + 1
                        else upb(Lfeat) - lwb(Lfeat)
            let Psize := if upb(Pfeat) > 0 then upb(Pfeat) - lwb(Pfeat) + 1
                        else upb(Pfeat) - lwb(Pfeat)

            let totalsize := Asize + Lsize + Psize
            let features := vector 1::totalsize of nil
            let nfeatures := 0
            if Asize > 0 do
                for i=1 to Asize do
                    begin
                        let Aid := Afeat(i) (AID)
                        let stripcode := codestrip(Aid)
                        for j=1 to size do
                            if stripcode = layerid(j) do
                                begin
                                    nfeatures := nfeatures + 1
                                    thepic := Afeat(i) (anypic)
                                    Enhpic := Afeat(i) (FA.pic)
                                    let theid := Afeat(i) (ATN)
                                    pol.count := pol.count + 1
                                    polygon.vec(pol.count) := theid
                                    features(nfeatures) := Lstr("A", theid, thepic, Enhpic)
                                    if XZR = 0 then { draw(gw, thepic, XOL, XOR, YOL, YOR)
                                                            draw(gw, Enhpic, XOL, XOR, YOL, YOR) }
                                    else { draw(gw, thepic, XZL, XZR, YZL, YZR)
                                            draw(gw, Enhpic, XZL, XZR, YZL, YZR) }
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end

```

```

if Lsize > 0 do
for i=1 to Lsize do
begin
let Lid := Lfeat(i) (LID)
let stripcode := codestrip(Lid)
for j=1 to size do
if stripcode = layerid(j) do
begin
nfeatures := nfeatures + 1
thepic := Lfeat(i) (L.pic)
Enhpic := Lfeat(i) (FL.pic)
let theid := Lfeat(i) (LTN)
line.count := line.count + 1
line.vec(line.count) := theid
features(nfeatures) := Lstr("L", theid, thepic, Enhpic)
if XZR = 0 then draw(gw, Enhpic, XOL, XOR, YOL, YOR)
else draw(gw, Enhpic, XZL, XZR, YZL, YZR)
end
end
end
if Psize > 0 do
for i=1 to Psize do
begin
let Pid := Pfeat(i) (PID)
let stripcode := codestrip(Pid)
for j=1 to size do
if stripcode = layerid(j) do
begin
nfeatures := nfeatures + 1
let px := Pfeat(i) (PX); let py := Pfeat(i) (PY)
thepic := [ px(1), py(1) ]
Enhpic := Pfeat(i) (P.pic)
let theid := Pfeat(i) (PTN)
point.count := point.count + 1
point.vec(point.count) := theid
features(nfeatures) := Lstr("P", theid, thepic, Enhpic)
end
end
!if XZR = 0 then draw(gw, Enhpic, XOL, XOR, YOL, YOR)
!else draw(gw, Enhpic, XZL, XZR, YZL, YZR)
end
end
let RetApic = proc(string MapChosen; pntr Atable)
begin
let Afeat = s.lookup(MapChosen, Atable) (AST)
let asize := upb(Afeat) - lwb(Afeat) + 1
let features := vector 1::asize of nil
let nfeatures := 0
let a.pic := [ 0.0, 0.0 ]
let ao.pic := [ 0.0, 0.0 ]
for i=1 to asize do {
nfeatures := nfeatures + 1
a.pic := Afeat(i) (FA.pic)
ao.pic := Afeat(i) (anypic)
let theid := Afeat(i) (ATN)
pol.count := pol.count + 1
polygon.vec(pol.count) := theid
features(nfeatures) := Tstr1("A", theid, a.pic, ao.pic)
if XZR = 0 then { draw(gw, ao.pic, XOL, XOR, YOL, YOR)
draw(gw, a.pic, XOL, XOR, YOL, YOR) }
else { draw(gw, ao.pic, XZL, XZR, YZL, YZR)
draw(gw, a.pic, XZL, XZR, YZL, YZR) } }
end
let RetLpic = proc(string MapChosen; pntr Atable)
begin
let Lfeat = s.lookup(MapChosen, Atable) (LST)
let asize := upb(Lfeat) - lwb(Lfeat) + 1
let features := vector 1::asize of nil
let nfeatures := 0
let a.pic := [ 0.0, 0.0 ]
let ao.pic := [ 0.0, 0.0 ]
for i=1 to asize do {
nfeatures := nfeatures + 1
a.pic := Lfeat(i) (FL.pic)
ao.pic := Lfeat(i) (L.pic)
let theid := Lfeat(i) (LTN)
line.count := line.count + 1

```

```

        line.vec(line.count) := theid
        features(nfeatures) := Tstr2("L", theid, a.pic, ao.pic)
        if XZR = 0 then draw(gw, a.pic, XOL, XOR, YOL, YOR)
        else draw(gw, a.pic, XZL, XZR, YZL, YZR) }
    end
let RetPpic = proc(string MapChosen; pntr Atable)
begin
    let Pfeat = s.lookup(MapChosen, Atable) (PST)
    let xlen := X.dim(gw)
    let ylen := Y.dim(gw)
    let xmin := 0.0; let xmax := 0.0
    let ymin := 0.0; let ymax := 0.0
    if XZR = 0 then {
        xmin := XOL; xmax := XOR; ymin := YOL; ymax := YOR }
    else { xmin := XZL; xmax := XZR; ymin := YZL; ymax := YZR }
    let asize := upb(Pfeat) - lwb(Pfeat) + 1
    let features := vector 1::asize of nil
    let nfeatures := 0
    let xpos := 0; let ypos := 0; let xposn := 0; let xposx := 0
    let a.pic := [ 0.0, 0.0 ]
    for i=1 to asize do {
        a.pic := if i=1 then Pfeat(i) (P.pic)
                  else a.pic & Pfeat(i) (P.pic)
        let stype := Pfeat(i) (PT)
        let x := Pfeat(i) (PX); let y := Pfeat(i) (PY)
        if stype = "geo" then {
            xpos := truncate( x(1)-20)
            ypos := truncate( y(1)-15); xposn := 0; xposx := 40 }
        else {
            xpos := truncate( x(1)-20); ypos := truncate( y(1))
            xposn := -20; xposx := 20 }
        xpos := truncate( (xpos - xmin) * xlen / (xmax - xmin))
        ypos := truncate( (ypos - ymin) * ylen / (ymax - ymin))
        if XZR = 0 then {
            if ypos+40 < YOR and ypos > YOL do
                if xpos+40 < XOR and xpos > XOL do {
                    let plotwin := limit screen to 40 by 40 at xpos, ypos
                    draw(plotwin, a.pic, xposn-10, xposx+10, 0, 60) } }
            else {
                if ypos+40 < YZR and ypos > YZL do
                    if xpos+40 < XZR and xpos > XZL do {
                        let plotwin := limit screen to 40 by 40 at xpos, ypos
                        draw(plotwin, a.pic, XZL, XZR, YZL, YZR) } }
            }
        }
    }
end
let retType = proc(*string types.vec; string MapChosen; pntr Atable)
begin
    let size := if upb(types.vec) > 0 then upb(types.vec) - lwb(types.vec) + 1
                  else upb(types.vec) - lwb(types.vec)

    if size > 0 do {
        let typeid := vector 1::size of ""
        let typetable := vector 1::size of nil
        Rec(range+4, Y.dim(screen)-116, out.range-8, 36,"Operation :", "fixbl3",
            "begining", false)
        let thepic := [ 0.0, 0.0 ]
        let Enhpic := [ 0.0, 0.0 ]
        for i=1 to size do {
            if types.vec(i) = "Polygons" then RetApic(MapChosen, Atable)
            else if types.vec(i) = "Lines" then RetLpic(MapChosen, Atable)
            else if types.vec(i) = "Points" do RetPpic(MapChosen, Atable)
            ! Enhpic := if i = 1 then thepic
                        ! else Enhpic & thepic }
            !if XZR = 0 then draw(gw, Enhpic, XOL, XOR, YOL, YOR)
            !else draw(gw, Enhpic, XZL, XZR, YZL, YZR)
        } }
    end
let class.code = proc(int stored.code; *int retobj -> bool)
begin
    let match := false
    let the.code := retobj(1)
    let the.class := retobj(2)
    if the.class = 4 then
        if stored.code = the.code do match := true
    else if the.class = 3 then {
        stored.code := truncate(stored.code/1000) * 1000
    }

```

```

    if stored.code = the.code do match := true }
else if the.class = 2 then {
    stored.code := truncate(stored.code/100000) * 100000
    if stored.code = the.code do match := true }
else if the.class = 1 do {
    stored.code := truncate(stored.code/10000000) * 10000000
    if stored.code = the.code do match := true}
!write the.code, stored.code
match
end
let retfromA = proc(*int retvec; string mapchosen; pntr atable)
begin
    !write mapchosen
    let Afeat := s.lookup(mapchosen, atable) (AST)
    ! write aname
    let asize := upb(Afeat) - lwb(Afeat) + 1
    let featid := 0
    let a.pic := [ 0.0, 0.0]
    let ao.pic := [ 0.0, 0.0]
    let features := vector 1::asize of nil
    let nfeatures := 0
    for i=1 to asize do {
        featid := Afeat(i) (AID)
        let does.it.match := class.code(featid, retvec)
        if does.it.match do {
            nfeatures := nfeatures + 1
            a.pic := Afeat(i) (FA.pic)
            ao.pic := Afeat(i) (anypic)
            pol.count := pol.count + 1
            polygon.vec(pol.count) := Afeat(i) (ATN)
            !features(nfeatures) = Estr("E", featid, a.pic, ao.pic)
            if XZR = 0 then { draw(gw, ao.pic, XOL, XOR, YOL, YOR)
                            draw(gw, a.pic, XOL, XOR, YOL, YOR) }
            else { draw(gw, ao.pic, XZL, XZR, YZL, YZR)
                  draw(gw, a.pic, XZL, XZR, YZL, YZR) }
        }
    }
end
let retfromL = proc(*int retvec; string mapchosen; pntr atable)
begin
    let Lfeat := s.lookup(mapchosen, atable) (LST)
    let lsize := upb(Lfeat) - lwb(Lfeat) + 1
    let featid := 0
    let a.pic := [ 0.0, 0.0]
    let ao.pic := [ 0.0, 0.0]
    let features := vector 1::lsize of nil
    let nfeatures := 0
    for i=1 to lsize do {
        featid := Lfeat(i) (LID)
        let does.it.match := class.code(featid, retvec)
        if does.it.match do {
            nfeatures := nfeatures + 1
            a.pic := Lfeat(i) (FL.pic)
            line.count := line.count + 1
            line.vec(line.count) := Lfeat(i) (LTN)
            !features(nfeatures) = Estr("E", featid, a.pic, ao.pic)
            if XZR = 0 then draw(gw, a.pic, XOL, XOR, YOL, YOR)
            else draw(gw, a.pic, XZL, XZR, YZL, YZR)
        }
    }
end
let retfromP = proc(*int retvec; string mapchosen; pntr atable)
begin
    let Pfeat := s.lookup(mapchosen, atable) (PST)
    let psize := upb(Pfeat) - lwb(Pfeat) + 1
    let featid := 0
    let a.pic := [ 0.0, 0.0]
    let ao.pic := [ 0.0, 0.0]
    let features := vector 1::psize of nil
    let nfeatures := 0
    for i=1 to psize do {
        featid := Pfeat(i) (PID)
        let does.it.match := class.code(featid, retvec)
        if does.it.match do {
            nfeatures := nfeatures + 1

```

```

        a.pic := Pfeat(i) (P.pic)
        point.count := point.count + 1
        point.vec(point.count) := Pfeat(i) (PTN)
        !features(nfeatures) = Estr("E", featid, a.pic, ao.pic)
    }
end
let retEntity = proc(*int retent; string MapChosen; pnter Atable)
begin
    retfromA(retent, MapChosen, Atable)
    retfromL(retent, MapChosen, Atable)
    retfromP(retent, MapChosen, Atable)
end
let retT.E = proc(*int retent; *string retent1; string MapChosen; pnter Atable)
begin
    let NofType := upb(retent1) - lwb(retent1) + 1
    for k = 1 to NofType do
        case true of
            retent1(k) = "Polygons" : retfromA(retent, MapChosen, Atable)
            retent1(k) = "Lines"    : retfromL(retent, MapChosen, Atable)
            retent1(k) = "Points"   : retfromP(retent, MapChosen, Atable)
            default                  : { }
        end
    end
end
let selectEntity = proc( -> *int)
begin
    let the.code.detail := vector 1::2 of 0
    the.code.detail := obtain.code()
    xor detail onto detail
    the.code.detail
end
let retT.Layer = proc(*string lvec, tvec; string MapChosen; pnter Atable)
begin
    let lsize := upb(lvec) - lwb(lvec) + 1
    let layerid := vector 1::lsize of 0
    for i=1 to lsize do
        layerid(i) := case true of
            lvec(i) = "Building"      : 2
            lvec(i) = "Delimiter"     : 6
            lvec(i) = "Desig. Area"   : 1
            lvec(i) = "Hydrography"   : 7
            lvec(i) = "Hypsography"   : 8
            lvec(i) = "Land Cover"    : 9
            lvec(i) = "Road & Rail"   : 4
            lvec(i) = "Structure"     : 3
            lvec(i) = "Utility"       : 5
            default                    : 0
        end
    end
    let the.pic := [ 0.0, 0.0]
    let tsize := upb(tvec) - lwb(tvec) + 1
    for k=1 to tsize do
        if tvec(k) = "Polygons" then {
            let Afeat := s.lookup(MapChosen, Atable) (AST)
            let asize := if upb(Afeat) > 0 then upb(Afeat) - lwb(Afeat) + 1
                        else upb(Afeat) - lwb(Afeat)
            if asize > 0 do
                for i=1 to asize do
                    begin
                        let Aid := Afeat(i) (AID)
                        let stripcode := codestrip(Aid)
                        for j=1 to lsize do {
                            if stripcode = layerid(j) do
                                the.pic := Afeat(i) (anypic) & Afeat(i) (FA.pic)
                                if XZR = 0 then draw(gw, the.pic, XOL, XOR, YOL, YOR)
                                else draw(gw, the.pic, XZL, XZR, YZL, YZR) }
                            end
                        }
                    end
                }
            else if tvec(k) = "Lines" do {
                let Lfeat := s.lookup(MapChosen, Atable) (LST)
                let Lsize := if upb(Lfeat) > 0 then upb(Lfeat) - lwb(Lfeat) + 1
                            else upb(Lfeat) - lwb(Lfeat)
                if lsize > 0 do
                    for i=1 to Lsize do
                        begin
                            let Lid := Lfeat(i) (LID)
                            let stripcode := codestrip(Lid)
                            for j=1 to lsize do {

```



```

        if stripcode = layerid(j) do
            the.pic := Lfeat(i) (FL.pic)
            if XZR = 0 then draw(gw, the.pic, XOL, XOR, YOL, YOR)
            else draw(gw, the.pic, XZL, XZR, YZL, YZR) }
        end
    }

end

let retrieve = proc()
begin
    xor screen onto screen
    prepform("Data Retrieval")
    let total.menu = limit screen to 242 by 410 at xt-91, 95!Frog
    xor total.menu onto total.menu
    let atable := s.lookup("Maps", mainDB )
    let key.names = table.to.text(atable)
    let the.top := upb(key.names)
    let choosemap = set.up.choose(key.names) (do.choose)
    let mapchosen = choosemap("Choose Map of:", xmstart+50, 120 )
    XOL := s.lookup(mapchosen,atable) (XL)
    YOL := s.lookup(mapchosen,atable) (YL)
    XOR := s.lookup(mapchosen,atable) (XR)
    YOR := s.lookup(mapchosen,atable) (YR)
    let parwin := limit screen to length(mapchosen)*8 +30 by 36 at range+80,
        Y.dim(screen)-80
    text.write(10, 12,mapchosen,"fix13",parwin)
    let the.grid := s.lookup(mapchosen, atable) (Grid)
    let x.grid := XOL + the.grid
    let y.grid := YOL + the.grid
    let PIC := [0.0,0.0]
    let i := 1
    while x.grid <= XOR do
        begin
            PIC := if i = 1 then [ x.grid,YOL ] ^ [ x.grid,YOR ] &
                [ XOL,y.grid ] ^ [ XOR,y.grid ]
            else PIC & [ x.grid, YOL] ^ [ x.grid, YOR] &
                [ XOL,y.grid ] ^ [ XOR,y.grid ]
            draw (gw, PIC, XOL, XOR, YOL, YOR )
            x.grid := x.grid + the.grid
            y.grid := y.grid + the.grid
            i := i + 1
        end
    !let Afeat := s.lookup(mapchosen, atable) (AST)
    !let Lfeat := s.lookup(mapchosen, atable) (LST)
    !let Pfeat := s.lookup(mapchosen, atable) (PST)
    !let Text := s.lookup(mapchosen, atable) (TST)
    let operations = @1 of string [ "Zoom In", "Zoom Out", "Start" ]
    let parwin1 := limit screen to 120 by 30 at range+110,Y.dim(screen)-115
    let finished := false
    while ~finished do {
        let chooseoperation = set.up.choose(operations) (do.choose)
        let operationchosen = chooseoperation( "OPTIONS      ", xmstart+50, 120 )
        if operationchosen = "Zoom In" then {
            xor parwin1 onto parwin1
            text.write(10, 12,operationchosen,"fix13",parwin1)
            zoomcounter := zoomcounter + 1
            zoomin( PIC, thegrid, range, zoomcounter, 0, 0, gw )
            xor parwin1 onto parwin1 }
        else if operationchosen = "Zoom Out" then {
            xor parwin1 onto parwin1
            text.write(10, 12,operationchosen,"fix13",parwin1)
            zoomout(PIC, thegrid, range, 0, 0, gw)
            xor parwin1 onto parwin1
            zoomcounter := 0 }
        else if operationchosen = "Start" then {
            copy eimage onto emenu
            copy timage onto tmenu
            copy limage onto lmenu
            let total.image := image 242 by 410 of off
            copy total.menu onto total.image
            let start := false
            while ~start do
                begin
                    xor total.menu onto total.menu
                    copy total.image onto total.menu

```

```

if def.menu = "l" then copy limage onto lmenu
else if def.menu = "t" then copy timage onto tmenu
else if def.menu = "e" do copy eimage onto emenu
let retLvec = vector 1::12 of ""
let retTvec = vector 1::4 of ""
let retevec = vector 1::10 of ""
let retEvec := vector 1::10 of 0
let ii := 0; let jj := 0; let kk := 0; let kkk := 0
let done := false
while ~done and ~start do
  begin
    let lo := locator()
    while ~lo(the.buttons)(1) do lo := locator()
    let xo := lo(X.pos)
    let yo := lo(Y.pos)
    while lo(the.buttons)(1) do lo := locator()
    let am := checkm(xo, yo)

    if def.menu = "l" and am = "l" then {
      ii := ii + 1
      retLvec(ii) := clayer(xo, yo)
      if retLvec(ii) = "Done" do {
        done := true
        xor total.menu onto total.menu }
      if retLvec(ii) = "Quit" do {
        start := true
        xor total.menu onto total.menu } }

    else if def.menu = "t" and am = "t" then {
      jj := jj + 1
      retTvec(jj) := ctype(xo, yo)
      if retTvec(jj) = "Done" do {
        done := true
        xor total.menu onto total.menu }
      if retTvec(jj) = "Quit" do {
        start := true
        xor total.menu onto total.menu } }

    else if def.menu = "e" and am = "e" do {
      kkk := kkk + 1
      retevec(kkk) := centity(xo, yo)
      if retevec(kkk) = "Proceed" then
        done := true
      else if retevec(kkk) = "Get Entity" then {
        kk := 1
        retEvec := selectEntity() }
      else if retevec(kkk) = "Quit" do {
        !
        start := true
        xor total.menu onto total.menu } }
    if ~start then
      activate(xo, yo)
    else {
      xor menuwin1 onto menuwin1
      ii := 0; jj := 0; kk := 0 }
  end
if ii >= 1 and jj >= 1 then {
  if ii = 1 do ii := ii + 1
  if jj = 1 do jj := jj + 1
  let retent1 := vector 1::ii-1 of ""
  let retent2 := vector 1::jj-1 of ""
  for k = 1 to ii-1 do
    retent1(k) := retLvec(k)
  for k = 1 to jj-1 do
    retent2(k) := retTvec(k)
  retT.Layer(retent1, retent2, mapchosen, atable) }
else
  if jj >= 1 and kk >= 1 then {
    if jj = 1 do jj := jj + 1
    let retent1 := vector 1::jj-1 of ""
    for k = 1 to jj-1 do
      retent1(k) := retTvec(k)
    retT.E(retEvec, retent1, mapchosen, atable)
  }
else {

```

```

        if ii > 1 do {
            let retent := vector 1::ii-1 of ""
            for k=1 to ii-1 do {
                retent(k) := retLvec(k)
                retLayer(retent, mapchosen, atable) } }
            if jj > 1 do {
                let retent := vector 1::jj-1 of ""
                for k=1 to jj-1 do {
                    retent(k) := retTvec(k)
                    retType(retent, mapchosen, atable) } }
            if kk >= 1 do
                retEntity(retEvec, mapchosen, atable)
        }
    end
}
else if operationchosen = "" do
    finished := true }
let saving := message.proc("Save Results ?", "Yes", "No", xend-210, 120,
                           205, 140)
if saving do
    begin
        let atext := seditor("Enter Image Name", "", xstart+50, ystart+150,
                              200, 50 )
        let animage = image X.dim(screen) by Y.dim(screen) of off
        not screen onto screen
        copy screen onto animage
        not screen onto screen
        let QDB := open.database("MapImages", "query", "write")
        if QDB is error.record do QDB := create.database("MapImages", "query")
        if QDB is error.record do { write "Failed to open the database'n";
                                   abort }
        let Queries := s.lookup("TQ", QDB)
        if Queries = nil do
            Queries := table()
            s.enter("TQ", QDB, Queries)
            structure Query(string mapname; real XQL, YQL, XQR, YQR; *int PV, LV,
                           AV, TV; #pixel MapImage)
            let Xl := 0.0; let Yl := 0.0; let Xr := 0.0; let Yr := 0.0
            if YZR = 0 then {
                Xl := XOL; Yl := YOL; Xr := XOR; Yr := YOR }
            else { Xl := XZL; Yl := YZL; Xr := XZR; Yr := YZR }
            if point.count = 0 do point.count := 1
            let pv := vector 1::point.count of 0
            for k = 1 to point.count do
                pv(k) := point.vec(k)
            if line.count = 0 do line.count := 1
            let lv := vector 1::line.count of 0
            for k = 1 to line.count do
                lv(k) := line.vec(k)
            if pol.count = 0 do pol.count := 1
            let av := vector 1::pol.count of 0
            for k = 1 to pol.count do
                av(k) := polygon.vec(k)
            if text.count = 0 do text.count := 1
            let tv := vector 1::text.count of 0
            for k = 1 to text.count do
                tv(k) := text.vec(k)
            s.enter(atext, Queries, Query(mapchosen, Xl, Yl, Xr, Yr, pv, lv, av,
                                         tv, animage) )
            if commit() ~= nil do
                { error.message("Query Results are not stored'n", -1, -1); abort }
        end
    end
end
end
structure modules3( proc()dataret)
let moduleDB := open.database("M", "m", "write")
if moduleDB is error.record do moduleDB := create.database("M", "m")
let the.module := s.lookup("Module3", moduleDB)
if the.module = nil do
    begin
        the.module := table()
        s.enter("Module3", moduleDB, the.module)
    end
end
s.enter("dataret", the.module, modules3(DataRet) )
?

```

APPENDIX G

APPENDIX G: HARD-COPY DATA OUTPUT MODULE

This Appendix contains the listing of the program dealing with Hard-Copy Data Output. First, the module retrieves the Global Procedures needed from the database 'Global.Proc', then it lists the different procedures required and described in Chapter 10. Finally, the module is stored in the database '%\$Modules'.

```

let DataOut = proc()
begin
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
do {write "No utilities database - do procsdbmaker first'n"; abort}
let prcget=
begin
structure procpak(proc(string -> pnter) xproc)
s.lookup("prcget",procsdb) (xproc)
end
let seditor={structure procpak(proc(string,string,int,int,int,int->
string) xproc)
prcget("seditor") (xproc) }
let error.message={structure procpak(proc(string,int,int) xproc)
prcget("error.message") (xproc) }
let more={structure procpak(proc(*string,int,int) xproc)
prcget("more") (xproc) }
let form.generate={structure procpak(proc(-> pnter) xproc)
prcget("form.generate") (xproc)}
let form.null={structure procpak(proc(string,int,int,int,int,int,pnter) xproc)
prcget("form.null") (xproc) }
structure form.package( proc(pnter) Form.show;
proc() Form.all.show;
proc(string,int,int,int,int,int,bool,proc(),pnter ->
pnter) Form.add;
proc(pnter) Form.remove;
proc(string,pnter) Form.update;
proc() Form.clear;
proc(-> pnter) Form.mouse;
proc() Fender;
proc() Form.monitor )
let set.up.choose = {structure procpak(proc(*string -> pnter) xproc)
prcget("set.up.choose") (xproc)}
structure chooser.pack( proc(string, int, int -> string) do.choose;
proc(string) add.choose;
proc(string) remove.choose;
proc(int, int) list.choose )
let table.to.text = {structure procpak(proc(pnter -> *string) xproc)
prcget("table.to.text") (xproc)}
let QDB := open.database("MapImages", "query", "write")
if QDB is error.record do { write "Failed to open the database'n"; abort }
let Queries := s.lookup("TQ", QDB)
structure Query(string mapname; real XQL, YQL, XQR, YQR; *int PV, LV, AV,
TV; #pixel MapImage)
let mainDB := open.database("MDB","data","write")
if mainDB is error.record do mainDB := create.database("MDB", "data")
if mainDB is error.record do { error.message("Cannot open database'n",-1,-1)
; abort }
let mapstable := s.lookup("Maps", mainDB )
structure maps( string Package, SerialNo, Scale, North;real XL, YL, XR, YR,
Grid; *pnter AST, LST, PST, TST )
structure Aholder(int ATN, AID; bool AP; real Xc, Yc; string AFN, AC;
*real AX, AY; pic anypic; real AD1, AD2; int NOI; *int Inc
; pic FA.pic )
structure Lholder(int LTN, LID; bool LP; string LFN, LC; *real LX, LY;
pic L.pic; real LD1, LD2; pic FL.pic )
structure Pholder(int PTN, PID; bool PP; string PFN, PC, PT; *real PX, PY;
pic P.pic)
structure Tholder(int Layer, Location; real xpos, ypos; string thetext, Font)
let xmstart := X.dim(screen)-250
let Sht := X.dim(screen) - 540
structure global.proc(proc(string, string, string, int, int, int, int ->
bool)MessageProc;
proc(int, int, string, string, #pixel)Text.Write;
proc(string -> real)StringToReal;
proc(string -> int)StringToInt;
proc(*real, int -> *real)MinMax;
proc(int, int, string -> int)Icon;
proc(real, real, real, string -> pic)Polygon;
proc(*real, *real -> pic)highlight;
proc(real, real,real, real -> pic)Drawline;
proc(int, int, int)box;
proc(int, int, int, int)rectangle;
proc(int, int, int, int, string, string, string, bool)Rectan
proc(int, #pixel)North.Dir;

```

```

        proc(int -> string)Feature.Type;
        proc(pic, pic, int, int, int, #pixel)Zoomout;
        proc(pic, pic, int, int, int, int, #pixel)Zoomin;
        proc(*real, *real, real, real, real, real, int -> bool)Check
        proc(string)Prepform )
let ProcDB := open.database("Proc.Lib", "proc", "write")
let GLOBALS := s.lookup("Procedures", ProcDB)
let message.proc = GLOBALS(MessageProc)
let text.write = GLOBALS(Text.Write)
let stringtoreal = GLOBALS(StringToReal)
let stringtoint = GLOBALS(StringToInt)
let minmax = GLOBALS(MinMax)
let icon = GLOBALS(Icon)
let polygon = GLOBALS(Polygon)
let Highlight = GLOBALS(highlight)
let drawline = GLOBALS(Drawline)
let Box = GLOBALS(box)
let rec = GLOBALS(rectangle)
let Rec = GLOBALS(Rectangle)
let north.dir = GLOBALS(North.Dir)
let feature.type = GLOBALS(Feature.Type)
let checkin = GLOBALS(Checkin)
let prepform = GLOBALS(Prepform)
let out.range := X.dim(screen) - Y.dim(screen)
let range := Y.dim(screen)
let gw = limit screen to range by range at 0, 0
let xl := 0.0; let yl := 0.0
let xr := 0.0; let yr := 0.0
let the.win = limit screen to out.range-30 by 60 at range+10, 200
let win1 = limit screen to 40 by 30 at range+20, 202
let win2 = limit screen to 40 by 30 at range+135, 202
let B1 = vector 1::100 of 0.0
let B2 = vector 1::100 of 0.0
let fod := nullfile
let N := 0
let tempN := 0
let showSymbol := proc( pntr P; string I; int W ); nullproc
showSymbol := proc( pntr P; string I; int W )
begin
    !write I, class.identifier( P )( 1 | 10 )
    case true of
        P is trnsfrm.strc:
            begin
                case true of
                    P(trnsfrm)=1: write I, "Scale by ", P(trnsfrm.x):6, ", ",
                        P(trnsfrm.y):6
                    P(trnsfrm)=2: write I, "Shift by ", P(trnsfrm.x):6, ", ",
                        P(trnsfrm.y):6
                    P(trnsfrm)=3: write I, "Rotate by ", P(trnsfrm.x):6, " degrees "
                    default: write "?????"
                showSymbol( P(mrtre), I ++ " ", 0 )
            end
        P is oprtn.strc:
            begin
                if P(opoo) = W
                then begin
                    showSymbol( P(lft), I, P(opoo) )
                    showSymbol( P(rght), I, P(opoo) )
                end
                else begin
                    if N > 0 do
                        begin
                            if N=1 and B1(N) = 0 and B2(N) = 0 then { }
                            else if N > 1 and B1(N) = 0 and B2(N) = 0 then
                                { N := N - 1
                                output fod, N,"'n"
                                for i=1 to N do
                                    output fod, B1(i), B2(i),"'n" }
                            else {
                                output fod, N,"'n"
                                for i=1 to N do
                                    output fod, B1(i), B2(i),"'n" }
                            N := 0
                        end
                    end
                case true of

```

```

        P(opoo) = 1: {}
        P(opoo) = 2: {}
        default: write "?????"
        showSymbol( P(lft), I ++ " ", P(opoo) )
        showSymbol( P(right), I ++ " ", P(opoo) )
    end
end
P is poin.strc:
begin
    N := N+1
    B1(N) := P(pnx)
    B2(N) := P(pny)
end
default: write "?????" ",class.identifier( P ),"n"
end

let showOne = proc( pic apic -> bool )
begin
    N := 0
    showSymbol( pic.pntr(apic), "", 0 )
    if N > 0 do
        begin
            tempN := N
            N := 0
        end
    true
end
let Laserdump = proc()
begin
    error.message("When Ready Click the Mouse", -1, -1)
    let dummy := system( "date '"+Map Output for $USER on %a %h %d, %y at
                        %H:%M.) show'" > /tmp/lsd89.$$" )
    dummy := system("screendump | strip head.sun3 | bin_hex.sun3 |
                    cat lsd89.head - lsd89.rear /tmp/lsd89.$$ lsd89.tail | lpr -Plws101" )
end
let Plotdump = proc(string o.mapname, n.mapname; *int Aid, Lid, Pid; pntr atable)
begin
    rec(range+11, 201, out.range-32, 58)
    text.write(10, 40, "Total          Current", "fixb13", the.win)
    output fod, o.mapname,"n"
    output fod, xl, " ", yl, " ", xr, " ", yr, "n"
    let asize := if upb(Aid) > 0 then upb(Aid) - lwb(Aid) + 1
                else upb(Aid) - lwb(Aid)
    let lsize := if upb(Lid) > 0 then upb(Lid) - lwb(Lid) + 1
                else upb(Lid) - lwb(Lid)
    let psize := if upb(Lid) > 0 then upb(Pid) - lwb(Pid) + 1
                else upb(Pid) - lwb(Pid)
    let the.total := asize + lsize + psize
    let abool := false; let lbool := false; let pbool := false
    !if asize = 1 then {
        !let Afeat := s.lookup(o.mapname, atable)(AST)
        while ~abool do
            if asize > 0 do
                begin
                    if Aid(1) = 0 then { write Aid(1); the.total := the.total - 1;
                                        abool := true; write abool }
                    else {
                        xor win1 onto win1
                        xor win2 onto win2
                        print the.total at range+30, 210
                        let Afeat := s.lookup(o.mapname, atable)(AST)
                        for i = 1 to asize do {
                            let k := Aid(i); write k
                            !let k := Afeat(kk)(ATN)
                            print i at range+145, 210
                            let thepic := Afeat(k)(anypic)
                            let rpik := Afeat(k)(FA.pic)
                            let fpic := thepic & rpik
                            let thebool := showOne(fpic)
                            if i = asize do abool := true }
                        end
                    }
                end
            }
        if lsize > 0 do {
            xor win1 onto win1
            xor win2 onto win2
            print the.total at range+30, 210

```



```

let lfeat := s.lookup(o.mapname, atable) (LST)
for i = 1 to lsize do {
    print i+asize at range+145, 210
    let k := lfeat(i) (LTN)
    let thepic := lfeat(k) (FL.pic)
    let thebool := showOne(thepic)
} }
if psize > 0 do {
    xor win1 onto win1
    xor win2 onto win2
    print the.total at range+30, 210
    let Pfeat := s.lookup(o.mapname, atable) (PST)
    for i = 1 to lsize do {
        print i+asize+lsize at range+145, 210
        let k := Pfeat(i) (PTN)
        let thepic := Pfeat(k) (P.pic)
        let thebool := showOne(thepic)
    } }
    xor the.win onto the.win
end
let parwin1 := limit screen to 120 by 30 at range+110,Y.dim(screen)-85
prepform("Data Output")
let MapChosen := ""
let OMapName := ""
let finished := false
while ~finished do
    begin
        let key.queries = table.to.text(Queries)
        let choosemap = set.up.choose(key.queries) (do.choose)
        let mapchosen = choosemap("Choose Map of:", xmstart+50, 120)
        if mapchosen = "" or mapchosen = " " then finished := true
        else {
            let theimage := s.lookup(mapchosen, Queries) (MapImage)
            OMapName := s.lookup(mapchosen, Queries) (mapname)
            text.write (10, 14, OMapName,"fix13",parwin1)
            xl := s.lookup(mapchosen, Queries) (XQL)
            yl := s.lookup(mapchosen, Queries) (YQL)
            xr := s.lookup(mapchosen, Queries) (XQR)
            yr := s.lookup(mapchosen, Queries) (YQR)
            copy theimage onto gw
            MapChosen := mapchosen
            fod := create(MapChosen ++ ".coo", 422)
        }
    end
let aid := s.lookup(MapChosen, Queries) (AV)
let lid := s.lookup(MapChosen, Queries) (LV)
let pid := s.lookup(MapChosen, Queries) (PV)
let operations = @1 of string [ "Laser Printer", "Plotter" ]
!write MapChosen
finished := false
while ~finished do
    begin
        let chooseoperation = set.up.choose(operations) (do.choose)
        let operationchosen = chooseoperation( "Select Output Device",
                                                xmstart+50, 120)
        if operationchosen = "Laser Printer" then Laserdump()
        else if operationchosen = "Plotter" then Plotdump(OMapName, MapChosen,
                                                         aid, lid, pid, mapstable)
        else if operationchosen = "" do finished := true
    end
end
close(fod)
end
structure modules4( proc()dataout)
let moduleDB := open.database("M", "m", "write")
if moduleDB is error.record do moduleDB := create.database("M", "m")
let the.module := s.lookup("Module4", moduleDB)
if the.module = nil do
    begin
        the.module := table()
        s.enter("Module4", moduleDB, the.module)
    end
end
s.enter("dataout", the.module, modules4(DataOut) )
?

```

APPENDIX H

APPENDIX H: THE OPERATIONAL MANAGEMENT SYSTEM

This Appendix lists the main part of the system. It is concerned with linking the different modules into an operational organization, thus forming the operational management system discussed in Chapter 6. This program also handles the calls to the Global Procedures.

```

let FONTsdb:=open.database("FONTS","friend","read")
let fix13 = s.lookup( "fix13", FONTsdb)
let bold = s.lookup("fixb13",FONTsdb)
let big = s.lookup("met22",FONTsdb)
let large = s.lookup("hci45i",FONTsdb)
let procsdb:=open.database("rutilities","friend","read")
if procsdb is error.record
do {write "No utilities database - do prcdbmaker first'n"; abort}
let prcget=
begin
structure procpak(proc(string -> pntr) xproc)
s.lookup("prcget",procsdb) (xproc)
end
let seditor={structure procpak(proc(string,string,int,int,int,int-> string) xproc
prcget("seditor") (xproc) }
let error.message={structure procpak(proc(string,int,int) xproc)
prcget("error.message") (xproc) }
let more={structure procpak(proc(*string,int,int) xproc)
prcget("more") (xproc) }
let form.generate={structure procpak(proc( -> pntr ) xproc)
prcget("form.generate") (xproc)}
let form.null={structure procpak(proc( string,int,int,int,int,int,pntr ) xproc)
prcget("form.null") (xproc) }
structure form.package( proc( pntr) Form.show;
proc( ) Form.all.show;
proc( string,int,int,int,int,bool,proc(),pntr -> pntr
) Form.add;
proc( pntr ) Form.remove;
proc( string,pntr ) Form.update;
proc( ) Form.clear;
proc( -> pntr ) Form.mouse;
proc( ) Fender;
proc( ) Form.monitor )
let set.up.choose = {structure procpak(proc(*string -> pntr) xproc)
prcget("set.up.choose") (xproc)}
structure chooser.pack( proc( string, int, int -> string ) do.choose;
proc( string ) add.choose;
proc( string ) remove.choose;
proc( int, int ) list.choose )
let table.to.text = {structure procpak(proc(pntr -> *string) xproc)
prcget("table.to.text") (xproc)}
structure global.proc(proc(string, string, string, int, int, int, int ->
bool)MessageProc;
proc(int, int, string, string, #pixel)Text.Write;
proc(string -> real)StringToReal;
proc(string -> int)StringToInt;
proc(*real, int -> *real)MinMax;
proc(int, int, string -> int)Icon;
proc(real, real, real, string -> pic)Polygon;
proc(*real, *real -> pic)highlight;
proc(real, real,real, real -> pic)Drawline;
proc(int, int, int)box;
proc(int, int, int, int)rectangle;
proc(int, int, int, int, int, string, string, string, bool
)Rectangle;
proc(int, #pixel)North.Dir;
proc(int -> string)Feature.Type;
proc(pic, pic, int, int, int, #pixel)Zoomout;
proc(pic, pic, int, int, int, int, #pixel)Zoomin;
proc(*real, *real, real, real, real, real, int -> bool)Check
proc(string)Prepform )
let ProcDB := open.database("Proc.Lib", "proc", "write")
let GLOBALS := s.lookup("Procedures", ProcDB)
let message.proc = GLOBALS(MessageProc)
let text.write = GLOBALS(Text.Write)
let stringtoreal = GLOBALS(StringToReal)
let stringtoint = GLOBALS(StringToInt)
let minmax = GLOBALS(MinMax)
let icon = GLOBALS(Icon)
let polygon = GLOBALS(Polygon)
let Highlight = GLOBALS(highlight)
let drawline = GLOBALS(Drawline)
let Box = GLOBALS(box)
let rec = GLOBALS(rectangle)
let Rec = GLOBALS(Rectangle)

```

```

let north.dir = GLOBALS(North.Dir)
let feature.type = GLOBALS(Feature.Type)
let checkin = GLOBALS(Checkin)
let prepform = GLOBALS(Prepform)
structure modules1( proc()dataentry)
structure modules2( proc()cartorep)
structure modules3( proc()dataret)
structure modules4( proc()dataout)
let moduleDB := open.database("M", "m", "write")
let Module1 := s.lookup("Module1", moduleDB)
let trans.code = Module1(dataentry)
let Module2 := s.lookup("Module2", moduleDB)
let data.prep = Module2(cartorep)
let Module3 := s.lookup("Module3", moduleDB)
let retrieve = Module3(dataret)
let Module4 := s.lookup("Module4", moduleDB)
let result = Module4(dataout)
let first.screen = proc()
begin
    Rec(0,0,X.dim(screen),Y.dim(screen),"Ps - GIS","cou20",
        "middle", true)
    let border1 = limit screen to X.dim(screen)-10 by Y.dim(screen)-43 at 5,5
    let border2 = limit screen to X.dim(screen)-20 by Y.dim(screen)-53 at 10,10
    nor border1 onto border1
    nor border2 onto border2
end
let finished := false
let first := 1
let mttitle = string.to.tile(" OPTIONS ", "cou20")
let mentries = @1 of #pixel
[ string.to.tile("DATA TRANSFER", "fix13"),
  string.to.tile("DATA ENHANCEMENT", "fix13"),
  string.to.tile("DATA RETRIEVAL", "fix13"),
  string.to.tile("OUTPUT", "fix13"),
  string.to.tile("QUIT", "fix13")]

let mactions = @1 of proc(c#pixel,cint)
[proc(c#pixel Y;cint z);trans.code (),
 proc(c#pixel Y;cint z);data.prep (),
 proc(c#pixel Y;cint z);retrieve (),
 proc(c#pixel Y;cint z);result (),
 proc(c#pixel Y;cint z);finished := true]
xor screen onto screen
let locate = proc(string anystring, Font -> int)
begin
    let size := case true of
        Font = "met22" : 14
        Font = "hcl45i" : 22
        Font = "cou20" : 14
        default : 8
    let alength := truncate(length(anystring)* size / 2)
    let pos := X.dim(screen) div 2 - alength
    pos
end
let h.xpos := locate("GLASGOW UNIVERSITY", "met22")
copy string.to.tile("GLASGOW UNIVERSITY", "met22") onto limit screen at h.xpos,
Y.dim(screen) -100
h.xpos := locate("Ps-GIS", "hcl45i")
copy string.to.tile("Ps-GIS", "hcl45i") onto limit screen at h.xpos,
Y.dim(screen) -200
h.xpos := locate("Persistent Geographic Information System", "cou20")
copy string.to.tile("Persistent Geographic Information System", "cou20") onto
limit screen at h.xpos, Y.dim(screen) -250

let mmenu = menu(mttitle,mentries,true,mactions)
write code(27), "N", code(27), "I"
while ~finished do
begin
    let pdone := false
    let mxpos := X.dim(screen) - 190
    let mypos := 120
    while pdone = false do pdone := mmenu(mxpos,mypos)
end
write code(27), "E", code(27), "W"
finished := false

```