

**A MULTIMEDIA PROTOTYPE
FOR ANNOTATION AND ILLUSTRATION USING
THE MICROSOFT FOUNDATION CLASS LIBRARY AND C++**

Eur Ing Pratul Chandra Chatterjee BTech (Hons), PhD, CEng, MRINA
Department of Computing Science
University of Glasgow

THIS THESIS IS SUBMITTED FOR THE DEGREE OF
MASTER OF SCIENCE



**UNIVERSITY
of
GLASGOW**



© Pratul C. Chatterjee, 2000

ProQuest Number: 13818554

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 13818554

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

GLASGOW
UNIVERSITY
LIBRARY

12050-Copy 1

DECLARATION

Except where reference is made to the work of others,
this thesis is believed to be original.

1999

10

ACKNOWLEDGEMENTS

I am grateful to the Department of Mechanical Engineering, University of Glasgow for their financial support which enabled the work presented in this thesis.

I wish to express my deepest gratitude to my Supervisor, Dr R. Poet for his advice and timely encouragement throughout the duration of my research.

I would like to thank Dr R. C. McGregor of the Mechanical Engineering Department for his invaluable help.

I would like to take this opportunity to thank all the staff in the Research and Development Department at Lloyd's Register, London who in one way or another helped me in producing this work. This especially includes Dr S. Rutherford, who reviewed some chapters of this thesis.

Finally, I am forever indebted to my wife, Keya for her patience and invaluable support throughout my study.

THESIS LAYOUT

The thesis is arranged in eight chapters, each of which has its own tables, figures and references. Citations in the text contain the author's name and date and they are linked to the alphabetical list of references at the end of each chapter. The numbering system for sections, tables and figures starts with the chapter number in front. The text is written in British English with the exception of words such as serialization to keep the text consistent with the C++ code fragments. The word processing application used to prepare all the chapters is Microsoft Word 97.

CONTENTS

Declaration	<i>i</i>
Acknowledgements	<i>ii</i>
Thesis Layout	<i>iii</i>
Contents	<i>iv</i>
Summary	<i>x</i>

CHAPTER 1 INTRODUCTION

1.1	Software Systems Based on Procedural Code	1-1
1.2	Problems with Earlier User Interfaces	1-2
1.3	Object-Oriented Design and Implementation	1-3
1.4	Evolution of Application Frameworks	1-4
1.4.1	GUI Libraries	1-5
1.4.2	Framework Class Libraries	1-6
1.4.3	Examples of Framework Class Libraries	1-7
1.5	The Main Research Objectives	1-9
1.6	Application Framework Selection	1-11
1.6.1	The Essentials of a <i>Windows</i> Program	1-12
1.6.2	The MFC Library	1-13
1.7	References	1-14

CHAPTER 2 OBJECT PERSISTENCE AND C++

2.1	Introduction	2-1
2.2	Stream Facilities in C++	2-1
2.3	Hidden Pointers	2-2
2.3.1	Retrieval Using a Special Constructor	2-3
2.3.2	Problem with the Special Constructor	2-4
2.3.3	Retrieval Using the Assignment Operator	2-5
2.4	Memory Management and Smart Pointers	2-6
2.4.1	Smart Pointers	2-7
2.4.2	Evaluation of Smart Pointers	2-9
2.5	Relational Databases	2-9
2.5.1	Mapping Objects to Tables	2-9

2.5.2	Binary Large Objects (BLOB)	2-10
2.6	Object Oriented Databases	2-11
2.6.1	Querying an OODB	2-11
2.7	Pointer Swizzling at Page-Fault Time	2-12
2.7.1	Two Level Storage Approach	2-13
2.7.2	Difficulties in Using the OS Support for Persistence	2-14
2.7.3	The Essentials of Pointer Swizzling	2-15
2.7.4	Wilson's Approach	2-16
2.7.5	ObjectStore	2-16
2.8	Orthogonal Persistence	2-17
2.9	Closure	2-17
2.10	References	2-18

CHAPTER 3 SERIALIZATION AND MFC

3.1	Introduction	3-1
3.2	The Problems	3-2
3.2.1	Pointer Storage	3-2
3.2.2	Virtual Constructor	3-2
3.2.3	Base Class Pointer	3-3
3.3	A Serializable Class	3-3
3.4	The Macros in MFC	3-4
3.4.1	Run Time Class Information (RTCI)	3-4
3.4.2	Dynamic Creation	3-8
3.5	A Sample Data Structure for Serialization	3-8
3.5.1	An Illustration in GGS	3-11
3.6	Storing Objects	3-12
3.6.1	Storing Vertices and Edges	3-15
3.7	The Serialization Macros	3-17
3.8	On-the-fly Registration and Restoration	3-17
3.9	Versionable Objects	3-18
3.10	Limitations	3-21
3.11	Closure	3-22
3.12	References	3-23

CHAPTER 4 USER INTERFACE AND CLASSES IN GGS

4.1	Principles of User Interface Design	4-1
4.2	Standard GUI of Applications for <i>Windows</i>	4-2
4.3	Separation of Document and View	4-3
4.4	The Document/View Architecture in MFC	4-5
4.4.1	Documents	4-5
4.4.2	Views	4-5
4.4.3	View Frames	4-6
4.4.4	Document Templates	4-6
4.5	Multiple Document Interface (MDI)	4-8
4.6	The Contributions of MFC in GGS	4-9
4.6.1	ToolTips	4-10
4.6.2	Message Boxes	4-10
4.6.3	Status Bar Messages	4-10
4.7	High Level Decomposition of GGS	4-10
4.8	A Case Study Walkthrough	4-12
4.8.1	A Protein Molecule	4-13
4.8.2	Importing the Picture	4-14
4.8.3	The Rectangle Around the Picture	4-15
4.8.4	Recording Sound Objects	4-16
4.8.5	A Text Object	4-17
4.8.6	Creating Timers	4-18
4.8.7	Creating Arrows	4-18
4.8.8	Arranging the Sequence of Objects in the Animation Editor	4-19
4.8.9	Animation	4-20
4.9	Class Hierarchy in GGS	4-20
4.10	Class Construction	4-21
4.10.1	Public Data Member?	4-22
4.10.2	Thin API Wrapper	4-22
4.10.3	Copy Constructor and Assignment Operator	4-23
4.10.4	Other Guidelines	4-24
4.10.5	Commenting Convention	4-24
4.11	Closure	4-25
4.12	References	4-25

CHAPTER 5	GRAPHICS PRIMITIVES IN GGS AND OTHER FEATURES	
5.1	Collections of Objects	5-1
5.2	Drawing GGS Document	5-2
5.2.1	The Parent Class CGGXObject	5-4
5.2.2	Drawing Sound and Timer Objects!	5-5
5.3	Rubber Banding	5-6
5.3.1	Programming the Mouse	5-7
5.3.2	Raster Operations	5-8
5.3.3	Drawing Curve Objects	5-9
5.4	Deleting and Moving Shapes	5-11
5.4.1	Highlighting Shapes	5-11
5.4.2	Moving Shapes	5-12
5.4.3	Masked Objects	5-13
5.5	Implementing Scrolling with Scaling	5-14
5.5.1	Scaleable Mapping Modes	5-15
5.5.2	Transformation of Coordinates	5-15
5.5.3	Restoring Scrolling	5-17
5.6	Text Objects	5-18
5.6.1	Moving Text Objects	5-19
5.7	Benefits of Serialization	5-20
5.8	Multi-page Printing	5-21
5.8.1	Paper Size	5-23
5.8.2	Preview and Printing	5-24
5.9	References	5-25
CHAPTER 6	BITMAPS AND METAFILES	
6.1	Graphics Device Interface (GDI)	6-1
6.2	Bitmaps and Metafiles	6-1
6.3	Colours in Bitmaps	6-2
6.4	The Requirements in GGS	6-3
6.5	The DIB's in GGS	6-4
6.5.1	Construction and Destruction	6-5
6.5.2	Reading a BMP File	6-6
6.5.3	User Defined DIB Size	6-7
6.5.4	Drawing DIB's	6-8
6.5.5	Moving DIB's	6-8

6.6	DIB Compression	6-9
	6.6.1 RLE Compression	6-10
6.7	Palette Programming	6-10
	6.7.1 Palettes in GGS	6-11
6.8	DIB Serialization	6-12
6.9	Different Types of Metafiles	6-13
	6.9.1 <i>Windows</i> Metafiles Versus Enhanced Metafiles	6-14
	6.9.2 Device Independence	6-15
6.10	Metafiles in GGS	6-17
	6.10.1 Construction and Destruction	6-18
	6.10.2 Reading a Metafile	6-18
	6.10.3 Drawing a Metafile	6-19
	6.10.4 Basic OpenGL Operations	6-19
	6.10.5 Enhanced Metafiles with OpenGL Records	6-21
6.11	Closure	6-22
6.12	References	6-23

CHAPTER 7 SOUND, TIMERS AND STILL ANIMATION

7.1	Audio Clips	7-1
7.2	Digital Sound	7-1
	7.2.1 What is MP3?	7-1
	7.2.2 WAVE and MIDI Files	7-2
	7.2.3 The Contents of a WAVE File	7-3
7.3	Waveform Audio and <i>Windows</i>	7-4
	7.3.1 The PlaySound API	7-4
	7.3.2 Low-level Audio Services	7-5
7.4	The Media Control Interface (MCI)	7-5
7.5	MCI Audio Architecture	7-7
7.6	Sound Objects in GGS	7-7
	7.6.1 Importing WAVE Files	7-8
	7.6.2 Playing WAVE Files	7-9
	7.6.3 Recording WAVE Files	7-10
7.7	Timers in <i>Windows</i>	7-12
	7.7.1 The Recording in Progress Dialogue	7-13
	7.7.2 Other Member Functions in CSound	7-13
7.8	Still Animation	7-14
7.9	Timers in GGS	7-15
7.10	References	7-16

CHAPTER 8 USER FEEDBACK AND CONCLUSIONS

8.1	Software Validation	8-1
8.2	Software Quality	8-2
	8.2.1 Process or Product?	8-2
8.3	User Feedback on GGS	8-3
	8.3.1 The Checklist	8-3
	8.3.2 User Feedback	8-4
	8.3.3 Feedback from MFC Developers	8-5
8.4	GGs as an OLE Server - an Afterthought	8-5
	8.4.1 Steps to Provide OLE Server Support After the Fact	8-6
	8.4.2 Embedded GGS Items	8-6
	8.4.3 Why not an ActiveX Document Server?	8-8
8.5	Conclusions	8-9
	8.5.0 Criteria for Evaluating Application Frameworks for Developing Multimedia Applications	8-9
	8.5.1 Code Reusability	8-10
	8.5.2 Fast Development Time	8-11
	8.5.3 Rapid Software Prototyping	8-12
	8.5.4 Separation of Concerns	8-14
	8.5.5 A Rich Set of Widgets	8-15
	8.5.6 Serialization	8-15
	8.5.7 Escape Mechanisms	8-15
	8.5.8 Defensive Programming	8-16
8.6	Last Words	8-17
8.7	References	8-17

SUMMARY

One of the major claims of the object-oriented programming approach is that it facilitates the development of complex programs by allowing reuse of components. Most compilers for object-oriented languages are now supplied with class libraries. In addition to those provided with the compilers, there are many others in the public domain or available from commercial suppliers. Code reuse can be maximised through the exploitation of framework class libraries for creating interactive programs. A framework library can be viewed as providing a skeleton application that can be extended and specialised through class inheritance. The evolution of application frameworks is discussed briefly in Chapter 1 with an objective to utilise one of them to develop a prototype multimedia application for annotation and illustration. This prototype is referred to as Glasgow Graphics and Sound (GGS) in this thesis.

GGS deals with externally created vector or bitmap images, graphics primitives and sound objects in any sequence. GGS is designed to provide the end-users with facilities to work on external images with free-hand curves and other graphics tools, record their voice, save everything in one disk file and animate them later, if necessary. GGS has the responsibility to store different objects without knowing in advance the sequence of object types the user will create. The implementation language, C++ does not have any built-in support for object persistence. Hence, a number of techniques and strategies for adding persistence to C++ objects are reviewed in Chapter 2.

The Microsoft Foundation Class (MFC) library is selected as the application framework for developing GGS and the serialization mechanism in MFC is chosen to deal with the object persistence issues. Some of the techniques for persistence, discussed in Chapter 2, are powerful but incur unacceptable overheads for lightweight applications. On the other hand, the MFC serialization is found very useful in creating transportable stream of bytes that can be stored in a file and sent away as an e-mail attachment.

Chapter 3 presents the serialization internals in MFC and uncovers some undocumented details that are believed to be valuable for other MFC users. From an application programmer's viewpoint, it is straightforward to use the MFC serialization in most cases. However, the actual implementation details are complex. A sample data structure is serialized and analysed step-by-step to explain the MFC serialization mechanism.

The user-friendliness of applications comes not only from an iconic user interface but also from a uniform user interface across applications. Some common user interface elements and their importance are discussed in Chapter 4 along with the document/view architecture in MFC that separates an application's data management code from its user interface code. The multiple document interface (MDI) in GGS is based on this document/view architecture. A case study walkthrough is presented, purely from an end-user's viewpoint, to illustrate a simple use of GGS. The main classes and their hierarchy are drafted in Chapter 4 based on a high-level decomposition of GGS.

Chapter 5 presents the final class hierarchy, different drawing operations and other features involving graphics primitives. Template based type-safe collection classes are used in GGS to store pointers to objects of any type. This simplifies the interaction with the document class. Basic drawing operations such as moving, deleting and highlighting graphics primitives on the screen use an efficient raster drawing mode. The implementation of view magnification together with the standard scrolling capabilities in a window is discussed that requires some special techniques. The benefits of trapping some uncommon messages from the operating system are also discussed. Chapter 5 ends with an overview of the printing process and a description of the multi-page printing features in GGS.

Chapter 6 starts with a general discussion on bitmaps and metafiles. A bitmap is a complete digital representation of a picture. Each pixel in the image corresponds to one or more bits in the bitmap. A metafile, on the other hand, stores pictorial information as a series of records that correspond directly to the graphics device interface (GDI) calls. GGS can import externally created bitmaps and metafiles and treat them like any other graphic or sound objects. All commercial illustration programs do something similar. However, the motivation for developing GGS is slightly different. GGS allows the users to construct and manipulate a fairly complex picture, adding comments as they go. The process of constructing the picture is saved, not just the final picture.

Sound can be an effective form of information and interface enhancement when appropriately used. It can serve purposes other than the transmission of details or factual information. Chapter 7 describes the facilities in GGS to record someone's voice or import sounds from WAVE files which are digital copies of the air pressure alterations of recorded sounds. The sequence of graphic and sound objects can be edited and separated by timers. In the animation mode, graphic objects are rendered

on the screen, sounds are played and timers are activated. One of the major objectives is to produce GGS documents containing graphic and sound objects that can be delivered as e-mail attachments. GGS avoids fluid animation to restrict the size of these documents. However, the still animation in GGS can be an effective way to get some ideas across and create an impact with a meaningful sequence of objects.

Executable copies of GGS were distributed with an aim to receive feedback about their ease of use, compatibility with other products and stability. The feedback is discussed in Chapter 8. The idea of playing sounds and rendering graphic objects separated by timers was appreciated by a number of users. One of the suggestions was to convert GGS from a stand-alone application to a full OLE server to integrate and take advantage of the features offered by other Windows applications. With the help of MFC, the author found that the conversion was not a difficult exercise. Finally, Chapter 8 ends with the main conclusions.

CHAPTER 1

INTRODUCTION

1.1 SOFTWARE SYSTEMS BASED ON PROCEDURAL CODE

There are various programs available in the engineering domain that consist of several hundred thousand lines of procedural code, usually written in Fortran. They serve as special-purpose tools for the modelling and simulation of physical systems in diverse fields such as structural mechanics, fluid dynamics, electro-magnetics and many others.

For the sake of efficiency, various components of such program often directly access the program's data structures. This compounds the complexity of the components, by requiring knowledge of the program's data structures. Modification or extension to a component requires not only the knowledge of the component at hand, but also a high degree of knowledge of the entire program.

The components of the software system become intimately tied to the program's data structures. The access to the data structures easily becomes inseparable from the components' function. Since the layouts of the data structures are unique to each program, the possibility of the reuse of the code in other systems is greatly diminished. Also, code from other programs is difficult to adapt for use within the system.

Since the data structures are globally accessible, a small change in them can have a ripple effect throughout the program. All portions of the code that access the affected data structures must be updated. Consequently, the layouts of the data structures tend to become fixed regardless of how appropriate they remain as the code evolves.

The components of the software system become dependent on each other via their common access to the data structures. Little control can be placed on the access. As a result, these interdependencies are numerous. More importantly, they are implicit. One component can be completely unaware of another's reliance on the same data structure. Thus, when modification or extension to a component occurs, it is difficult to assure that all affected portions of the code are adjusted accordingly.

The access to the program's data structures is usually described by an interface. The interface may consist of a document describing the layout of the data, or may get as involved as a memory management tool that shields the data from the component routines. In either case, it is up to each programmer to honour the interface. The

best laid plans are easily subverted for the sake of efficiency and ease of implementation.

A large number of existing software systems based on procedural codes are inflexible and present a barrier to practising engineers and researchers. Recoding these systems in a new language will not remove this inflexibility. Instead, a redesign is needed.

1.2 PROBLEMS WITH EARLIER USER INTERFACES

Older style scientific and business programs tended to have a modal structure with very rigid control flows: 1) get the input data, 2) process the data, 3) print the results. Any data input mode would be under the program's control, with the program prompting the user for successive data elements that had to be provided in a fixed order. When interactions with users are under the program's control, the programmer's task is simplified. Because the sequence of any interactions is known in advance the programmer can plan to have all required data structures created and correctly initialised prior to use.

Early programs were "user-hostile" in a way, demanding their data in precise formats and requiring arcane command languages to control their operation. Such "user-hostility" did not matter because the users of such programs were neither perceptive enough to notice nor assertive enough to complain. Nowadays, every program is expected to be "user-friendly" and to indicate this "friendliness" through graphics displays replete with button controls, pop-up menus, and scrollable views.

Such user interface requirements add extra complexity to programs. Developers must implement a friendly user interface as well as implementing the data structures and algorithms required for some computational problem. Fortunately, one facet of "user friendliness" is consistency. All programs running on a particular platform are expected to work in similar style. Their windows should have the same functionality and controls (drag region, grow box, close box, etc.); a menu bar should be placed at the top of each window or at the top of the screen; dialog boxes are to be used to select processing options and should use a variety of standardised controls such as check boxes and groups of radio buttons. Such requirements for consistency really make the reuse of interface components a part of the specification of a program. Reuse becomes mandated, it is not simply a matter of enhanced productivity.

However, things get more difficult when the user acts and the program reacts. A

characteristic of a user driven program is complexity in the flow of control. User commands such as a change to the page setup, the resizing of a window, or an editing action that adds or removes data, can occur in any order. Typically, each such command will affect different aspects of the run-time environment - aspects such as menu options, data displays, and window/view structures. Since most of the user commands affect the underlying data, they may necessitate updating the amount of "travel" in any scroll-bars that may be associated with a window displaying the data. Consequently, the code determining this travel may have to be called from any of several quite different contexts. It is in cases like this that problems arise. A programmer may fail to perceive an implicit consequence of some user command; this failure could result in programs with abnormal behaviours (e.g., a program whose scroll-bars appear oblivious to data editing actions and only respond when a window is explicitly resized). Alternatively, the programmer produces some ad hoc solution with a complex flow graph.

Complexity of the control flow often results in program errors. After all, routines are typically written assuming a particular call pattern, with concomitant prior creation and initialisation of all necessary data structures. When other call sequences are imposed later, initialisation steps can be forgotten and so errors may arise. Often these errors seem intermittent and so are difficult to detect. Problems only arise with non-standard patterns of interaction. If the programmer implementing an interactive editor lacks a strong model, the code tends to become convoluted, unreliable, and difficult to maintain or extend.

1.3 OBJECT-ORIENTED DESIGN AND IMPLEMENTATION

The application of object-oriented design has proven to be very beneficial to the development of flexible programs. The basis of object-oriented design is abstraction. The abstraction forms a stable definition of objects in which the relationships among the objects are explicitly defined. The implicit reliance on another component's data does not occur. Thus, the design can be extended with minimal effort.

The object oriented paradigm provides four fundamental concepts: objects, classes, inheritance, and polymorphism. A software is organised into objects that store both its data and the operators that work on this data. This permits developers to abstract out the essential properties of an object; those that will be used by other objects. This abstraction allows the details of the implementation of the object to be hidden, and thus easily modified. Objects are instances described by a class definition. Classes are related by inheritance. A subclass inherits behaviour through the attributes and

operators of the base class. Polymorphism allows the same operation to behave differently in different classes and thus allows objects of one class to be used in place of those of another related class.

1.4 EVOLUTION OF APPLICATION FRAMEWORKS

One of the main claims of the object-oriented programming approach is that it facilitates the development of complex programs by allowing reuse of components. Most compilers for OO languages are now supplied with class libraries. In addition to those provided with the compilers, there are many other class libraries that are either available from commercial suppliers or which exist as public domain code (usually accessible from archive sites on the Internet). Most of the public domain class libraries are for C++ on UNIX, with a few for the PC platform.

Some of these class libraries are simply collections of supposedly useful components. The classes in these libraries will typically define "abstract" (programmer defined) data types. Instances of these classes can be used to manage particular resources. For example, a library might provide a class that represents a kind of "sparse array manager". An instance of this class could be created, in a client program, to provide an initially empty array to which elements could be added, accessed, modified, or removed without the client programmer having any need to be concerned with storage management issues or the coding of efficient access methods. As another example, a library class might simply provide a "wrapper" around some system data, for example the data defining the time and date. The library class would provide several alternative ways of accessing the time data, e.g., as a string, as millisecond count, etc.

The majority of classes in such a library will be "concrete" (i.e., directly instantiable) with minimal or no provision for further extension or specialisation. Although such class libraries are useful, they do not make a dramatic impact on program development. Nonetheless, it is often better to use the classes from a library. Usually, the library classes are thoroughly tested and shown to perform correctly whereas it is quite likely that a hurriedly implemented version will contain errors in the handling of less common cases.

Libraries with classes that are designed to be extended or specialised by client programmers do have a more significant impact on the cost of developing new applications. Some or most of the classes in these libraries will be partially implemented abstract classes. Each class will define some concept (abstract type)

and specify its behaviours. Default implementations of many of the behaviours may be provided, while the implementation of others will be “deferred” (i.e., left as pure virtual functions in C++). In some cases, the designer would have been able to define sensible default implementations for all the behaviours of a class. Any such class will be instantiatable, but will usually have provision for further specialisation by subclassing. The designer of each library class should consider the needs of programmers carefully who must implement the derived classes. Hence, protected access functions would typically be provided for most (maybe for all) private data members or the data members might be given protected access status. The majority of the member functions would be declared as being virtual; and, of course, the library classes would all have virtual destructors.

1.4.1 GUI Libraries

The graphical user interface (GUI) libraries provide individual building blocks and complete subassemblies for the construction of user interfaces. The subassemblies are provided in the form of clusters of related classes. GUI libraries are inherently platform specific. The classes define various types of visual element from which an interface can be constructed. Thus, there might be a set of classes that help handle scrollable views that are too large to be displayed in their entirety. Such a set would include classes that can be instantiated to provide various forms of scroll-bar that control movement, and a “scroller” class that maintains a coordinate frame and interacts with the scroll-bars. A programmer employing such a class library would probably be able to use the scroll-bar class unchanged. The scroller class might be abstract, with some member functions left as pure virtual functions. The programmer creating a new application would then create a specialised subclass of the class scroller, providing definitions for all pure abstract functions as well as adding new functionality and, possibly, replacing some inherited virtual functions.

The GUI libraries are more complex than the component libraries. Part of the complexity results from more extensive use of inheritance. Although the inheritance structures are usually just trees, they may be quite deep. A specialised element, such as a radio button, may be six or more branches away from the root of the tree [Gray94].

Class libraries like these GUI libraries can significantly enhance a programmer’s productivity. The ways in which instances of different classes interact will all have been sorted out, thus saving the application programmer from having to do a lot of design work. Typically, there are default implementations for the vast majority of

the member functions of the various classes. Consequently, instead of writing an entire interface, the application programmer needs to implement those functions that are inherently application-specific along with any others whose default implementations are considered to be unsatisfactory. Furthermore, the use of standard classes enhances consistency among different products because applications built using the same GUI classes will tend to have the same “look and feel”.

1.4.2 Framework Class Libraries

Code reuse can be maximised through the exploitation of framework class libraries. A framework can be viewed as providing a skeleton application that can be extended and specialised through class inheritance. The framework libraries were developed for creating interactive programs. In the run-time environment of an interactive program, there are several objects present that own some part of the program’s data and handle any commands that relate to their data. User actions, such as menu selections, keyed input, or mouse-based interactions, are translated into commands or messages that get sent to particular command-handling objects for processing.

The authors of the framework library started by identifying a set of “command-handler” objects that could be conceived of as being present in any executing program. The various tasks that a program has to perform were then partitioned amongst these objects, and so the forms of specialised kinds of command-handler classes evolved. For example, there was a “document” command-handler. A document “owned” the data structure that was being manipulated by the program’s user, it looked after changes to that structure, and it arranged transfers between disk files and the main memory. An “application” command-handler was responsible for overall organisation of the objects that made up a running program, and it performed much of the work involved in translating low-level operating system events into commands that could be sent to other objects. “View” command handlers provided a x, y -coordinate framework for the display of data, methods for drawing data, and methods for responding to mouse-based data selection and editing actions.

The authors of the frameworks sorted out all the typical patterns of interaction that occur among these principal command-handler objects. These interactions were then encoded in the member functions that were defined for these classes. For example, if an application object found that it was dealing with a “Quit” command, it would remember to warn any document objects that were present. Each document object

could then check to determine whether its data structure had been altered. If the data had been altered, the document would display a standard alert asking the user whether the data should be saved before the program terminated. Once it had either saved or abandoned its data, each document would proceed by disposing of any view objects that it had created to display its data. Here, there is a complex pattern of interactions among instances of the application, document, and view classes. But the normal responses can all be defined and provided as functions such as `Document::Close()`, `Application::Quit()`, etc. Of course, these functions would be specified as being virtual - a programmer should have the opportunity of changing standard patterns of behaviour if really necessary.

In a typical framework, there are hundreds of such patterns of interactions associated with standard user commands and run-time events. Each interaction may involve several objects; the work performed by each object may entail a series of nested function calls; and there may be some cycles in the call patterns. For example, the application object might ask a view to handle a mouse action and, in doing so, the view might ask the application object to perform some function such as adding a "mouse-command object" to a list.

The application, document and view classes in a framework library will be partially implemented abstract classes. While many of the behaviours of these classes, and patterns of interactions among instances of these classes, can be identified, actual details of implementation of some behaviours have to be deferred and left for definition in derived classes. It is not uncommon for these classes to have one hundred or more different member functions [Bisc92]; most of which will have default implementations.

1.4.3 Examples of Framework Class Libraries

MacApp-1.0 [Schm86, RoDo*86] was one of the earliest of these frameworks. It was created to facilitate the implementation of applications for Macintosh computers. It had been noted that, excluding games and other special programs, all the early Macintosh programs could be conceived as involving: 1) an application that handled interaction with users and managed documents, 2) one or more documents that managed the run-time data structures representing the information manipulated by the user, and that organised data transfers to and from disk files, 3) views of the data, and 4) application specific data structures (e.g., paragraphs in a word-processor, records in a database, or graphics elements in a MacDraw like graphics editor). The

first three of these four parts were essentially the same irrespective of the application. Inevitably, very similar code was being re-implemented in each new program.

It was realised that if suitable abstractions could be found, much of the standard code could be provided in a library (with the proposed standard Macintosh look and feel actually implemented correctly for once). For example, every Macintosh application had to include an event loop [WiRo*91] that received mouse-down, keystroke, update, and other events from the underlying operating system and which distributed these to other components in the code for handling. This standard code could be provided as a `Run()` method for an application class. This `Run()` method would sort out the events, passing updates to appropriate view objects, resolving mouse movements into menu requests and drawing actions. The general behaviour of an application object, and its modes of interaction with views and documents could be almost completely characterised with default implementations provided for all required functions.

MacApp started with Object Pascal as its implementation language but switched to C++ later on. It served as a model for many other framework libraries and it is still a fairly useful development aid for the Macintosh platform. Version numbering for MacApp changed after 3.3.3, so that the next major version was called Release 12. The most recent version of MacApp is Release 13, Update 4 [INET11].

ET++ [WeGa*89, WeGa94] is a public domain class library for UNIX/C++ environments. Its initial structure was based on MacApp-2 with C++ substituted for Object Pascal. Inspired in part by MacApp's design, ET++ took advantage of a more powerful platform, typically a Sun Workstation and provided many extensions with a more consistent design and implementation. Unidraw [VILi89] is another example of an application framework on UNIX for structured graphics editors.

Currently, the most active area for framework development is for the Windows™ operating system. The leading product was Borland's Object Windows Library (OWL) which provided most standard framework components. In addition to the main framework classes, the Borland product included an extensive set of collection classes. Microsoft entered this market relatively late but has introduced the Microsoft Foundation Class (MFC) library. In order to compete with the success of MFC, Borland has introduced the Visual Component Library (VCL) with its new product C++Builder [INET12]. Unfortunately, incorporating OWL classes in VCL applications is not straightforward [INET13].

1.5 THE MAIN RESEARCH OBJECTIVES

The author has experienced some of the problems with procedural codes in the structural engineering domain as outlined in Section 1.1. He has seen software systems with excellent technical contents suffering from poor user interface problems.

The author has realised that much of the user-friendliness of applications comes not only from an iconic user interface but also from a uniform user interface across applications. This leads to a significant amount of development redundancy because most of the code required by the user interface has to be reengineered for every new application.

This background motivated the author to undertake this research project to learn more about object-oriented software design and implementation techniques with an application framework. The actual project work has evolved from a slightly different motivation. As a Naval Architect, the author has witnessed the interactions among various organisations in the lifecycle of a ship:

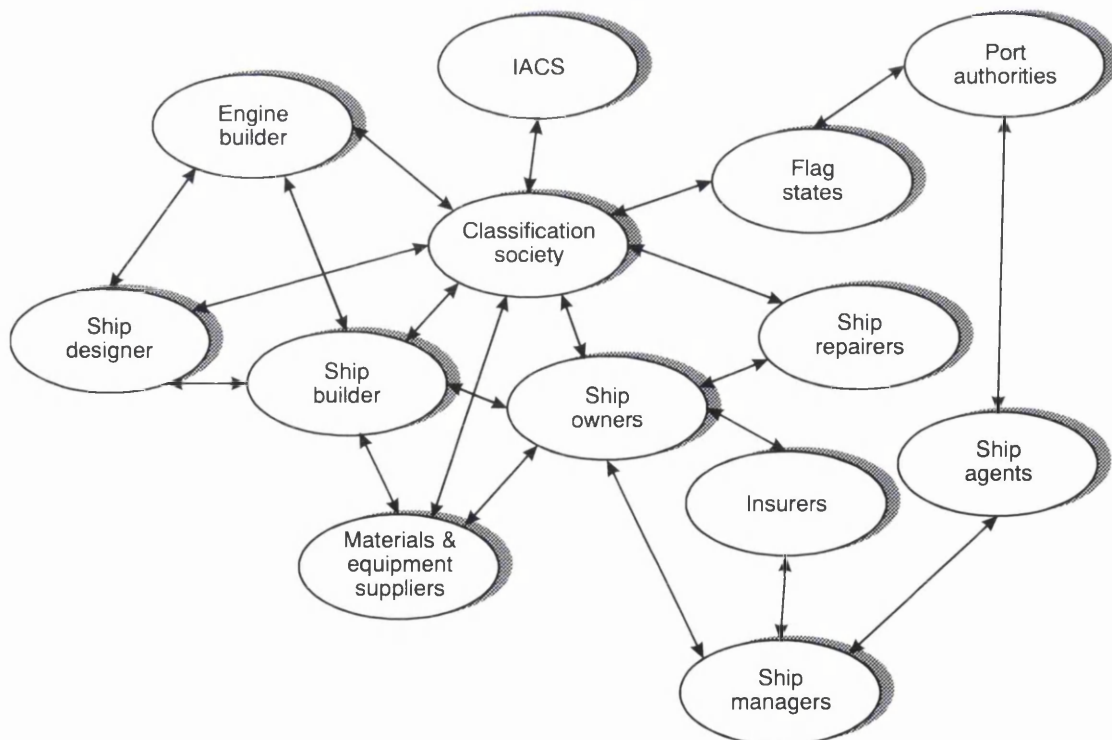


Figure 1.1 Interactions in the Lifecycle of a Ship

This type of interactions generate a huge volume of paper-based communications. The diagram in Figure 1.1 is just an example. In today's business environment, for

instance, when a company undergoes a private or public offering, a large number of documents must be drafted and agreed upon by several parties, including the company executives, the company's law firm, and the underwriting financial institution. During the drafting exercise, the document must be distributed to the different parties many times. Even though more than 90 percent of business documents originate in electronic form, individuals and organisations still spend 300 billion US dollars annually on physical document delivery services [INET14]. Despite the obvious speed and cost advantages of communicating over the Internet, most businesses still rely on traditional delivery methods when sending important information. Customer surveys indicate that the Internet lacks two qualities required for important business communications and transactions: reliability and trusted security. However, more communications and business processes are moving online everyday. The pace at which e-commerce and electronic document exchange applications are improving is whiplash fast.

However, the author is not concerned about the actual document or information exchange process. He is more interested in developing a simple tool which might be useful in resolving the queries or misunderstandings that arise sometimes specially from paper-based communications. For example, ambiguous official memorandums are not uncommon when the medium of communication is English but the participants are non-native English speakers. Annotations are helpful in this context. An annotation to a document is a comment on or question about the document. Traditionally, paper annotations are little notes jotted on the margin of a paper.

However, annotations may be organised in "conversations", where one annotation answers or refers to another. If all the annotations are collected and preserved, the logical structure of the conversation becomes more apparent. This raises the possibility of using sound objects for annotating a document. A short question in the form of natural conversation, recorded and preserved as a sound object can clarify things quickly and effectively rather than formal speech or writing. However, some graphics tools are also necessary to mark or highlight a document when sound objects are used for annotation.

An application, therefore, would be useful if it could deal with externally created vector or bitmap images, graphics primitives and sound objects in any sequence. The application should provide the end-users with facilities to work on external images with free-hand curves and other graphics tools, record their voice, save everything in one disk file and animate them later, if necessary. This type of

multimedia application would be useful not only for annotation with sound objects but also for various illustration purposes. One of the main objectives of this research is to develop a prototype of such an application. This prototype is referred to as Glasgow Graphics and Sound (GGS) in this thesis.

The software architecture of GGS should be flexible and robust. It must allow changes to be made easily and it must be decoupled from the way the objects are stored and retrieved later. One way of achieving this is to make the objects persistent and avoid deciding on a file format for storage altogether. However, it is necessary to understand the techniques or mechanisms available to add persistence to objects and select one of those techniques for GGS.

Now the research objectives can be summarised. The reasons for engaging in this line of research are fourfold:

- to learn object-oriented software design and implementation techniques and to exploit the opportunities provided by an application framework;
- to learn “user-friendly” graphical interface design principles;
- to develop a prototype multimedia application for annotation and illustration;
- to understand object persistence and select one suitable persistence mechanism for the multimedia prototype.

1.6 APPLICATION FRAMEWORK SELECTION

The recent advancement in the PC technology has already made a remarkable impact on the industry. The dramatic enhancement in the PC clock speed and associated technology has blurred the price/performance difference between UNIX workstations and PC's. A number of UNIX based technologies are no longer actively being developed to cope with the recent advancement. The Microsoft Windows™ market is the growing segment of the industry. In the present circumstances, the selection of Windows™ as the target platform for this research project has not been a difficult decision. However, Windows™ comes in various flavours. But only the 32-bit versions – Windows 95/98/NT 4.0 (collectively referred to as *Windows* later on) are targeted here for the development of the multimedia prototype.

To work with an application framework, the choice of C++ as the programming language is almost unavoidable since most of the frameworks are written in C++. Although Java is getting a lot of press, almost every major *Windows* application

shipping today is developed in C++ and most are created using Microsoft's Visual C++ [INET12].

1.6.1 The Essentials of a *Windows* Program

Windows is an event-driven operating system that sits on the computer monitoring the hardware for events. Whenever an event takes place, for example, when the mouse is moved, *Windows* detects it and passes the event information on to the applications it is hosting. The event information is called a message. If the message is important to a specific window on the screen, that window handles it.

The structure in Figure 1.2 is at the heart of all *Windows* programs. There are two essential pieces of a *Windows* program: the function `WinMain()`, which is called by *Windows* at the start of execution of the program, and a window procedure for each window class, often referred to as `WndProc()` or `WindowProc()`, which will be called by the operating system whenever a message is to be passed to the application's window.

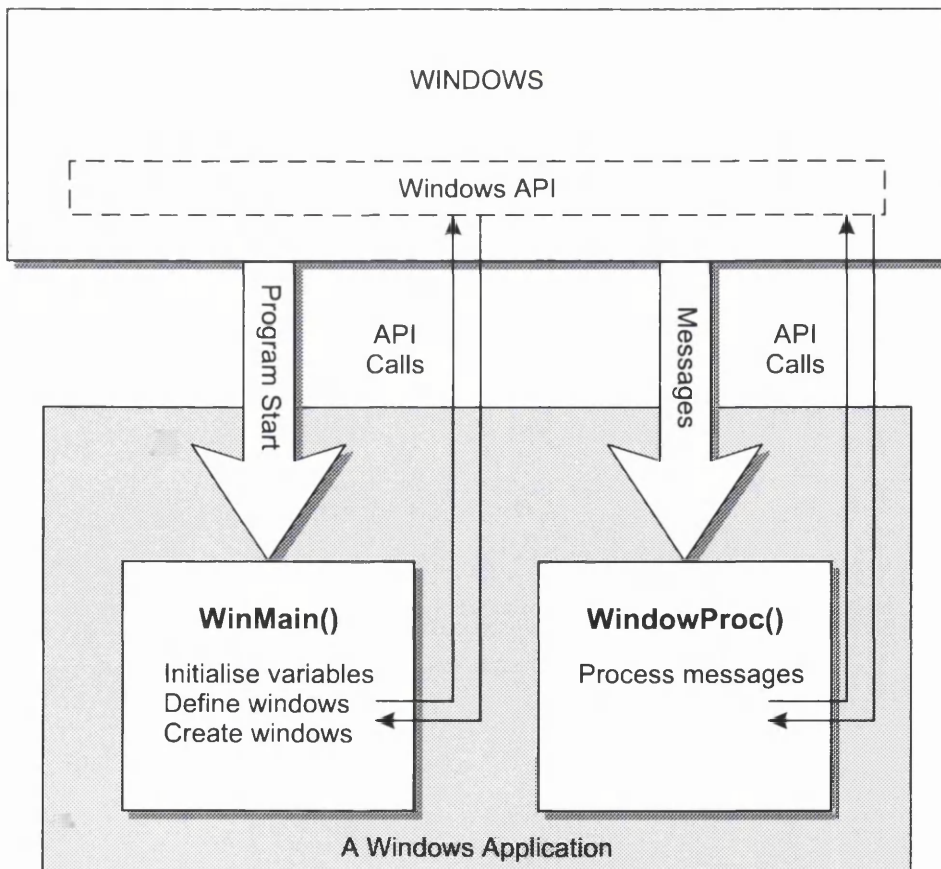


Figure 1.2 Two Essential Components of a *Windows* Program

The function `WinMain()` does any initialisation that is necessary and sets up the window or windows that will be the primary interface to the user. It also contains the message loop for retrieving messages that have been queued for the application. The function `WindowProc()` handles all the messages. `WindowProc()` contains the application-specific response to each *Windows* message. `WindowProc()` should handle all the communications with the user by processing the *Windows* messages generated by user actions, such as moving or clicking the mouse, or entering information at the keyboard. A program written in C with *Windows* API calls, hooks into the *Windows* machinery [ShWi97] by writing large `switch` statements in the `WindowProc()` function to handle the messages. It is certainly a better picture when MFC is used to develop a *Windows* application. To allow an application to receive *Windows* messages, MFC employs a class named `CCmdTarget` and a lookup mechanism called message maps.

1.6.2 The MFC Library

The core of the MFC library is an encapsulation of a large portion of the *Windows* API in C++ form. Library classes represent windows, dialog boxes, device contexts, common GDI objects such as brushes and pens, controls, and other standard *Windows* items. These classes provide a convenient C++ member function interface to the structures in *Windows* that they encapsulate.

But the MFC library also supplies a layer of additional application functionality built on the C++ encapsulation of the *Windows* API. This layer is a working application framework for *Windows* that provides most of the common user interface expected of programs for *Windows*, including toolbars, status bars, printing, print preview, database support and many others.

The MFC library has been chosen as the application framework for developing the multimedia prototype, not just because it provides an efficient C++ interface to *Windows* programming. An MFC program is portable to a wide variety of platforms. Since MFC is built on top of the Win32 API, any platform that supports the Win32 API can be targeted by an MFC application. An important aspect of MFC portability is that Win32-based platforms are not limited to *Windows*. Through the use of the Windows Portability Layer, MFC applications built on Win32 can also be built to run under the Macintosh operating system. MFC applications can target not only Intel processors but also PowerPC, Alpha AXP and MIPS processors since *Windows* NT runs on them. In addition, UNIX and VMS porting kits are also available for the developers to port their MFC applications to these platforms as well [MSDN11].

1.7 REFERENCES

- [Bisc92] Bischofberger, W.R.: "Sniff - A Pragmatic Approach to a C++ Programming Environment", USENIX Association C++ Technical Conference, El Cerrito, CA, pp. 67-81, 1992.
- [Gray94] Gray, N.A.B.: "Programming with Class: A Practical Introduction to Object-Oriented Programming with C++", John Wiley and Sons, 1994.
- [INET11] Apple Computer, Inc.: "Mac OS 8 and 9 Developer Documentation: MacApp", <http://developer.apple.com/techpubs/macos8/DevTools/MacApp/macapp.html>.
- [INET12] Heller, M.: "Keep Building in C++", <http://www.winmag.com/library/1998/0501/rev0082.htm>.
- [INET13] Reisdorph, K.: "Using OWL classes in C++Builder", <http://www.zdjournal.com/cpb/9708/cpb9781.htm>.
- [INET14] Tumbleweed Communications: "Tumbleweed IME Architecture White Paper", http://www.tumbleweed.com/solutions/ime_overview_archwhitepaper.htm.
- [MSDN11] Microsoft Developer Network Library: "From One Code Base to Many Platforms Using Visual C++", April 1999.
- [RoDo*86] Rosenstein, L.S., Doyle, K. and Wallace, S.: "Object-Oriented Programming for Macintosh Applications", ACM Fall Joint Computer Science Conference, Dallas, Texas, pp. 31-35, 1986.
- [Schm86] Schmucker, K.J.: "Object Oriented Programming for the Macintosh", Hayden, Hasbrouck Heights, New Jersey, 1986.
- [ShWi97] Shepherd, G. and Wingo, S.: "MFC and Windows Messages", Visual C++ Developers Journal, April 1997.
- [VILi89] Vlissides, J.M. and Linton, M.A.: "Unidraw: A Framework for Building Domain-Specific Graphical Editors", Technical Report: CSL-TR-89-380, Stanford University, Computer Systems Laboratory, July 1989.
- [WeGa*89] Weinand, A., Gamma, E. and Marty, R.: "Design and Implementation of ET++, A Seamless Object-Oriented Application Framework", Structured Programming, Vol. 10, No. 2, pp. 63-87, 1989.
- [WeGa94] Weinand, A. and Gamma, E.: "ET++ - A Portable, Homogenous Class Library and Application Framework", Proceedings of the UBILAB'94

Conference, Zurich, pp. 66-92, September 1994.

- [WiRo*91] Wilson, D.A., Rosenstein, L.S. and Shafer, D.: "C++ Programming with MacApp", Addison Wesley, 1991.

CHAPTER 2

OBJECT PERSISTENCE AND C++

2.1 INTRODUCTION

Persistence may well be the most important capability of a computing environment. Without it, every calculation would have to start anew. Databases and file systems could not exist. We would have a memory-less computing landscape.

Unfortunately, most of today's popular operating systems still adhere to the model of flat files developed in the early days of computing and leave it up to the application to provide any required structure. We can save data but the application has to contain logic for writing its data to a flat file in a way that will permit it to reconstruct the data structure later.

C++ has emerged as the *de facto* standard language for software development. It blends the C language with support for object oriented programming and its newly enhanced template features [ANSI97] bring another new programming methodology, generic programming. This triple heritage makes the language very powerful but unfortunately, C++ does not have any built-in support for object persistence. However, there are different techniques for adding persistence to C++ objects and some of these techniques and strategies are discussed in this chapter.

2.2 STREAM FACILITIES IN C++

C++ streams are close to offering some form of language-supported persistence. "The stream I/O family in C++ is exclusively concerned with the process of converting typed objects into sequences of discrete characters and vice versa" [Stro93]. All data is converted to a common ASCII text format, allowing data to be easily transferred back and forth from secondary storage. Stream-based persistence stores objects in a text-based data store. The C++ stream library converts and transfers the built-in types. Structures and classes are the programmer's responsibility to convert and transfer to a hard disk. They are not as difficult to stream as it may first appear, since they can be recursively broken down into a set of built-in types.

The most basic level of persistence is provided by overloading the shift operators to allow an object's data to be streamed. For example, a class designed to represent a line in a two-dimensional plane can use the start and end coordinates as the basic data necessary to describe the line object. Hence, a line object can have four built-in integer data members representing the coordinates. When saved, each integer is usually separated by a space to ensure correct interpretation when the data is

restored. The restoration of the line object has to be performed in the same sequence as when it was stored.

In addition to the correct order of instance data, the order in which the objects themselves are saved and restored must be maintained. An application's persistent store would normally contain more than one single data type. For example, a line class and a rectangle class are created, both using a pair of coordinates. The line coordinates describe the end points and the rectangle coordinates represent the locations of the opposite corners. For the items to be restored correctly, the application needs to be aware of the order in which each object was saved. Without the explicit knowledge of object restoration order, the application needs to determine the owner of each data item.

The stream facility is well suited for small amount of simple data types. The technique works only if the overloaded operators (i.e., << and >>) write and read data members of types for which the insertion and extraction operators have already been defined and known at compile time. The limits of this technique are revealed if the objects contain polymorphic data structures. The C++ polymorphism allows a base class pointer to refer to a base class object and any object derived from it. To restore a polymorphic pointer correctly, the object's exact type is important. A base class pointer must be restored with a pointer to an instance of the original polymorphic type. The stream facility in C++ cannot read back objects without knowing their exact types [LaSi93].

2.3 HIDDEN POINTERS

Each object of a class that has virtual functions, contains a hidden pointer that points to a virtual function table, called the `vtbl`. The `vtbl` contains the addresses of the virtual functions. It also contains the offsets or 'deltas' that are used to find the address of a derived class object given the address of a base class sub-object. Virtual function invocations involve an indirection that uses the `vtbl` pointer to access the entries in the virtual function table. Similarly, a virtual base class requires an indirection through a pointer, called the `vbase` pointer which is required to implement the sharing of the virtual base class in objects of types specified using multiple inheritance.

Virtual functions and virtual base classes have an impact on persistence because of the 'hidden' `vtbl` and `vbase` pointers generated by C++ compilers to implement these facilities. The `vtbl` and `vbase` pointers are called 'hidden' pointers because

they represent implementation related information and are invisible to the user. Many C++ programmers are not even aware of their existence [BiDa*93].

Unfortunately, the hidden pointers are volatile pointers, i.e. they are not valid beyond the lifetime of the program that created them. Saving objects containing hidden pointers on disk and then reading them back in another program means that the hidden pointer values in the objects read from disk are invalid. This can also be true for the same program if it gets relinked for any reason, changing the address assigned by the linker to the virtual function tables [Dixi98]. The same observation holds for the values of data members that are volatile pointers. The problems with volatile data members are discussed later in detail.

2.3.1 Retrieval Using A Special Constructor

There are couple of solutions to fix the hidden pointer problem. One of them is based on the fact that each class constructor, as translated by the C++ compilers, contains code to properly initialise the hidden pointers. This code is executed prior to the execution of the constructor body, written by the user. The basic scheme is as follows:

- An object is read from a hard disk. As a result, the page on which the object resides is brought from the disk to the main memory but the object contains bad hidden pointers.
- A special constructor is applied to the object without changing any data member.

There are two obstacles to implementing the above scheme. Firstly, C++ does not allow a constructor to be invoked in conjunction with an existing object. Secondly, a constructor defined by the user cannot be used to correctly initialise the hidden pointers in an object read from the disk because the constructor may modify the values of the data members of the object and even update other objects as well. It is essential to invoke a constructor that will not modify any data item. Hence, it should have a null body.

The first problem can be solved by defining an overloaded version of the global operator `new` function. When an object of class `CMyClass` is created by calling `new CMyClass(...)`, C++ does two things:

- The function operator `new` is called to allocate memory for the storage of the object.

- An appropriate constructor (as determined from the arguments to the constructor supplied with the invocation of `new`) is called to initialise the hidden pointers and other components of the object.

In the context of fixing hidden pointers, the storage allocation is completely unnecessary. We simply want the operator `new` to execute the constructor. The following overloaded version of `new` is an example:

Listing 2.1 One Example of Overloaded Operator New

```
void* operator new(size_t, void* p)
{
    return p ;
}
```

C++ requires the first parameter of an overloaded definition of the operator `new` to be of type `size_t` and that `new` returns a value of type `void*`. If the memory address of an object's location is passed to this overloaded function in Listing 2.1, it simply returns the input address as its result.

Now if we have the previous image of the object from a binary file, we could just do the following:

```
char* buffer = new char[sizeof(CMyClass)] ;
binary_read(buffer, sizeof(CMyClass)) ;
CMyClass* pMyClass = new(buffer) CMyClass ;
```

If the constructor has a null body, the data members will remain unchanged but the 'hidden' code in the constructor added by the compiler will reconstruct the `vtbl` and `vbase` pointers, if present. However, please note that `binary_read` is an illustration only; an appropriate I/O routine should be used to bring the object from the disk to the main memory.

2.3.2 Problem with the Special Constructor

Unless otherwise specified, a constructor for a class will invoke the argumentless constructor for each of its base class sub-objects and for every data member that is a class object. All constant and reference members are also initialised by the constructor. Hence, a special constructor that fixes the hidden pointers but does not alter the data members must invoke similar special constructors for each base class sub-object and for every data member that is a class object. This is possible to

implement by generating a default constructor with a null body for every class. But the classes must have other constructors for regular use. The constructor that takes no argument has to be reserved for fixing hidden pointers only. But this solution fails when a class has an array of class objects as a data member. In such a case, constructors with arguments cannot be used.

The database programming language O++ [AgGe89] is based on C++. Among other things, O++ provides facilities for making C++ objects persistent. O++ is an upward compatible extension of C++. O++ programs are translated into C++, compiled and linked together with the Ode Object Manager [DaAg*93]. The O++ implementation handles the hidden pointer problem by modifying each user-specified constructor so that it would do nothing (i.e. execute no statements) when it is called to initialise the hidden pointers. The value of an integer global variable `_fix_hidden` [BiDa*93] is used to determine whether or not the constructor has been invoked to fix the hidden pointers:

```

CMyClass::CMyClass (parameter declarations) initialiser-list
{
    if (!_fix_hidden)
    {
        ... ;
    }
}

```

Other initialisers that specify initial values for data members, are modified to change the values of the data members only if the constructor is called to initialise a newly created object. They have no effect if the constructor is invoked to fix the hidden pointers for an object that has been read from the disk. For example, an initialiser of the form:

```
m (initial_value)
```

where `m` is a data member, is transformed to the following:

```
m (_fix_hidden ? m : initial_value)
```

When `_fix_hidden` is nonzero (i.e., true), the initialiser effectively assigns the member to itself and consequently does not change the value of the data member.

2.3.3 Retrieval Using the Assignment Operator

Another solution to fix the hidden pointer problem is to use the assignment operator.

The basic scheme is as follows:

- Space is allocated for an object read from the disk. The object contains bad hidden pointers.
- A new object is created with correct hidden pointers.
- The object read from the disk is assigned to the new object.

The default assignment performs member-wise assignment of the components of the source object to the destination object. In particular, the hidden pointers are not copied from the source object to the destination object.

The disadvantage is that the storage has to be allocated twice for every object. In addition, this solution assumes that the assignment operator performs member-wise assignment. These are the semantics of the default assignment operator generated by the C++ compilers. However, users are allowed to define their own version of the assignment operator. This may invalidate this solution if the explicitly defined assignment operator does not perform member-wise assignment or has side-effects.

2.4 MEMORY MANAGEMENT AND SMART POINTERS

So far we have seen that the objective is to store C++ objects as simple sequences of bytes in disk files. Then the saving/loading of objects becomes the saving/loading of a portion of the main memory image. Unfortunately, saving objects on disk containing volatile pointers such as hidden pointers and embedded pointer data members and then reading these objects back from the disk does not solve the persistence issue because the volatile pointer values in the objects will be invalid. We have discussed about hidden pointers in Section 2.3 and realised that it is necessary to bypass the usual C++ object allocation mechanism and to get control on object allocation.

Memory management mechanisms for persistence are not new. In Smalltalk, the program's entire memory image is dumped on disk and restored when running the program later. This is one way of saving and loading the memory without worrying about its content, i.e., without distinguishing the data from the executable.

In a simple memory management approach, all kinds of objects that must be persistent can be stored in an array and the entire array can be saved on disk or loaded in memory instead of one object at a time. This is possible by overloading the operator `new` so that the objects can be allocated in a predetermined array instead

of scattering them in the heap. The concept of an array can be easily extended to coarse-grain objects, memory pages or files.

One possible version of the overloaded `new` is as follows:

```
void* operator new (size_t size, char* arrayName)
{
    void* ptr ;
    ptr = allocate (size, arrayName) ;
    return ptr ;
}
```

An appropriate `allocate()` function stores the new object in the array, `arrayName`. Here is an example of an allocation:

```
char array1[100] ;
CMyClass* pMyClass = new(array1) CMyClass(...) ;
```

Similar functions like `allocate()` are necessary for freeing objects from the array and the full exploitation of the memory space. The main advantage of this scheme is that the storing and loading operations are very simple because they just read and write to an array. For these operations, the array is like a piece of raw memory.

To deal with pointers in user-defined objects, they must be represented in an address-independent way so that they remain valid every time the array is reloaded. This can be solved through a logical pointer mechanism that allows the identification of objects independent from their actual physical address. One good option is to use an object's offset in the array because it does not change if the array is reloaded and can be managed easily. But built-in pointers have a behaviour that cannot be modified. Thus, something similar and more powerful, e.g., a special class can be used that behaves like a pointer but refers to the actual object's address by using its offset.

2.4.1 Smart Pointers

The "Smart Pointer" technique allows the implementation of a special class such as just described. Smart pointers are template classes that behave like pointers to other classes. One simple example [DaDa95], [Hors93] of their implementation is as follows:

Listing 2.2 One Example of a Smart Pointer Class

```

// For simplicity, let us consider array1 as a global variable
template <class X>
class Pointer
{
    Pointer() : nStore(0), bStoreAssigned(false) {}
    // p should point to an object inside array1.
    // That must be checked before calculating nStore
    Pointer(X* p)
        {nStore = (char*) p - array1; bStoreAssigned = true;}
    X* operator->()
        { assert (bStoreAssigned); return (X*)&array1[nStore]; }
    X& operator*()
        { assert (bStoreAssigned); return *((X*)&array1[nStore]);}
private:
    X*    nStore ;
    bool  bStoreAssigned ;
}

```

Except for the type declaration, instances of this `Pointer` class can be used in the same way as regular pointers:

```

Pointer<Employee> pe = new(array1) Employee("James Bond") ;
strcpy( s, pe->name() ) ;
Employee Agent_007 = *pe ;

```

The pointers are smarter for two reasons. Firstly, they are always initialised to 0, not to some random value. Secondly, any attempt to indirect through a smart pointer when it is not properly assigned, will lead to an assertion failure. Storing/loading operations on the array (i.e., the object container) do not change the offsets of the objects. Hence, pointer data members can be stored without problem using the smart pointer mechanism as illustrated in Listing 2.2. The mechanism solves the problems of recognising pointers and references among objects inside the array. However, it does not consider any attempt to access the objects from outside the array.

The mechanism in Listing 2.2 can be extended with an array of smart pointers stored and loaded together with the arrays containing objects. A solution in [DaDa95] demonstrates a variation of reference counting technique through the handle class idiom [Booc91].

2.4.2 Evaluation of Smart Pointers

The persistence mechanism using smart pointers offers some advantages. It could be associated with a large number of applications without modifying their design. If the user needs an object to be persistent, s/he must declare a smart pointer to it. The user can then execute all the operations usually done with regular pointers. The object pointed to by the smart pointer can reside in an array and the user can store and load this object without worrying about its content. The concept of an array holding objects can be extended to specialised object containers [FuDa93].

A drawback is that the entire array and therefore all the objects must reside in the main memory. When more than one array is involved, the level of indirection increases which has its usual penalties.

Unfortunately, smart pointers do not respect a public inheritance hierarchy. If two classes, `Apple` and `Orange` are derived from `Fruit`, `Pointer<Fruit>` cannot be used like built-in base class pointers in place of `Pointer<Apple>` or `Pointer<Orange>`. As far as compilers are concerned they are three separate classes and they have no relationship to one another. Fortunately, there is a way to get around this limitation by using member function templates [Mey96] so that compilers can generate implicit type conversion functions to help with the inheritance relationships.

2.5 RELATIONAL DATABASES

Let us now move on to databases before describing other techniques for persistence. Most real-world commercial applications store their data to a relational database and with good reason. Most business systems are not created in a vacuum. Typically, they must interoperate with legacy systems and they must access legacy data. In other words, in many circumstances there is no choice other than a particular relational database as the persistence mechanism.

Relational database technology is mature. It has been tested, its performance is well understood and there are good tools to support the development and maintenance of large relational database applications.

2.5.1 Mapping Objects to Tables

If a relational database management system (RDBMS) is chosen as the persistence mechanism, then an object's state will be mapped to a set of rows in one or more

tables in the RDB. Usually the decisions taken in this context affect the system's performance. Mapping C++ objects to database tables involves the translation of an object's primitive data types to database data types and the representation of its hierarchies and associations.

An application programmer has to program in two different languages with distinct syntax, semantics and type systems, namely, the application programming language (e.g., C++) and the data manipulation language of the RDBMS (i.e., SQL). The logic of the application is implemented using the programming language while SQL is used to create and manipulate the data in the database.

When any data are retrieved from a relational database, they have to be translated from their database representation to the in-memory programming language specific representation. Similarly, any data updates have to be explicitly communicated to the database using another SQL statement, causing the data to be translated from the in-memory representation back to the database representation. All this communication back and forth between the database and the application leads to some unnecessary processing of data which is commonly known as the impedance mismatch problem [SrCh97].

2.5.2 Binary Large Objects (BLOB)

Mapping an object to database columns can dramatically slow the performance of the system. An alternative method is to store the entire object as a binary stream of bytes, known as BLOB. The key values are stored as attributes in columns, but the bulk of the data is not differentiated. However, the database cannot manipulate or sort on that data but by using the keys, it can retrieve the BLOB which can then be reconstituted in the application.

The decision on which parts of an object to put in the BLOB is critical. This decision changes as the physical design of the database is modified during the development and maintenance cycles. Indexes can only be put on columns, not on a part of a BLOB because the BLOB is just binary bits to the database. This forces the developer to decide once and for all time which fields to key on. The decision to add new keys requires a change to the definition of the table as well as a redefinition of the BLOB which can be an expensive maintenance decision. This can make the design more viscous because it is harder to change as requirements shift. For this reason, many developers do not use a BLOB. They just make columns for their data and accept the trade-off in performance.

Classically, when a database-centric system is built first, the developer spends a good deal of time tweaking the database to make it run faster. The database definitions, the SQL embedded in the application code, the database utilities are all suspects when it comes to finding performance bottlenecks [Libe97].

2.6 OBJECT ORIENTED DATABASES

While many systems routinely manage the conversion between objects and relational databases, there is something unnatural about this relationship. The issues can be encapsulated, but it is not possible to escape from the fact that relational databases think in terms of relations, not objects.

An obvious alternative is to build a system on top of a true object-oriented database (OODB). Just as fixed length rows in tables are the natural unit of storage in a RDB, objects are the natural unit of storage in an OODB. An OODB is a natural match when using objects in programs. From the application programmer's viewpoint, there is no need to transform his/her objects. The database management system takes care of any transformations required to transmit the objects from the program to the OODB and vice versa.

There are several advantages of using an OODB when working with a language like C++:

- There is no need to do any "data modelling" and type conversion as the unit of storage is an object.
- Translation of objects into rows and rows into objects is not necessary, as there is when using a RDB. Hence, for certain types of applications, the runtime performance of an OODB can be superior to a RDB.
- Usually OODBs use implicit persistence. The user of an OODB does not have to worry about loading and saving his objects and whether they are in the main memory or in the OODB. This is all done transparently by the OODB.

These advantages are especially important in applications where the data consists of a small number of objects that have complex inter-relationships and not many queries are necessary. RDBs tend to perform better when a large number of objects are present with simpler inter-relationships but with many queries [Libe97].

2.6.1 Querying An OODB

OODBs for C++ typically use native C++ as the language to define and manipulate objects. Definitions are essentially just the header files normally produced as part of

developing the application. Data manipulation is accomplished through the normal C++ syntax of accessing data members of objects and pointers to objects.

There are two groups on the subject of query languages. One group tries to embed SQL in C++, so that the same types of queries are available as in traditional RDBs. The other group tries to stay with C++. For example, one approach is not to query a whole database, but to query a container [SrCh97]. A query on a container is a binary expression which returns the subset of valid elements in the container that are true for the expression.

As the OODB market matures, clients expect that these products will support the same set of multi-user features that RDBs support. This includes transactional control, journalling, security, protection, etc. OODBs are beginning to penetrate the transaction processing market (e.g., banks and insurance companies), and this will require them to support thousands of simultaneous users and to store terabytes of data.

2.7 POINTER SWIZZLING AT PAGE-FAULT TIME

A number of object database management systems (ODBMS) interact with the virtual memory mechanism of underlying operating systems and provide implicit persistence to objects. The particular technique used to achieve this object persistence is often referred to as pointer swizzling. To understand this technique, it is important to look at the demand-driven interaction between a modern operating system (OS) and a persistent storage medium (e.g., a hard disk).

Modern operating systems have developed sophisticated techniques to run huge programs simultaneously by keeping only a small portion of them in the main memory at any given time. When the data or program accessed by an instruction is not currently in the main memory, the OS page faults which raises a hardware interrupt that allows the OS to call a special interrupt service routine [Vada95a] that brings the required page from the disk after a number of security checks and then resumes control from where the page-fault occurred. The program runs as if it had the entire data and code in the main memory all along and the programmer is completely unaware of the fact that his/her program is straddling the two memories and thinks that the entire memory is available to use. Thus, the hardware interrupt processing is done in a way that is transparent to the programmer. The virtual memory management part of the OS decides (a) the pages to be evicted to bring new page requests, (b) the pages not to be evicted because they are currently being used,

(c) the size of the buffer, etc. The programmer sees only a single level of storage: the main memory of size equal to the address space even though the real main memory or his/her share of it is much less than the address space.

2.7.1 Two Level Storage Approach

Almost all DBMSs maintain a buffer or pool of objects. They bring in an object that is dereferenced if it is not in the main memory and decide when to flush some objects and when to lock, when to bring in new objects, etc. This is quite similar to what an OS does.

However, there is one main difference. The programmer can see the difference between dereferencing a persistent pointer and a regular pointer. In case of a persistent pointer, the programmer has to explicitly call a function that does the following:

- If the corresponding object is in the main memory, the function returns the object's address.
- If the object is not in the main memory, the function retrieves it from disk and puts it in memory and then returns the object's new address.

However, in case of a regular pointer, the programmer simply dereferences it. If the corresponding location is not in the main memory, it is the OS that brings the page containing the object transparently.

In this two level storage approach, whenever a programmer needs to dereference a persistent pointer, s/he has to check if the corresponding object is resident in the main memory. However, dereferencing a transient pointer is fully supported by the underlying OS and the hardware. The programmer is completely unaware of any disk interaction as a result of such a dereferencing.

The most important disadvantage of this two level storage approach is the time taken for residency check. If main-memory objects are accessed frequently, then each such access becomes a few orders of magnitude more expensive because of this residency check. A natural question in this context is, why not treat the persistent objects as if they are main-memory objects and when an object is not in the main memory, raise an interrupt. Then the DBMS would process this interrupt just like an OS processes an interrupt raised at page-fault time. Thus, the DBMS would bring the disk-resident object into a main memory location, give its main-memory address to the dereferencing program and resume computation as usual. This way, the application

programmer does not see any difference between persistent and main memory storage. The memory mapped architectures achieve essentially this uniform storage view. The exact way in which each implementation achieves this single level storage may differ from one another but they all use an essential technique called pointer swizzling.

2.7.2 Difficulties in Using the OS Support for Persistence

An important consequence of relying on the OS support for dereferencing persistent pointers is that the management of object buffers may also have to be done by the OS. If the OS finds that there is no space available in the virtual memory allocated for the process to bring in a new object, the OS decides which pages owned by the process to evict and make room for bringing in the new object. Such a buffer management by the OS may not exactly be in tune with the requirements of the DBMS.

The question of using the OS support for persistence has been discussed quite extensively in the literature and some of the main objections to such a dependence are the following [Ston84], [Trai82]:

- **Lack of control on committing to disk:** Databases must be able to commit the changes they made to the objects when the programmer wants to commit. The operating systems, on the other hand, commit their files when the virtual memory manager finds the need to do so, or when an explicit 'close file' command is issued. The grain of control provided by OS's is thus inadequate for databases.
- **Limitations on the size:** In the virtual memory architecture, the size of the address space, and hence the size of a database if it is managed by an OS for persistence, will be limited to the 32-bit address space in most hardware. This may not be adequate for a number of applications.
- **Look-ahead algorithms:** For reasons of efficiency, one would like to be able to load database objects ahead of the time they are needed. If we rely completely on the OS for buffer management, loading ahead may not be possible with the DBMS.
- **Taking advantages of the usage patterns:** It is possible that the usage of database objects follows certain usage patterns and for reasons of efficiency, the DBMS may want to exploit this pattern to decide which objects should be flushed out from the buffer pool. Once again, relinquishing the control to the OS may not be the best idea.

Traditionally, for the above reasons, DBMS's have maintained their own buffers rather than depending on OS's. However, recently a number of ODBMS developers have started taking advantage of the 'hooks' provided by modern OS's to provide an efficient single level storage interface and also to circumvent the problems mentioned above.

2.7.3 The Essentials of Pointer Swizzling

Let us assume that all objects in an application are initially disk-resident. Then, the inter-relationship between any two objects is through some disk address. In other words, one object points to another object using a disk address. When disk-resident objects are brought into the main memory, the inter-relationship between any two objects should be maintained in the main memory as well. If objects on disk are viewed as a network or a graph, then the same topological relationship should be maintained across the disk-to-main-memory transfer. This is achieved by replacing all disk addresses by appropriate virtual memory addresses in the objects brought into the main memory. This process of transforming disk addresses into corresponding virtual memory addresses is called pointer swizzling.

When an object is brought into the main memory from the disk, it may have member variables that are references to other objects. These data members are swizzled by examining their disk addresses and retrieving the referred-to objects. These objects in turn may need to swizzle their member objects in memory. Bringing a disk-resident object may cascade through a number of swizzling operations and can, therefore, be a very expensive process.

One answer to this problem is to defer swizzling the member objects until they are needed. This lazy evaluation of member references can dramatically improve performance at run time. The principle is to maintain the following invariant [Vada95b]:

- Every persistent pointer field of a persistent object that is currently in the main memory must point to either
 - a valid virtual memory address where the valid persistent object currently resides or
 - an unallocated virtual memory address that is access protected and will be occupied by the valid persistent object when the pointer is dereferenced.

Thus, whenever a pointer field is dereferenced in the main memory, either the program reaches a valid object or it reaches an access protection violation and hence

an interrupt is raised and the corresponding object is brought into the main memory. This process of replacing the pointers at the time of page-fault is called pointer swizzling at page-fault time and was developed independently by ObjectStore developers [LaOr*91] and Wilson [Wils91].

When a pointer field is dereferenced a second time, the referred-to object is already in memory and, therefore, the system will not page-fault and the program runs as if the referred-to object has always been in the main memory. Thus, further dereferencing of persistent objects takes place at the same speed as the usual main memory access, eliminating further residency checks. This is the main advantage of the pointer swizzling at page-fault time technique.

2.7.4 Wilson's Approach

Motivated by the requirements to support huge address spaces (i.e., bigger than what the virtual address space can support) at a uniform level, Wilson developed [Wils91] an interesting way to map a 64-bit address space into a 32-bit virtual memory address space in a uniform way. Essentially, Wilson uses 64-bit pointers for referring to the disk-resident objects and the usual 32 bits for referring to the virtual memory pointers. His scheme transforms the pointers from one format to the other at the time of page-fault. Wilson used 64 bits as an example size but his methods work with other sizes as well.

2.7.5 ObjectStore

Object Design International independently implemented an idea that is essentially very similar to Wilson's pointer swizzling at page-fault time and built a commercial ODBMS by the late 1990's. This system is called ObjectStore [LaOr*91]. It differs from Wilson's approach in the size. The addresses of the disk-resident objects and the memory resident objects are of the same size. Whereas Wilson's approach uses 64-bit addresses for disk-resident pointers, ObjectStore uses 32 bits to refer to both main memory and disk-resident pointers. Also different are their motivations: Wilson's approach is motivated by the need to provide huge persistent stores and ObjectStore is motivated by the need to eliminate residency checks as far as possible.

Within a short time after the first few approaches were produced, a number of variations on ObjectStore and Wilson's method were developed. QuickStore of White and DeWitt [WhDe94] and Texas [SiKa*92] are some examples of these.

2.8 ORTHOGONAL PERSISTENCE

So far we have discussed different extensions to C++ for persistence and some features of database management systems that support C++. However, there are persistent programming languages such as PS-Algol [AtBa*83] and Napier88 [SjWe*97] that recognise certain principles for persistent data:

- **Orthogonal persistence:** The principle of orthogonality states that all data objects, whatever their type, have equal rights to persistence. This is in line with the principle of data type completeness that all data objects should be allowed the full range of persistence.
- **Persistence independence:** The principle of persistence independence states that all codes should have the same form irrespective of the longevity of the data on which they act.
- **Transitive persistence:** There should be a straightforward and consistent mechanism for determining the longevity of data objects. The mechanism, in most cases, is the persistence by reachability from other persistent or root objects.

These three principles [AtMo95] have been deployed recently in developing Persistent Java (PJava) which is a persistent programming environment for the Java programming language. PJava attempts to support a wide range of applications. A major goal of PJava is to support arbitrary Java code to participate unchanged in PJava applications [Jord96].

Unfortunately any further discussion on these language-based approaches is beyond the scope of this thesis.

2.9 CLOSURE

In this chapter, we have reviewed a number of techniques and strategies for adding persistence to C++ objects. The techniques have their advantages and limitations. For example, OODB's are quite powerful but there are situations where they are inappropriate. In particular, OODB's may cause unacceptable overheads for certain types of applications. The basic problem is that small applications have to pay for features they do not need. For example, an application that does not require concurrency control or support for architectural heterogeneity still has to incur additional time, space and development overheads [Cohe96].

The multimedia prototype, GGS as introduced in Chapter 1 is not a data-access

application. Data-access applications typically update their data on a “per-transaction” basis. They update the records affected by the transaction rather than reading and writing a whole data file at once. An important objective is to develop GGS as a standalone, lightweight and “e-mailable” application. Adding a database support to GGS for persistence is clearly not the best option.

The MFC serialization is chosen for providing object persistence in GGS. Serialization is an easy-to-use and easy-to-implement data storage method that eliminates worries about file formats. It creates transportable stream of bytes that can be stored in a file, sent around the world as an e-mail attachment or transported to a remote client or server for later reconstitution. Serialization is discussed in great detail in the next chapter.

2.10 REFERENCES

- [AgGe89] Agrawal, R. and Gehani, N.H.: “Ode (Object Database and Environment): the Language and the Data Model”, Proceedings of the International Conference on Management of Data, Portland, Oregon, 1989, pp. 36-45.
- [ANSI97] ANSI / ISO C++ Committee: “International Standard for Information Systems – Programming Language C++”, ISO / IEC IS 14882, November 1997.
- [AtBa*83] Atkinson, M.P., Bailey, P.J., et al.: “An Approach to Persistent Programming”, The Computer Journal, Vol. 26(4), 1983.
- [AtMo95] Atkinson, M.P. and Morrison, R.: “Orthogonally Persistent Object Systems”, VLDB Journal, Vol. 4(3), 1995.
- [BiDa*93] Biliris, A., Dar, S. and Gehani, N.H.: “Making C++ Objects Persistent: the Hidden Pointers”, Software Practice and Experience, Vol. 23(12), pp. 1285-1303, December 1993.
- [Booc91] Booch, G.: “Object Oriented Design with Applications”, Benjamin/Cummings, Redwood City, CA, 1991.
- [Cohe96] Cohen, S.: “Lightweight Persistence in C++”, C++ Report, May 1996.
- [Chan96] Channon, D.: “Persistence for C++”, Dr. Dobb’s Journal, October 1996.
- [DaAg*93] Dar, S., Agrawal, R. and Gehani, N.H.: “The O++ Database Programming Language: Implementation and Experience”, IEEE Conference on Data Engineering, Vienna, Austria, 1993.

- [DaDa95] Dabbene, D. and Damiani, S.: "Adding Persistence to Objects Using Smart Pointers", *Journal of Object Oriented Programming*, June 1995.
- [Dixi98] Dixit, S.: "Memory Blasting: Persistent Heaps and Smart Pointers", *C++ Report*, January 1998.
- [FuDa93] Fu, M.M. and Dasgupta, P.: "A Concurrent Programming Environment for Memory Mapped Persistent Object Systems", *Proceedings of 17th Annual International Computer Software and Applications Conference*, pp. 291-297, 1993.
- [Hors93] Horstman, C.S.: "Memory Management and Smart Pointers", *C++ Report*, March-April 1993.
- [Jord96] Jordan, M.J.: "Early Experiences with Persistent Java", *Proceedings of the First International Workshop on Persistence and Java*, Drymen, Scotland, September 1996.
- [LaOr*91] Lamb, C., Orenstein, J. and Weinreb, D.: "The ObjectStore Database System", *Communications of the ACM*, 34(10), October 1991.
- [LaSi93] Laurent, P. and Silverio, N.: "Persistence in C++", *Journal of Object Oriented Programming*, October 1993.
- [Libe97] Liberty, J.: "Beginning Object-Oriented Analysis and Design with C++", Wrox Press, 1997.
- [Lipp96] Lippman, S.B.: "Inside the C++ Object Model", Addison-Wesley, 1996.
- [Meye96] Meyers, S.: "Smart Pointers, Part 3", *C++ Report*, September 1996.
- [SiKa*92] Singhal, V., Kakkad, S. and Wilson, P.: "Texas: An Efficient Portable Persistent Store", *Proceedings of the 5th International Workshop on Persistent Object Systems*, San Minato, Italy, September 1992.
- [SjWe*97] Sjøberg, D.I.K., Welland, R., et al.: "The Persistent Workshop - A Programming Environment for Napier88", *Nordic Journal of Computing*, Vol. 4, pp. 123-149, 1997.
- [SrCh97] Srinivasan, V. and Chang, D.T.: "Object Persistence in Object-Oriented Applications", *IBM Systems Journal*, Vol. 36(1), 1997.
- [Ston84] Stonebraker, M.: "Virtual Memory Transaction Management", *ACM Operating Systems Review*, Vol. 18(2), 1984.
- [Stro93] Stroustrup, B.: "The C++ Programming Language", Second Edition, Addison-Wesley, 1993.

- [Trai82] Traiger, I.: "Virtual Memory Management for Database Systems", *ACM Operating Systems Review*, Vol. 16(4), 1982.
- [Vada95a] Vadaparty, K.: "Memory-Mapped Architectures", *Journal of Object-Oriented Programming*, 8(6) : 18-26, 1995.
- [Vada95b] Vadaparty, K.: "Pointer Swizzling at Page-Fault Time", *Journal of Object-Oriented Programming*, 8(7) : 12-20, 1995.
- [WhDe94] White, S.J. and DeWitt, D.J.: "QuickStore: A High Performance Mapped Object Store, *International Conference on ACM SIGMOD*, Minneapolis, 1994.
- [Wils91] Wilson, P.R.: "Pointer Swizzling at Page-Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", *Computer Architecture News*, pp. 6-13, June 1991.

CHAPTER 3

SERIALIZATION AND MFC

3.1 INTRODUCTION

Serialization is the process of writing or reading one or more objects to or from a persistent storage medium. In this process, an object is transformed into a linear stream of bytes, and deserialization is the reverse process of restoring the object from a byte stream. Serialization is not only used for storing objects in persistent memory such as a disk file, but also for transporting them from one place to another, for example, through a network connection.

The basic idea of serialization is that an object should be able to write its current state, usually indicated by the values of its member variables to a persistent storage. Later, the object can be re-created by reading or deserializing the object's state from the storage. A key point is that the object itself is responsible for reading and writing its own state. Thus, for a class to be serializable, it must implement the basic serialization operations.

This chapter presents an in-depth dissection of the implementation of serialization in the Microsoft Foundation Class (MFC) library. As indicated before in Chapter 1, this implementation was important for the development of Glasgow Graphics and Sound (GGS). The MFC serialization provides the persistence mechanism to the multimedia objects in GGS. In normal circumstances, the MFC user does not need to know the serialization internals. It is easy and straightforward to add the serialization support to an MFC application created by the AppWizard. This has been demonstrated in [Jack97] by adding only three lines of additional codes to an existing project. However, when something is that easy, there must be a lot of work going on behind the scenes. This is especially true for MFC serialization.

It is interesting to look at the details of MFC serialization because it is not clearly understood by most of the application developers [ShWi96]. MFC is full of undocumented classes. The only way to explore and understand these classes is to study the library source code. The discussions and illustrations presented in this chapter are believed to be valuable for the future MFC users.

MFC supplies built-in support for serialization in the class `CObject`. All classes derived from `CObject` can take advantage of its serialization protocol. There is a lot of debate about whether it is a good design to have almost every class in MFC derived from `CObject`. C++ linguists, in general, do not prefer a single rooted hierarchy. They argue

that such a design causes the virtual tables to grow and dramatically decreases the runtime performance. Some others argue that it is not a good object-oriented design. After revealing a number of important `CObject`'s internals, we shall revisit this question and compare the functionality that `CObject` delivers with the performance hit taken by adding its virtual table entries.

3.2 THE PROBLEMS

Serializing an object containing non-dynamic data is a fairly trivial process of storing/retrieving each data member. For each member of a built-in type (such as `int` or `float`) its value is written directly to the stream. But it begs the question of how to read and write complex user-defined types. The answer is that each object delegates the responsibility to read and write any non-primitive data members to the member itself [Libe97]. That is, user-defined types write out their primitive members and tell their user-defined members to write themselves. Each one, in turn, recursively delegates this responsibility. Ultimately, every object can be written and read because every user-defined type is composed of primitive types. The serialization of dynamic data structures, on the other hand, introduces several complicated issues.

3.2.1 Pointer Storage

Dynamic data structures by definition contain pointers and, as discussed in Chapter 2, it is meaningless to store pointers in a byte stream since their values will be invalid when restored later. In the context of serialization, one solution to this problem is to store a unique object ID in place of the pointer to represent the object referred to by the pointer. That same ID must then be used in the stream everywhere pointers to that object appear in the data structure. When the data structure is later restored from the stream, the dynamic object must be recreated and all occurrences of its ID must become pointers to the recreated object.

3.2.2 Virtual Constructor

Arbitrary types must be created as needed, but the `new` operator can only create an explicit type, so a "virtual constructor" for `CObject` is necessary. A virtual constructor is used when the type of an object needs to be determined from the context in which the object is constructed [Cop192]. In MFC, the context is based on the information read from a serialized archive. However, a virtual constructor is only a concept and not part of C++ language.

3.2.3 Base Class Pointer

Another issue arises due to the compatibility in C++ between derived and base class pointers. When storing an object of a derived type through a base class pointer, we must store (and later restore) the derived type object, even though the static types in the code performing the serialization are all of the base class pointer type. Performing this restoration correctly requires the insertion of some class information into the serialization stream and a mechanism to dynamically create objects of a type indicated by the information in the stream.

3.3 A SERIALIZABLE CLASS

The following steps establish the general algorithm for serializing an object:

- The parents in the inheritance hierarchy are asked to serialize themselves.
- Primitive data members are serialized.
- Then object pointers are serialized.
- Sub-objects (i.e. user-defined objects) are asked to serialize themselves.

This last act of asking member objects to serialize themselves, allows the implementation of serialization to be fully encapsulated and localised in that object which knows best how to do it. In other words, the serializable base class does not change when new classes are added. Classes that contain a new type do not need to change either. The responsibility for serializing that new type is encapsulated in the new type itself.

In MFC, five main steps are required to make a class serializable [MSDN31]:

- The class should be derived from `CObject` or its children.
- The class should override the `Serialize()` member function.
- The `DECLARE_SERIAL` macro should be placed in the class declaration.
- A constructor that takes no argument is necessary.
- The class implementation should contain the `IMPLEMENT_SERIAL` macro.

MFC uses an object of the `CArchive` class as an intermediary between the object to be serialized and the storage medium. This `CArchive` object is always associated with a `CFile` object, from which it obtains the necessary information for serialization, including the file name and whether the requested operation is a read or write. The object that performs a serialization operation can use the `CArchive` object without

regard to the nature of the storage medium.

A `CArchive` object uses overloaded insertion (`<<`) and extraction (`>>`) operators to perform writing and reading operations. If the `Serialize()` member function is called directly rather than through the `>>` and `<<` operators of `CArchive`, the last three steps (i.e. the macros and the default constructor) are not required for serialization.

3.4 THE MACROS IN MFC

One common pattern in MFC is the use of pairs of macros (`DECLARE/IMPLEMENT`) to add various functionalities to new classes. The `DECLARE` macros always declare some member variables and functions for a class, while the `IMPLEMENT` macros, as the name suggests, always implement the member functions. The `DECLARE` macros always go in the header file and the `IMPLEMENT` macros always stay in the C++ file. The functionalities offered by these macros can be divided in three categories as shown in Table 3.1.

Table 3.1 The DECLARE / IMPLEMENT Macro Pairs in MFC

Macro Pair	RTCI	Dynamic Creation	Serialization
<code>DECLARE_DYNAMIC/ IMPLEMENT_DYNAMIC</code>	✓	✗	✗
<code>DECLARE_DYNCREATE/ IMPLEMENT_DYNCREATE</code>	✓	✓	✗
<code>DECLARE_SERIAL/ IMPLEMENT_SERIAL</code>	✓	✓	✓

3.4.1 Run Time Class Information (RTCI)

The serialization mechanism is built on top the `CObject`'s run time class information (RTCI) feature that lets the developer determine information about an object such as class name and parent at run time. The code to support RTCI lives in all `DECLARE/IMPLEMENT` macros. The following example shows how to use RTCI to check that a polymorphic cast to a derived class is safe:

```

CObject* pObject = new CMyClass ;
if (pObject->IsKindOf(RUNTIME_CLASS(CMyClass)))
    CMyClass* pMyObject = (CMyClass*) pObject ;

```

It is important to note that the above example is using another macro, `RUNTIME_CLASS`. The natural question is why the MFC developers did not use the C++ run-time type information (RTTI) features [ANSI97] instead of going to all this trouble with macros. In fact, they needed RTTI support in the very early days of MFC. A future version of MFC may replace [ShWi96] the implementation of the RTCI macros to use the C++ RTTI features. However, this will not affect existing applications because the macros isolate them from any changes.

Now we look at the definition of `DECLARE_DYNAMIC` and apply the macro to a sample class name:

```
#define DECLARE_DYNAMIC(class_name) \  
public: \  
    static CRuntimeClass class##class_name; \  
    virtual CRuntimeClass* GetRuntimeClass() const; \  

```

One trick this macro employs is the preprocessor concatenation operator `##`. Operator `##` tells the preprocessor to concatenate what is on the right of the operator onto what is on the left. For example, `class##CMyClass` would generate `classCMyClass`. If arguments appear in the concatenation, they are replaced and then concatenated.

Plugging `CMyClass` into the `DECLARE_DYNAMIC` macro and running the preprocessor causes the following code to be added in the class definition:

```
public: \  
    static CRuntimeClass classCMyClass ; \  
    virtual CRuntimeClass* GetRuntimeClass() const ;
```

The static `CRuntimeClass` declaration is very important:

Listing 3.1 The `CRuntimeClass` Declaration

```
struct CRuntimeClass \  
{ \  
    // Attributes \  
    LPCSTR m_lpszClassName; \  
    int m_nObjectSize; \  

```

```

UINT m_wSchema; // schema number of the loaded class
CObject* (PASCAL* m_pfnCreateObject)();
CRuntimeClass* m_pBaseClass;
// Operations
CObject* CreateObject();
BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass) const;
// Implementation
void Store(CArchive& ar) const;
static CRuntimeClass* PASCAL Load(CArchive& ar, UINT* pwSchemaNum);
// CRuntimeClass objects linked together in simple list
CRuntimeClass* m_pNextClass;
};

```

The name "CRuntimeClass" appears to be an MFC misnomer. However, C++ structures are just classes in which everything defaults to public. Because CRuntimeClass is a structure, all of its members are public.

To understand more about CRuntimeClass, it is essential to expose the IMPLEMENT_DYNAMIC(class_name, base_class_name) macro. After running the preprocessor with the arguments, CMyClass and CObject as the base class:

```

AFX_DATADEF CRuntimeClass CMyClass::classCMyClass =
{
    "CMyClass", sizeof(CMyClass), 0xFFFF, NULL,
    RUNTIME_CLASS(CObject), NULL
};
static const AFX_CLASSINIT _init_CMyClass (&CMyClass:: classCMyClass);
CRuntimeClass* CMyClass::GetRuntimeClass() const
{
    return &CMyClass::classCMyClass;
}

```

First of all, IMPLEMENT_DYNAMIC initialises the static CRuntimeClass structure as follows:

```

m_lpszClassName = "CMyClass" ;
m_nObjectSize = sizeof(CMyClass) ;

```

```

m_wSchema = 0xFFFF ;
m_pfnCreateObject = NULL ;
m_pBaseClass = RUNTIME_CLASS(CObject) ;

```

Second, `IMPLEMENT_DYNAMIC` creates `AFX_CLASSINIT` which is a static structure with only a constructor:

```

struct AFX_CLASSINIT
{ AFX_CLASSINIT (CRuntimeClass* pNewClass) ; }

```

Therefore, `IMPLEMENT_DYNAMIC` creates a static `AFX_CLASSINIT` structure and calls its constructor with a pointer to the static structure, `classCMyClass`. This causes the class to be added to an MFC state list. This is discussed in detail later.

Finally, the `IMPLEMENT_DYNAMIC` macro generates the overridden member function `GetRuntimeClass()`. `GetRuntimeClass()` just returns the address of the static `CRuntimeClass` structure, `classCMyClass` just like the `RUNTIME_CLASS` macro.

This discussion brings up some interesting questions about `CRuntimeClass`. What are members like `m_wSchema` and `m_pfnCreateObject` used for? Why does it appear that `CRuntimeClass` has a linked list? To help answer some of these questions, it is necessary to look at how `CObject::IsKindOf()` is implemented:

Listing 3.2 The Implementation of `CObject::IsKindOf()`

```

BOOL CObject::IsKindOf (const CRuntimeClass* pClass) const
{
    CRuntimeClass* pClassThis = GetRuntimeClass() ;
    while (pClassThis != NULL)
    {
        if (pClassThis == pClass)
            return TRUE;
        pClassThis = pClassThis->m_pBaseClass;
    }
    return FALSE;          // walked to the top, no match
}

```

`IsKindOf()` takes the static `CRuntimeClass` pointer and then calls `GetRuntimeClass()` for the current object (or `this`). It then compares the `CRuntimeClass` pointers to see if they are pointing to the same static structure. If they are the same object, a match has been made. If not, `IsKindOf()` walks up the inheritance tree looking for a match.

3.4.2 Dynamic Creation

`CObject::IsKindOf()` is the key to RTCI. Similarly, the dynamic creation of objects is based on the `CreateObject()` method of the `CRuntimeClass` structure. The `DECLARE_DYNCREATE` macro adds only one line on top of the `DECLARE_DYNAMIC` macro:

```
static CObject* CreateObject() ;
```

`IMPLEMENT_DYNCREATE` generates the `CreateObject()` member function for the class in addition to the usual `IMPLEMENT_DYNAMIC` output. The `CreateObject()` implementation is fairly simple:

```
CObject* PASCAL CMyClass::CreateObject()
{
    return new CMyClass ;
}
```

`IMPLEMENT_DYNAMIC` initialises `CRuntimeClass::m_pfnCreateObject` function pointer with `NULL` because it does not use this feature. On the other hand, `IMPLEMENT_DYNCREATE` passes the address of `CMyClass::CreateObject()` to initialise `m_pfnCreateObject`. The static structure, `CRuntimeClass` uses `m_pfnCreateObject` to dynamically create an object of type `CMyClass` at run time. The following is an example of how to use dynamic creation:

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CMyClass) ;
CObject* pObject = pRuntimeClass->CreateObject() ;
ASSERT(pObject->IsKindOf(RUNTIME_CLASS(CMyClass))) ;
```

3.5 A SAMPLE DATA STRUCTURE FOR SERIALIZATION

From an application programmer's viewpoint, it is straightforward to use the MFC serialization in most cases. However, the implementation details are not easy to grasp

as we have seen in the earlier sections. It would be much easier to explain the actual serialization mechanism with the help of an example.

One example with many patterns is that of graphs, made up of vertices and edges. Vertices can exist in isolation but edges must connect two existing vertices. Edges know about their vertices and vertices know the edges belonging to them and this information is always correct and consistent. Listing 3.3 looks at the declarations of two classes, `CVertex` and `CEdge`, deriving them from `CObject`. All member functions except `Serialize()` are left out for clarity:

Listing 3.3 Vertices and Edges

```
class CVertex : public CObject
{
    . . .
private:
    int      m_nCordX ;
    int      m_nCordY ;
    int      m_nEdgeNum ;
    CEdge*   m_pEdges[MAX_EDGES] ;

protected:
    virtual void Serialize(CArchive& ar) ;
    DECLARE_SERIAL(CVertex)
};

class CEdge : public CObject
{
    . . .
private:
    int      m_nLineThickness ;
    CVertex* m_pVertex1 ;
    CVertex* m_pVertex2 ;

protected:
    void Serialize(CArchive& ar) ;
```

```
    DECLARE_SERIAL(CEdge)
};

IMPLEMENT_SERIAL(CVertex, CObject, 1)
IMPLEMENT_SERIAL(CEdge, CObject, 1)

void CEdge::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_nLineThickness << m_pVertex1 << m_pVertex2 ;
    else
        ar >> m_nLineThickness >> m_pVertex1 >> m_pVertex2 ;
}

void CVertex::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_nCordX << m_nCordY << m_nEdgeNum ;
        for (int i=0; i< m_nEdgeNum; i++)
            ar << m_pEdges[i] ;
    }
    else
    {
        ar >> m_nCordX >> m_nCordY >> m_nEdgeNum ;
        for (int i=0; i< m_nEdgeNum; i++)
            ar >> m_pEdges[i] ;
    }
}
```

Two dimensional Cartesian coordinates can be expressed using the MFC class `CPoint` which is also serializable. In practice, `CVertex` should maintain a variable-length array or list of `CEdge` pointers instead of a fixed-length array, `m_pEdges`. However, data members in Listing 3.3 are kept in simple forms so that the serialization mechanism can be understood clearly.

3.5.1 An Illustration in GGS

To illustrate how MFC serialization works, `CVertex` and `CEdge` classes are used to construct a data structure. Figure 3.1 presents this data structure drawn in GGS. GGS can create such illustrations just like any other graphics packages.

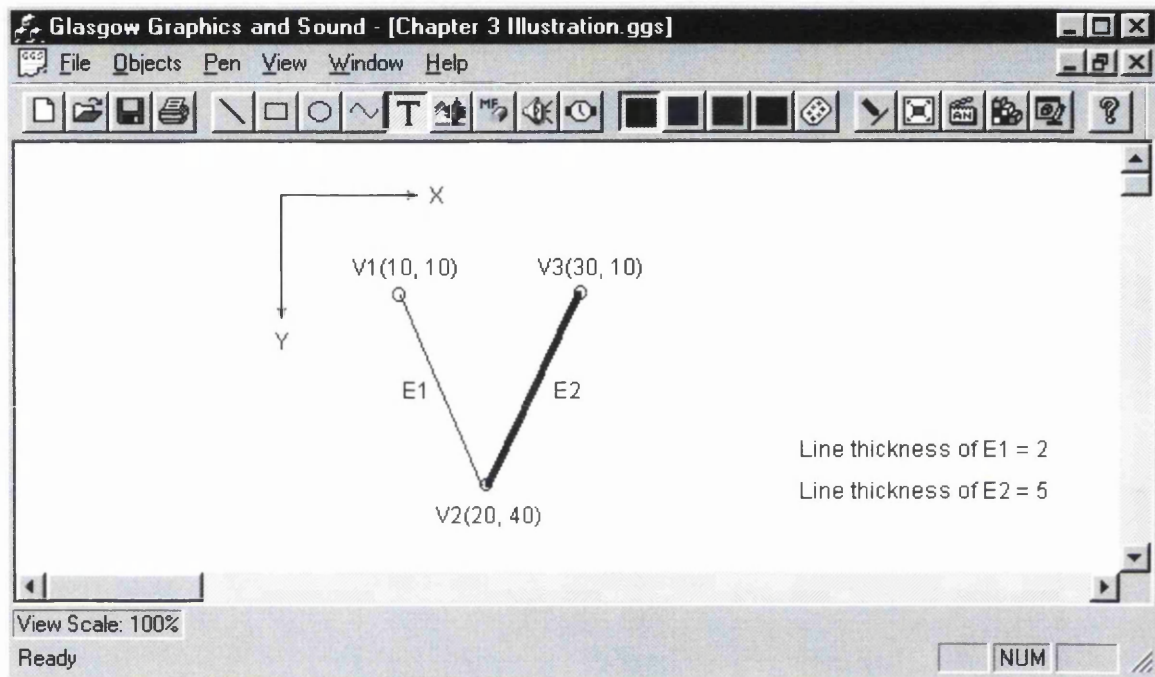


Figure 3.1 A Sample Pattern of Vertices and Edges

The coordinates and the line thickness values are in arbitrary units. For the sake of simplicity, let us consider a document class containing pointers to these vertices and edges:

```
class CMyDoc : public CDocument
{
    . . .
    CVertex*   V1 ;
    CVertex*   V2 ;
    CVertex*   V3 ;
    CEdge*     E1 ;
    CEdge*     E2 ;
};
```

The `Serialize()` function of that document class would be equally simple:

Listing 3.4 Serialization of Sample Vertices and Edges

```

void CMyDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << V1 << V2 << V3 << E1 << E2 ;
    else
        ar >> V1 >> V2 >> V3 >> E1 >> E2 ;
}

```

Once these objects have been initialised, they could be connected together as shown below in Figure 3.2:

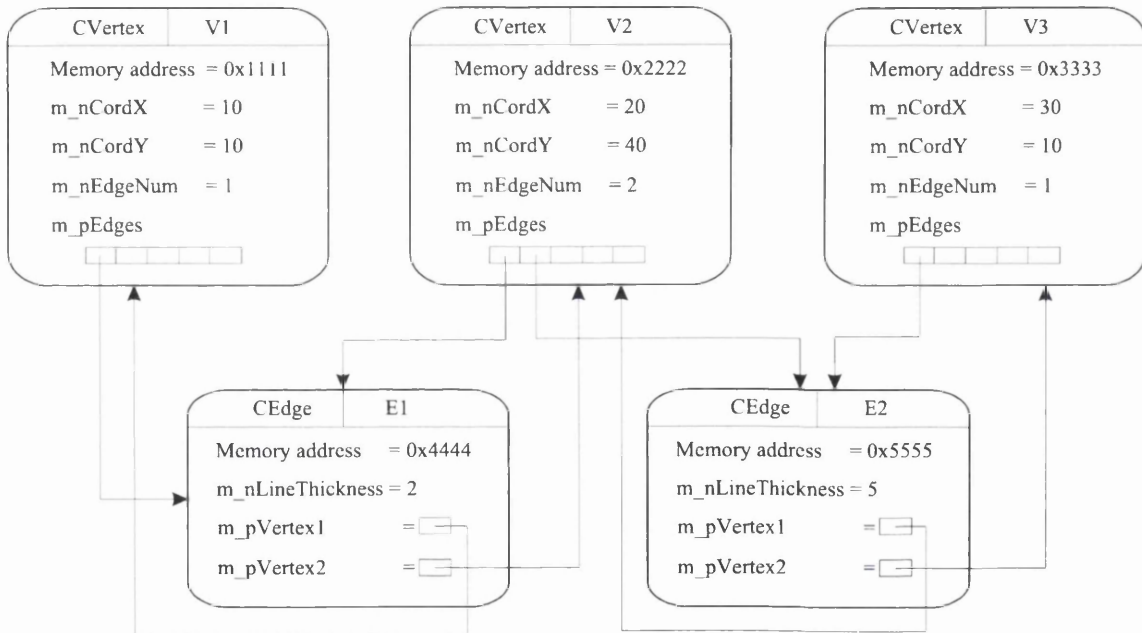


Figure 3.2 Possible Data Structure Arrangement in Memory

3.6 STORING OBJECTS

The MFC implementation for persistent data relies on a compact binary format for saving the data for many objects in a single contiguous part of a file. This binary format defines the structure for how the data is stored, but it is the object's `Serialize()` member function that provides the actual data saved by the object.

MFC solves the structuring problem by using the class `CArchive`. A `CArchive` object

provides a context for persistence that lasts from the time the archive is created until the `CArchive::Close()` member function is called, either explicitly by the programmer or implicitly by the destructor when the scope containing the `CArchive` is exited.

Two member functions, `ReadObject()` and `WriteObject()` in `CArchive` do most of the serialization work. These functions are not called directly from the user code [OnHa96a]. They are used by class-specific type-safe insertion and extraction operators. For example:

```

CMyClass* pObj;
CArchive& ar;
ar << pObj; // Calls ar.WriteObject(pObj)
ar >> pObj; // Calls ar.ReadObject(RUNTIME_CLASS(CMyClass))

```

`CArchive::WriteObject()` writes the header data used to reconstruct the object. This data consists of two parts: the type of the object and the state of the object. This member function is also responsible for maintaining the identity of the object being written out, so that only a single copy is saved, regardless of the number of pointers to that object (including circular pointers).

Saving and restoring objects relies on some “manifest constants” or tags [MSDN32]. These are values that are stored in binary and provide important information to the archive (‘w’ prefix indicates 16-bit quantities):

Table 3.2 Tags in MFC Serialization

Tags	Description	Value
wNullTag	Used for NULL object pointers.	0
WNewClassTag	Indicates the class description that follows is new to this archive context.	-1
WOldClassTag	Indicates the class of the object being read has been seen in this context.	0x8000

When storing objects, `CArchive` maintains a store map (`m_pStoreMap` data member of type `CMapPtrToPtr`) which is a mapping from a stored object to a 32-bit persistent identifier (PID). A PID is assigned to every unique object and every unique class name

that is saved in the context of the archive. These PID's are handed out sequentially starting at 1. It is important to note that these PID's have no significance outside the scope of the archive and, in particular, are not to be confused with any database terminology.

Starting with MFC version 4.0 the `CArchive` class has been extended to support very large archives. In previous versions, a PID was a 16-bit quantity [Onio95a], limiting the archive to 0x7FFE (32766) objects. PID's are now 32-bit, but they are written out as 16-bit unless they are larger than 0x7FFE. In other words, they are 16-bit until 0x7FFE is reached but 32-bit thereafter. This technique maintains a backward compatibility with earlier archives.

When a request is made to save an object to an archive (usually through the global insertion operator), a check is made for a `NULL CObject` pointer. If the pointer is `NULL`, the `wNullTag` is inserted into the archive stream.

If there is a real object pointer that is capable of being serialized, the store map is checked to see if the object has been saved already. If it has, the 32-bit PID associated with that object is inserted in the stream.

If the object has not been saved before, there are two possibilities to take into account: either both the object and the exact type (i.e. the class) of the object are new to this archive context, or the object is of an exact type already seen. To determine if the type has been seen already, the store map is queried for a `CRuntimeClass` structure that matches the `CRuntimeClass` structure associated with the object being saved. If this class has been seen before, `WriteObject()` inserts the bit-wise OR'ing of the `wOldClassTag` and the PID of this class. If the `CRuntimeClass` structure is new to the archive context, then `WriteObject()` assigns a new PID to this class and writes the class description into the archive, preceded by the `wNewClassTag`. The descriptor for this class is written using `CRuntimeClass::Store()`:

```
void CRuntimeClass::Store(CArchive& ar) const
{
    WORD nLen = (WORD)lstrlenA(m_lpszClassName) ;
    ar << (WORD)m_wSchema << nLen ;
    ar.Write(m_lpszClassName, nLen*sizeof(char)) ; }

```

`CRuntimeClass::Store()` inserts the schema number of the class (discussed later) and the name of the class as an ASCII string, preceded by the string length. Following the insertion of the class information, the archive places the object in the `m_pStoreMap` and then calls the `Serialize()` member function to insert class-specific data into the archive. Placing the object in the `m_pStoreMap` before calling `Serialize()` prevents multiple copies of the object from being saved to the archive.

Hence, `CArchive` uses its store map to access quickly the class information of similar objects. `CArchive` also uses the map to ensure that it will write out `CRuntimeClass` information only once for a certain class then reference it later in the serialization stream.

3.6.1 Storing Vertices and Edges

To help understand how serialization works, we shall trace a complete write operation in the earlier example of vertices and edges. In addition to the class instances shown in Figure 3.2, which are part of the document (i.e. `CMyDoc`) class, each class type will have an instance of the `CRuntimeClass` structure:

Table 3.3 `CRuntimeClass` Structures of `CVertex` and `CEdge`

Class	<code>CVertex</code>	<code>CEdge</code>
Schema	1	1
Length of name string	7	5
Name string	" <code>CVertex</code> "	" <code>CEdge</code> "

`CArchive::WriteObject()` handles writing both the runtime class structure and the data associated with a given object. In Listing 3.4, `V1` is serialized first. When neither the runtime class structure nor the object itself has been previously written out, what is actually written to the stream will be as follows:

Tag	RUNTIME_CLASS (<code>CVertex</code>)			Part of <code>V1</code>		
	Schema	Length	String	Coordinates		No of Edges
<code>WNewClassTag</code>	1	7	" <code>CVertex</code> "	10	10	1

The PID of the `CRuntimeClass` structure of `CVertex` and `V1` will be 1 and 2

respectively. When `m_pEdges` of `V1` is serialized, `E1` is serialized as a result because `m_pEdges` of `V1` contains a pointer to `E1`. In fact, all objects directly or indirectly connected to `V1` are serialized.

At the end of the sample data structure serialization, the store map will appear as shown in Table 3.4:

Table 3.4 PID's of Vertices and Edges

Objects	PID
CRuntimeClass structure of CVertex	1
V1	2
CRuntimeClass structure of CEdge	3
E1	4
V2	5
E2	6
V3	7

If the `CRuntimeClass` structure has already been encountered and written out previously, then its PID will suffice for all subsequent uses. When the object (and hence the runtime class information) has been seen before, all that will be written to the stream is the object's PID. Therefore, the actual data that will be written out for the sample data structure of vertices and edges is as follows:

WNewClassTag	1	7	"CVertex"	10	10	1	wNewClassTag	1	5	
"CEdge"	2	2	wOldClassTag 1	20	40	2	4	wOldClassTag 3		
5	5	wOldClassTag 1	30	10	1	6	5	7	4	6

The PID's are shown with gray fill. It is important to note that each class instance is actually stored only once with the PID's referencing previously read in classes. The bitwise OR'ing of `wOldClassTag` with the PID's of `CRuntimeClass` structures is important to distinguish the class PID's from the object PID's. This information is essential when the archives are restored.

3.7 THE SERIALIZATION MACROS

There is a good reason for not describing the serialization macros before this section. It was mentioned earlier in Section 3.4 that `DECLARE_SERIAL` and `IMPLEMENT_SERIAL` provide all the supports offered by the `DYNCREATE` macro pair. Here is the definition of `DECLARE_SERIAL`:

```
#define DECLARE_SERIAL(class_name) \
DECLARE_DYNCREATE(class_name) \
friend CArchive& operator>> (CArchive& ar, class_name* &pOb);
```

The `DECLARE_SERIAL` macro is actually adding only one line of code to the RTCI and dynamic creation sections. The extra line declares a global extraction operator as a friend to the new class.

`IMPLEMENT_SERIAL` generates the implementation of the global extraction operator:

```
CArchive& operator>> (CArchive& ar, CMyClass*& pOb)
{
    pOb = (CMyClass*) ar.ReadObject (RUNTIME_CLASS(CMyClass)) ;
    return ar ;
}
```

It is quite natural to expect a global insertion operator as well but that is not included in the `SERIAL` macro pair. The internal details are explained as follows.

3.8 ON-THE-FLY REGISTRATION AND RESTORATION

As we have seen before, the persistence and dynamic object creation mechanisms of MFC use the `CRuntimeClass` data structure to uniquely identify classes. MFC associates one structure of this type with each dynamic and/or serializable class in the application. These structures are initialised at application startup time using a special static object of type `AFX_CLASSINIT` as mentioned in Section 3.4.1. The constructor of `AFX_CLASSINIT` links `CRuntimeClass` structures into the MFC type registry. `AFX_CLASSINIT` does not take up any data space because it has no member data. This mechanism allows on-the-fly registration of object types whenever they are linked into the program. The idea of types registering themselves is the core of object-oriented design [Beve95]. If a type registers its own existence with a registry instead of

hardcoding the type into the registry, then the type can be freely added and removed from the program without any code changes to the registry.

The `SERIAL` macros define the `operator>>` for a new class because it is called with a pointer to the class, but no instance of the class will exist until the instance has been restored from the file. Without an instance of the class, MFC cannot access the run-time class information to ensure that the object being loaded is equivalent to or is a derived class of the given pointer. By overloading `operator>>`, MFC is able to pass a pointer to the run-time class information so that the serialization mechanism will be typesafe.

A mapping scheme is required to allow a class to be created based on the information read from a file. The actual class name is the ideal candidate to write to a file in order to identify a class because C++ class names have to be unique inside an application. That is why `CRuntimeClass::Store()` writes the class name as an ASCII string while storing objects. When the name of the class is loaded from the file, the type registry is searched for that name. While reading, `CArchive` maintains an array of objects already created and stores the already-read `CRuntimeClass` structure information in the array. This way, `CArchive` can look it up when it finds a reference that was written in the serialization stream. As long as the type exists in the registry and was declared with either `DECLARE_DYNCREATE` or `DECLARE_SERIAL`, the object can be constructed. The actual loading of the data appropriate to that kind of object is delegated to the object itself by calling its `Serialize()` member function. The type of object created is separated from the type of object requested. If a derived class is loaded into a pointer to a base class, then the correct derived class will still be created.

It is time to answer why there is no global insertion operator. `WriteObject()` in `CArchive` is similar to its `ReadObject()`. It writes out the `CRuntimeClass` information and then calls the object's `Serialize()` function. Since `WriteObject()` does not need any specific `CRuntimeClass` information, the `CObject` insertion operator is sufficient and a new insertion operator for the `CObject` derivative is not necessary.

3.9 VERSIONABLE OBJECTS

The `IMPLEMENT_SERIAL` macro takes an unsigned integer that defines the schema or version information for the object. In fact, the data member `m_wSchema` of the

`CRuntimeClass` structure (Listing 3.1) is initialised with this number. If the schemas do not match when the data is read, the framework will not read in the object.

Serialization, to some extent is an all-or-nothing proposition [Howa97]. Especially, the MFC serialization mechanism does not record the length of each object into the archive. If one object cannot be loaded, MFC will not be able to skip that object and load the rest of the archive. This can be a serious pitfall unless versionable object creation is possible.

The basic schema technique is adequate for objects that do not exist in multiple versions. One solution to supporting backward-compatible objects is to use the constant, `VERSIONABLE_SCHEMA`. Versionable objects in MFC are easily defined by OR'ing the `VERSIONABLE_SCHEMA` constant with the schema number in `IMPLEMENT_SERIAL`:

```
IMPLEMENT_SERIAL(CMyClass, COBJECT, VERSIONABLE_SCHEMA | 1)
```

This simple change can make the objects backward compatible. For example, if the `CVertex` class design in Listing 3.3 is changed to include the Z coordinate value, `CVertex::Serialize()` can be rewritten as follows:

```
void CVertex::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_nCordX << m_nCordY << m_nCordZ << m_nEdgeNum ;
        for (int i=0; i< m_nEdgeNum; i++)
            ar << m_pEdges[i] ;
    }
    else
    {
        switch (ar.GetObjectSchema())
        {
            case 1: // Load old version
                ar >> m_nCordX >> m_nCordY >> m_nEdgeNum ;
                break;
            case 2: // Load current version
                ar >> m_nCordX >> m_nCordY >> m_nCordZ >> m_nEdgeNum ;
        }
    }
}
```



```

    }
    for (int i=0; i< m_nEdgeNum; i++)
        ar >> m_pEdges[i] ;
    }
}

```

`CArchive::GetObjectSchema()` returns the schema for the current object. There is a caveat with `GetObjectSchema()`. It can be called only once during the serialization of an object; subsequent calls return -1. Therefore, it is not possible to build versionable class hierarchies using `VERSIONABLE_SCHEMA`. The following example illustrates this point:

```

// Base and child classes have different schema numbers
class CBase : public CObject { ... };
class CChild : public CBase { ... };
IMPLEMENT_SERIAL(CBase, CObject, VERSIONABLE_SCHEMA | 1);
IMPLEMENT_SERIAL(CChild, CBase, VERSIONABLE_SCHEMA | 2);

void CBase::Serialize(CArchive& ar)
{
    if ( ar.IsLoading() )
    {
        UINT nSchema = ar.GetObjectSchema(); // Returns 1
        // Load versionable object...
    }
}

void CChild::Serialize(CArchive& ar)
{
    CBase::Serialize(ar); // Call base class Serialize()
    if ( ar.IsLoading() )
    {
        UINT nSchema = ar.GetObjectSchema(); // Returns -1
        // Second call to GetObjectSchema() FAILS
    }
}

```

The call to `GetObjectSchema()` in the `Serialize()` function of `CChild` fails because it was already called in the base class code. There is one way to avoid this problem by storing the version numbers explicitly [Stou97], instead of relying on MFC's `VERSIONABLE_SCHEMA`. The following code shows one simple method:

```
class CBase : public CObject { enum { verBase = 1 }; ... };

void CBase::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);
    if ( ar.IsStoring() )
    {
        ar << verBase;    // Explicitly save version
        // ... then save everything else
    }
    else
    {
        UINT nMySchema;    // Load version number
        ar >> nMySchema;
        switch ( nMySchema )
        {
            case verBase:
                // Load current version of object
                // Other cases for older versions
            }
        }
    }
}
```

This custom versioning scheme can be implemented in child classes and it does not have any problem in building versionable class hierarchies.

3.10 LIMITATIONS

By creating a foundation of a run-time type mechanism with a type registry and building serialization on top, MFC implements a fast, flexible, typesafe serialization mechanism. This mechanism should be powerful enough to satisfy most design requirements. There are limitations but it is possible to get around at least some of

them. For example, the serialization macros cannot be used with abstract base classes. The solution [INET31] is to define a macro similar to `IMPLEMENT_SERIAL` but without the `CreateObject()` reference.

The current implementation of `CRuntimeClass` does not support multiple inheritance (MI). This does not mean that MI cannot be used in an MFC application. The responsibilities involved when working with objects that have more than one base class are discussed in [MSDN33].

The Standard Template Library (STL) is now part of the C++ language [ANSI97]. Unfortunately, the STL template classes do not have any persistence support [Stev98]. They cannot be serialized using MFC either. The class names must be known at pre-processing time for this to work, but templates are expanded by the compiler long after the preprocessor is gone. Consequently, there is no way for the preprocessor to know the compiler-generated class name [Holu96].

However, the general idea that the MFC serialization does not work with templates is not right. The STL equivalent template classes in MFC are serializable and typesafe. The typesafe template classes have inline functions using the C++ type-checking facility to eliminate errors caused by mismatched pointer types. When serializing the template based collection classes in MFC, the `Serialize()` member function should only be used and not the insertion and extraction operators.

3.11 CLOSURE

The advanced memory diagnostic features of `CObject` are beyond the scope of this thesis. The diagnostics are implemented by two virtual functions, `AssertValid()` and `Dump()`. `AssertValid()` validates the object's integrity and `Dump()` produces a diagnostic dump of the object. But they work in debug builds only to avoid any performance hit taken by adding their virtual table entries. [ShWi96] is recommended as a detailed source of information.

Let us revisit the question posed earlier in the chapter: "Does the MFC architecture with `CObject` as root cause performance problems?" C++ has to index the class virtual table at run-time to determine the correct function to execute. Out of five virtual functions in `CObject`, two are in debug builds only, so they do not really affect the release-build

performance. The largest overhead with using virtual functions is the increased instance size from a virtual table. Therefore, the trade-off in deriving from `CObject` is three extra virtual functions (i.e. `GetRuntimeClass()`, `Serialize()` and the destructor) in exchange for RTCI, dynamic creation, serialization and memory diagnostics. The choice, of course, depends on the MFC user.

The multimedia objects in GGS are derived from `CObject` to take advantage of these attractive features.

3.12 REFERENCES

- [ANSI97] ANSI/ISO C++ Committee: "International Standard for Information Systems – Programming Language C++", ISO / IEC IS 14882, November 1997.
- [Beve95] Beveridge, J.: "Inside MFC Serialization: Typesafe Serialization that's Fast and Flexible", Dr. Dobb's Journal, October 1995.
- [Cop192] Coplien, J.O.: "Advanced C++ Programming Styles and Idioms", Addison Wesley, 1992.
- [Holu96] Holub, A.: "Roll Your Own Persistence Implementations to Go Beyond the MFC Frontier", Microsoft Systems Journal, June 1996.
- [Howa97] Howard, R.R.: "Simple Object Persistence with STORE_TABLE", Journal of Object Oriented Programming, June 1997.
- [INET31] Wingo, S.: "Visual C++/MFC Frequently Asked Questions", http://www.stingray.com/mfc_faq/.
- [Jack97] Jackson, A.G.: "Adding Serialization", Visual C++ Developers Journal, 1997.
- [Libe97] Liberty, J.: "Beginning Object-Oriented Analysis and Design with C++", Wrox Press, 1997.
- [MSDN31] Microsoft Developer Network Library: "Serialization (Object Persistence)", April 1999.
- [MSDN32] Microsoft Developer Network Library: "TN002: Persistent Object Data Format", April 1999.
- [MSDN33] Microsoft Developer Network Library: "TN016: Using C++ Multiple

Inheritance with MFC", April 1999.

- [OnHa96a] Onion, F. and Harrison, A.: "Dynamic Data Structure Serialization in MFC", C++ Report, March 1996.
- [Onio95a] Onion, F.: "Object Persistence in MFC", C++ Report, Nov/Dec 1995.
- [ShWi96] Shepherd, G. and Wingo, S.: "MFC Internals: Inside the Microsoft Foundation Class Architecture", Addison-Wesley Developers Press, 1996.
- [Stev98] Stevens, A.: "The Persistent Template Library", Dr Dobb's Journal, March 1998.
- [Stou97] Stout, J.: "Object Persistence and Versioning: Serialization in MFC", Visual C++ Developers Journal, 1997.

CHAPTER 4

USER INTERFACE AND CLASSES IN GGS

4.1 PRINCIPLES OF USER INTERFACE DESIGN

An important principle of user interface design is that the user should always feel in control of the software rather than feeling controlled by the software. This has various implications. The first implication is the operational assumption that the user initiates actions, not the computer or software. The user plays an active, rather than a reactive role.

Visibility of information and choices also reduce the user's mental workload. Users can recognise a command easier than they can recall its syntax. Often they like to explore an interface and learn by trial and error. An effective interface allows for interactive discovery. It provides only appropriate sets of choices and warns users about potential situations where they may damage the system or data, or better, makes actions reversible or recoverable.

A good interface should always provide feedback for the user's actions. Visual and sometimes audio cues should be presented with every user interaction to confirm that the software is responding to their input and to communicate details that distinguish the nature of the action.

An interface should be simple (not simplistic), easy to learn and easy to use. It must also provide access to all functionality provided by an application. Maximising functionality and maintaining simplicity work against each other in the interface. An effective design balances these objectives.

One way to support simplicity is to reduce the presentation of information to the minimum required to communicate adequately. For example, wordy descriptions for command names or messages should be avoided. Irrelevant or verbose phrases clutter the design, making it difficult for users to easily extract essential information. Another way to design a simple but useful interface is to use natural mappings and semantics. The arrangement and presentation of interface elements affect their meaning and association.

The interface designer can also help users manage complexity by using "progressive disclosure". Progressive disclosure involves careful organisation of information so that it is shown only at the appropriate time. By "hiding" information presented to the user, the designer reduces the amount of information to process. For instance, a menu displays its choices when clicked and some of these choices can use dialogue boxes to present various options effectively.

What we see influences how we feel and what we understand. Visual information communicates nonverbally but very powerfully. It can include hints that motivate, direct or distract the user. Effective visual design serves a greater purpose than decoration; it is an important tool for communication. How the interface designer organises information on the screen can make the difference between a design that communicates a message and the one that leaves a user feeling puzzled or overwhelmed. Good graphic designers provide a perspective on how to take the best advantage of the screen and how to use effectively the concepts of shape, colour, contrast, focus, and composition. Moreover, they understand how to design and organise information and the effects of fonts and colour on perception.

4.2 STANDARD GUI OF APPLICATIONS FOR WINDOWS

Figure 4.1 presents the standard graphical user interface (GUI) of common applications designed to run on *Windows* (i.e., Microsoft Windows 95 / 98 / NT 4.0 as discussed and explained in Chapter 1). The detailed descriptions of different GUI components can be found in [Micc95].

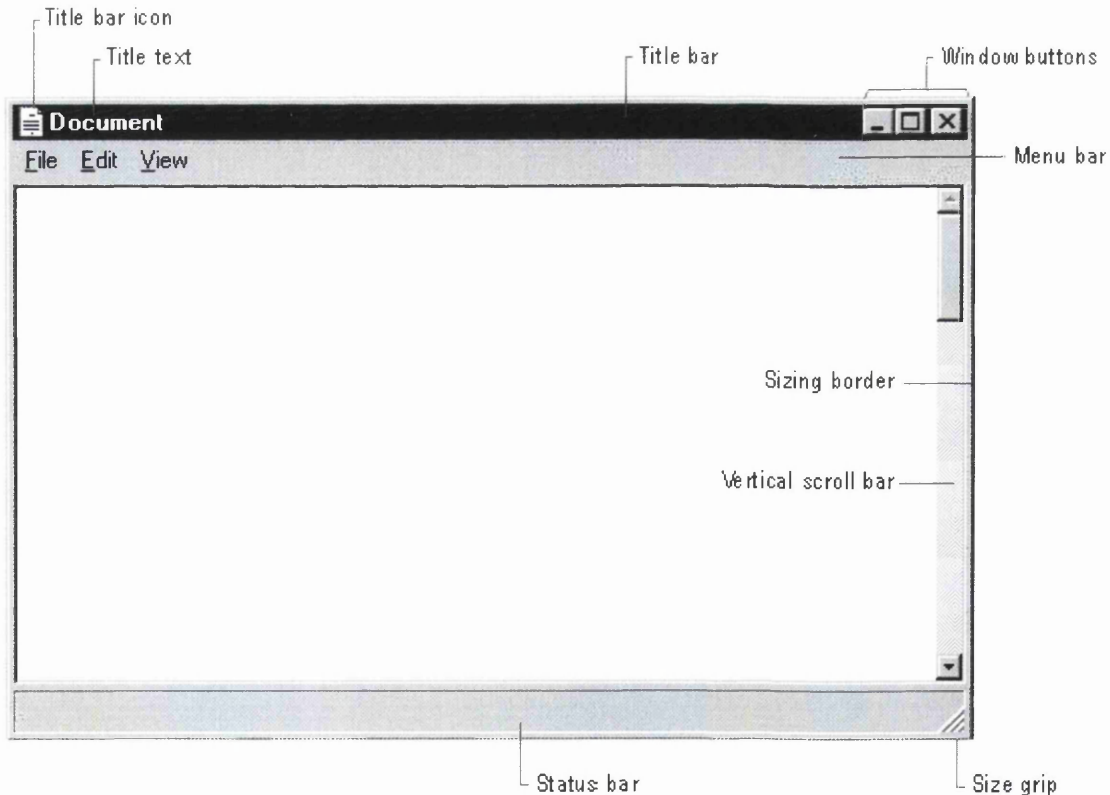


Figure 4.1 Typical GUI of *Windows* Applications

Recently, the design of this interface supports a model where users can browse for data and edit data directly instead of having to first locate an appropriate editor or

application. As users interact with data, the corresponding commands and tools to manipulate the data or the view of the data become available automatically. This frees the user to focus on the information and tasks rather than on applications and how applications interact.

In this context, a “document” is a common unit of data used in tasks and exchanged between users. The term document is a little misleading because it seems to imply the output of a word-processing or spreadsheet application. A document is just a place to keep data, and a view is a place to represent that data. The use of the term, document emphasises the fact that the focus of design is on data, rather than the underlying application.

4.3 SEPARATION OF DOCUMENT AND VIEW

Almost every software development effort has had to deal with data management in some form or other. Finding out ways of managing an application’s data has been a problem for software developers throughout the history of the IT industry. After all, information and data management is one of the primary uses of computer technology.

There are many issues involved in separating data from the user interface display code, such as (1) deciding which part of the application owns the data, (2) resolving which part of the application is responsible for updating the data, (3) determining how to display multiple renderings of the data, (4) coordinating data updates, (5) storing the data, and (6) managing the user interface. The last item can be difficult especially when multiple document types are involved because they often require updating the toolbars and menus.

The application programmers from the C/SDK background [Petz92] understand how difficult it can be to manage data because of the structure of an SDK program. There is no easy way to decide where to put the data management code.

The basic idea behind any data management scheme is to split the task into two conceptual parts: (1) data management and (2) user interface management. In this context, MFC is very successful in separating the code that manages an application’s data from the code that renders that data. This specific feature – the separation of an application’s data management code from its user interface code is implemented in the document / view architecture in MFC.

MFC's document/view is not a new idea. It was first created by the computer scientists at Xerox PARC and was a key part of the Smalltalk environment [Kras83]. Smalltalk's version of the document/view is called model-view-controller (MVC) where the model is equivalent to the document in MFC.

The MVC paradigm is a way of breaking an application, or even just a piece of an application's interface into three parts: the model, the view, and the controller. MVC was originally developed to map the traditional input, processing, output roles into the GUI realm:

Input \Rightarrow Processing \Rightarrow Output

Controller \Rightarrow Model \Rightarrow View

The user input, the modelling of the external world, and the visual feedback to the user are separated and handled by the model, viewport and controller objects. The controller interprets the mouse and keyboard inputs from the user and maps these user actions into commands that are sent to the model and / or viewport to effect the appropriate change. The model manages one or more data elements, responds to queries about its state, and responds to instructions to change the state. The viewport manages a rectangular area of the display and is responsible for presenting data to the user through a combination of graphics and text.

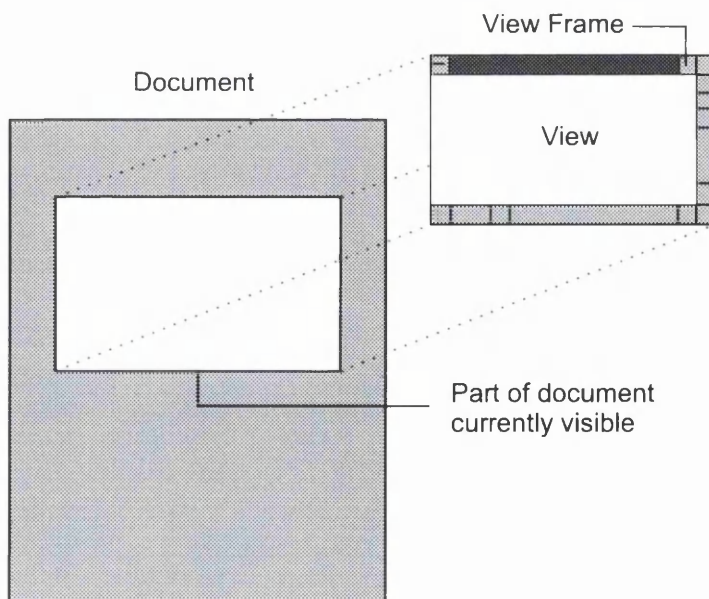


Figure 4.2 The Document / View Architecture in MFC

4.4 THE DOCUMENT/VIEW ARCHITECTURE IN MFC

The MFC document/view architecture provides a consistent way of coordinating application data and representations of that data. In MFC, the document handles the data management and an application's views handle the user interface management (Figure 4.2). In effect, an application's data is centralised in one place and the user interface code is packaged separately.

There are four main components of the architecture: (1) documents, (2) views, (3) view frames and (4) document templates. These components are briefly discussed in the following sections.

4.4.1 Documents

In MFC, the document is embodied by the `CDocument` class. In fact, when AppWizard generates an application, it derives a class from `CDocument`. `CDocument` has various functions for performing such operations as managing file I/O and updating the representation of the data.

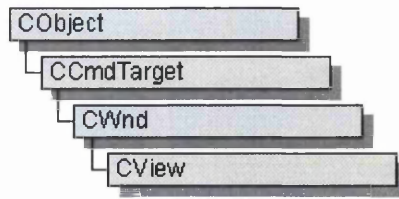
MFC's `CDocument` class is two layers deep in the MFC hierarchy. It is derived from `CCmdTarget`, which is derived from `CObject`. As a result of being derived from `CObject`, `CDocument` is eligible for all the support offered by the `CObject` class as explained in Chapter 3. In addition, by virtue of being derived from `CCmdTarget`, the `CDocument` class can handle command messages (i.e., `WM_COMMAND` messages). Finally, `CDocument` can support component object model (COM) interfaces, as well as OLE automation, because it is derived from `CCmdTarget`.

The document can receive programmatic commands, and at the same time allow the user interface to directly bind to its functionality. In other words, the programmer can implement command handlers in the document itself. Because of the command routing, the commands find their way to the document from the user interface elements, such as toolbars and menus, which actually belong to totally different objects. This fact can be abused to make dangerous access to the document, but if used correctly, it can be a valuable technique [ShWi96].

4.4.2 Views

An application with only a document would be fairly uninteresting. There has to be some way of rendering the data on the screen as well as providing a user interface for manipulating the data. This functionality is supplied by MFC's `CView` class. In

an MFC program using the document / view architecture, views are responsible for rendering a document's data. The `CView` class is three layers deep in the MFC hierarchy:



Because `CView` is derived from `CCmdTarget`, it is eligible to receive command messages from menus and controls which is a good feature since the view is often responsible for handling the user interface as well. Since `CView` is derived from `CWnd`, it is also eligible for receiving the *Windows* messages, such as `WM_PAINT`, `WM_SIZE`, etc.

Standard MFC applications using the document / view architecture do their drawing within a view. `CView` includes a function called `OnDraw()`, which is called by the framework whenever a view needs to be updated.

4.4.3 View Frames

MFC views are simply borderless windows that supply data renderings. However, the drawings appear within a window that has a border and menus. This window is called a frame window and it is necessary for the architecture to work properly.

Now the question is why the menu management is handled by a separate frame and not by the view itself. It is a good design practice to isolate this type of functionality and not tightly couple it. That way, the dependencies are reduced, which makes it much easier to work on a single piece of code at a time. This technique is also the cornerstone for isolating the differences between the single document interface (SDI), the multiple document interface (MDI), and the OLE in-place editing. Separating the view and the frame makes the `CView` objects more flexible, so that they can be used in different situations.

SDI applications use a class derived from `CFrameWnd` as the frame housing the view. MDI applications derive a class from `CMDIChildWnd` as the frame housing the document's views.

4.4.4 Document Templates

One interesting aspect of the document / view architecture is that the three previous

components are treated as a unit. In other words, the ideas of a document, its representation, and its user interface are treated together as a whole. Document templates tie the whole picture together:

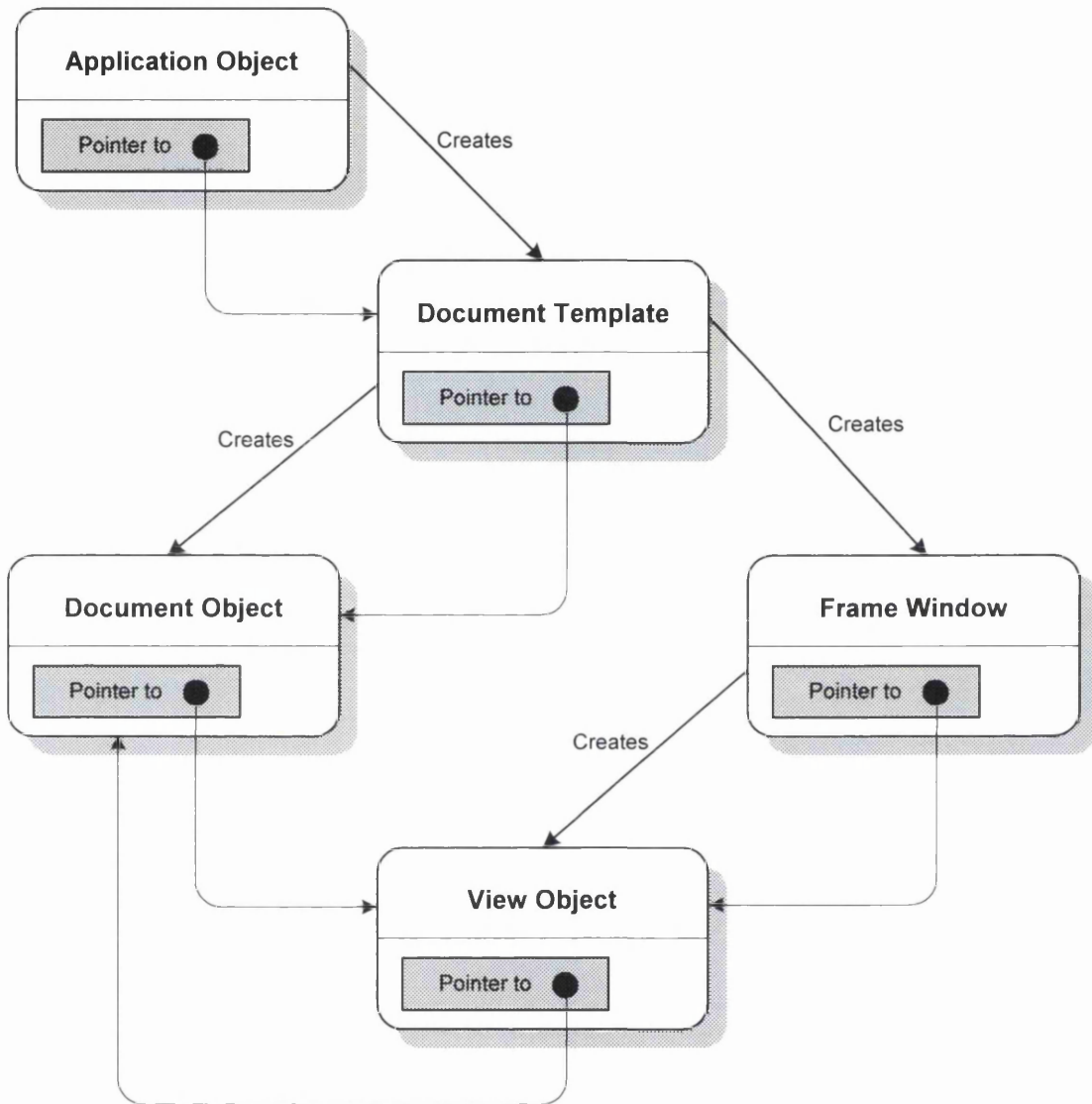


Figure 4.3 The Interrelationships between Objects in the Architecture

The controller in the Smalltalk MVC architecture acts like a shield between the model and the view so that they do not get too dependent on each other. The document template has a similar role in MFC. An MFC application has one document template for each type of document that it supports. For example, if the application supports both a spreadsheet and a database, the application will have two document template objects. Each document template is responsible for creating and managing all the documents of its type.

`CDocTemplate` is an abstract base class that defines the base functionality for

handling documents, frames and views. `GetFirstDocPosition()`, `GetNextDocPosition()` and `OpenDocumentFile()` are the pure virtual functions that make `CDocTemplate` an abstract base class. MFC defines two other document template classes that can be used directly within applications. They are `CSingleDocTemplate` and `CMultiDocTemplate`. `CMultiDocTemplate` is for applications using more than one type of document. `CMultiDocTemplate` is very similar to `CSingleDocTemplate`. `CMultiDocTemplate` holds a list of document types, whereas `CSingleDocTemplate` holds only one document type.

4.5 MULTIPLE DOCUMENT INTERFACE (MDI)

For some applications, the *Windows* taskbar may not be sufficient for managing a set of related windows. For example, it can be more effective to present multiple views of the same data or multiple views of related data in windows that share the common interface elements. The multiple document interface (MDI) is used for this kind of situation.

The MDI technique uses a single primary window, called the parent window to visually contain a set of related child windows. Each child window is essentially a primary window, but is constrained to appear only within the parent window instead of on the desktop as shown in Figure 4.4.

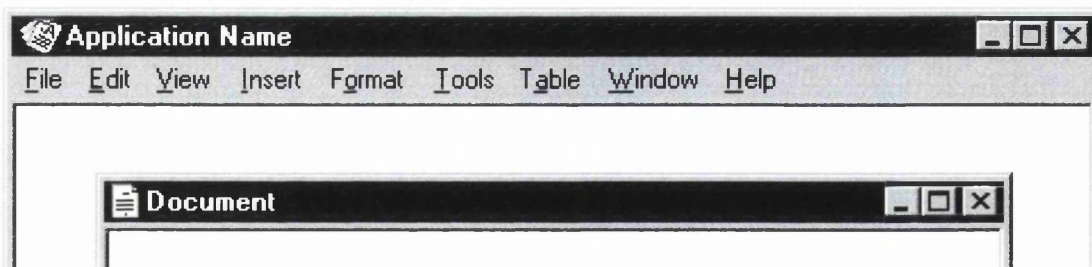


Figure 4.4 A Child Window Constrained within its Parent

The parent window also provides a visual and operational framework for its child windows. For example, child windows typically share the menu bar of the parent window and can also share other parts of the parent's interface, such as a toolbar or a status bar. These interface elements can be changed to reflect the commands and attributes of the active child window.

Secondary windows such as dialog boxes, message boxes, or property sheets – displayed as a result of interaction within the MDI parent or child are typically not contained or clipped by the parent window. These windows are activated and

displayed following the common conventions [MSDN41] for secondary windows associated with a primary window, even if they apply to individual child windows.

4.6 THE CONTRIBUTIONS OF MFC IN GGS

The software developers are moving more and more towards an industry of frameworks. This is quite a paradigm shift from the old idea of building an entire program on top of an OS [Walk97]. Nowadays, we purchase a framework like MFC or Borland OWL [Newa98] or Java's class library AWT [ChLe97]. The advantage is obvious; we can have an entire *Windows* application up and running in minutes. The developer can concentrate on adding components and implementing the special features of his / her application. S/he does not need to waste time finding answers to the common questions, such as, "Where and how do I place my first window?". If someone tries to write a *Windows* application from scratch, the result could be a whole lot less attractive and it would certainly take more time.

The skeleton MFC applications created by AppWizard, provide a good starting point to develop an application that conforms to the published user interface guidelines [Micr97]. Not only the application developer but also the end-user benefits from a familiar and consistent user interface because s/he is not forced to re-learn common operations. A user who regularly prints documents from Microsoft Word intuitively looks for a Print option on the File menu when confronted with the task of printing in an unfamiliar application.

Any solution to the problems encountered when attempting to develop and maintain a software system written using object-oriented languages has to fulfil two important requirements [LeMe*92]. Firstly, it has to be able to describe the structure of the system being supported, and secondly, it has to provide an efficient and user-friendly interface.

To address the first of these requirements, the author decided to use the MFC serialization as opposed to other techniques for object persistence. This has been discussed in detail in Chapters 2 and 3. The second requirement is satisfied by incorporating the MFC document / view architecture in GGS to provide the "look and feel" expected by experienced users of the *Windows* environment.

It is unnecessary to reiterate the descriptions of the WIMP (i.e., windows, icons, menus and pointers) interface for *Windows*. However, it is still important to discuss briefly the ways GGS provides feedback to the user.

4.6.1 ToolTips

GGs uses ToolTips to help the user to identify the icons in the toolbar. A Tooltip is a small pop-up window that displays a single line of text that describes the purpose of a toolbar button in an application. A Tooltip control is hidden most of the time, appearing only when the user puts the cursor on a toolbar button and leaves it there for approximately one-half second [MSDN42]. The Tooltip control appears near the cursor and disappears when the user clicks a mouse button or moves the cursor off the toolbar button.

4.6.2 Message Boxes

Message boxes are used in GGS with caution. It only takes one line of code to display a message box on the screen:

```
AfxMessageBox("Simple message box.") ;
```

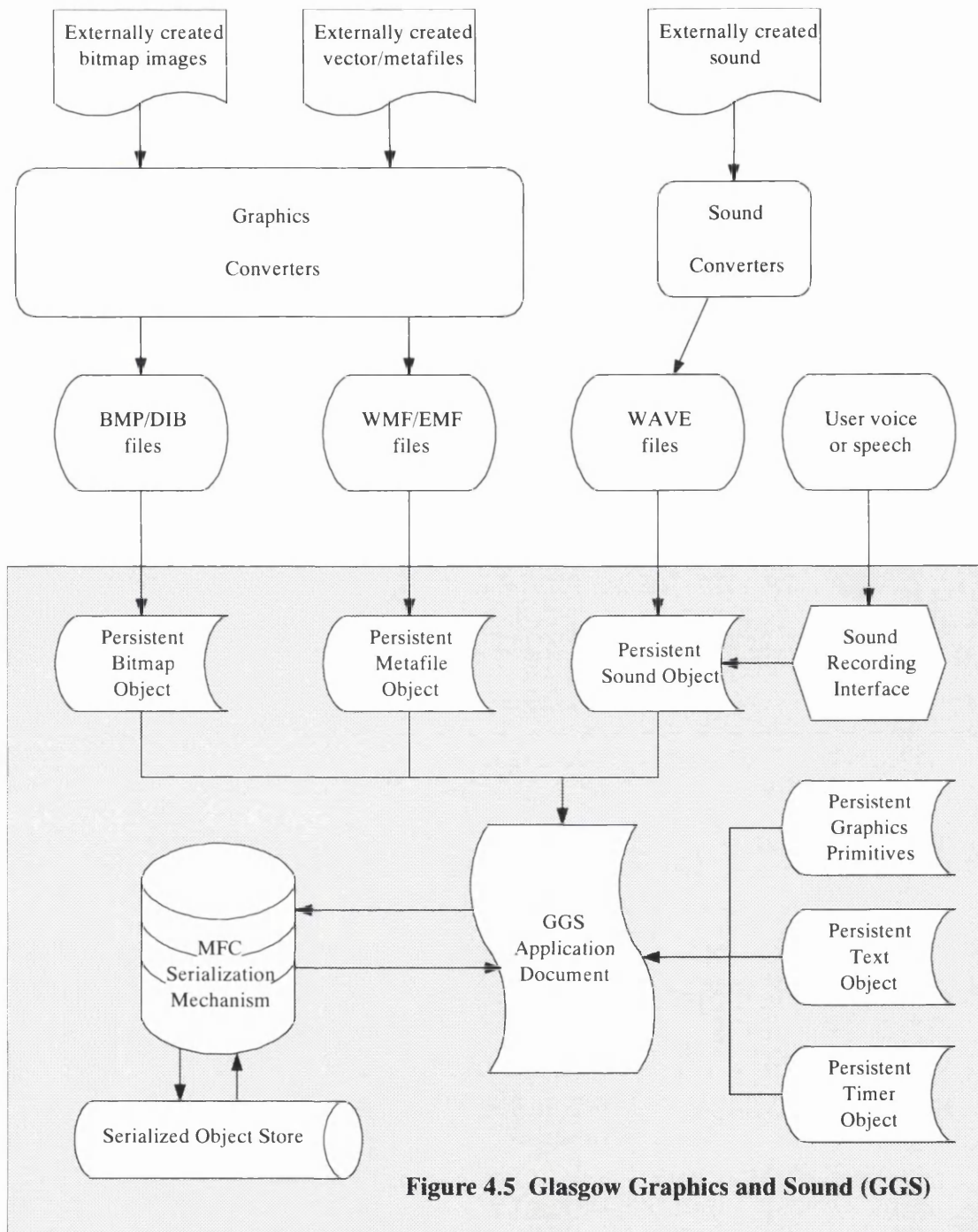
However, the message boxes can distract and annoy the user if frequently used.

4.6.3 Status Bar Messages

The status bar in a *Windows* application is a convenient place to display messages without interrupting the user action. The ToolTips are usually accompanied by the flyby texts. A flyby text is a message that is displayed on the status bar, most commonly in the first "stretchy" pane. The help text provided in a flyby text can be a bit longer than the ToolTips because there is more space to display the text. In addition to displaying help texts about the menu and toolbar buttons, GGS displays descriptive messages on the status bar about the context of any activity in a view window. Status bar messages can be helpful and they do not have the adverse effects of message boxes.

4.7 HIGH LEVEL DECOMPOSITION OF GGS

As discussed in Chapter 1, one of the major objectives in developing GGS is to improve technical communications between engineers separated by distance and language barriers. In order to achieve such an objective, this prototype multimedia application should be able to deal with externally created vector or bitmap images, graphics primitives and sound objects in any sequence. The end-user should be able to open an external image, record his/her voice wherever appropriate, draw / scribble on the images and save them in one single file and animate them later, if necessary. Now with the MFC document / view architecture and serialization in the background, it is possible to present a high level decomposition of GGS.



It is worth noting from Figure 4.5 that GGS only supports the “core” file formats for sound, bitmaps and metafiles. They are core file formats because the OS (i.e., *Windows*) heavily relies on them and internally uses them most of the time. However, there are several other popular graphics and sound file formats and commercial applications tend to support as many as possible. But writing file converters is not a primary issue in developing a research prototype such as GGS. Source codes of several converters are available in the public domain [INET41] and they can be added on later to expand the scope of GGS. This also raises an important

issue that there should be plenty of room for expansion.

4.8 A CASE STUDY WALKTHROUGH

Before discussing the software engineering issues involved in the development of GGS, it is important to introduce the GGS interface and create a short and simple illustration as an end-user. Figure 4.6 presents the GGS interface where the users can select commands from the drop-down menus or click the buttons on the toolbar that represent the frequently used commands.

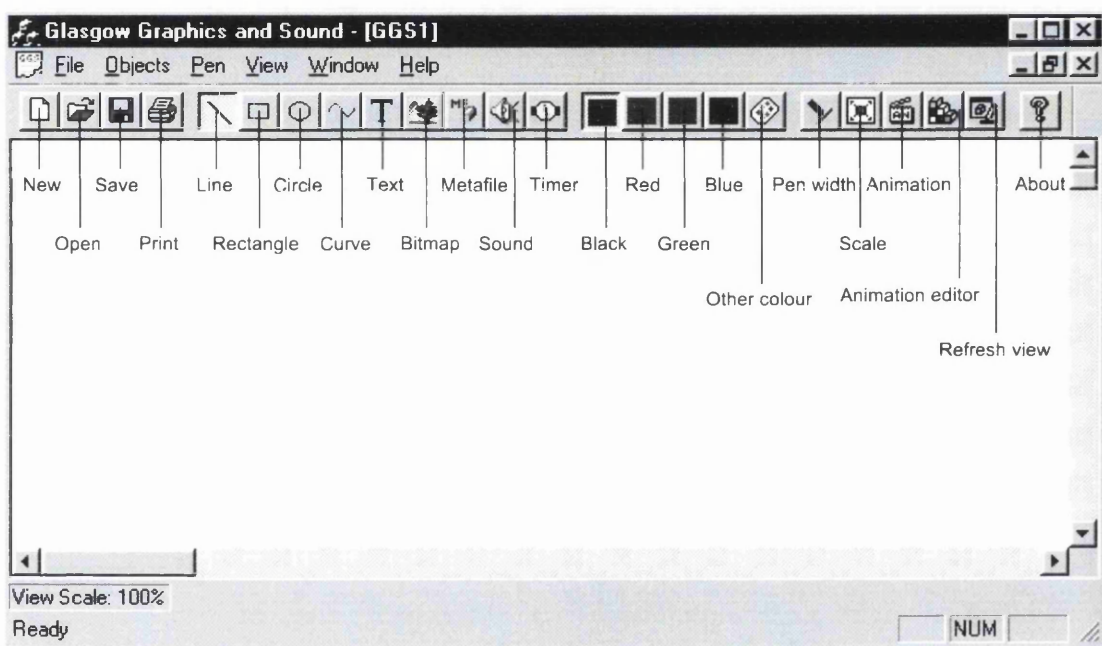


Figure 4.6 The User Interface of GGS

The buttons on a toolbar are analogous to the items in a menu. Both types of user interface objects generate commands, which a *Windows* program handles by providing handler functions. Often toolbar buttons duplicate the functionality of menu commands, providing an alternative user interface to the same functionality.

In addition to the toolbar buttons, the user can get to the context sensitive menus by clicking the right mouse button:

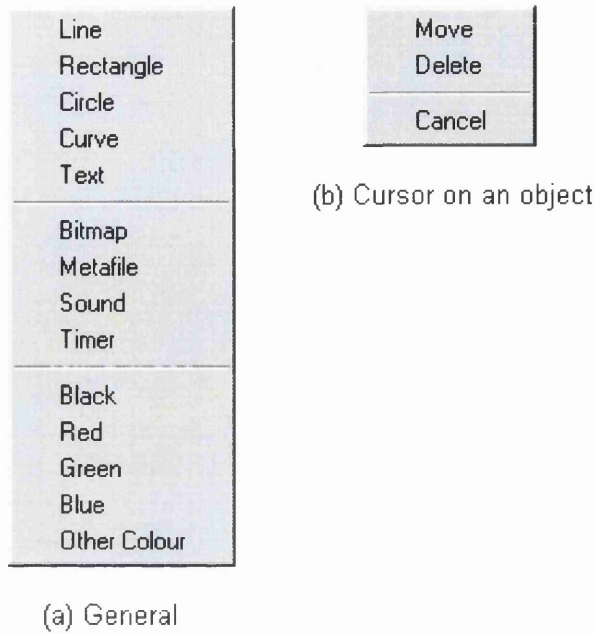


Figure 4.7 The Context Sensitive Menus

4.8.1 A Protein Molecule

Three-dimensional structures of proteins, the relationships between them and their sequence are by no means simple. In the computer generated pictures, different colours are used to highlight the important features found in proteins. Perceptual response of the eye to visual information is very important in this context. The depth perception in these images has made it possible to create models of proteins that stimulate the imagination of the end-user. Figure 4.8 presents the atomic details of a portion of Thermolysin (3TLN) protein [Belh90]:

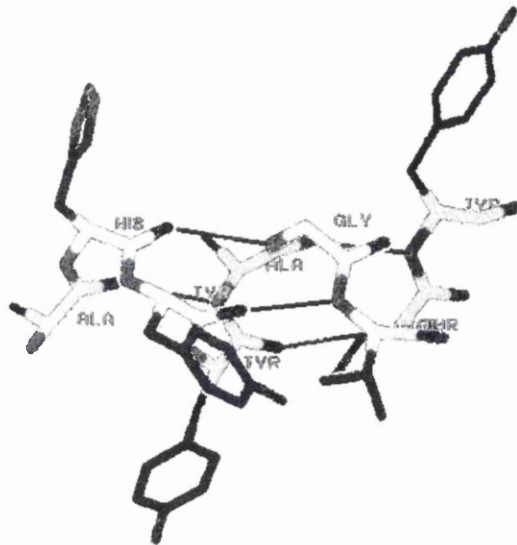



Figure 4.8 The Atomic Details of a Portion of Thermolysin (3TLN) Protein

We could use GGS to create a simple presentation to explain different colours used in Figure 4.8. Unfortunately, greyscale hardcopies are not good in representing colours. However, the accompanying disc with this thesis contains the presentation and an executable copy of GGS.

A definite theme or story is very important for any presentation. For the protein molecule, we can record some sound clips, use one line of text as the caption, create a rectangle to frame the picture and draw some arrows and circles to highlight the atomic presentation of the molecule. Some timers are also required to separate the objects and adjust the pace of the animation. The sequence of objects can be edited in the GGS Animation Editor.

4.8.2 Importing the Picture

First of all, the picture of the Thermolysin (3TLN) protein molecule should be imported into GGS. We press the  button and click in the active view window where we want to place the picture. A standard File Open dialogue box pops up and we navigate through the folders to find the BMP file for the molecule:

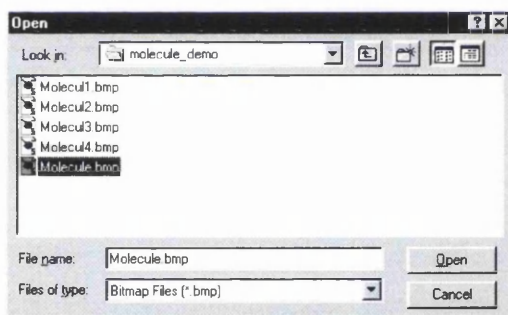


Figure 4.9 The Picture Comes from a BMP File

After the file selection, GGS reads it and provides us with the following information:

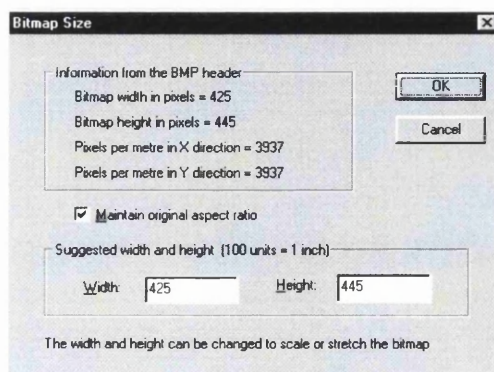



Figure 4.10 Bitmap Size Related Options

Although we have an opportunity to scale or stretch the bitmap, in this case, we accept the width and height values suggested by GGS. The bitmap is then placed with its top-left corner at the point where we clicked before with the left mouse button. If this is not convenient, we can always move the bitmap by clicking on it with the right mouse button and then selecting the Move option.

4.8.3 The Rectangle Around the Picture

A rectangle can be used to frame the image of the protein molecule. To select an appropriate colour, we click the  button on the toolbar which presents the following dialogue box:

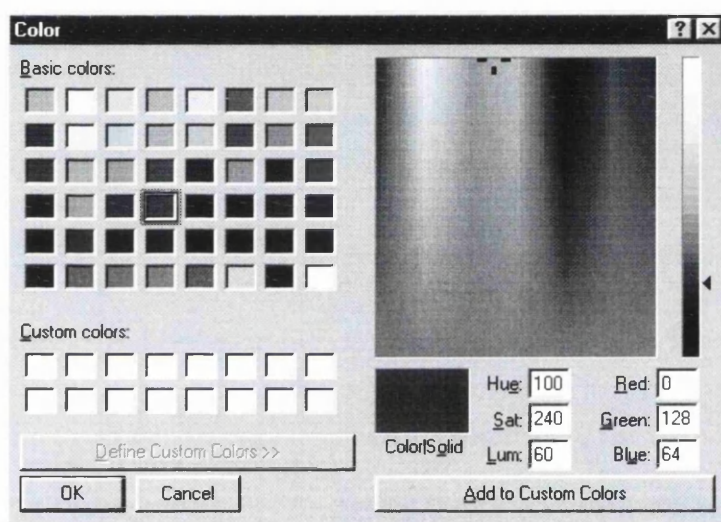



Figure 4.11 Selection of Colours

After choosing the colour as shown in Figure 4.11, the next step is to select the pen thickness. We click the  button on the toolbar and select 0.03 inches as our choice:

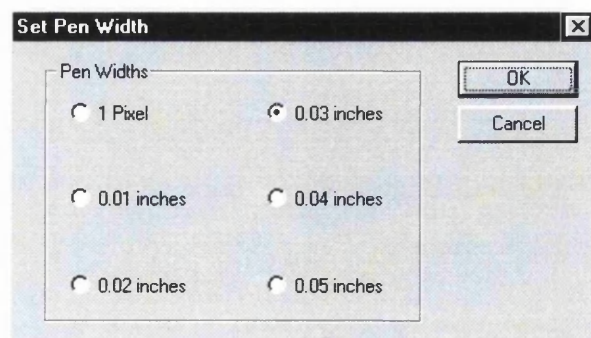



Figure 4.12 Pen Width Selection

Finally, we click the  button and draw a rectangle around the picture of the

molecule in the active view window by clicking and dragging the mouse. The rectangle can be moved or deleted like any other object in GGS.

4.8.4 Recording Sound Objects

The sound object creation is very similar. We select the  button on the toolbar and click anywhere in the active view window and the following dialogue box pops up:

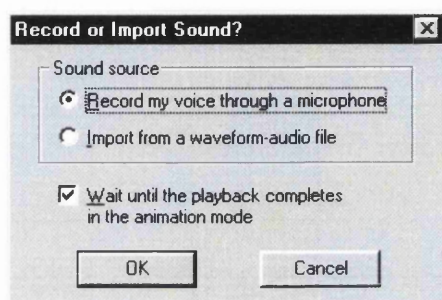



Figure 4.13 Sound Recording Options

We accept the default options and click the  button. GGS presents another dialogue box:

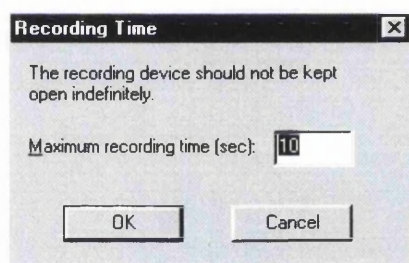




Figure 4.14 Recording Time

We select 10 seconds as our maximum recording time, click the  button, and then record, “*This picture comes from an earlier work done in the Department of Computing Science*”, when the following dialogue box appears, and finally, click the  button when we finish:

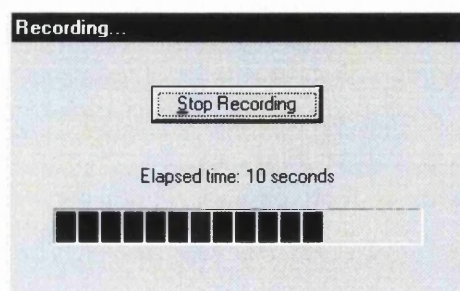


Figure 4.15 Progress Dialogue

The same steps are followed to record the other sound clips as shown in Table 4.1:

Table 4.1 Other Sound Clips

Item No.	Sound Clips
2	This is a portion of an Alpha Helix in Thermolysin protein.
3	Green segments represent nitrogen atoms.
4	The blue ones are oxygen.
5	The red shaded ones represent carbon atoms.
6	Three letter codes display different Amino acids.
7	This presentation is an example of creating a sequence of objects in GGS.

4.8.5 A Text Object

A text object is necessary for the caption. Unlike others, a text object is associated with several font characteristics:

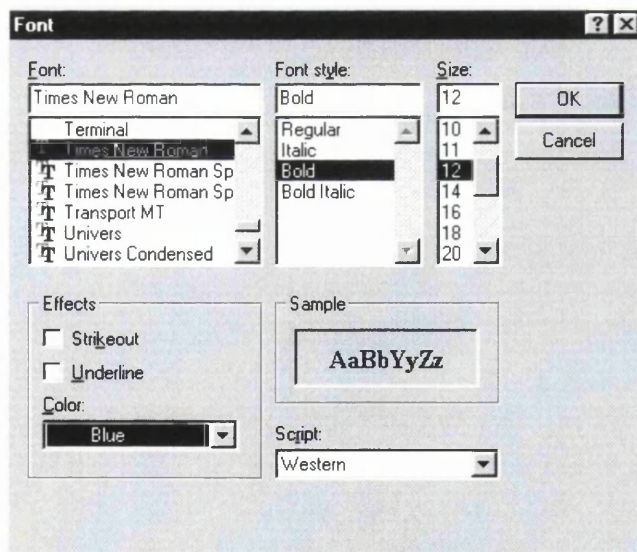


Figure 4.16 Font Properties

In order to create the caption, we press the **T** button and then left click on the view window where we want to place the text. GGS presents the following dialogue box and we type, "A Portion of an Alpha Helix in Thermolysin Protein":

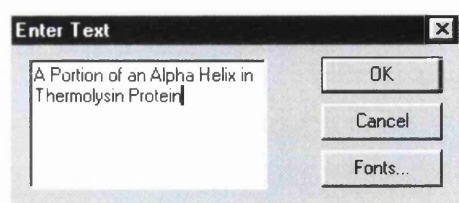



Figure 4.17 The Caption

Now, if we click the **Fonts...** button, the dialogue box in Figure 4.16 will appear and we can select a combination of different font features. If we do not go that far, GGS will render the text with the default font properties.

4.8.6 Creating Timers

Seven timer objects are necessary for this presentation. To create a timer object, we select the  button and click anywhere in the active view window. GGS presents the following dialogue and we type 2000 in the edit box:

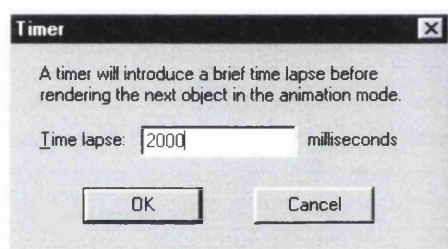


Figure 4.18 Time Lapse of 2 Seconds

Timers and sound objects in GGS do not have any top-left corner, but the user is still expected to click the left mouse button in the active view window. This way multiple sound or timer objects can be created in one sequence. In addition, when multiple GGS documents are open, it is also necessary to indicate where the new object will be placed.

We create another six timers with the time lapse of 2 seconds. Timers can be fine-tuned in the Animation Editor. It is easier to adjust the time lapse after watching the animation.

4.8.7 Creating Arrows

Some arrows would be ideal to highlight specific areas on the protein molecule. An arrowhead (e.g., \blacktriangleright) can be drawn with the freehand curve and the rest with a line. We select the Line and the Curve buttons on the toolbar to accomplish this task. The pen colour and its width are selected as described in Section 4.8.3. We also

encircle a three-letter code with the Circle tool. The final arrangement should look like the following:

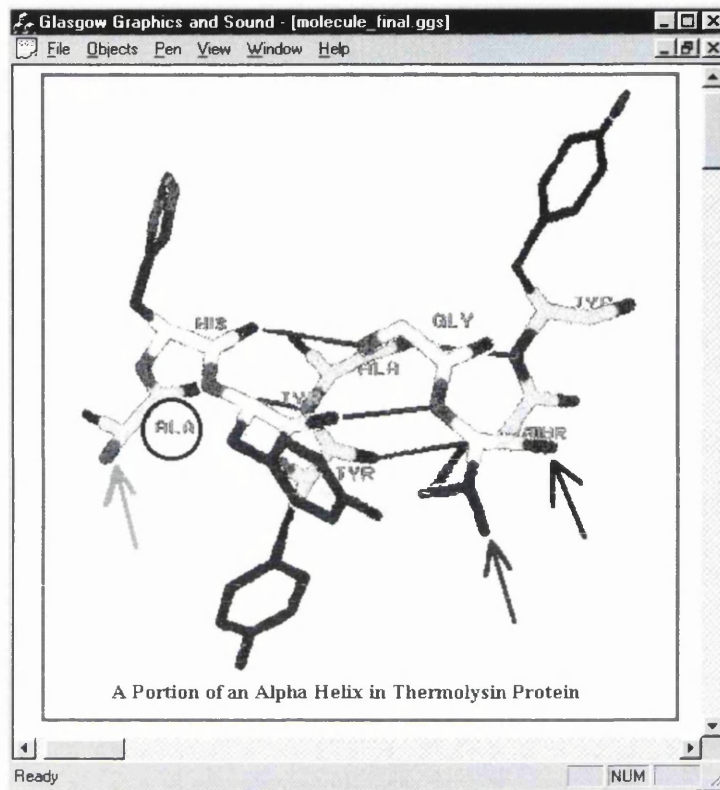



Figure 4.19 Explaining Thermolysin Protein

4.8.8 Arranging the Sequence of Objects in the Animation Editor

Now it is necessary to arrange the sequence of objects in the Animation Editor. We invoke the Editor by clicking the  button on the toolbar:

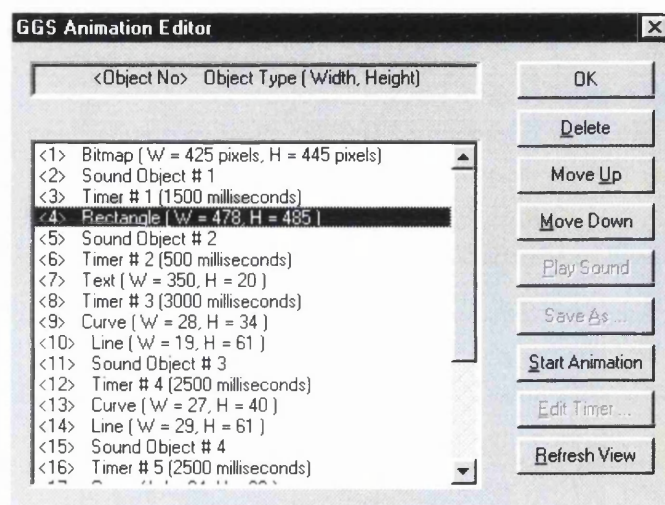



Figure 4.20 The Animation Editor

We arrange the objects with the **Move Up** and **Move Down** buttons. The graphic objects are highlighted in the view window when they are selected in the Editor. The sound objects can be checked with **Play Sound** and the timers can be altered with the **Edit Times ...** button. The animation can be tested incrementally by clicking **Start Animation** which switches to **Stop Animation** and awaits the user intervention. The selection of colours, pen widths, timers, etc. are very subjective. However, the author's final selection of the objects and their sequence can be found in the accompanying disk.

4.8.9 Animation

It is possible to animate the sequence of objects without invoking the Animation Editor. If we click the  button, GGS offers the following alternatives:

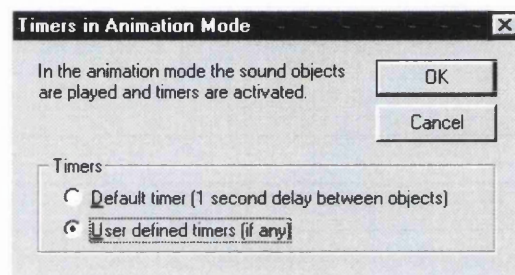


Figure 4.21 Timers in Animation

If the **OK** button is pressed, GGS animates the whole sequence of objects from the start to finish.

4.9 CLASS HIERARCHY IN GGS

Now it is time to come back to the software development issues. It is clear from the case study walkthrough that a potential user of GGS should be able to, (a) import externally created images in BMP, WMF and EMF formats, (b) import sound as WAVE files, (c) record his / her own speech, (d) use graphics primitives such as lines, circles, rectangles, etc., (e) scribble with freehand curves, (f) add text wherever necessary, (g) move objects anywhere on the screen, and (h) control the static animation sequence by inserting timers.

It is not known in advance what sequence of object types the user will create. GGS must be able to handle any sequence of objects. This suggests that using a base class pointer for selecting a particular member function might simplify the design. For example, there is no need to know what an object is, in order to draw it. As long as the object is accessed through a base class pointer, the object can draw itself by using a virtual function. To achieve this, we need to make sure that the classes defining

similar characteristics, share a common base class. In this base class, the functions to be selected automatically at run time should be declared as virtual.

The MFC library would impose some restrictions on the design of this class hierarchy. For example, serializable classes in GGS should define `CObject` as their base class. Sound and timer objects are quite different in nature from others. Hence, one class for sound objects and another for timers can be derived straight from `CObject`. Similarly, `CExternalImage`, derived from `CObject`, can deal with the import issues of externally created bitmaps and metafiles. On the other hand, lines, circles, curves and even text objects have similar characteristics and they can share a common base class. In essence, the features of GGS can be divided in four main domains as shown in Figure 4.22. The MFC classes have names beginning with the letter "C", such as `CDocument` or `CView`. Data members of an MFC class are prefixed with "m_". This convention is followed in developing GGS because it will be easier to identify and understand the derived classes.

4.10 CLASS CONSTRUCTION

The proposed classes in Figure 4.22, in a way, extend the MFC library. In view of this, it is important to understand the philosophy and development strategies used by the MFC developers when adding new classes to the library. This also raises some controversial class design issues.

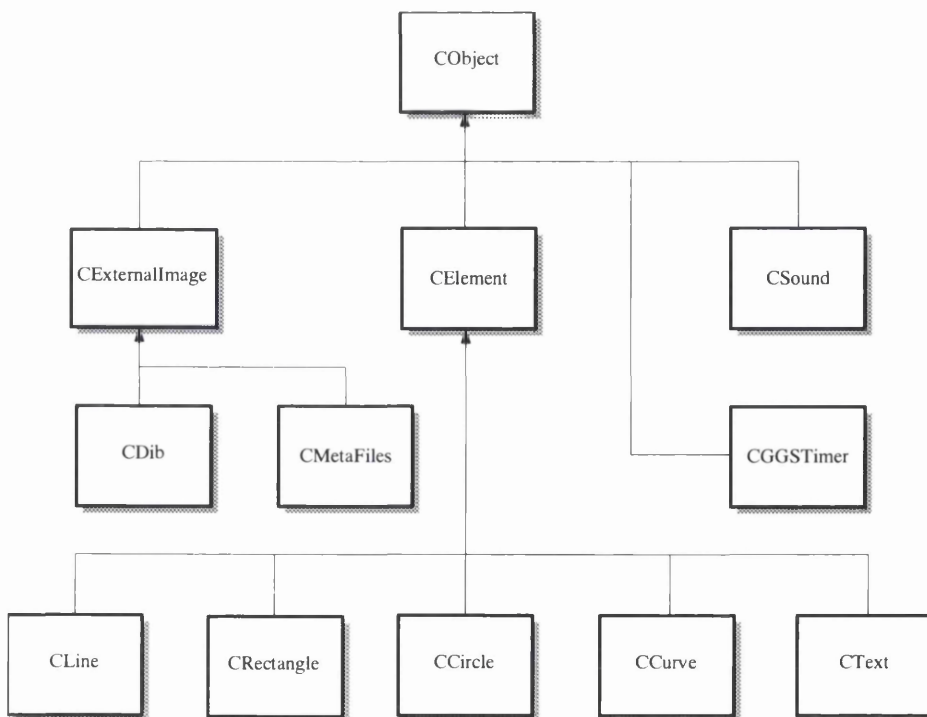


Figure 4.22 Initial Class Hierarchy

4.10.1 Public Data Member?

The MFC library does not follow an important principle that all data members should be private or protected and accessor functions should be used to change the values in those data members. MFC frequently offers both a public data member and an accessor function for use by the developer. For example, the class `CWnd` provides both a public data member, `CWnd::m_hWnd`, and an accessor function, `CWnd::GetSafeHwnd()`. Depending on the needs of a particular application, a developer might use the data member or the member function, or both.

More often, MFC uses accessor functions in significant operations such as setting and retrieving styles, returning pointers, and obtaining values used in other operations. The MFC developers encourage application programmers who are writing MFC-friendly classes to use public data members as often as possible [Mirc97], and reserve accessor functions for operations that do more than simply change values in a data member, such as incrementing a counter or updating another function.

In other words, the MFC developers encourage delegating the security related responsibilities to the class user. However, the author supports a different opinion that classes should be easy to use but difficult to abuse [Meye98]. The classes in GGS are, therefore, not designed to offer any public data member.

To some extent, it is agreeable that accessor functions or `Set()` / `Get()` pairs indirectly expose the data members and they are not necessary unless they add value or do some additional work. However, this may not be the most important reason for the MFC architects to use and encourage public data members. The *Windows* environment was designed long before the C++ language became popular. Thousands of applications still use the procedural C-language API. Procedural programmers are more familiar with the task of assigning values directly to variables and structures. This might be another reason behind the existence of public data members in MFC. From a commercial viewpoint, the MFC designers and developers have to think about a vast number of *Windows* programmers with the procedural background.

4.10.2 Thin API Wrapper

The core of the MFC library encapsulates a large portion of the *Windows* API in C++ form. The classes represent windows, dialog boxes, device contexts, common GDI objects and other standard *Windows* items. These classes provide a convenient C++ member function interface to the structures in *Windows* that they encapsulate. The

MFC library also supplies a layer of additional application functionality built on the C++ encapsulation of the Windows API. This layer is a working application framework for *Windows* that provides most of the common user interface expected of programs for *Windows*.

One important characteristic that sets MFC apart from other class libraries for *Windows* is the close mapping to the *Windows* API written in the C language. An MFC user can generally mix calls to the class library freely with direct calls to the *Windows* API. Part of MFC's success lies in the fact that the framework is a very thin layer over the *Windows* API. Most of the time, MFC "wraps" the *Windows* functionality in C++ without adding other functionality. In some cases (for example, dialogue boxes), MFC simplifies the *Windows* API by managing the details, but the general rule is to keep the layer as thin as possible.

4.10.3 Copy Constructor and Assignment Operator

It is generally regarded as a good practice to explicitly define copy constructors and assignment operators. If not defined by the programmer, the C++ compilers generate them as memberwise initialisers and memberwise assignment operators. However, implicitly declared copy constructors and assignment operators, most of the time, do not perform as intended when pointer data members are involved. Even when explicitly declared, they can be a source of problems [Mey96].

CObject derived classes in MFC avoid copy constructors and assignment operators altogether. The copy constructor and the operator = are private in CObject with no implementation. This, in fact, forces the C++ compilers to generate an error if one CObject is assigned to another via operator =, or any attempt is taken to create one CObject from another using the copy constructor. In addition, CObject derived classes do not get the compiler generated default copy constructors and assignment operators since their base class has declared them private.

In many MFC classes, provision is made for copying in such a way to protect the integrity of data. For example, in the template collection classes, it is not possible to do the following:

```
CArray myarray1 ;
CArray myarray2 ;
// Insertion of data ...
myarray1 = myarray2 ; // Error
```

The programmer can, however, do this:

```
CArray myarray1 ;
CArray myarray2 ;
// Insertion of data ...
myarray2.RemoveAll() ;
myarray2.Append(myarray1) ;
```

4.10.4 Other Guidelines

Using `const` is a way to protect the accessibility of data. In general, programmers could use `const` more often than they do, and in MFC `const` is used whenever feasible to make the data less vulnerable. The MFC developers encourage programmers to apply `const` to data members as often as possible as a guarantee to the class users that the accessor functions do not have hidden side effects.

On the other hand, the use of multiple inheritance is discouraged because it adds a high degree of complexity to application development. The MFC library does not use multiple inheritance in the design or implementation of any of its classes. However, the use of templates is encouraged although the level of complexity inherent to templates is quite high.

4.10.5 Commenting Convention

In both the MFC source files and the files that AppWizard creates, the following comments are found within the class declarations, usually in this order:

- `// Constructors`: Obviously, the section headed by this comment will house the class constructor declarations, but other functions that are used in the initialisation of class members may also appear here.
- `// Attributes`: This comment indicates that the statements following it define properties of objects of the class - typically these will be the data members of the class, but they can also be `Get()` / `Set()` types of functions that supply information about the class or just change the data member values.
- `// Operations`: The function members following this comment act on the data members of the class, so they change the attributes of a class object in some way.
- `// Overridables`: This defines a section of the class which declares function members that can be overridden in a derived class. Pure virtual

functions may also appear in this section.

- `// Implementation`: This indicates that everything following it is not guaranteed to be the same in the next release of the library. Anything can be included in here - data members as well as function members.

MFC recommends these commenting conventions to delineate the sections of the class declarations containing similar types of class members. Some classes may omit some sections but all classes should have at least the `//Implementation` section [Micr97].

4.11 CLOSURE

In this chapter, we have discussed some principles of graphical user interface design, the MFC document/view architecture and general recommendations from the MFC developers regarding class construction. A high-level decomposition of GGS is also presented based on the requirements set out in Chapter 1. The main classes and their hierarchy are proposed based on this high-level decomposition. A case study walkthrough is added, purely from an end-user's viewpoint, to illustrate a simple use of GGS.

C++ Class design is a complex issue and one may take years to learn how to design classes and templates well. What is valuable in a class is not the implementation code, but the insight of the developer. The next chapter will endeavour to improve the initial class hierarchy for GGS.

4.12 REFERENCES

- [ANSI97] ANSI/ISO C++ Committee: "International Standard for Information Systems – Programming Language C++", ISO/IEC IS 14882, November 1997.
- [Belh90] Belhadj-Mostefa, K.: "Molecular Graphics: Protein Visualisation", PhD Thesis, Department of Computing Science, University of Glasgow, May 1990.
- [ChLe97] Chan, P. and Lee, R.: "The Java Class Libraries : Java.Applet, Java.AWT, Java.Bans", Addison-Wesley, 2nd Edition, Vol. 2, October 1997.

- [How196] Howlett, V.: "Visual Interface Design for Windows : Effective User Interfaces for Windows 95, Windows NT, and Windows 3.1", John Wiley and Sons, April 1996.
- [INET41] Liverpool HP-UX Porting and Archive Centre, "Software Porting And Archive Centre for HP-UX", <http://hpux.csc.liv.ac.uk/hppd/hpux>.
- [Kras83] Krasner, G.: "Smalltalk-80: Bits of History, Words of Advice", Addison-Wesley, 1983.
- [LeMe*92] Lejter, M., Meyers, S. and Reiss, S.P.: "Support for Maintaining Object-Oriented Programs", Report No. CS-91-52, Department of Computer Science, Brown University, January 1992.
- [Meye96] Meyers, S.: "More Effective C++: 35 New Ways to Improve Your Programs and Designs", Addison-Wesley, 1996.
- [Meye98] Meyers, S.: "Effective C++: 50 Specific Ways to Improve Your Programs and Designs", 2nd Edition, Addison-Wesley, 1998.
- [Micr95] Microsoft Press: "The Windows Interface Guidelines for Software Design : An Application Design Guide", 2nd Edition, July 1995.
- [Micr97] Microsoft Corporation: "Microsoft Foundation Class Library Development Guidelines", March 1997.
- [MSDN41] Microsoft Developer Network Library: "Displaying Secondary Windows", Windows User Interface: Platform SDK, April 1999.
- [MSDN42] Microsoft Developer Network Library: "Using CToolTipCtrl", Visual C++ Programmer's Guide, April 1999.
- [Newa98] Neward, T.: "Advanced OWL 5.0 : Power Tools for OWL Programmers", Independent Publishers Group, January 1998.
- [Petz92] Petzold, C.: "Programming Windows 3.1", Microsoft Press, 3rd Edition, 1992
- [ShWi96] Shepherd, G. and Wingo, S.: "MFC Internals: Inside the Microsoft Foundation Class Architecture", Addison-Wesley Developers Press, 1996.
- [Walk97] Walker, C.: "Using Frameworks - For Beginners Only?", <http://www.kinetica.com/oootips/using-frameworks.html>.

CHAPTER 5

GRAPHICS PRIMITIVES IN GGS AND OTHER FEATURES

5.1 COLLECTIONS OF OBJECTS

The document class in the GGS application needs to be able to store an arbitrary collection of lines, rectangles, externally created images, sound and other objects in any sequence. An appropriate vehicle for handling this could be a doubly linked list. A doubly linked list collection is an ordered arrangement of data items, where each item has two pointers associated with it which point to the next and previous items in the list. This type of list collection can be searched in either direction because it has both backward and forward pointing links.

The MFC collection classes provide two approaches to implementing each type of collection. One approach is based on the use of class templates and provides type-safe handling of data in a collection. Type-safe handling means that the data passed to a function member of the collection class will be checked to ensure that its type can be processed by the function.

The other approach makes use of a range of collection classes (rather than templates), that perform no data checking. The MFC user has the responsibility to include additional code to make these collection classes type-safe. Clearly, the better option is to work with the template-based classes, since they would provide the best chance of avoiding errors in the application.

The Standard Template Library (STL) is now part of the C++ language [ANSI97]. If we consider the portability issues, STL could be a better choice. Unfortunately, as discussed in Section 3.10, the STL template classes do not have any persistence support [Stev98]. They cannot be serialized using MFC either.

The template-based type-safe collection classes in MFC support collections of objects of any type, and collections of pointers to objects of any type. A doubly linked list in the GGS document class could store pointers to the objects created on the heap, rather than objects themselves to avoid unnecessary duplication.

The `CTypedPtrList` class template is selected as a basis for managing objects in the GGS document class. A typed pointer list class can be declared as the following statement:

```
CTypedPtrList<BaseClass, Type*> ListName;
```

The first argument specifies a base class that must be one of the two pointer list classes defined in MFC, i.e., either `CObList` or `CPtrList`. Using `CObList` creates a

list supporting pointers to objects derived from `CObject`, while `CPtrList` supports lists of `void*` pointers. Since the main classes in GGS (i.e. other than dialogue template based and helper classes) have `CObject` as their base class, the choice is straightforward. The second argument to the `CTypedPtrList` template is the type of the pointer to be stored in the list. Based on the initial class hierarchy as presented in Figure 4.22, this would be `CObject*` because `CObject` is the common root. But the disadvantage is that the list could contain an object of any class derived from `CObject`. To increase the level of security in GGS, it is desirable to store pointers to objects of a user defined class derived from `CObject` and not `CObject` itself.

5.2 DRAWING GGS DOCUMENT

As the GGS document owns the list of objects, and if the list is protected, the GGS view class cannot use it directly. For example, the `OnDraw()` member of the view should be able to call the `Draw()` member of each object in the list. So we need to consider how best to do this with the following options:

- The list can be made public, but this would rather defeat the goal of maintaining protected members of the document class, as it would expose all the function members of the list object.
- A member function could be added to return a pointer to the list, but this would effectively make the list public and also incur overhead in accessing it.
- A public function could be added to the document which would call the `Draw()` or similar member for each object. This public function of the document could be called from the view. This would not be a bad solution, as it would still maintain the privacy of the list. The negative point is that the function would need access to a device context, and this is really the domain of the view.
- The GGS view class can be a friend of the document class, but this would expose all the members of the document, which is not desirable, particularly with a complex class.
- We could add a function to provide a `POSITION` value for the first list object, and a second member to iterate through the list. This would not expose the list, but it would make the object pointers available.

The last option appears to be the best alternative. However, if the document stores `CObject` pointers, it would not be easy for the member functions of the view class to

deal with them. The `OnDraw()` member function could skip the sound and timer objects but the following code would be necessary:

```
// The view class retrieves a CObject pointer, m_pSelected
// from the document
if (m_pSelected->IsKindOf(RUNTIME_CLASS(CElement)))
    ((CElement*)m_pSelected)->Draw(pDC);
if (m_pSelected->IsKindOf(RUNTIME_CLASS(CExternalImage)))
    ((CExternalImage*)m_pSelected)->Draw(pDC);
```

`Draw()` is not a member function of the `CObject` class. Hence, not only casting is necessary, but also RTCI, as described in Section 3.4.1. The C++ run-time type information (RTTI) could be used instead of RTCI, but that would not improve the situation. In fact, the above code completely defeats the expected polymorphism in GGS.

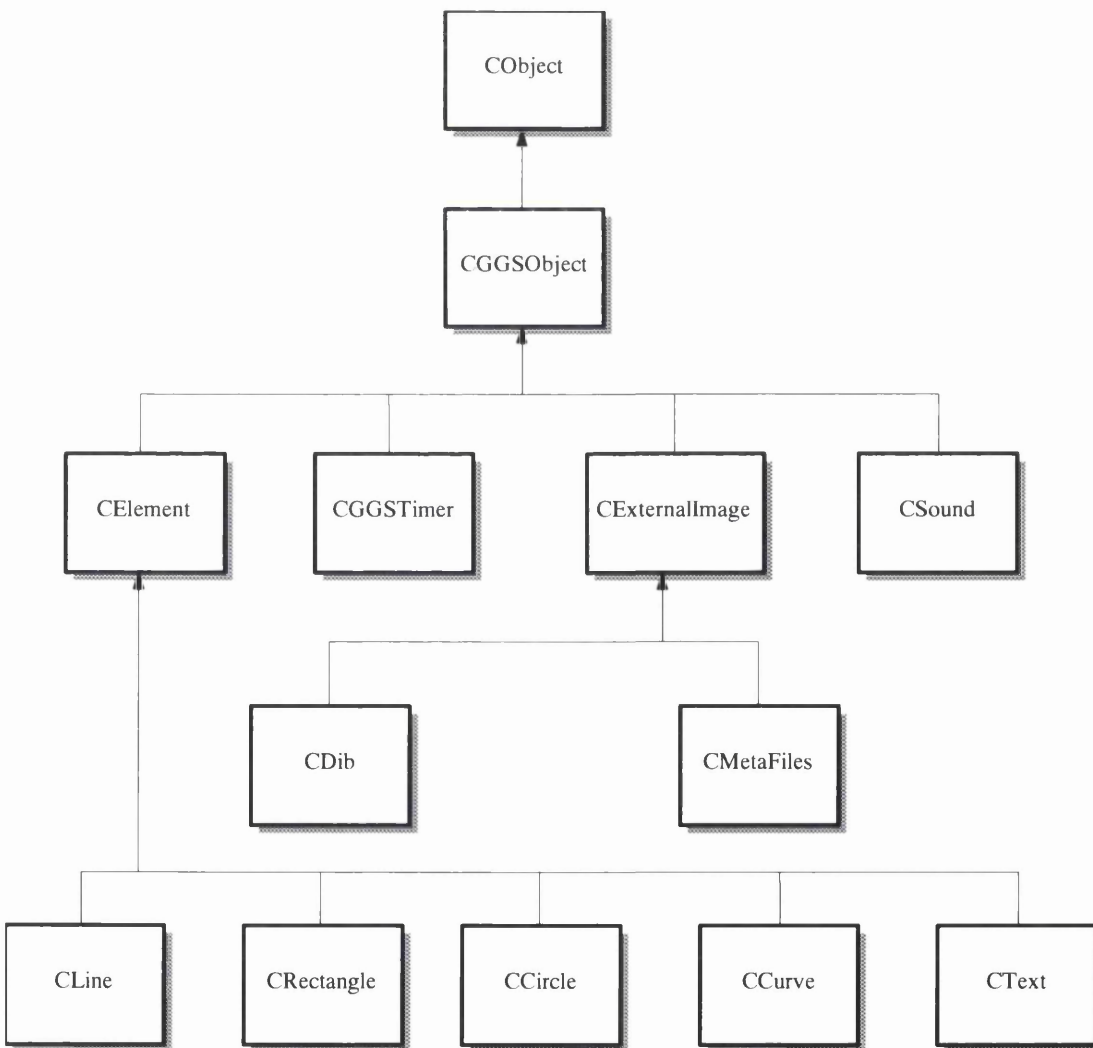


Figure 5.1 Revised Class Hierarchy

Clearly, a parent class is necessary for all GGS objects to define the common features as virtual functions and each derived class would override these functions in order to define its own characteristics.

Now based on the revised class hierarchy in Figure 5.1, the list in the GGS document class can be revised as well:

```
CTypedPtrList<CObList, CGGSObject*> m_ObjectList ;
```

The declaration of `m_ObjectList` ensures that only `CGGSObject` pointers can be stored. This provides an increased level of security in GGS.

5.2.1 The Parent Class `CGGSObject`

Listing 5.1 presents the `CGGSObject` class declaration. A virtual destructor is there to ensure that derived class objects are destroyed properly. The default constructor is in the protected section of the class to ensure that it cannot be used externally.

```
// Generic CGGSObject class
class CGGSObject : public CObject
{
    DECLARE_SERIAL(CGGSObject)
public:
    virtual ~CGGSObject(){}
    virtual void Draw(CDC* pDC, BOOL bSelect=FALSE){}
    virtual CRect GetBoundRect() ;
    virtual void Serialize(CArchive& ar) ;
protected:
    // Default constructor for serialization
    CGGSObject(){}
};
```

Listing 5.1 The `CGGSObject` Declaration

The `Draw()` function will need a pointer to a `CDC` object to provide access to the *Windows* drawing functions. In fact, the `Draw()` member should be declared as a pure virtual function in the `CGGSObject` class - after all, it cannot have any meaningful content in this class. This would also force any derived class to define it. However, `CGGSObject` inherits serialization from `CObject` and that requires an instance of the class be created. It is possible to include abstract classes in the MFC serialization, as discussed in Section 3.10, but it is best to avoid redefining the framework macros.

5.2.2 Drawing Sound and Timer Objects!

`CSound` and `CGGSTimer` objects do not have anything to draw on the screen. In the animation mode as explained in Chapter 1, sound objects are played and timers are activated. Apparently, one alternative solution could be the replacement of the virtual `Draw()` by a virtual `Activate()` member function. `Activate()` could draw the graphic objects, play sound and enliven timers.

A *Windows* application does its “permanent” drawings in response to the `WM_PAINT` message. The `OnDraw()` function in the view class should be the only place initiating any drawing operations for the document data. This ensures that the view is drawn correctly whenever *Windows* deems it necessary.

An application draws in a window at a variety of times: when first creating a window, when changing the size of the window, when moving the window from behind another window, when minimising or maximising the window, when displaying data from an opened file, and when scrolling, changing, or selecting a portion of the displayed data.

The OS manages actions such as moving and sizing a window. If an action affects the content of the window, the OS marks the affected portion of the window as ready for updating and, at the next opportunity, sends a `WM_PAINT` message to the application. The message is a signal to the application to determine what must be updated and to carry out the necessary drawing.

Clearly, the drawing operations are frequently needed but the animation is a special case. Hence, it is necessary to separate them altogether which would finally help in implementing the incremental animation and other difficult issues. In fact, `CSound` and `CGGSTimer` do not need to override the `Draw()` member of `CGGSObject`:

```
POSITION aPos = pDoc->GetListHeadPosition();
CGGSObject* pObject = 0; // Store for an object pointer
while(aPos) // Loop while aPos is not null
{
    pObject = pDoc->GetNext(aPos);
    // If the object is visible...
    if(pDC->RectVisible( pObject->GetBoundRect() ))
        pObject->Draw(pDC) ; // ...draw it
}
```

`CDC::RectVisible()` is used for efficiency [ShWi96] to save clock cycles from drawing invisible objects. It is not an uncommon practice to add “nil” virtual functions to a base class such as `CObject::Draw()` so that derived classes are not burdened with the need to implement them. However, the class designer should be aware of the fact that such a practice violates the Liskov Substitution Principle (LSP), leading to reusability problems [Mart96].

5.3 RUBBER BANDING

“Rubber banding” refers to a variety of visual effects to graphic designers. To draw a line, for instance, the user could position the cursor and press the left mouse button where s/he wanted the line to start, and then define the end of the line by moving the cursor with the left button held down. It would be ideal if the line was continuously drawn as the cursor was moved with the left button down. This continuous updating of the line is an example of rubber banding effects:

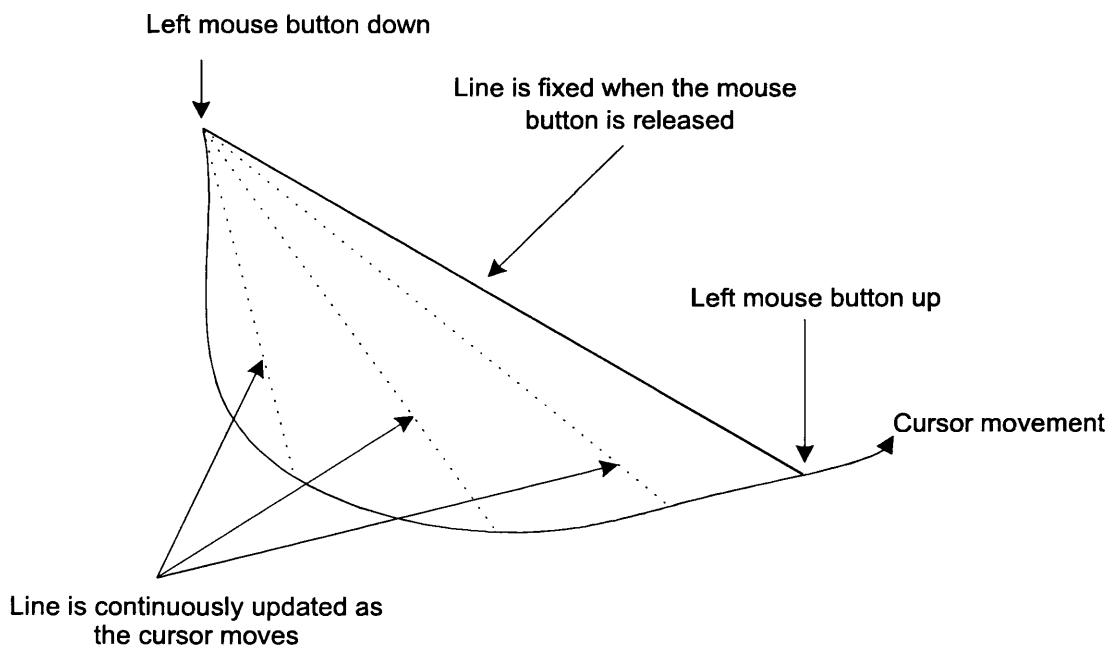


Figure 5.2 A Line Defined by the Mouse

It is not necessary to worry about coordinates since the graphics primitives in GGS are there only to support sketching, scribbling, etc. The easiest mechanism for drawing would be just using the mouse.

A circle could be drawn in a similar fashion. The first press of the left mouse button would define the centre, and as the cursor is moved with the button down, the program would track it. The circle would be continuously redrawn, with the current

cursor position defining a point on the circumference of the circle. As with drawing a line, the circle would be fixed when the left mouse button is released.

There is no difference in drawing a rectangle either. The first point is defined by the position of the cursor when the left mouse button is pressed. This is one corner of the rectangle. The position of the cursor when the mouse is moved with the left button held down defines the diagonally opposite corner of the rectangle. The final rectangle is the last one defined when the left mouse button is released.

A curve will be somewhat different. It may be defined by an arbitrary number of points as illustrated below:

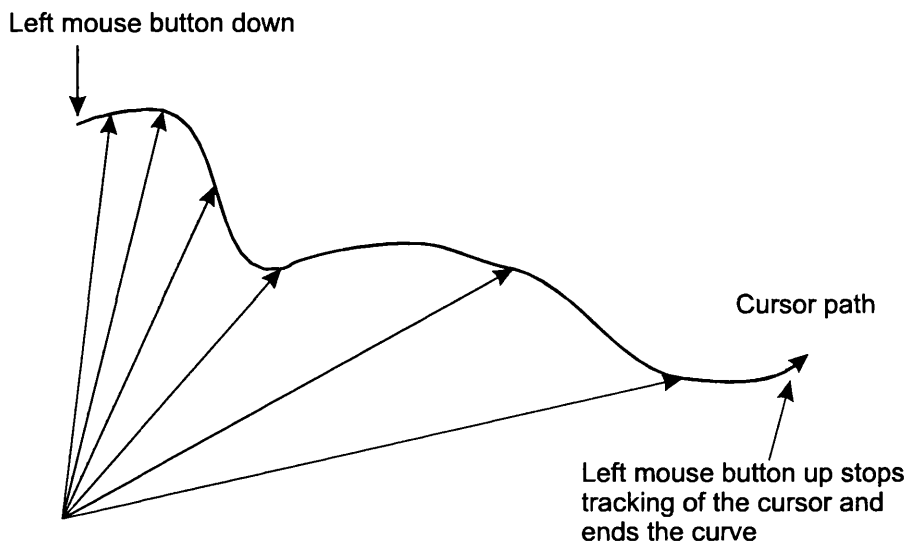


Figure 5.3 A Curve Defined by Straight Line Segments Joining Successive Cursor Positions

As with the other shapes, the first point is defined by the cursor position when the left mouse button is pressed. Successive positions recorded, when the mouse is moved, are connected by straight line segments to form the curve. In other words, the mouse track defines the curve.

5.3.1 Programming the Mouse

Information regarding mouse activities is provided by *Windows* in the form of messages. The following three mouse messages are important to GGS:

Table 5.1 Mouse Messages

Message	Occurs...
WM_LBUTTONDOWN	when the left mouse button is pressed.
WM_LBUTTONUP	when the left mouse button is released.
WM_MOUSEMOVE	when the mouse is moved.

These messages are quite independent of one another. It is quite possible for a window to receive a WM_LBUTTONUP message without having previously received a WM_LBUTTONDOWN message. This can happen if the button is pressed with the cursor over one window and then moved to another window before being released.

When the user of GGS is drawing a graphics primitive, s/he will be interacting with a particular document view. The view class is, therefore, the obvious place to put the message handlers for the mouse. Capturing the *Windows* messages is a standard programming practice. The implementation details that are standard and discussed in various books and manuals are not described here.

In short, the WM_LBUTTONDOWN message handler records the starting position for a graphics primitive. The WM_MOUSEMOVE handler will only be concerned with drawing a succession of temporary versions of an object as the cursor is moved, because the final object will be created when the left mouse button is released. We can therefore treat the drawing of temporary objects to provide rubber banding as being entirely local to the WM_MOUSEMOVE handler, leaving the final version of the object being created, to be drawn by the OnDraw() member of the view. This approach will result in the drawing of the rubber-banded objects being reasonably efficient without involving the OnDraw() member, which will be responsible ultimately for the drawing of the entire document.

5.3.2 Raster Operations

The CDC::SetROP2() function sets the drawing mode for all subsequent output operations in the device context associated with a CDC object. When the mode is set as R2_NOTXORPEN, it can work some magic for the application programmer! For example, the first time a particular shape is drawn on the default white background, it will be drawn normally in the pen colour specified. If the same shape is drawn again, overwriting the first, the shape will disappear, because the colour that the shape will be drawn in corresponds to that produced by exclusive ORing the pen

colour with itself. The drawing colour that results from this will be white. This is illustrated through an example in Table 5.2. The colour, white is formed from equal proportions of the 'maximum' amounts of red, blue, and green. For simplicity, this can be represented as 1,1,1 being the RGB components of the colour. In the same scheme, red is defined as 1,0,0.

Table 5.2 Drawing First Time

Colour	R	G	B	Comments
Background - white	1	1	1	White
Pen - red	1	0	0	Red
XOR	0	1	1	Cyan
NOT XOR	1	0	0	Red

So, the first time a red line is drawn on a white background in the R2_NOTXORPEN mode, it comes out red. If the same line is drawn a second time, overwriting the existing line, the background pixels to be written over are red. The resultant drawing colour works out as follows:

Table 5.3 Drawing Second Time

Colour	R	G	B	Comments
Background - red	1	0	0	Red
Pen - red	1	0	0	Red
XOR	0	0	0	Black
NOT XOR	1	1	1	White

Since the rest of the background is white, the line will disappear. Hence, a shape or a graphics primitive being drawn should be stored temporarily so that it could be redrawn in the R2_NOTXORPEN mode in order to remove it from the client area of the view. This will automatically rubber-band the shape being created, so it will appear to be attached to the cursor position as it moves.

5.3.3 Drawing Curve Objects

Drawing a curve is different from drawing a line or a circle. With a line or a circle, as the cursor is moved with the left button down, a succession of different lines or

circles are created that share a common reference point - the point where the left mouse button was pressed.

On the contrary, when the cursor is moved, while drawing a curve, the user is not creating a sequence of new curves, but extending the same curve, so each successive point adds another segment to the curve's definition. Therefore, a `CCurve` object needs to be created as soon as two points are obtained from the `WM_LBUTTONDOWN` message and the first `WM_MOUSEMOVE` message. Points defined by subsequent mouse move messages then add new segments to the existing curve object. Hence, the `CCurve` class should maintain a list of all joining points and a member function, such as `AddSegment()` to extend the curve once it has been created by the constructor.

A further point to consider is how to calculate the enclosing rectangle. This is defined by getting the minimum x and y pair from all the defining points to establish the top left corner of the rectangle, and the maximum x and y pair for the bottom right corner. This involves going through all the points in the list. The enclosing rectangle, therefore, is computed incrementally in the `AddSegment()` function as points are added to the curve.

An object of type `CCurve` should be treated as a special case once it has been created. This is because, on all subsequent calls to the `WM_MOUSEMOVE` handler, the `AddSegment()` function should be called for the existing curve, rather than constructing a new curve to replace the old. The `R2_NOTXORPEN` drawing mode is not necessary to erase the previous curve each time. The way to handle this is by moving the call to `SetROP2()` to a position after the code processing a curve:

```

if (CURVE == GetDocument()->GetObjectType()) // Is it a curve?
{
    // We are drawing a curve
    // so add a segment to the existing curve
    ((CCurve*)m_pTempObject)->AddSegment(m_ptSecondPoint);
    m_pTempObject->Draw(&aDC); // Now redraw it
    return ;
}
// Redraw objects other than curves to erase them from the view
aDC.SetROP2(R2_NOTXORPEN); // Set the drawing mode
m_pTempObject->Draw(&aDC); // Disappear now ...

```

5.4 DELETING AND MOVING SHAPES

Being able to delete shapes is a fundamental requirement in a drawing program. One question relating to this, that we need to find an answer for, is how the user is going to select the element to be deleted. Of course, once the process of selecting an element is decided, this will apply equally well if the user wants to move an element. Hence, moving and deleting elements can be treated as related problems.

One conventional way of providing move and delete functions would be to have a pop-up context menu appear at the cursor position when the right mouse button is clicked. "Move" and "Delete" will be the items on the menu. A pop-up that works like this is a handy facility that can be used in various other situations.

How should the pop-up be used? The standard way that context menus work is that the user moves the mouse over a particular object and right-clicks on it. This selects the object and pops up a menu containing a list of items which relate to actions that can be performed on that object. This means that different objects can have different menus. The menu that appears is sensitive to the context of the cursor, hence the term "context menu". There are two contexts to consider in GGS. The user could right click on an object with the cursor, or s/he could right click when there is no object under the cursor. This can be resolved simply by creating two menus: one for an object under the cursor, and one for the rest of the client area. After a right click, if there is an object under the cursor, that should be highlighted so that the user knows exactly which object the context pop-up is referring to.

MFC provides a class called `CMenu` for managing and processing menus. The pop-up menu appears when the user presses (or more specifically, releases) the right mouse button. Clearly, the implementation code is necessary in the message handler for `WM_RBUTTONDOWN` in the view class of GGS.

5.4.1 Highlighting Shapes

Highlighting a shape is important when it is selected. Before deleting a shape, the user must know which element s/he is operating on. The highlighting can be done in the `Draw()` member of each shape class. In Listing 5.1, the virtual `Draw()` member function has a Boolean flag specifically for this purpose. The default value for this flag is `FALSE` when the shape is drawn normally. However, when the flag is `TRUE`, in other words, when the shape is selected, it is drawn with the highlighting colour.

5.4.2 Moving Shapes

In GGS, if the user chooses to move a selected object, the mouse pointer is first moved automatically to the centre of the highlighted object. Then the user can move (i.e., not click and drag) the pointer and the object will move as well, as if glued to the pointer.

Moving a shape in a view window is implemented using the `R2_NOTXORPEN` drawing mode since it is easy and fast. This is exactly the same mechanism as that used during the creation of a graphics primitive. The selected object is redrawn to reset it to the background colour, and then the function `Move()` is called to relocate the object by the distance measured between the current and previous cursor position. For example, the following is the implementation of the `Move()` function in the `CLine` class:

```
// Distance between current & previous cursor position = aSize
void CLine::Move(const CSize& aSize)
{
    m_StartPoint += aSize; // Move the start point
    m_EndPoint += aSize; // and the end point
    m_EnclosingRect += aSize; // Move the enclosing rectangle
}
```

This is easy because of the overloaded `+=` operators in the `CPoint` and `CRect` classes. They all work with `CSize` objects, so the relative distance specified by `aSize` can be added to the start and end points for the line, and to the enclosing rectangle.

Moving `CRectangle` or `CCircle` objects is even easier. Only the enclosing rectangle is required to move because it defines these objects. Moving a `CCurve` object is a little more complicated because it is defined by an arbitrary number of points. It is necessary to iterate through all the points defining the curve, moving each one in turn with the overloaded `+=` operator available in the `CPoint` class.

All that remains is to drop the object in position once the user has finished moving it, or to abort the whole move. In GGS, to drop the object in its new position, the user will click the left mouse button. But a right click will cancel the move operation and the highlighted object will be moved back to its original position.

5.4.3 Masked Objects

The enclosing rectangles of objects are very likely to overlap. This would create a situation where multiple objects could be found under the cursor. In this case, the common practice followed in different drawing packages is to select the first or the last object in sequence and allow the user to send it “back”, if necessary. That can alter the sequence of objects in the internal list maintained by the package. However, the sequence of objects in GGS is important for still animation, and must not be changed to facilitate the selection process. Hence, the user is requested to resolve the ambiguity when multiple objects are found under the cursor:

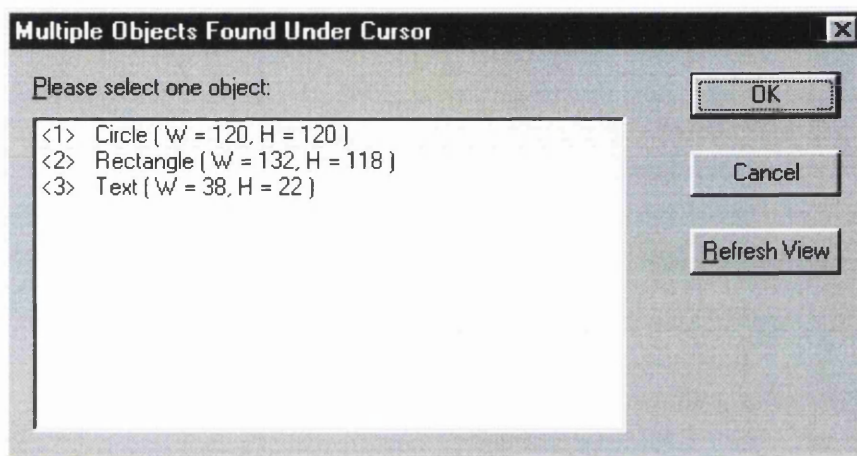


Figure 5.4 Dialogue Box Resolving Ambiguity

A dialogue box is actually a window and each control in a dialogue box is also a specialised window. When *Windows* runs, most of the things we see on the screen are windows! There are two aspects to programming for a dialogue: getting it displayed and handling the effects of its controls. Usually, an application programmer using MFC creates a dialogue resource by placing different controls on it, and then associates the new resource with a new class typically derived from the MFC class `CDialog`.

The class associated with the dialogue box in Figure 5.4 receives the pointers to the GGS objects found under the cursor and a pointer to the active view as well. The list box displays the object type and its width and height. The objects are highlighted in the active view window, when they are selected in the list box so that they could be identified. This requires trapping a *Windows* message. The `LBN_SELCHANGE` notification message is sent when the selection in a list box is about to change. The parent window of the list box which is the dialogue in this case, receives this notification message. The following code illustrates the `LBN_SELCHANGE` message handler:

```

void CMultipleObjectsDlg::OnSelchangeObjectsList()
{
    int nSelection = m_listbox.GetCurSel() ;
    if( nSelection != LB_ERR )
    {
        // Draw previously selected object without highlight
        if( m_pObject != NULL )
            m_pObject->Draw(m_pDC, FALSE) ;
        m_pObject = (CGGSObject*) m_listbox.GetItemDataPtr
            (nSelection) ; // New selection
        m_pObject->Draw(m_pDC, TRUE) ;
    }
}

```

Listing 5.2 Highlighting Objects in a View from a Dialogue Box

The list box is a `CListBox` object and it is worth noting that `CListBox::GetItemDataPtr()` returns a void pointer that requires suitable casting. The user not only highlights objects for selection but also refreshes the view window, if necessary using the “Refresh View” button as shown in Figure 5.4. Bitmaps and metafiles can obstruct other objects when deselected. Hence, the view may need to be refreshed.

5.5 IMPLEMENTING SCROLLING WITH SCALING

The MFC class `CScrollView` is very useful. It is possible to implement standard scrolling capabilities in any class derived from `CView` by overriding the message-mapped `OnHScroll` and `OnVScroll` member functions. But `CScrollView` adds the following features [MSDN51]:

- It manages the window and viewport sizes and the mapping modes.
- It scrolls automatically in response to the scroll-bar messages.
- It scrolls automatically in response to messages from the keyboard, a non-scrolling mouse, or the IntelliMouse wheel.

To take advantage of automatic scrolling, application programmers should derive their view class from `CScrollView` instead of `CView`.

On the other hand, scaling with *Windows* usually involves using one of the scaleable mapping modes, `MM_ISOTROPIC` or `MM_ANISOTROPIC`. By using one of these

mapping modes, the programmer can get *Windows* to do most of the work. Unfortunately, it is not as simple as just changing the mapping mode, because neither of these mapping modes is supported by `CScrollView`.

5.5.1 Scaleable Mapping Modes

`MM_ISOTROPIC` and `MM_ANISOTROPIC` allow the mapping between the logical and device coordinates to be altered. The logical coordinates (also referred to as the page coordinates) are determined by the mapping mode. For example, the `MM_LOENGLISH` mapping mode has logical coordinates in units of 0.01 inches, with the origin at the top left corner of the client area, and the positive y axis direction running from bottom to top. The logical coordinates are used by the device context drawing functions.

The device coordinates (also referred to as the client coordinates in a window) are measured in pixels with the origin at the top left corner of the client area, and with the positive y axis direction running from top to bottom. These are used outside of a device context, for example for defining the position of the cursor in the mouse message handlers.

The screen coordinates are measured in pixels and have the origin at the top left corner of the screen, with the positive y axis direction from top to bottom. These are used when getting or setting the cursor position.

`MM_ISOTROPIC` has the property that *Windows* will force the scaling factor for both the x and y axes to be the same, which has the advantage that the circles will always appear as circles. But the disadvantage is that a document cannot be mapped to fit into a rectangle of a different shape. `MM_ANISOTROPIC`, on the other hand, permits scaling of each axis independently. Because it is more flexible, `MM_ANISOTROPIC` is used for scaling operations in GGS.

5.5.2 Transformation of Coordinates

The way in which logical coordinates are transformed to device coordinates is dependent on the parameters presented in Table 5.1. With mapping modes other than `MM_ISOTROPIC` and `MM_ANISOTROPIC`, the window and the viewport extent are fixed and they cannot be changed. Calling the functions `SetWindowExt()` or `SetViewportExt()` of the CDC object to change them will have no effect, although the position of (0,0) in the logical reference frame can be moved by calling `SetWindowOrg()` or `SetViewportOrg()`.

Table 5.1 Coordinate Transformation Parameters

Parameter	Description
Window origin	The logical coordinates of the top left corner of the window. This is set by calling the function <code>CDC::SetWindowOrg()</code> .
Window extent	The size of the window specified in logical coordinates. This is set by calling the function <code>CDC::SetWindowExt()</code> .
Viewport origin	The device coordinates of the top left corner of the window (pixels). This is set by calling the function <code>CDC::SetViewportOrg()</code> .
Viewport extent	The size of the window in device coordinates (pixels). This is set by calling the function <code>CDC::SetViewportExt()</code> .

The viewport referred to here has no physical significance by itself; it serves only as a parameter for defining how coordinates are transformed from logical coordinates to device coordinates. For a given document size which will be expressed by the window extent in logical coordinate units, the scale can be adjusted at which objects are displayed by setting the viewport extent appropriately. The formulae that are used by *Windows* to convert from logical coordinates to device coordinates are:

$$x_{Device} = (x_{Logical} - x_{WindowOrg}) \times \frac{x_{ViewPortExt}}{x_{WindowExt}} + x_{ViewportOrg} \quad (5.1)$$

$$y_{Device} = (y_{Logical} - y_{WindowOrg}) \times \frac{y_{ViewPortExt}}{y_{WindowExt}} + y_{ViewportOrg} \quad (5.2)$$

If the equations are simplified for converting between device and logical coordinates by setting the window origin and the viewport origin both at (0,0), they become:

$$x_{Device} = x_{Logical} \times \frac{x_{ViewPortExt}}{x_{WindowExt}} \quad (5.3)$$

$$y_{Device} = y_{Logical} \times \frac{y_{ViewPortExt}}{y_{WindowExt}} \quad (5.4)$$

If the viewport extent values are multiplied by the scale, the objects will be drawn according to the value of the scale. The equations with the scale included will be:

$$x_{Device} = x_{Logical} \times \frac{x_{ViewPortExt} \times Scale}{x_{WindowExt}} \quad (5.5)$$

$$y_{Device} = y_{Logical} \times \frac{y_{ViewPortExt} \times Scale}{y_{WindowExt}} \quad (5.6)$$

It is clear from the above equations that a given pair of device coordinates will vary in proportion to the scale value. The coordinates at a scale of 3 will be three times the coordinates at a scale of 1. Of course, as well as making objects larger, increasing the scale will also move them away from the origin.

This logic is implemented in the `OnPrepareDC()` member of the GGS view class. `OnPrepareDC()` is always called for any `WM_PAINT` message. However, for scaling it is necessary to call it before drawing temporary objects in the mouse message handlers. Although there is nothing else required to scale the view, the scrolling will not work with the new arrangement because the length of the scroll bars will not scale appropriately with the rest of the view. Just setting the mapping mode is clearly not enough.

5.5.3 Restoring Scrolling

The solution to get around the problem with scrolling could be the use of another mapping mode to set up the scrollbars. The easiest way to do this is to use `MM_TEXT`, because in this case the logical coordinates are the same as the device coordinates - pixels, in other words. The necessary step is to calculate how many pixels are equivalent to the logical document extent for the scale at which the objects are being drawn. Hence, a member function is added to the GGS view class to take care of the scrollbars:

```
void CCGSView::ResetScrollSizes()
{
    CClientDC aDC(this);
    OnPrepareDC(&aDC); // Set up the device context
    CSize DocSize = GetDocument()->GetDocSize();
    aDC.LPtoDP(&DocSize); // Get the document size in pixels
    SetScrollSizes(MM_TEXT, DocSize); // Set up the scrollbars
}
```

After creating a local `CClientDC` object for the view, `OnPrepareDC()` is called to set up the `MM_ANISOTROPIC` mapping mode. Because this takes account of the scaling, the `CDC::LPtoDP()` member will convert the document size stored in the

local variable `DocSize` to the correct number of pixels for the current logical document size and scale. The total document size in pixels defines how large the scrollbars must be in the `MM_TEXT` mode. It is important to remember that the `MM_TEXT` logical coordinates are also in pixels. Based on this, the `SetScrollSizes()` member of `CScrollView` is called to set up the scrollbars by specifying `MM_TEXT` as the mapping mode.

It may seem strange that the mapping mode can be changed in this way, but it is important to note that the mapping mode is nothing more than a definition of how logical coordinates are to be converted to device coordinates. Whatever mode (and therefore, coordinate conversion algorithm) has been set up, will apply to all subsequent device context functions until it is changed again. When a new mode is set, subsequent device context functions just use the conversion algorithm defined by the new mode.

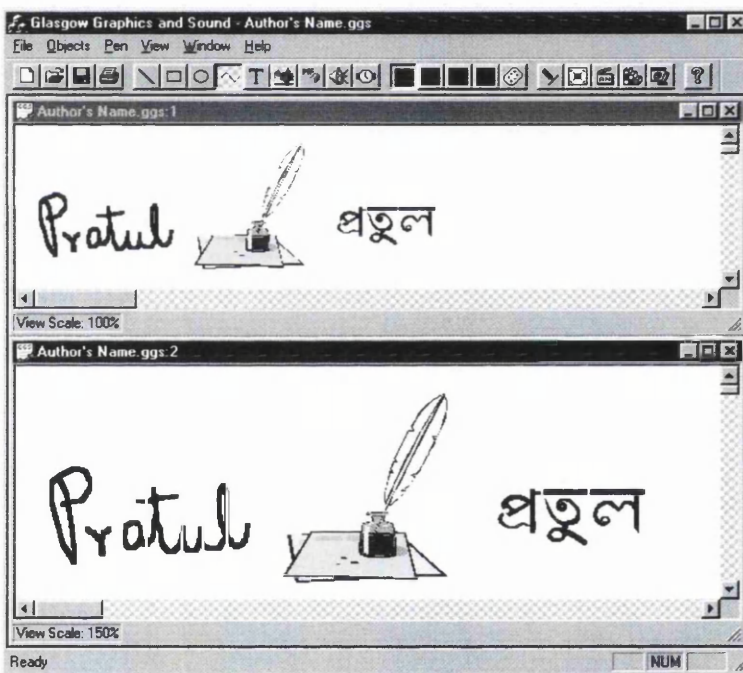


Figure 5.5 Scaling and Scrolling in Action

5.6 TEXT OBJECTS

Figure 5.5 presents the author's name in two different ways. Freehand curves are used on the left-hand side and a scaleable foreign (Bengali) font is used on the right-hand side. As mentioned earlier, the text objects in GGS are slightly different from other primitives, although they are derived from the same base class, `CElement`. The `CText` objects not only store the text string and its position, but also the font information as well.

```

class CText:public CElement
{
    DECLARE_SERIAL(CText)
public:
    virtual void Draw(CDC* pDC, BOOL bSelect = FALSE);
    CText(CPoint ptStart, CPoint ptEnd, CString aString,
        LOGFONT logFont, COLORREF aColor);
    virtual void Move(CSize& aSize);
    virtual void Serialize(CArchive& ar);
protected:
    BOOL m_bFontCreated;
    LOGFONT m_logFont; // Font details
    CFont m_font; // Encapsulates the font handle
    CPoint m_StartPoint; // Position of a text element
    CString m_String; // Text to be displayed
    CText(); // Protected default constructor
};

```

Listing 5.3 The CText Class Declaration

The CFont object, `m_font` in Listing 5.3 is initialised with the characteristics stored in the LOGFONT structure, `m_logFont`. The font can subsequently be selected as the current font for any device. When the font is selected by using `CDC::SelectObject()`, the *Windows* font mapper attempts to match the logical font with an existing physical font. If it fails to find an exact match for the logical font, it provides an alternative whose characteristics match as many of the requested characteristics as possible.

5.6.1 Moving Text Objects

A CText object in GGS is displayed using the `CDC::TextOut()` function. Since `TextOut()` does not use a pen, it would not be affected by setting the drawing mode of the device context. In other words, the raster operations using the `SetROP2()` function to move the elements will leave temporary trails behind when applied to the CText objects. To get around this problem, a method of invalidating the rectangles [Hort98] that are affected by the moving elements could be used. This can, however, cause some flicker when an element is moving fast. A better solution would be to use the invalidation method only for the text objects and the original ROP method for all other graphics primitives.

In order to use separate code for the text objects, the runtime class information is necessary:

```

if (m_pSelected->IsKindOf(RUNTIME_CLASS(CText)))
{
    CRect OldRect = m_pSelected->GetBoundRect();    // Get old rect
    m_pSelected->Move(aSize);                       // Move the object
    CRect NewRect = m_pSelected->GetBoundRect();    // Get new rect
    OldRect.UnionRect(&OldRect,&NewRect);          // Combine rects
    aDC.LPtoDP(OldRect);                           // Convert to device coords
    OldRect.NormalizeRect();                        // Normalize combined area
    InvalidateRect(&OldRect);                      // Invalidate combined area
    UpdateWindow();                                // Redraw immediately
    m_pSelected->Draw(&aDC,TRUE);                  // Draw highlighted
    return;
}

```

Clearly, the above code for invalidating the rectangles for moving the text objects is much less elegant than the ROP code used for other objects.

5.7 BENEFITS OF SERIALIZATION

The application data to be stored in a document often originates in an unstructured and unpredictable way. The MFC serialization is not the best solution for object persistence as discussed in Chapters 2 and 3 but it is relatively easy-to-use and highly effective in the case of GGS. This section briefly reviews the implementation of serialization starting with `CGGSObject::Serialize()`:

```

void CGGSObject::Serialize(CArchive& ar)
{
    CObject::Serialize(ar);    // Call the base class function
}

```

Apart from calling the base class `Serialize()` function, it does not have to do anything else. The same function in the `CElement` class serializes the object colour, the enclosing rectangle and the pen width:

```

void CElement::Serialize(CArchive& ar)
{
    CGGSObject::Serialize(ar);    // Call the base class function
    if (ar.IsStoring())
    {

```

```

        ar << m_Color           // Store the colour,
        << m_EnclosingRect     // and the enclosing rectangle,
        << m_Pen ;             // and the pen width
    }
else
{
    ar >> m_Color           // Retrieve the colour,
    >> m_EnclosingRect     // and the enclosing rectangle,
    >> m_Pen ;             // and the pen width
}
}

```

`Serialize()` in the `CRectangle` and `CCircle` class is very simple and identical since these classes have no additional data member. All they do is to call the direct base class function to ensure that the inherited data members are serialized. For the `CCurve` class, the `Serialize()` function is surprisingly simple:

```

void CCurve::Serialize(CArchive& ar)
{
    CElement::Serialize(ar);    // Call the base class function
    m_PointList.Serialize(ar); // Serialize the list of points
}

```

It is only necessary to call the `Serialize()` function for the `CList` object, `m_PointList`. Objects of the `CList`, `CArray`, and `CMap` classes can be serialized in this way, since they are all derived from `CObject`.

`CText::Serialize()` is also similar but has the responsibility to take care of a `LOGFONT` structure as illustrated in Listing 5.3. The members of the `LOGFONT` structure are individually serialized using the overloaded extraction and insertion operators of the `CArchive` class.

5.8 MULTI-PAGE PRINTING

Printing a document is controlled by the current view. The process is quite complicated since printing is inherently a complex business and it potentially involves the application programmer in implementing his/her versions of quite a number of inherited functions in the view class. The logic of the process and the functions involved are summarised in Figure 5.6.

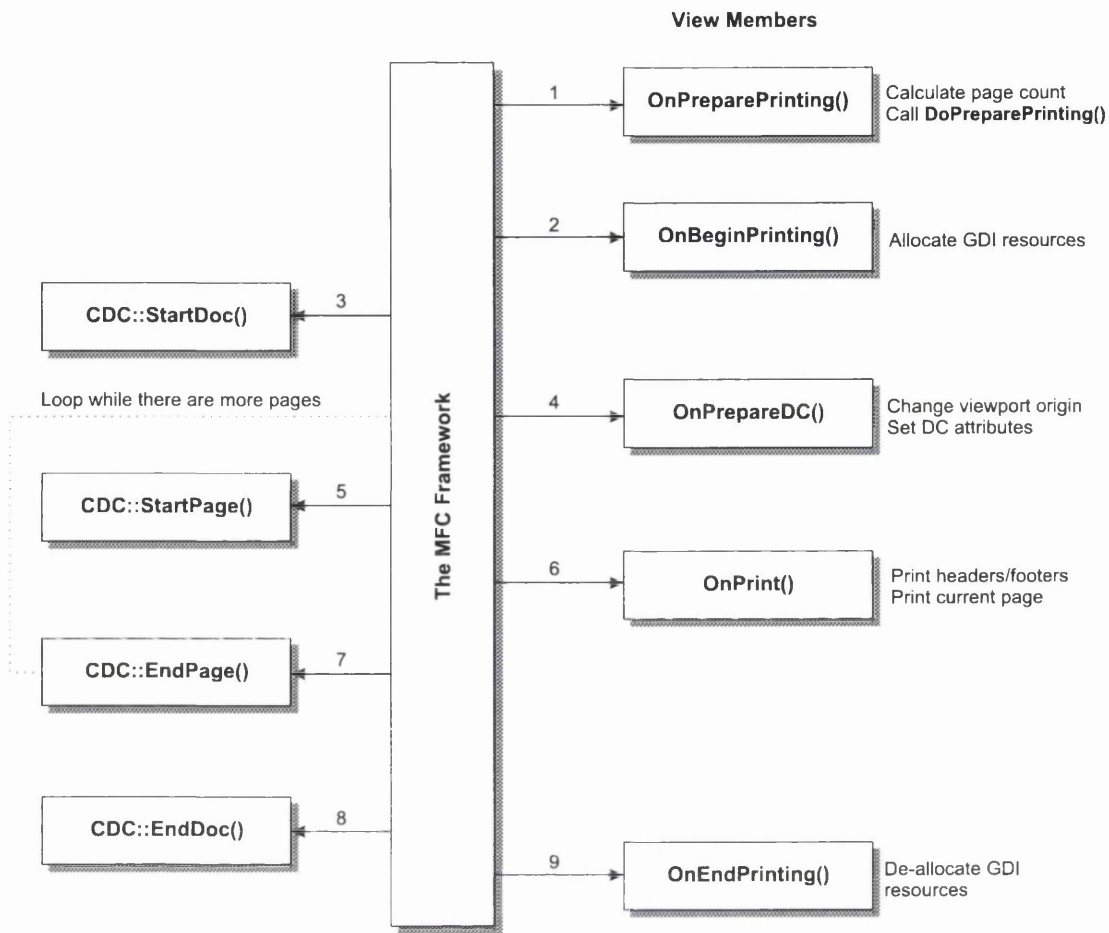


Figure 5.6 The Printing Process

Figure 5.6 shows how the sequence of events is controlled by the MFC framework and involves calling five inherited members of the view class, which may need to be overridden. The CDC member functions shown on the left side of the diagram communicate with the printer device driver and are called automatically by the framework.

The typical role of each of the functions in the current view during a print operation is specified in the notes alongside it. The sequence in which they are called is indicated by the numbers on the arrows. In practice, it is not necessary to implement all of these functions. Typically, `OnPreparePrinting()`, `OnPrepareDC()` and `OnPrint()` are implemented based on particular printing requirements of an application.

The output of data to a printer is done in the same way as outputting data to the display through a device context. The GDI calls that are used to output text or graphics are device independent, so they work just as well for a printer as they do for a display. The only difference is the device that the CDC object applies to.

The CDC functions in the process diagram in Figure 5.6 communicate with the device driver for the printer. If the document to be printed requires more than one printed page, the process loops back to call the `OnPrepareDC()` function for each successive new page, as determined by the `EndPage()` function.

All the functions in the view class that are involved in the printing process are passed a pointer to an object of type `CPrintInfo` as an argument. This object provides a link between all the functions that manage the printing process. A `CPrintInfo` object has a fundamental role in the printing process, since it stores information about the print job being executed and details of its status at any time. It also provides functions for accessing and manipulating this data. This object is the means by which information is passed from one view function to another during printing, and between the framework and the view functions.

5.8.1 Paper Size

To avoid overcomplicating the problem, GGS supports printing on either A4 or US Letter size (i.e., 8½ by 11 inches) paper only. However, GGS supports both portrait and landscape orientation. With either paper size, the document is printed in a central portion of the paper measuring 6 inches by 9 inches in the portrait orientation, and 9 inches by 6 inches in landscape. With these simplifications, GGS does not need to worry about the actual paper size. However, for a document larger than one page, it is necessary to divide up the document into chunks of 6 inches by 9 inches (or the other way around).

AppWizard added versions of `OnPreparePrinting()`, `OnBeginPrinting()` and `OnEndPrinting()` to the GGS view class at the outset. The base code provided for `OnPreparePrinting()` calls `DoPreparePrinting()` in the return statement:

```
BOOL CGGSView::OnPreparePrinting(CPrintInfo* pInfo)
{
    return DoPreparePrinting(pInfo) ;
}
```

The `OnPreparePrinting()` function in the view class is called by the application framework to initialise the printing process for the document. The basic initialisation that is required is to provide information about how many pages are in the document for the print dialogue that will be displayed. The `DoPreparePrinting()` function displays the print dialogue using information about the number of pages to be printed. Whenever possible, the number of pages to be

printed, should be calculated and stored in the `CPrintInfo` object before this call occurs. Of course, in many circumstances information is required from the device context for the printer before this can be done. For example, the number of pages may depend on the size of the font available in the printer, in which case it would not be possible to get the page count before calling `OnPreparePrinting()`. There are two ways to solve the problem. The number of pages could be computed in the `OnBeginPrinting()` member, which receives a pointer to the device context as an argument. This function is called by the framework after `OnPreparePrinting()`. Otherwise, the page count could be done after `DoPreparePrinting()` returns within `OnPreparePrinting()` because the information entered in the print dialogue would be available. This also means that the paper size and other options selected by the user in the print dialogue could be accounted for. The following code fragment is an example:

```
// The print dialogue box returned TRUE
//Paper size info is available now
LPDEVMODE pDevMode = pInfo->m_pPD->GetDevMode() ;
if ((pDevMode->dmPaperSize != DMPAPER_A4) &&
    (pDevMode->dmPaperSize != DMPAPER_LETTER))
{
    AfxMessageBox ("Please select A4 or Letter size paper") ;
    return FALSE ;
}
```

5.8.2 Preview and Printing

As mentioned earlier, if a document is larger than one page, it is divided into suitable rectangular areas depending on the paper size and orientation. In the `OnPrint()` function, each rectangular area of the document is mapped to the current page. Then the document is drawn on the page by calling the `OnDraw()` function that is used to display the document in the view. This potentially draws the entire document, but what appears on the page is restricted by defining a clip rectangle. A clip rectangle encloses a rectangular area in the device context within which output appears. Outside the clip rectangle, output is suppressed.

Internally, the objects and the view extents in GGS are measured in terms of hundredths of an inch. With the unit of size a fixed physical measure, objects are printed at the same size as they appear in a view window. The unit of object size corresponds to the `MM_LOENGLISH` mapping mode. In fact, this mapping mode is suitable where high precision is not necessary. In `MM_HIMETRIC`, for instance, each

logical unit corresponds to 0.01 millimetre. However, coordinate overflow problems are not uncommon when the `MM_HIMETRIC` mapping mode is used [Gery99]. Some application programmers do not realise that the coordinate systems in *Windows NT* are 32-bit but they are still 16-bit in *Windows 95*.

The printing section in GGS is complete with a preview option. The print preview functionality comes completely free in any AppWizard generated application with the MFC document/view architecture. If the application developer writes code for the multi-page printing operation, the framework uses that code to produce page images in the print preview window.

Print preview is somewhat different from screen display and printing because, instead of directly drawing an image on a device, the application must simulate the printer using the screen. When the user selects the Print Preview command from the File menu, the framework creates a `CPreviewDC` object. Whenever the application performs an operation that sets a characteristic of the printer device context, the framework also performs a similar operation on the screen device context. For example, if the application selects a font for printing, the framework selects a font for screen display that simulates the printer font. Whenever the application would send output to the printer, the framework instead sends the output to the screen.

Print preview also differs from printing in the order that each draws the pages of a document. During printing, the framework continues a print loop until a certain range of pages has been rendered. During print preview, one or two pages are displayed at any time, and then the application waits; no further pages are displayed until the user responds. During print preview, the application must also respond to the `WM_PAINT` messages, just as it does during ordinary screen display.

5.9 REFERENCES

- [ANSI97] ANSI/ISO C++ Committee: "International Standard for Information Systems – Programming Language C++", ISO/IEC IS 14882, November 1997.
- [Gery99] Gery, R.: "Coordinate Mapping", Technical Articles, Microsoft Developer Network Library, April 1999.
- [Hort98] Horton, I.: "Beginning Visual C++ 6", Wrox Press Ltd., August 1998.
- [Mart96] Martin, R.C.: "The Liskov Substitution Principle", C++ Report, March

1996.

- [MSDN51] Microsoft Developer Network Library: "CScrollView", Microsoft Foundation Class Library Reference, April 1999.
- [ShWi96] Shepherd, G. and Wingo, S.: "MFC Internals: Inside the Microsoft Foundation Class Architecture", Addison-Wesley Developers Press, 1996.
- [Stev98] Stevens, A.: "The Persistent Template Library", Dr Dobb's Journal, March 1998.

CHAPTER 6

BITMAPS AND METAFILES

6.1 GRAPHICS DEVICE INTERFACE (GDI)

The graphics device interface (GDI) forms part of the *Windows* operating system. GDI manages all graphics output from a *Windows* program. This means that no matter if a window is displayed on the screen or a screen saver displays some dazzling graphics or if an application prints a document, GDI is involved in making it happen.

Windows itself uses GDI to draw user interface elements such as windows, menus and dialogue boxes. The mouse pointer is displayed using GDI even though it appears to float over other screen objects. GDI was created to decouple rendering graphics from the underlying hardware. GDI provides high level drawing functions that produce generally the same results regardless of the underlying hardware.

The MFC class, `CDC` is the C++ wrapper of various GDI functions. In the earlier chapters, member functions of `CDC` or its derived classes have been used frequently to illustrate the implementation issues in developing GGS. In this chapter, some other aspects of GDI will be explored while dealing with externally created bitmaps and metafiles.

6.2 BITMAPS AND METAFILES

Bitmaps and metafiles represent two very different ways of storing pictorial information. A bitmap is a complete digital representation of a picture. Each pixel in the image corresponds to one or more bits in the bitmap. Monochrome bitmaps require only one bit per pixel; colour bitmaps require additional bits to indicate the colour of each pixel.

A metafile, on the other hand, stores pictorial information as a series of records that correspond directly to the GDI calls. A metafile is thus a description of a picture rather than a digital representation of it. Metafiles are more closely associated with *Windows* drawing and computer-aided design (CAD) programs. The user of these programs draws an image with lines, rectangles, circles, text, and other graphics primitives. Although drawing or CAD programs generally use a private data format for storing the picture in a file, they can usually transfer the picture in the form of a metafile understood by other applications.

Both bitmaps and metafiles have their place in computer graphics. Bitmaps are very often used for very complex images originating from the real world, such as digitised photographs. Metafiles are more suitable for human or machine generated images,

such as architectural drawings. Both bitmaps and metafiles can exist in memory or be stored on a disk as files, and both can be transferred among *Windows* applications using the clipboard.

Bitmaps have two major drawbacks. First, they are highly susceptible to problems involving device dependence. The most obvious device dependency is colour. Displaying a colour bitmap on a monochrome device is often unsatisfactory. Another problem is that a bitmap implies a particular resolution and aspect ratio of an image. Although bitmaps can be stretched or compressed, this process generally involves duplicating or dropping rows or columns of pixels and can lead to distortion in the scaled image. A metafile can be scaled to almost any size without distortion.

The second major drawback of bitmaps is that they require a large amount of storage space. Metafiles usually require much less storage space than bitmaps. The storage space for a bitmap is governed by the size of the image and the number of colours it contains, whereas the storage space for a metafile is governed by the complexity of the image and the number of individual GDI instructions it contains. One advantage of bitmaps over metafiles, however, is speed. Copying a bitmap to a video display is usually much faster than rendering a metafile.

6.3 COLOURS IN BITMAPS

Each pixel in an image corresponds to one or more bits in a bitmap. A monochrome image requires 1 bit per pixel. A colour image requires more than 1 bit per pixel. The number of different colours that can be represented by a bitmap is equal to 2 to the power of the number of bits per pixel. For example, a 16-colour bitmap requires 4 bits per pixel, and a 256-colour bitmap requires 8 bits per pixel.

The PC platform and its operating systems have come a long way since the early 1980s. Prior to *Windows* 3.0, the only bitmaps supported under *Windows* were GDI objects, which were referenced using a bitmap handle. These bitmaps were either monochrome or had the same colour organisation as a real graphics output device, such as a video display. For example, a bitmap compatible with a VGA monitor had four colour planes. The problem was that those colour bitmaps could not be saved and used on a graphics output device with a different colour organisation.

For *Windows* 3.0, a new bitmap format was defined, called the device independent bitmap, or DIB. The DIB included its own colour table that showed how the pixel

bits correspond to RGB colours. DIBs can be displayed on any raster output device but the DIB colours must be converted to the nearest colours that the device can actually render.

The DIB format is called “device independent” because it contains a colour table. With the introduction of the DIB, the GDI bitmap objects are sometimes called “device dependent bitmaps”. They are device dependent because they must be compatible with a specific graphics output device.

DIBs offer many programming advantages over GDI bitmaps. Because a DIB carries its own colour information, the colour palette management is easier. DIBs also make it easy to control grey shades when printing. Any computer running *Windows* can process DIBs, which are usually stored in BMP disk files or as resources in a program’s EXE or DLL file. The wallpaper background on a computer monitor is read from a BMP file when *Windows* starts. Other graphic interchange formats are available such as TIFF, GIF and JPEG, but only the DIB format is directly supported by the Win32 API.

6.4 THE REQUIREMENTS IN GGS

The MFC class, `CBitmap` encapsulates a GDI bitmap and provides member functions to manipulate the bitmap. But there is no equivalent MFC class to encapsulate the DIBs. This opens the door for others to write their own versions of a `CDib` class. Most of these implementations [DiLa97], [Krug97], [LeAr*98] concentrate on opening BMP files and displaying their contents on the screen. However, GGS requires a lot more functionality in its `CDib` class. The user of GGS should be able to open any BMP file and see the original size of the DIB inside. Then s/he should be able to scale it up or down before placing it in a GGS view. Once imported, a DIB should behave like other shape objects discussed in Chapter 5. The user should be able to import multiple bitmaps from various BMP files, place them side by side or in any sequence, and move them around whenever necessary.

The DIBs should be serializable but they must be retrievable. The user may create a complex image with various shapes, bitmaps and other objects but at a later stage, s/he should be able to retrieve each and every DIB object, and save them in separate BMP files. For example, one user of GGS could scan a drawing and save it in a BMP file. Later on, s/he could import the drawing into GGS and add some graphics primitives and sound objects for illustration and send the whole document as an e-mail attachment. The person at the receiving end could open the e-mail attachment

and animate the illustration and then retrieve the original picture and save in a BMP file.

6.5 THE DIB'S IN GGS

Figure 6.1 presents a part of the revised class hierarchy from Figure 5.1. The `CExternalImage` class has a protected default constructor for serialization only. Otherwise, `CExternalImage` cannot be instantiated. The `CExternalImage` class is supposed to abstract the characteristics of externally created images. At present, it only adds one virtual function for the move operation. In future, if GGS supports other graphic interchange formats, they can be implemented as classes derived from `CExternalImage` and the implementation of some common functionality can be moved up in the class hierarchy.

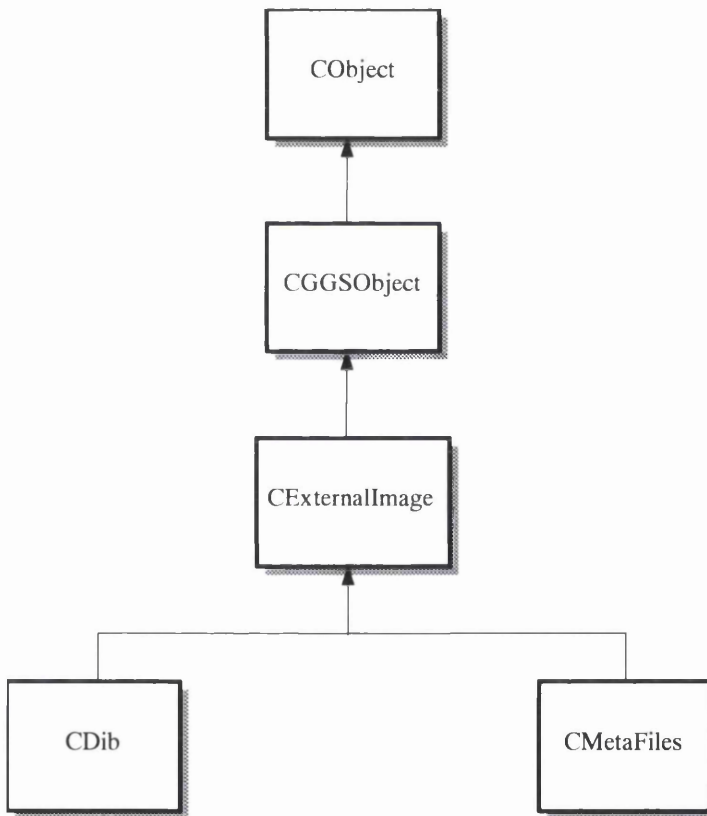


Figure 6.1 Bitmaps and Metafiles in the GGS Class Hierarchy

The best way to get to know the `CDib` class is to look at the public member functions. Listing 6.1 presents the `CDib` class declaration. The inline functions and the data members are omitted from Listing 6.1 so that the class interface does not look intimidating.


```

class CDib : public CExternalImage
{
    DECLARE_SERIAL(CDib)
public:
    CDib(CPoint point) ;
    ~CDib();
    virtual void Move(CSize& aSize);
    virtual void Draw(CDC* pDC, BOOL Select=FALSE);
    virtual void Serialize(CArchive& ar);
    BOOL Read(CFile* pFile);
    BOOL Write(CFile* pFile);
    void Empty();
    UINT UsePalette(CDC* pDC, BOOL bBackground = FALSE);
    BOOL MakePalette();
    BOOL SetSystemPalette(CDC* pDC);
    BOOL Compress(CDC* pDC, BOOL bCompress = TRUE);
    HBITMAP CreateBitmap(CDC* pDC);
protected:
    CDib(); // Default constructor for serialization
    BOOL BitmapSize();
    void ComputePaletteSize(int nBitCount);
    void ComputeMetrics();
};

```

Listing 6.1 The CDib Class Declaration with Some Details Omitted

6.5.1 Construction and Destruction

CDib objects are constructed in the WM_LBUTTONDOWN mouse message handler in the view class, like other objects in GGS. The position of the mouse where the user has decided to create a CDib object is passed on to the public constructor. This position indicates the top-left corner of the bitmap. Other than initialising some data members, the constructor does not do anything else. The general MFC principle of two-stage construction is followed here. Once the object is created, the Read() function is called to deal with the BMP file selected by the user from a standard file open dialogue box.

The destructor simply calls the Empty() function. Empty() does the cleaning up operations. It is also called by other functions to recover from I/O and other error conditions.

6.5.2 Reading a BMP File

Figure 6.2 shows a layout for a BMP file. The `BITMAPFILEHEADER` structure contains the offset to the image bits, which can be used to compute the combined size of the `BITMAPINFOHEADER` structure and the colour table that follows. The `BITMAPFILEHEADER` structure contains a file size member which is not trustworthy. The original documentation in *Windows 3.0* did not specify the unit of file size measurement. Hence, there are several BMP files around with the size measured in bytes, words or double words!

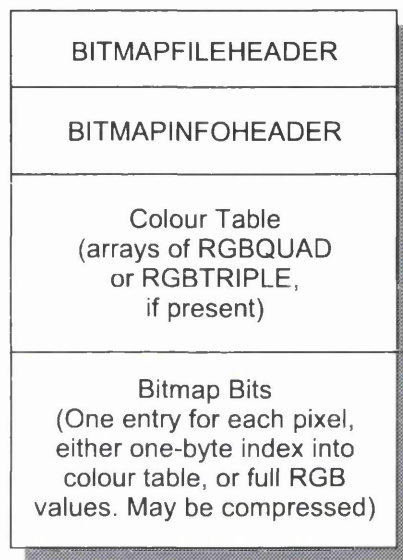


Figure 6.2 BMP File Structure

The `BITMAPINFOHEADER` structure contains the bitmap dimensions, the bits per pixel, compression information for 4-bpp and 8-bpp bitmaps, and the number of colour table entries. If the DIB is compressed, this header contains the size of the pixel array; otherwise, the size can be calculated from the dimensions and the bits per pixel. Immediately following the header is the colour table (if the DIB has a colour table). The DIB image comes after that. The DIB image consists of pixels arranged by column within rows, starting with the bottom row. Each row is padded to a 4-byte boundary.

The `CDib::Read()` function accepts a pointer to a `CFile` object. Firstly, `Read()` verifies the file type and then it tries to read the `BITMAPFILEHEADER` structure at the current file position to establish the size of the `BITMAPINFOHEADER` and the colour table. Secondly, it reads the info header and the colour table. Finally, it reads the image bits after constructing the colour palette, if present. Functions such as `Read()`

in GGS use the `try/catch` loops to trap any exception raised by any I/O error.

As indicated earlier, the user of GGS does not need to accept a DIB at its original size. Before `Read()` returns, it checks a Boolean flag to see if the user defined size is available. When the answer is "no", `Read()` returns `BitmapSize()`.

6.5.3 User Defined DIB Size

`BitmapSize()` presents a dialogue box to the user as shown in Figure 6.3.

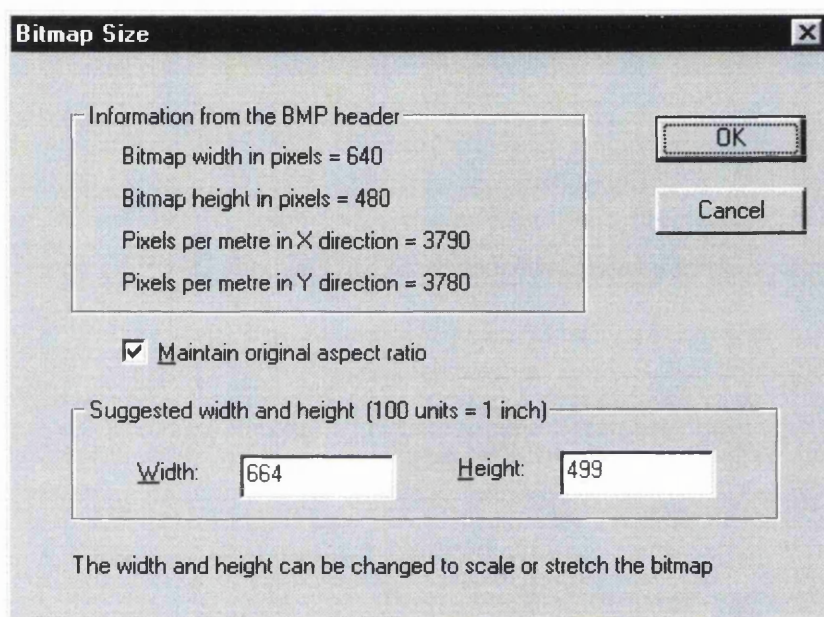


Figure 6.3 The User Can Specify the Bitmap Size

The user gets an opportunity to modify the default bitmap size. But if the suggested width and height values are accepted, the bitmap will appear on the screen at its original size.

There are two edit controls in the dialogue box in Figure 6.3. An edit control is a rectangular control window typically used in a dialog box to permit the user to enter and edit text from the keyboard. When the original aspect ratio of the DIB is to be maintained, the user can either specify the new width or the new height of the DIB. The height and width values in the edit controls are dynamically and synchronously changed. In other words, when the user is typing in one edit control, the other one is continuously updated. Many commercial applications use this trick to impress the user! This involves trapping an edit control message - `EN_UPDATE`. The `EN_UPDATE` notification message is sent when an edit control is about to redraw itself. This notification message is sent after the control has formatted the text, but

before it displays the text. If any other edit controls have to be updated simultaneously, the best place to change them is in the `EN_UPDATE` message handler.

6.5.4 Drawing DIB's

There are several Win32 functions to display a DIB directly on the screen or printer. `SetDIBitsToDevice()` is an example. However, it does not do any scaling; one bitmap bit corresponds to one display pixel or one printer dot. This scaling restriction limits the function's usefulness. `BitBlt()` is similar which copies a bitmap from the source device context to the current device context using logical coordinates. The `CDib` class in GGS uses `StretchDIBits()` that copies the colour data for a rectangle of pixels in a DIB to the specified destination rectangle. If the destination rectangle is larger than the source rectangle, this function stretches the rows and columns of colour data to fit the destination rectangle. If the destination rectangle is smaller than the source rectangle, this function compresses the rows and columns of the DIB. It is worth noting that the `StretchDIBits()` function has been extended to allow a JPEG or PNG image to be processed as the source image [MSDN61]. But this new facility is available only in *Windows 98* and *2000*. This could be used to extend GGS in the near future.

6.5.5 Moving DIB's

When a `CDib` object is underneath the mouse pointer and the right button is pressed, the object is highlighted and a menu pops up with an option to move or delete the `CDib` object. Highlighting a `CDib` object is slightly different from highlighting graphics primitives that are drawn in a different colour when selected. GGS highlights a `CDib` object by drawing a rectangle around it with the highlighting colour.

After selecting a `CDib` object, if the user chooses to move it, the mouse pointer is moved automatically to the centre of the highlighted object. Then the user could simply move (i.e., not click and drag) the pointer and the rectangle around the `CDib` object would move as well, as if glued to the pointer. In this mode, if the left mouse button is clicked, the `CDib` object is moved to the new position. But a right click would cancel the move operation and the highlighting rectangle would disappear. In other words, the original object is not moved until the user has decided its new position. While in the move mode, the temporary rectangle moves with the cursor to provide guidance to the user.

6.6 DIB COMPRESSION

Large bitmaps require a large amount of storage. This puts heavy demands on transmission and display systems. For this reason, various forms of data compression can be used to minimise the storage requirements.

Data compression techniques work on the basis that there are redundant, repeating elements in any data set. These repeating elements are identified and encoded to achieve compression. The degree to which image data can be compressed is image dependent. Images that contain lots of repeated information can be compressed more efficiently than images containing lots of different information.

There are two fundamental types of image compression. Lossless compression allows the image to be reconstituted exactly. Lossy compression discards data that is less important. Both types of compression may be combined to yield high compression ratios.

Lossless data compression schemes generally involve the substitution of shorter codes for common patterns in the image data, thus reducing the amount of storage space required. This can work well for simple graphic images. However, such conventional data compression methods are not very effective for greyscale or colour images. Higher resolution colour images do not generally display simple arithmetic relationships or redundancies among nearby pixels. In the worst case, complex images can result in negative compression, requiring more storage once data compression has been applied.

The simplest form of data compression is known as Run Length Encoding or RLE. In RLE, a run of a certain value is represented by the length of the run followed by the value. Such a simple scheme works well for images with large areas of uniform value.

More complex algorithms include Huffman encoding, Lempel-Ziv Welch or LZW compression, and Arithmetic compression [KaLe94]. These algorithms use statistical methods to replace common values with shorter codes, so achieving higher compression ratios of up to 3:1, although 10:1 is possible for some images. The Arithmetic compression comes close to the theoretical limit for lossless data compression, although it has not been widely adopted due to patent protection [Coop92].

6.6.1 RLE Compression

Windows supplies some important DIB access functions. `GetDIBits()` constructs a DIB from a GDI bitmap. To some extent, this function can control the format of the DIB because the number of colour bits per pixel and the compression type can be specified.

The `Compress()` function in the `CDib` class for GGS can be used for compressing or decompressing DIBs that define their colours with 8 or 4 bits per pixel (bpp). Firstly, `Compress()` calls another member function `CreateBitmap()` to make a GDI bitmap from the existing DIB. `CreateBitmap()` calls the *Windows* function `CreatedDIBitmap()` with a device context pointer. Secondly, `Compress()` makes a new DIB from the GDI bitmap with compression (or decompression). This involves calling `GetDIBits()` twice - once to calculate the memory needed and again to generate the DIB data. And finally, `Compress()` replaces the original DIB with the new one.

6.7 PALETTE PROGRAMMING

Windows palette programming is quite complex and outdated, but it is not sensible to ignore the possibility that the users may run their displays in the 8-bpp mode if they have video cards with memory of 1 MB or less. Video adapters that support 24-bit colour, or *true colour*, are becoming increasingly common. But *Windows* still runs on PC's with video adapters limited to 4 or 8 bpp. Typically, these devices are "palettised" devices, meaning that they support a wide range of colours but can display only a limited number of colours at a time. The common case is a video adapter that can display more than 16.7 million colours in total but only 256 colours at once.

Windows handles palettised devices by pre-programming a standard selection of colours into the adapter's hardware palette. A 256-colour adapter is pre-programmed with 20 static colours. When an application draws on a palettised device, the GDI maps each colour value to the nearest static colour using a simple colour matching algorithm.

For many applications, the primitive form of colour mapping that *Windows* performs using static colours is good enough. But for others, accurate colour output is a foremost concern and 20 colours are not enough. In a single-tasking environment such as MS-DOS, an application running on a 256-colour adapter can program the hardware palette itself and use any set of 256 colours. In *Windows*, applications are

not allowed to program the hardware palette directly because the video adapter is a shared resource. Applications can take advantage of the 236 colours left unused in a 256-colour adapter by using a GDI object known as a logical palette.

A logical palette is a table of RGB colours that tells *Windows* what colours an application would like to display. The term “system palette” refers to the hardware colour palette. At an application’s request, the palette manager built into *Windows* will transfer the colours in a logical palette to unused entries in the system palette - a process known as realising a palette, so that the application can take full advantage of the video adapter’s colour capabilities. With the help of a logical palette, an application running on a 256-colour video adapter can use the 20 static colours plus an addition of 236 colours of its own. And because all requests to realise a palette go through the GDI, the palette manager can serve as an arbitrator between programs with conflicting colour needs and thus ensure that the system palette is used cooperatively.

The palette manager assigns colour priorities based on each window’s position in the z-order. The window at the top of the z-order (the foreground window) receives the top priority, the window that is second gets the next highest priority, and so on. If the foreground window realises a palette of 200 colours, all of them get mapped to the system palette. If a background window then realises a palette of, say, 100 colours, 36 of them get programmed into the system palette entries left over after the foreground application has realised its palette, and the remaining 64 get mapped to the nearest matching colours. However, that may not be the case when these applications have colours in common. Unless directed to do otherwise, the palette manager avoids duplicating entries in the system palette.

6.7.1 Palettes in GGS

Each view object in GGS maintains a logical palette. The palette programming has been made easy by the `CPalette` class in MFC which encapsulates a *Windows* colour palette. The user of GGS has the freedom to create several `CDib` objects from different BMP files and place them anywhere on the screen. When a BMP file is opened and a `CDib` object is created in GGS, and if there is a colour table, the entries are added to the logical palette in the view class, using the `CPalette::SetPaletteEntries()` member function.

16-bpp, 24-bpp or 32-bpp DIB’s do not have colour tables because their image bits contain actual RGB values. To display DIB’s with more than 8-bit colour, it is

important to call the `CreateHalftonePalette()` function because if the application is running on a 256-colour palettised display, and if there is no palette available, only the 20 static colours will appear in the DIB's. `CPalette` provides two member functions for palette creation: `CreatePalette()` creates a custom palette from RGB values specified; `CreateHalftonePalette()` generates a "halftone" palette containing a generic and fairly uniform distribution of colours. Custom palettes give better results when an image contains few distinctly different colours but many subtle variations in tone. Halftone palettes are ideal for images containing a wide range of colours.

6.8 DIB SERIALIZATION

It has been mentioned earlier that the user should be able to retrieve the DIB in its original format from a `CDib` object whenever necessary. Therefore, on demand, a `CDib` object should be able to generate a BMP file from its contents. The `Write()` function writes out a DIB from the `CDib` object to a file which has been successfully opened or created. `Write()` accepts a pointer to a `CFile` object as a parameter just as the `Read()` function. Both of them read or write data at the current file position. This fact has been utilised in the implementation of serialization in the `CDib` class:

```
void CDib::Serialize(CArchive& ar)
{
    CExternalImage::Serialize(ar); // Base class function called
    if(ar.IsStoring()) {
        ar << m_szDibSize
            << m_bBitmapSizeDefined
            << m_ptTopLeftCorner ;
    }
    else {
        ar >> m_szDibSize
            >> m_bBitmapSizeDefined
            >> m_ptTopLeftCorner ;
    }
    ar.Flush() ; // Necessary before direct read/write
    if(ar.IsStoring())
        Write(ar.GetFile());
    else
        Read(ar.GetFile());
}
```

Listing 6.2 The Implementation of Serialization in the `CDib` Class

`CArchive::GetFile()` retrieves the `CFile` object pointer associated with a serialized archive. The `Flush()` member function forces any data remaining in the archive buffer to be written to the file. This ensures that all data is transferred from the archive to the file. Hence, the `Read()` function can be used to examine a BMP file and create a `CDib` object. The function can also be used to read a DIB among other objects in a serialized archive. The `Write()` function behaves exactly the other way around.

6.9 DIFFERENT TYPES OF METAFILES

Like DIB's, the tried-and-true *Windows* metafile has been an invaluable aid to the development of numerous drawing and presentation applications for *Windows*. However, the *Windows* metafile did not address issues related to scalability and device independence. Left on their own, developers attempted to address this issue in various ways. Some developers embedded application, location, or scaling comments in the metafiles. This resulted in extremely nonportable metafiles. Others added headers to the metafile that provided various application-specific information. The net result of most of these efforts was, once again, nonportable metafiles. However, one of these endeavours—placeable metafiles—caught on. Developed by Aldus Corporation, placeable metafiles include a 22-byte header that provides, among other things, mapping and measurement information that can be used to scale the metafile.

The proliferation of the placeable metafile, other home-grown formats, and the confusion of many developers regarding the use of metafiles led to a demand for a metafile format that addressed the development community's needs. Thus the Win32 enhanced metafile was born [Crai93a]. Developed by Microsoft, the enhanced metafile distinguishes itself from the *Windows* metafile in that it is device-independent and not difficult to use [Chat97]. In order to create a *Windows* metafile, developers had to code two paths to deal with the drawing operations. One code path drew on the screen and the second code path drew to the metafile. The only way to get around this was to use a subset of GDI functions that used logical coordinates. Although this permitted limited scaling capabilities, it restricted the use of many helpful GDI functions. It was not possible to query the metafile device context (DC) for information such as window origins and extents. The advent of the enhanced metafile has made those restrictions obsolete. A single code path is all that is required to draw to any DC, whether it be a metafile, screen, or printer DC.

6.9.1 Windows Metafiles Versus Enhanced Metafiles

A *Windows* metafile is used for applications written using the *Windows* version 3.x application programming interface (API). The format of a *Windows* metafile consists of a header and an array of metafile records. *Windows* metafiles are limited in their capabilities and should rarely be used in Win32-based applications. However, they are still supported in Win32 to maintain backward compatibility with applications that use the older *Windows* metafiles.

An enhanced metafile is used in applications written using the Win32 API (Win32s™, however, does not implement enhanced metafiles). The enhanced format consists of a header, a table of handles to GDI objects, a private palette, and an array of metafile records. Enhanced metafiles provide true device independence. The picture stored in an enhanced metafile can be visualised as a snapshot of the video display taken at a particular moment. This snapshot maintains its dimensions no matter where it appears: on a printer, a plotter, the desktop, or in the client area of any Win32-based application.

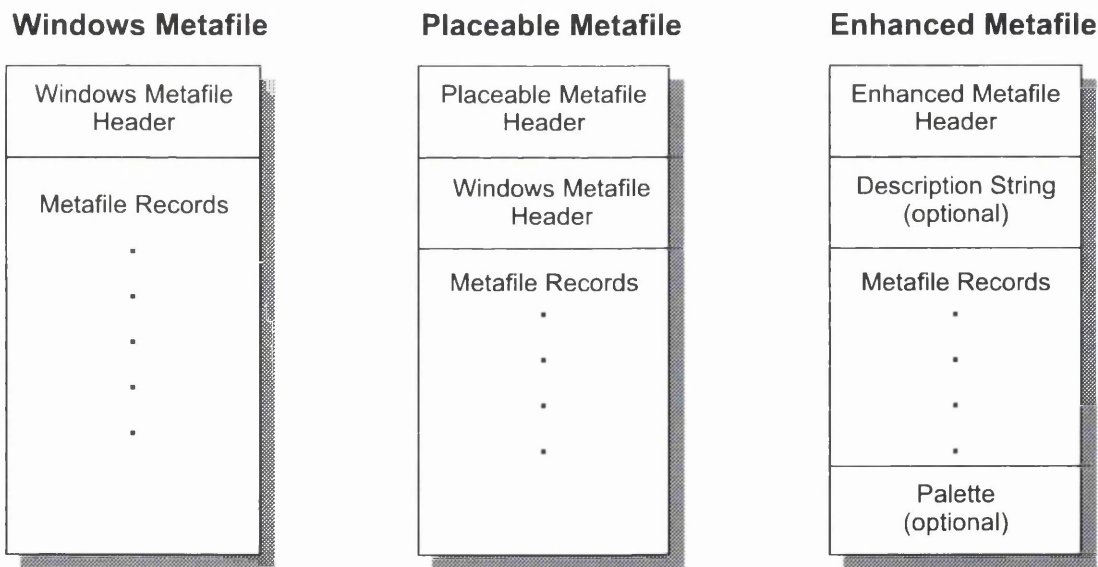


Figure 6.4 Three Different Metafile Formats

At first glance, *Windows* metafiles and enhanced metafiles may appear to share the same overall structure. They have an array of variable length structures called metafile records. The first record in a metafile specifies general information such as the resolution of the device on which the picture was created, the dimensions of the picture, and so on. The remaining records, which constitute the bulk of a metafile, correspond to the GDI functions required to draw the picture.

However, a closer inspection reveals a number of differences between a *Windows* metafile and an enhanced metafile, as presented in Figure 6.4. Unlike the *Windows* metafile format, the enhanced metafile has a different header and may include a description string and an optional palette stored in a special end-of-file record. The enhanced metafile format also provides support for additional types of records.

6.9.2 Device Independence

Achieving device independence was very difficult, if not impossible, with *Windows* metafiles. The *Windows* metafile header has the following form:

```
typedef struct tagMETAHEADER {
    WORD    mtType;
    WORD    mtHeaderSize;
    WORD    mtVersion;
    DWORD   mtSize;
    WORD    mtNoObjects;
    DWORD   mtMaxRecord;
    WORD    mtNoParameters;
} METAHEADER;
```

It contains only size and version information. The placeable variant of the *Windows* metafile had the best shot at this. The additional header in a placeable metafile provided an opportunity for an application to render the metafile in a device relative way:

```
typedef struct tagPLACEABLEMETAFILEHEADER {
    DWORD   key;
    HANDLE  hmf;
    RECT    bbox;
    WORD    inch;
    DWORD   reserved;
    WORD    checksum;
} PLACEABLEMETAFILEHEADER;
```

Device independence was typically achieved by setting the mapping mode to `MM_ANISOTROPIC`, setting the viewport extents to the physical dimensions of the device, and finally setting the window's extents to the product of the device's physical dimensions (in inches) and the metafile units per inch (contained in the `inch` member of the header structure). The biggest problem with that approach was the variants of the placeable *Windows* metafile that began surfacing. Often the

mapping mode and the viewport extents were included in the metafile as records. This necessitated enumerating the metafile as a method of filtering out undesirable records. Unfortunately, the bounding box and the metafile unit per inch often did not match the environment being set by the undesirable metafile records! This led to the situation in which even the placeable metafiles were, once again, application-specific.

Device independence is a key feature of enhanced metafiles. [Micr93] states that "...when an application creates a picture measuring 2 inches by 4 inches on a VGA display and stores that picture in a metafile, it (the picture) will maintain those original dimensions when it is printed on a 300 dpi laser printer or copied over a network and displayed in another application that is running on an 8514/A video display". The key to achieving this device independence lies inside the enhanced metafile header:

```
typedef struct tagENHMETAHEADER
{
    DWORD   iType;           // Record type EMR_HEADER.
    DWORD   nSize;          // Record size in bytes. This may be greater
                          // than the sizeof(ENHMETAHEADER).
    RECTL   rclBounds;     // Inclusive-inclusive bounds in device units.
    RECTL   rclFrame;     // Inclusive-inclusive Picture Frame of
                          // metafile in .01 mm units.
    DWORD   dSignature;    // Signature. Must be ENHMETA_SIGNATURE.
    DWORD   nVersion;      // Version number.
    DWORD   nBytes;        // Size of the metafile in bytes.
    DWORD   nRecords;      // Number of records in the metafile.
    WORD    nHandles;      // Number of handles in the handle table.
                          // Handle index zero is reserved.
    WORD    sReserved;     // Reserved. Must be zero.
    DWORD   nDescription;  // Number of chars in the description string.
                          // Zero if there is no description string.
    DWORD   offDescription; //Offset to the metafile description record.
                          // Zero if there is no description string.
    DWORD   nPalEntries;   // No of entries in the metafile palette.
    SIZEL   szlDevice;     // Size of the reference device in pixels.
    SIZEL   szlMillimeters; //Size of the reference device in mm.
    DWORD   cbPixelFormat; // Size of the last recorded pixel format.
    DWORD   offPixelFormat; // Offset of the last pixel format.
}
```

```

    DWORD bOpenGL;           // TRUE if any OpenGL records are present.
} ENHMETAHEADER;

```

Listing 6.3 The Enhanced Metafile Header

The enhanced metafile header contains dimension and resolution information, as well as size and version information. To achieve device independence, a reference device context is used. The reference device context is where the picture was formed. When an enhanced metafile is created, information regarding the reference DC is placed in its header. More specifically, GDI calls `GetDeviceCaps()` and assigns the `HORZSIZE` and `VERTSIZE` return values to `szlMillimeters` and assigns the `HORZRES` and `VERTRES` values to `szlDevice`. The `rclFrame` member is assigned the bounding rectangle specified in the `lpRect` parameter of `CreateEnhMetaFile()`. If `lpRect` is `NULL`, GDI determines the bounding rectangle and assigns it to `rclFrame`. This information is sufficient to enable the playback functions to achieve device independence. When a metafile is played back, the picture undergoes a series of transformations that scale and translate the picture to the output rectangle specified in the call to the `PlayEnhMetaFile()` or `EnumEnhMetaFile()` functions. These transformations rely on the dimensions of the picture frame (`rclFrame`), the dimensions of the device upon which the metafile was created (`szlMillimeters` and `szlDevice`), and the world-to-page transformation values set in the destination DC [Crai93b].

6.10 METAFILES IN GGS

`CMetaFile` was an old class in MFC to support very basic metafile capabilities. `CMetaFiles` in GGS is very different - it supports all three types of metafiles discussed in Section 6.9. The main public members and some protected members of the `CMetaFiles` class are presented in Listing 6.4. Once again, the inline functions and the data members are omitted for clarity:

```

class CMetaFiles : public CExternalImage
{
    DECLARE_SERIAL(CMetaFiles)
public:
    CMetaFiles(CPoint point);
    ~CMetaFiles();
    virtual void Move(CSize& aSize);
    virtual void Draw(CDC* pDC, BOOL Select=FALSE);
    BOOL Read(CFile* pFile);

```

```

    BOOL Write(CFile* pFile);
    void Serialize(CArchive& ar);
protected:
    CMetaFiles(); // For serialization
    BOOL CreateTempFile();
    BOOL MetaFileSize();
    BOOL CleanupOpenGL();
    BOOL PrepareOpenGL(HDC hDC, HENHMETAFILE hMeta);
    BOOL CopyBytesToTempFile(CFile* pFile, DWORD dwLength);
};

```

Listing 6.4 Main Public and Protected Member Functions in CMetafiles

6.10.1 Construction and Destruction

Objects of the `CMetaFiles` class are constructed exactly in the same way as `CDib` objects. The user decides the position of the top-left corner of the metafile to be imported by clicking the mouse. Then a standard file open dialog box is presented to help selecting the file. The public constructor initialises some data members and the `Read()` function deals with the WMF or EMF file selected by the user.

6.10.2 Reading a Metafile

The `Read()` function in the `CMetaFiles` class uses a number of *Windows* API functions. First of all, `Read()` assumes that the file selected by the user contains an enhanced metafile. The `GetEnhMetaFile()` API is called which creates a handle that identifies the enhanced-format metafile stored in the specified file. If this API function succeeds, the return value is a handle to the enhanced metafile. Otherwise, the return value is `NULL`. When the call is successful, `GetEnhMetaFileHeader()` retrieves some header information and then, a dialogue box, very similar to the one in Figure 6.3, is thrown so that the user can specify the size of the metafile on the screen. Unlike bitmaps, metafiles can be scaled up or down without any distortion unless they contain bitmaps inside!

However, the user has the full freedom to select old *Windows* metafiles. When a *Windows* metafile is passed to `GetEnhMetaFile()`, the return value is `NULL`. In that case, the `Read()` function calls `GetMetaFile()`. `GetMetaFile()` does not form part of the Win32 API. This function is provided for compatibility with 16-bit versions of *Windows*. As the name suggests, `GetMetaFile()` creates a handle for the given *Windows*-format metafile.

In the next step, `Read()` calls `GetMetaFileBitsEx()` to copy the contents of the *Windows* metafile into a buffer space. The `SetWinMetaFileBits()` API function then creates an enhanced metafile from the data in the buffer and stores the new metafile in memory.

Finally, `Read()` considers the possibility of encountering a placeable metafile. `GetMetaFile()` cannot recognise a placeable metafile because of its additional header. When `GetMetaFile()` returns `NULL`, the `Read()` function checks the file signature in the first 22 bytes of the file. If the signature matches that of a placeable metafile, `Read()` treats the rest of the file as an old *Windows* metafile. The user, however, always gets an option to specify the onscreen size of the metafile irrespective of its format.

6.10.3 Drawing a Metafile

It is a common practice to enumerate *Windows* metafiles, rather than simply to play them back, to achieve better control over positioning, scaling, getting access to application-specific comments, or manipulating the palette records. However, the improvements to enhanced metafiles reduce the need for enumeration of the metafile. In Win32, most applications need to use only `PlayEnhMetaFile()` unless they need to edit the enhanced metafile by adding, deleting, or modifying records, in which case they should use `EnumEnhMetaFile()`.

`PlayEnhMetaFile()` is sufficient for GGS. The GDI metafile player encapsulates all the details inside `PlayEnhMetaFile()`. The `Draw()` member in the `CMetaFiles` class calls `PlayEnhMetaFile()`. It is reasonable to think that the implementation of the `Draw()` function is simple and straightforward. In fact, the implementation was simple when enhanced metafiles did not contain OpenGL records.

6.10.4 Basic OpenGL Operations

Originally developed by Silicon Graphics Inc., OpenGL is an industry-standard procedural software interface for producing 3-D graphics. The OpenGL interface provides around 120 commands to draw various primitives including points, lines, and polygons in various modes [Open92]. As a software interface for graphics hardware, OpenGL's main purpose is to render two and three dimensional objects into a frame buffer. These objects are described as sequences of vertices or pixels. OpenGL performs several processing steps on these objects to convert them to pixels to form the final desired image in the frame buffer.

The high-level block diagram in Figure 6.5 illustrates how OpenGL processes the image data. In the diagram, commands enter from the left and proceed through what can be considered a processing pipeline. Some commands specify geometric objects to be drawn, and others control how the objects are handled during various processing stages.

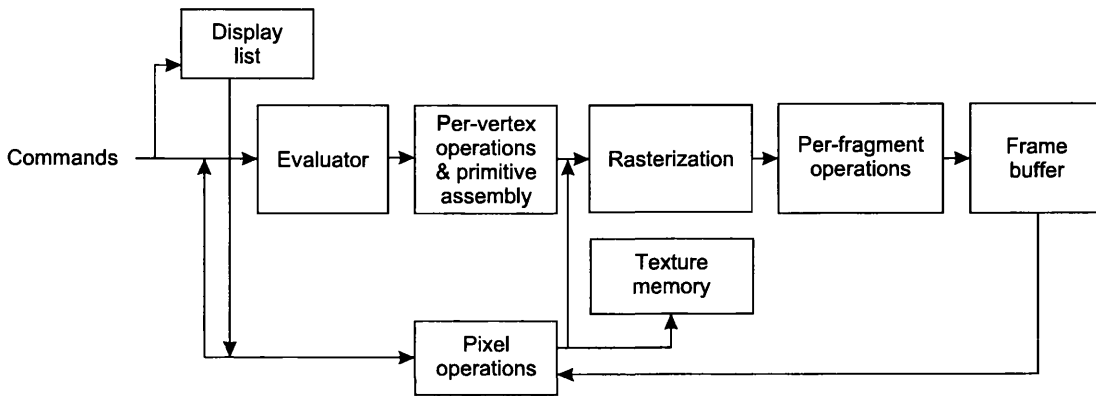


Figure 6.5 Basic OpenGL Operations

The processing stages in basic OpenGL operations are as follows:

- **Display list:** Rather than having all commands proceed immediately through the pipeline, some of them can be accumulated in a display list for processing later.
- **Evaluator:** The evaluator stage of processing provides an efficient way to approximate curve and surface geometry by evaluating polynomial commands on input values.
- **Per-vertex operations and primitive assembly:** OpenGL processes geometric primitives – points, line segments, and polygons – all of which are described by vertices. Vertices are transformed and lit, and primitives are clipped to the viewport in preparation for rasterisation.
- **Rasterisation:** The rasterisation stage produces a series of frame-buffer addresses and associated values using a two dimensional description of a point, line segment or polygon. Each fragment so produced is fed into the last stage, per-fragment operations.
- **Per-fragment operations:** These are the final operations performed on the data before storing it as pixels in the frame buffer.

Per-fragment operations include conditional updates to the frame buffer based on incoming and previously stored *z* values (for *z* buffering) and blending of incoming

pixel colours with stored colours, as well as masking and other logical operations on pixel values.

6.10.5 Enhanced Metafiles with OpenGL Records

In Listing 6.3, the last member (i.e., `bOpenGL`) of the `ENHMETAHEADER` structure indicates whether any OpenGL record is present in an enhanced metafile. In fact, the last three members are added to the `ENHMETAHEADER` structure later so that EMF files can capture OpenGL images. The other two members, `cbPixelFormat` and `offPixelFormat`, as their names suggest, specify the recorded pixel format(s) in a metafile which is very important for an OpenGL rendering.

There are two major steps before an OpenGL rendering can take place. Firstly, the pixel format of the device must be set up where the OpenGL image will be drawn. Secondly, a rendering context should be created.

The `Read()` function in the `CMetaFiles` class, raises a Boolean flag when it encounters an enhanced metafile with OpenGL records. The `Draw()` function checks the flag and calls `PrepareOpenGL()` which attempts to set up the target DC for OpenGL rendering based on the metafile's `PIXELFORMATDESCRIPTOR`. After preparing the target DC, the metafile is played using the `PlayEnhMetaFile()` API. A pointer to a `RECT` structure that contains the coordinates of the bounding rectangle for the display of the metafile is required as a parameter to `PlayEnhMetaFile()`. This bounding rectangle is constructed from the top-left corner and the metafile size specified by the user. Finally, `CleanupOpenGL()` is called to free the rendering context and the OpenGL libraries loaded by `PrepareOpenGL()`.

`CMetaFiles::PrepareOpenGL()` in GGS is based on the sample code provided by Microsoft in [MSDN61]. The OpenGL rendering to enhanced metafiles does not work on all PC's running *Windows 95*. The original version of *Windows 95* was not OpenGL aware. In the OEM system release 2, *Windows 95* was updated with the OpenGL 1.1 implementation [MSDN62, MSDN63]. OpenGL first became part of a Microsoft operating system when *Windows NT* version 3.5 was released [Crai94].

It is therefore important to examine the operating system modules for OpenGL support before setting up a rendering context. `PrepareOpenGL()`, at first, tries to load some OpenGL dynamic link libraries (DLL's) and checks if appropriate functions are available. It also examines one of the core OS modules, `GDI32.DLL` to see if the module exports the `GetEnhMetaFilePixelFormat()` function. Special

programming techniques are necessary to query a DLL. The following code fragment is from [MSDN61] which was written in C:

```
FARPROC fnPointer = NULL; // A function pointer
m_hOpenGL32 = LoadLibrary("OpenGL32.dll");
if (! m_hOpenGL32)
    return FALSE;
fnPointer = GetProcAddress(m_hOpenGL32, "wglDeleteContext");
(fnPointer)(m_hRC); // m_hRC is an OpenGL HANDLE
```

The `GetProcAddress()` function can retrieve addresses of exported functions in DLL's. However, the C style function pointer casting is generally not accepted by the C++ compilers because they cannot do the type checking. Hence, the above code fragment has been rewritten in `CMetaFiles::PrepareOpenGL()` as follows:

```
BOOL (*fnPointer)(HGLRC) = NULL;
m_hOpenGL32 = LoadLibrary("OpenGL32.dll");
if (! m_hOpenGL32)
    return FALSE;
fnPointer = (BOOL (*)(HGLRC))GetProcAddress(m_hOpenGL32,
    "wglDeleteContext");
fnPointer(m_hRC);
```

6.11 CLOSURE

Other member functions such as `Move()`, `Write()` and `Serialize()` in the `CMetaFiles` class essentially follow the same principles as those in the `CDib` class. Although DIB's and metafiles represent two very different ways of storing pictorial information, GGS treats them as similar objects but with different file formats.

Now the user can open one or more externally created bitmaps and metafiles in GGS and arrange them in any sequence suitable for his/her illustration. Figure 6.6 presents some alien characters playing music somewhere near Loch Ness!

Full-blooded illustration programs can produce such an image very easily. They can also do a lot more with their advanced editing features. However, the motivation for developing GGS is slightly different. GGS allows users to construct and manipulate a fairly complex picture, adding comments as they go. The process of constructing the picture is saved, not just the final picture. There is also an opportunity to add music as we shall see in the next chapter.



Figure 6.6 Alien Music Somewhere Near Loch Ness!

6.12 REFERENCES

- [Chat97] Chatterjee, P.C.: "Exporting graphics from UNIX-based FE software to *Windows*", *Pro/E: The Magazine*, Vol. 5, No. 6, 1997.
- [Coop92] Cooper, W.: "Image Formats: An Introduction to Image Formats for Use in Computer Based Learning and Multimedia", <http://cblmsu.leeds.ac.uk/WWW/projects/cblmsu/images.html>.
- [Crai93a] Crain, D.: "Enhanced Metafiles in Win32", Microsoft Developer Network Technology Group, June 1993.
- [Crai93b] Crain, D.: "EMFDCODE.EXE: An Enhanced Metafile Decoding Utility", Microsoft Developer Network Technology Group, July 1993.
- [Crai94] Crain, D.: "Windows NT OpenGL: Getting Started", Microsoft Developer Network Technology Group, April 1994.

- [DiLa97] DiLascia, P.: "More Fun with MFC: DIBs, Palettes, Subclassing, and a Gamut of Reusable Goodies", *Microsoft Systems Journal*, Vol. 12, No. 1, 1997.
- [KaLe94] Kay, D.C. and Levine, J.R.: "Graphics File Formats", Windcrest/McGraw-Hill, August 1994.
- [Krug97] Kruglinski, D.J.: "Inside Visual C++", 4th Edition, Microsoft Press, 1997.
- [LeAr*98] Leinecker, R.C., Archer, T., et al.: "Visual C++ 6 Programming Bible", IDG Books Worldwide Inc., 1998.
- [Micr93] Microsoft Corporation: "Microsoft Windows NT Resource Guide", Vol. 1, October 1993.
- [MSDN61] Microsoft Developer Network Library: "SAMPLE: How to Create & Play Enhanced Metafiles in Win32", Article ID: Q145999, July 1999.
- [MSDN62] Microsoft Developer Network Library: "SAMPLE: OpenGL 1.1 Release Notes & Components", Article ID: Q154877, July 1999.
- [MSDN63] Microsoft Developer Network Library: "INFO: README for Win32 Software Development Kit, Part 1 of 2", Article ID: Q167799, July 1999.
- [Open92] OpenGL Architecture Review Board: "OpenGL Reference Manual, The Official Reference Document for OpenGL", Release 1, Addison Wesley, 1992.

CHAPTER 7

SOUND, TIMERS AND STILL ANIMATION

7.1 AUDIO CLIPS

Sound can be an effective form of information and interface enhancement when appropriately used. Any presentation material comes to life with audio clips. This is because audio clips can serve purposes other than transmission of details or factual information. A greeting or a recitation of a short poem in both text and audio might help emphasise the tone of the author of the page as effectively as the layout, colours, and images on the page. All of us know how pervasive the effect of musical score is on the way we experience a movie. The analogy is not exact, but short audio sections might help create the tone that we hope for in presenting ideas, opinions, facts, and/or art.

7.2 DIGITAL SOUND

The number of applications for high quality audio functions on the PC, including music synthesis, grew explosively after the introduction of *Windows 3.0* with multimedia extensions ("*Windows with Multimedia*") in 1991. These extensions are also incorporated and enhanced in the 32-bit versions of *Windows*. The Multimedia PC (MPC) specification, originally published by Microsoft in 1991 [Micr91a], states the minimum requirements for multimedia-capable PC's to ensure compatibility in running multimedia applications. The audio capabilities of an MPC system must include digital audio recording and playback (linear PCM sampling), music synthesis and audio mixing.

Windows applications address hardware devices such as the Musical Instrument Digital Interface (MIDI) or synthesisers through the use of drivers. The drivers provide software applications with a common interface through which hardware may be accessed, and this simplifies the hardware compatibility issues. Multimedia applications store audio data in different file formats. At present, MP3 is the most popular format for downloaded music. In fact, MP3 is the second most frequently requested search term on the Internet search engines [Hedt99].

7.2.1 What is MP3?

The Motion Picture Experts Group (MPEG) is a set of standards for compressing and storing digital audio and video. MP3 is an abbreviation of MPEG Audio Layer 3, and it identifies a way to store digital audio files. MP3 files offer high-quality sound in a file format that requires roughly 1 MB for every minute of sound. CD's and WAVE files, by contrast, require about 11 MB per minute. This means that a single song or track in the MP3 format usually takes up between 3MB and 5MB - a reasonable

download even at a speed of 28.8 Kbit/s. Because of this, a profusion of MP3 Web sites, newsgroups and FTP sites has sprouted up across the Internet. It also means that one can create a DVD disc containing over 80 hours' worth of music.

Regardless of where a sound comes from, what we actually hear is analogue. Computers translate and store this information as digital sound, however. This is done through sampling – the process of taking a snapshot of the sound many times per second. Audio CD's store information in a digital audio format known as CD-DA, which is very similar to the standard WAVE format and samples the analogue source 44,100 times per second.

The compression techniques used to create MP3 files are based on psychoacoustics, the study of how the human brain perceives sound. This science has determined that not all of the sound we hear is perceived by the brain. To create an MP3 file, an MP3 encoder reads a WAVE file and then strips out the parts that we will not miss hearing, at least in theory. For example, most adults experience attenuated hearing response at high frequencies, so the encoder strips out any sounds above a preset threshold frequency, say above 16kHz. Loud sounds will tend to mask quieter sounds at or near the same frequency, so the encoder removes these too. By whittling away the parts we are less likely to miss, the encoder creates a file that sounds very similar to the original, but is dramatically smaller.

MP3 files are not illegal just because they are MP3 files, but there are many files around that violate someone's copyright and are, therefore, illegal. Almost everything found on MP3 newsgroups falls into this category and record companies worldwide are starting to take action against what they perceive as a huge loss of revenue. As a result of all this attention, most of the large MP3 sites are distributing only authorised MP3 files. This still lets people download files as music shareware and if they like the music, then they can buy the CD.

7.2.2 WAVE and MIDI Files

Before MP3 was born, WAVE and MIDI were the most popular audio file formats for PC's. WAVE and MIDI files contain coded sound information. When played back in a modern system, equipped with proper software and hardware, they can result in impressive sound. Nevertheless, the type of information and the way they are coded are quite different, in a degree that an attempt for comparison could seem almost meaningless to specialists.

A WAVE file contains information on the 'form' of sound, obtained by digitisation of some analogue signal. A WAVE file is basically a digital copy of the air pressure alterations of a recorded sound or an artificial sound-like signal (i.e., synthesised sound). Playing back this copy may result in some quality degradation (harmonic and dynamic distortions, added noise, pitch deviation, etc.), but not in the modification of the basic form of the sound signal. A WAVE-coded record of someone's voice will sound as his/her voice and not something else.

A MIDI file, on the other hand, has information on the musical contents of sound, expressed in terms similar to those used in the note score. The data refer to the pitch, duration and volume of different notes, the instrument which they have to be played with, the way they are supposed to sound (vibrato, echo, reverberation, sustain, etc.). Playing back a MIDI file may result in quite different sound pictures, depending on the synthesiser – the device producing sound under the control of the MIDI file data, but the basic music information will be preserved in any case. A MIDI file essentially contains a sequence of bytes representing various MIDI events of a musical composition, which can be directed to a synthesiser or can be translated to a perfect staff notation to be used in printed form by an interpreter to play the music.

In the context of developing GGS, audio clips of human voice are more important. The handling of WAVE files is more appropriate. MIDI files are mainly for music and MP3 is a lossy compression of WAVE files. Writing an MP3 encoder and a player from scratch is not only challenging and difficult but also beyond the scope of this research.

7.2.3 The Contents of a WAVE File

A WAVE file consists of a sequence of bytes representing the amplitude of the sound signal in consequent time moments close enough to represent its form with acceptable precision. These bytes result from a sound digitisation process. There are three parameters that determine the inherent quality of the digital copy:

- **Sampling frequency (F):** The sampling frequency is equal to the number of samples per second. F determines the maximum signal frequency ($= F/2$) that can be digitally coded. Since the (young) human ear perceives frequencies up to 20 kHz, an F value higher than 40 kHz is required to avoid audible spectral degradation of the sound. Three standard F values are used: 11025 Hz, 22050 Hz and 44100 Hz, providing digital copies with frequency spectrum extending to 5, 10 and 20 kHz respectively.

- **Sampling accuracy:** The sampling accuracy or “depth” depends on the number of bits per sample, which determines the number of discrete steps available to represent the amplitude of each sample. Two basic depths are used in WAVE files. The 8-bit (i.e., 1 byte per sample) depth provides 255 steps (-127 to 127) of amplitude digitisation, resulting in amplitude accuracy of about $\pm 0.4\%$ of the highest signal level. The 16-bit (i.e., 2 bytes) depth creates 65535 (-32767 to 32767) steps and amplitude accuracy about $\pm 0.0015\%$ of the highest signal level. The 8-bit sampling results in some degradation of sound, expressed in audible noise and nonlinear distortion, while the 16-bit sampling virtually creates noise- and distortion-free copy.
- **Sound channels:** The number of sound channels recorded is also important. There are mono and stereo WAVE files, the last containing digitised data of two sound channels.

The sound obtained by various combinations of the above parameters (mainly F) is often characterised as being of “AM”, “FM” or “CD” quality, which are good approximations with respect to the frequency spectrum, but not always adequate with respect to other sound quality parameters. The final sound quality depends not only on the contents of a WAVE file, but also on the quality of the whole recording and playback system. Nevertheless, a 44 kHz / 16-bit / stereo WAVE file can sound very good if played back on a top-level hardware. The above parameters coincide with those used in CD’s, while the 22 kHz / 16-bit quality is slightly inferior to the actual FM sound quality. The 22 kHz / 16-bit / mono is possibly the bottom-level combination which is still acceptable in sound quality terms [INET71].

7.3 WAVEFORM AUDIO AND WINDOWS

There are several ways to play waveform audio in *Windows*: using PlaySound API, using the Media Control Interface (MCI) or the low-level audio services. Recording of waveform audio can be done through MCI or the low-level audio services.

7.3.1 The PlaySound API

The PlaySound API plays waveform audio, as long as the sound file fits into the available memory. The sound source can be specified in three different ways:

- As a system alert, using the alias stored in the system registry
- As a filename
- As a resource identifier

If the source specified is a file and it does not fit into the available memory, `PlaySound` plays the default system sound. If no default system sound has been defined, `PlaySound` fails without producing any sound. Hence, it is suitable for simple applications dealing with small files or resources.

7.3.2 Low-level Audio Services

Playing waveform audio using the low-level services involves the application in opening an output device and sending a series of one or more blocks of waveform data to the device. The output device driver sends a notification message to the application each time a block has finished playing.

In order to record waveform audio, an application must supply a series of data buffers to the wave device driver. The device driver fills the buffers with data as it becomes available, and when each buffer is full, it posts a message to the application saying that the buffer is full and the application may now process it.

This is rather an over-simplified description of the process. Using the low-level services provides the most control over what is going on during recording and playback but most applications such as GGS do not need to deal with audio data at the buffer level or deal directly with the audio device drivers either.

7.4 THE MEDIA CONTROL INTERFACE (MCI)

The Media Control Interface (MCI) provides applications with device independent capabilities for controlling devices such as audio and visual peripherals. There are two MCI interfaces that can be used to communicate with MCI devices: command-message functions and command-string functions. Either set of functions can be used to access all MCI device capabilities. The difference between the two interfaces is in their basic command structure and the method in which they pass information to the devices.

The command-message interface uses messages to control MCI devices. A bit-vector of flags and a pointer to a data structure are sent with each message. The flags and information data structure let an application send information to a device and receive returned data. MCI passes device messages and information directly to the device.

The command-string interface uses text commands to control the MCI commands. The text strings contain all the information needed to execute a command. MCI

parses the text string and translates it into the message, flags, and the data structure to be sent to the command-message interface. Because of this process, this interface is slightly slower than the command-message interface.

Each interface has unique properties. The command-message interface is more versatile if an application controls an MCI device directly. If this is the case, the application can directly and easily manipulate and decode data used by this interface. For example, an application can play an audio or video segment when the user successfully completes a task. GGS uses the command-message interface to control the recording and playback devices.

The command-string interface should be selected if an application uses a text-based interface to let the user control an MCI device. In such an application, the user can easily read and create the necessary command strings. For example, the application might read a user-written script that controls some MCI devices. The MCI commands in the script can be sent directly to MCI without intermediate processing by the application.

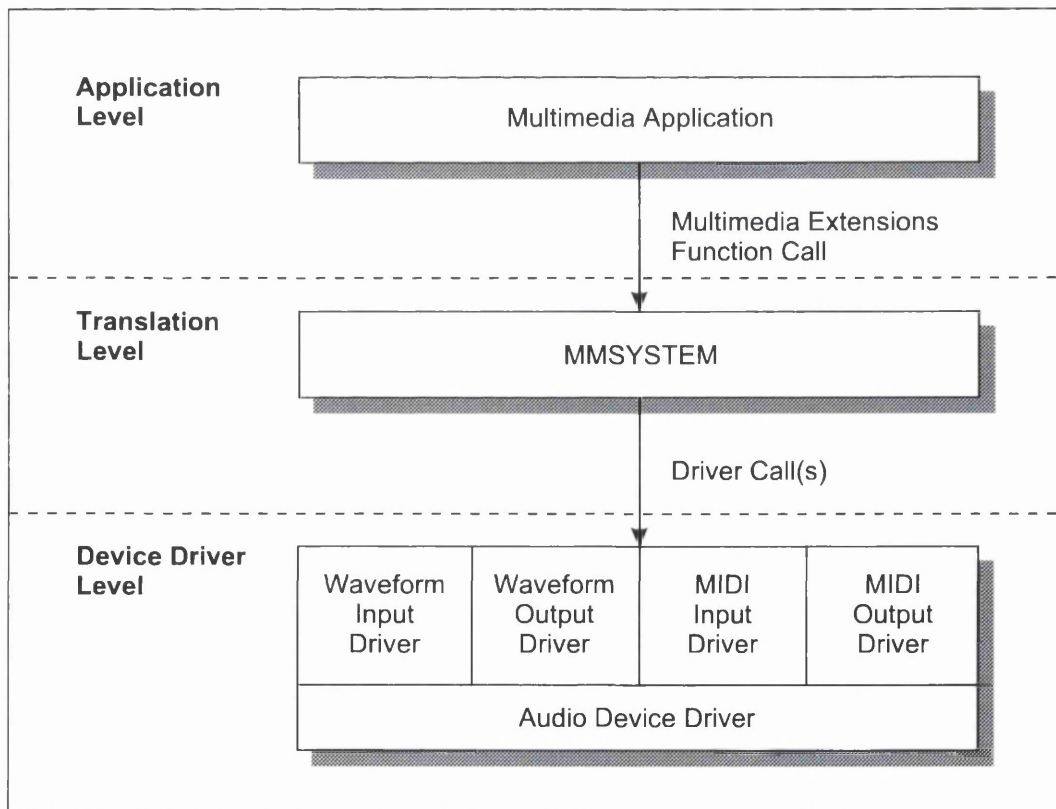


Figure 7.1 Relationship between an Application and Multimedia Device Drivers

7.5 MCI AUDIO ARCHITECTURE

The MCI audio architecture is designed around the concepts of extensibility and device-independence. Extensibility allows the software architecture to accommodate advances in technology without changes to the architecture itself. Device-independence allows multimedia applications to be developed that will run on a wide range of hardware providing different levels of multimedia support.

Three design elements of the system software provide extensibility and device-independence:

- A translation layer (MMSYSTEM) that isolates applications from device drivers and centralises device-independent code.
- Run-time linking that allows the MMSYSTEM translation layer to link to the drivers it needs.
- A well-defined and consistent driver interface that minimises special-case code and makes the installation and upgrade process easier.

Figure 7.1 illustrates how the translation layer translates an MCI function call into an audio device driver call. Some function calls might result in multiple driver calls, or they might be handled by MMSYSTEM without causing any driver calls.

7.6 SOUND OBJECTS IN GGS

The main public members and some protected members of the `CSound` class are presented in Listing 7.1. Once again, the inline functions and the data members are omitted for clarity:

```
class CSound : public CGGSObject
{
    DECLARE_SERIAL(CSound)
protected:
    BOOL CalculateFileSize() ;
    BOOL Read(CFile* pFile) ;
    BOOL CreateTempFile() ;
    BOOL Check() ;
    BOOL Import() ;
    BOOL Record() ;
public:
    CSound() ;
```

```

virtual ~CSound() ;
BOOL CreateSound() ;
BOOL Play() ;
BOOL Write(CFile* pFile) ;
virtual void Serialize(CArchive& ar) ;
};

```

Listing 7.1 The CSound Class Declaration with Some Details Omitted

The general MFC principle of two-stage construction is followed when CSound objects are created in GGS. Other than initialising some data members, the constructor does not do anything else. Once the object is created, the CreateSound() function should be called. CreateSound() offers two alternative ways of creating sound objects by displaying the following dialogue box:



Figure 7.2 Options for Selecting the Sound Source

7.6.1 Importing WAVE Files

If the user decides to import sound, GGS checks the selected WAVE file before creating a CSound object. A WAVE file is one type of RIFF (Resource Interchange File Format) file. RIFF is the tagged file structure developed for multimedia resource files. The basic building block of a RIFF file is called a chunk, which looks like the following:

```

typedef unsigned long    DWORD;
typedef unsigned char    BYTE;
typedef DWORD            FOURCC; // Four-character code

typedef struct {
    FOURCC ckID;
    DWORD cksize; // The size of field <ckdata>

```

```

    BYTE ckData[ckSize]; // The actual data of the chunk
} CK;

```

Four-character codes (FOURCC) are used extensively in RIFF files; they identify the sections of data contained in the file. A four-character code has the following characteristics:

- A 32-bit quantity represented as a sequence of one to four ASCII alphanumeric characters.
- Padded on the right with blank characters (ASCII character value 32).
- Contains no embedded blanks.

For example, the four-character code "RIFF" is stored as a sequence of four bytes ('R' 'I' 'F' 'F') in ascending addresses. For quick comparisons, a four-character code may also be treated as a 32-bit number. Two types of chunks, "LIST" and "RIFF" [Micr91b] may contain nested chunks, or subchunks.

A RIFF form is a chunk with a "RIFF" chunk ID. The first DWORD of chunk data in the "RIFF" chunk is also a four-character code identifying the data representation, or the 'form type' of the file. Following the form type code is a series of subchunks. Which subchunks are present depends on the form type. For a WAVE file, the form type is 'WAVE'. This is followed by the format chunk and then the data chunk. The format chunk must always occur before the data chunk. Programs must expect and ignore any unknown chunks encountered, as with all RIFF forms.

`CSound::Import()` calls the `Check()` member to verify the WAVE file selected by the user. In other words, GGS does not assume that a file with .WAV extension is a WAVE file. `Import()` returns the `Play()` function so that the user has an opportunity to listen to the imported file.

7.6.2 Playing WAVE Files

`CSound` member functions use the `mciSendCommand()` API to send a command message to the MCI devices. When an MCI driver receives a command, by default it should start the operation and then return control to the calling application. The driver should not wait for the operation to complete before returning. For example, if an application sends an `MCI_PLAY` command, the driver should start the play operation and immediately return. Optionally, an application sending any MCI command can request the driver to wait until the associated operation is complete

before returning. The application makes this request by including the `MCI_WAIT` flag as a command argument.

In Figure 7.2, the check box can be used to indicate how a `CSound` object should be played in the animation mode. When the user decides to wait until the playback of a `CSound` object is complete, GGS internally uses the `MCI_WAIT` flag to request the driver to complete the operation before returning control to GGS.

A GGS document may contain several sound objects. It is important to close the MCI playback device when `CSound::Play()` returns. This is certainly not a problem when the `MCI_WAIT` flag is used. The `Play()` function requests MCI to close the device knowing that the playback is complete. However, in the absence of the `MCI_WAIT` flag, the MCI driver returns immediately after an `MCI_PLAY` request. The play operation will not be complete if the playback device is closed at that point. Hence, `Play()` returns without closing the device. This brings into question the status of the playback device when it finishes playing the sound object.

GGS uses the `MCI_NOTIFY` flag to request notification when the playback operation has completed. When the `MCI_NOTIFY` flag is used, the `MM_MCINOTIFY` message is sent to an application indicating that an MCI device has completed an operation.

The MFC library includes macros that an application can include in the message map of a `CWnd` or `CWnd` derived object. These macros, such as `ON_WM_PAINT()` and `ON_WM_SIZE()`, map common messages to default handler functions. Similar macros are available for all standard *Windows* messages. To process user-defined messages or less-common *Windows* messages, such as `MM_MCINOTIFY`, the `ON_MESSAGE()` macro is used. The `ON_MESSAGE()` macro must be used in a `CWnd` derived class. For example, it cannot be used in a `CWinApp` class or a `CDocument` class because neither of these classes is derived from `CWnd`. In GGS, the main frame window handles the `MM_MCINOTIFY` message. The handler function retrieves the MCI device ID from the message and closes the device.

7.6.3 Recording WAVE Files

The users have an option of recording their voice instead of importing waveform audio from externally created WAVE files. As mentioned earlier, WAVE files are device-independent with respect to its sound content. However, they require a lot of storage space. A CD-quality stereo WAVE file for one minute measures $60 \text{ (sec)} \times 44100 \text{ (samples)} \times 2 \text{ (bytes)} \times 2 \text{ (channels)} \approx 10 \text{ MB}$ of storage space. It is crucial that

the waveform recording device is not left open indefinitely. Otherwise, it may fill up the hard disk and initiate serious problems. As a precautionary measure, GGS requests the user to specify the maximum time necessary for the recording before opening the recording device:

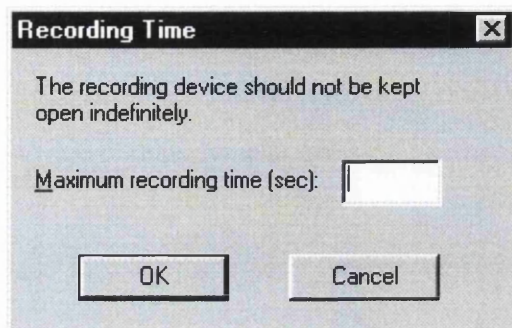


Figure 7.3 Maximum Recording Time

The sound quality can be controlled by passing relevant information to the `MCI_WAVE_SET_PARMS` structure for the `MCI_SET` command for waveform-audio devices. However, there is a straightforward way of controlling the sound quality:

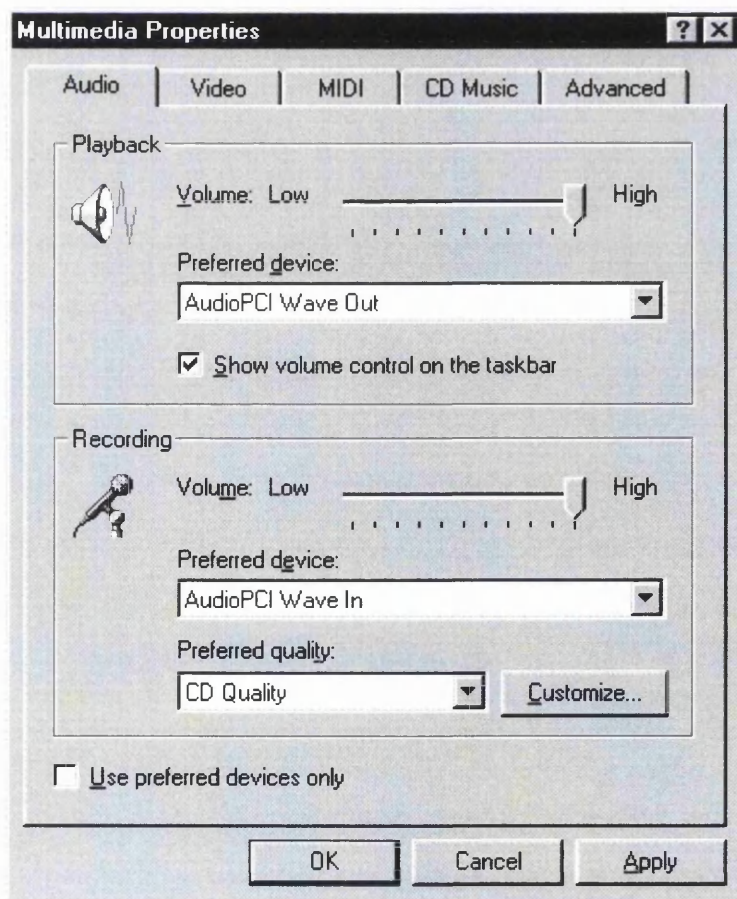


Figure 7.4 The Properties of Multimedia Devices in the Control Panel

The properties of multimedia devices can be changed in the Multimedia section of the Control Panel of *Windows*. GGS accepts the default settings for the recording device. In the recording mode, GGS presents the following dialogue box and automatically closes the recording device depending on the maximum record length specified by the user. But the recording process can be stopped much earlier. The user can press the 'Stop Recording' button at any time:

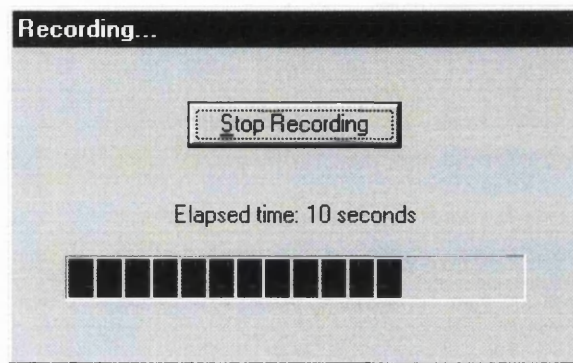


Figure 7.5 Recording in Progress

7.7 TIMERS IN WINDOWS

Windows API exports two types of timers to applications: message-based timers (sometimes referred to as standard timers) and multimedia timers. Standard timers rely on the message `WM_TIMER`. The message is sent (actually faked by `SendMessage()` [Marc98]) to the window whose handle was passed to the `SetTimer()` API during the timer object initialisation. The counterpart to `SetTimer()` is `KillTimer()`, whose job is to destroy the timer passed as parameter, thus stopping the delivery of the `WM_TIMER` expiration messages.

It would be a waste to require the presence of a window just for receiving time-out notifications. Fortunately, passing a `NULL` value as the window designated to handle the timer messages causes the default `WndProc` to invoke the `TimerProc` callback function whose address was passed to `SetTimer()`. This effectively permits windowless components like console applications to benefit from timers without resorting to dirty tricks.

Relying on message delivery provides standard timers advantages and disadvantages. The best advantage is simplicity. Handling a message and writing a callback function are straightforward tasks. The disadvantage is a lack of accuracy. Process scheduling and message queues may cause unpredictable delays, which might be acceptable in many conditions but are intolerable when the program relies

on timers for critical tasks.

The need for precise timers is better addressed by so-called multimedia timers. To achieve superior accuracy, multimedia timers are handled by user-provided callback functions running in a high-priority independent thread owned by the creating process. This approach greatly reduces the gap between the moment of the timeout and the actual execution of the handling code. The resolution should reach the millisecond [MSDN71], although system computational power and workload contribute to make this value significantly uncertain.

Using multimedia timers has its own drawbacks. Firstly, they impose a greater cost to the system than standard timers because multimedia timers run in a separate thread transparently managed by *Windows*. Hence, they should be adopted only when a high degree of precision is required. Secondly, the handling code must often pay close attention to synchronisation and execution time because of the intrinsic multithreaded nature of the mechanism. Thirdly, they should not call any system-defined function except `PostMessage()`, `OutputDebugString()`, and some multimedia API's.

7.7.1 The Recording in Progress Dialogue

The dialogue box in Figure 7.5 is different from many others in GGS. It is driven by a timer. The object associated with this dialogue template creates a *Windows* standard timer just before opening the recording device and displaying the dialogue box. A static function of this dialogue control class serves as the callback function. This callback function updates the 'Elapsed time' after receiving the time-out notifications. It can also close the recording device and the progress dialogue when the maximum recording time expires.

7.7.2 Other Member Functions in CSound

Sound objects can occupy a lot of space. It is necessary to store them in disk files and load them in memory for playback operations only. `CSound::CreateTempFile()` serves this purpose of creating a temporary file for storing the waveform audio data. The `Read()` and `Write()` functions are used in serializing the sound objects. The technique has been discussed already in Section 6.8. However, it is important to note that `Read()` is a protected member in Listing 7.1 but `Write()` is a public member. This is because `Read()` is called only by other member functions in the `CSound` class. But the Animation Editor calls the `Write()` function when the user of GGS wants to extract the audio data from a sound object and save it in a WAVE file.

7.8 STILL ANIMATION

The classic way to implement animation is to define a series of elements, making each one appear in succession. An animated movie, after all, is nothing but a series of images drawn by an artist that are displayed on screen one after another. In the world of animation, things improve, change and reinvent themselves very quickly. New animation techniques emerge everyday [INET72]. However, there is one thing in common among various types of animation. A good animation has a definite theme or story. Successful animators always keep the goal of their “story” in mind as they animate.

The animation in GGS is simplistic in nature. The objects do not move or the background does not change. But it is possible to create some impact and get some ideas across. For instance, structural engineers often face problems with detailed design drawings. Sometimes, the inquiries regarding drawings are misunderstood by other parties involved. Fax messages or telephone conversions do not always identify the exact nature of confusion or misunderstanding. In this type of situation, to clarify some of his/her questions, an engineer may scan the drawing and import it into GGS, mark the areas of interest with the free-hand curve tool and simply record the questions as sound objects. The final product is a GGS document that can be sent as an e-mail attachment. The person at the receiving end can open this document in his/her copy of GGS and press the ‘animation’ button. In the animation mode, graphic objects appear in succession, sound objects are played and timers are activated, depending on their position in the sequence of objects in the document.

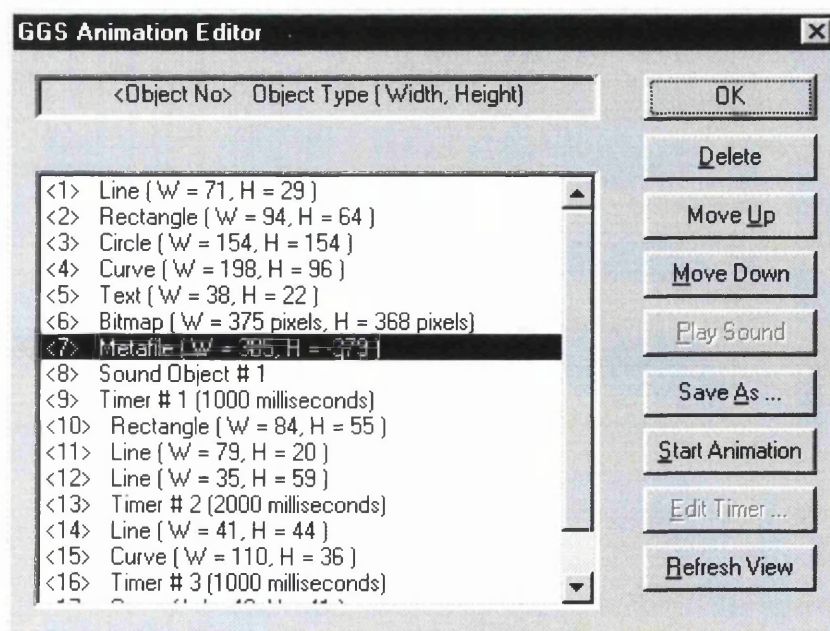


Figure 7.6 The Animation Editor in GGS

The sequence of objects is very important. Especially, sound objects can add realism to a situation or take it away. They can also create a mood, setting or pace. It is important to place them correctly, and if necessary, separate them with brief time lapses. `CGGSTimer` objects can be used for creating such time lapses or short pauses for better control on the animation sequence. The focus of attention can be moved as well by introducing these timers.

The Animation Editor, as presented in Figure 7.6, is an essential feature of GGS to modify the sequence of objects as they can be added later on and moved forward or backward using the “Move Up” or “Move Down” button. The Editor shows the width and height of the bounding rectangle for each object. In addition, the objects are highlighted in the active view window when they are selected in the Editor’s list box. The highlighting technique is similar to the one described in Section 5.4.3. Sound objects and timers can be deleted here which is not possible from the right mouse button pop-up menu. The “Save As” button is useful in extracting the internal bitmap, metafile or sound objects as BMP, EMF or WAVE files from GGS documents.

7.9 TIMERS IN GGS

GGS prompts the following dialogue box when the user wants to create a `CGGSTimer` object:

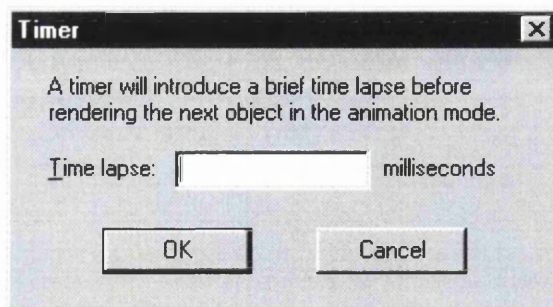


Figure 7.7 Timers in GGS

These timer objects do not use the standard or multimedia timers discussed in Section 7.7. The main purpose of introducing a `CGGSTimer` object in the sequence of other graphic and sound objects is to create a short pause. One effective solution is to use the `Sleep` API:

```
VOID Sleep(DWORD dwMilliseconds) ;
```

The `Sleep` function suspends the execution of the current thread for a specified interval. If the sleep time specified is zero millisecond, it causes the thread to

relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution. In other words, it is an efficient way of creating short pauses. The `CGGSTimer` objects use the `Sleep` API when activated in the animation mode.

7.10 REFERENCES

- [Hedt99] Hedtke, J.: "Hard Facts", PC Magazine, September 1999.
- [INET71] Kritidis, P.: "WAV Versus MID", http://jupiter.int-rpnet.ariadne.t.gr/erl/oth_wm2.html.
- [INET72] McMillan, A. and Hobson, E.: "Animation Tutorial", <http://www.hotwired.com/98/32/index0a.html>.
- [Marc98] Marcato, D.: "Encapsulating Windows Timers in MFC", Visual C++ Developers Journal, January 1998.
- [Micr91a] Microsoft Corporation: "Microsoft Windows Multimedia Programmer's Reference", Microsoft Press, 1991.
- [Micr91b] Microsoft Corporation: "Microsoft Windows Multimedia Programmer's Workbook", Microsoft Press, 1991.
- [MSDN71] Microsoft Developer Network Library: "About Multimedia Timers", *Windows Multimedia, Graphics and Multimedia Services*, Platform SDK, April 1999.

CHAPTER 8

USER FEEDBACK AND CONCLUSIONS

8.1 SOFTWARE VALIDATION

Software validation is the process of ensuring that the software being developed satisfies functional requirements and includes the intended products. Validation is a systematic evaluation of software and associated results of the development process. Reviews and tests are performed at each phase of the development process to ensure that the software requirements are complete and testable. Reviews include walkthroughs and are actually a form of inspection, rather than testing. Reviews include examinations of documentation to make sure that it will support operation, maintenance and future enhancements. Reviews are usually conducted at the end of each phase of the project life cycle to determine whether established requirements, design concepts, and specifications have been met.

Testing of a software is the operation with real or simulated inputs to demonstrate that a product satisfies the requirements. Test failures are used to identify the specific differences between expected and actual results. Software tests may include:

- Unit-level testing
- Various integration testing
- Performance testing
- Operational environment testing
- Acceptance testing

Informal tests are usually performed by the developers' team when the customer of the software is not involved and the "witnessing" of the testing by independent parties is not required. Unit, component, and subsystem integration tests are usually informal tests. Informal testing may be requirements-driven (black box) or design-driven (white box).

Requirements-driven testing is performed by selecting input data and other parameters, based on the software requirements, then observing the outputs and reactions of the software. Usually formal "test cases" are used. Most often, black box testing is repeated at each successive level of integration.

Design-driven testing is the process where the developer or the tester examines the internal workings of the code. Design-driven testing is performed by selecting the input data and other parameters based on the internal logical paths that are checked. The goal of design-driven testing is to determine that all logical paths through the code lead somewhere that was intended.

Formal testing, on the other hand, demonstrates that the software is ready for intended uses. A formal test may include an end-user approved test plan, quality assurance witnesses, a record of all discrepancies, and a test report. Formal testing is always requirements-driven and independent of design criteria [INET81].

After acceptance of a software product, any changes to the product require a formal test. Post acceptance testing normally includes a regression testing. A regression testing involves re-running previously used acceptance tests or test cases to ensure that the change did not disturb functions that were previously accepted.

8.2 SOFTWARE QUALITY

While the dependability of system hardware has undoubtedly improved over the last twenty years, the same cannot always be said of software. The complexity of software has expanded to match the capabilities of hardware, and it is often good software engineering practices which are sacrificed in an attempt to satisfy the demand for product.

Software engineering is characterised by a multitude of languages, tools and processes. In view of this, attempting to establish the fitness for purpose of the end product remains a difficult task. This section will consider some of the underlying issues which often give rise to the problems encountered.

8.2.1 Process or Product?

Software engineering differs from 'traditional' engineering in that its end product is rather nebulous. It cannot be pressure tested, nor can it be subjected to an over-voltage. Software can be subjected to various analysis techniques, but these are often esoteric and of questionable benefit.

Software can also be considered as merely a series of instructions which, when properly executed, serve to provide desired functions. From this viewpoint, software is essentially a process, rather than an invariant product. It is this duality which is the root of many of the problems associated with software based systems.

There are two common approaches to assessing software based systems, each with their own limitations. The first may be considered as the product view of software. This approach is purely functional and is based upon system testing and the rigorous definition and identification of component parts, both hardware and software. This is essentially testing a software as an invariant system component, tested under a

defined set of conditions. However, a 'black box' approach is inherently restrictive and often unrepresentative for complex or integrated systems as the testing can never be fully comprehensive. The approach does not reconcile itself to on-going modification since, in principle, the system should be fully tested and redefined after each alteration. Practice has shown that software based systems are rarely invariant and will be subject to changes throughout their life-cycle. Comprehensive testing becomes impractical once the system is installed in the working environment.

In contrast, the process view of software lends itself to an assessment approach based on the development process, with a corresponding emphasis on quality assurance. This approach requires software development activities to be subject to adequate procedural controls. Survey and audit activities provide an external cross check on the QA activities. Unfortunately, software QA does not in itself guarantee good software, nor does its absence automatically imply bad software [MeTw98]. However, it is increasingly recognised that a traceable, documented design and development process is of long term benefit in managing both cost and risk, particularly for complex systems.

8.3 USER FEEDBACK ON GGS

GGs is a research prototype and not a software system of commercial importance. It was recognised that software QA procedures might not be necessary but one C++ checklist must be used while developing GGS and some user feedback would be important at the end.

8.3.1 The Checklist

At first look, it would seem that operating in an object-oriented paradigm would make code inspections easier rather than harder. The encapsulation of data and functions results in simple design for the functions and high cohesion between the data and functions. And because an object-oriented design more closely models reality, inheritance and polymorphism are used to simplify the design. However, many of the concepts that work in the object-oriented paradigm also work against code inspections. The simple functions are easier to comprehend, but usually the complexity has been moved from the function to the interactions between functions. The smaller functions scattered across the code leads to "de-localised plans". In fact, these designs tend to be harder to comprehend. It is the opinion of the author that a checklist is very important when the implementation language is C++. Some examples of the items in the checklist are as follows:

- Does the class have any virtual functions? If so, ensure that the destructor is also virtual.
- Is each memory allocation in the constructor matched by a memory deallocation in the destructor? This is not always needed, but it is a warning flag.
- Does any method use one of the `malloc()` family rather than `new`?
- Are arrays deleted as though they were scalars? That is, code that reads “delete CharArray” should read “delete [] CharArray”.
- Pointers should be set to NULL following a deletion.
- Does any method return a pointer to class data? If so, is it justified?
- Does any method use a pointer to data outside the class? If so, is it justified?

8.3.2 User Feedback

Executable copies of GGS were distributed with an aim to receive feedback about their ease of use, compatibility with other products and stability. The distribution was very informal. The GGS executable was statically linked to the MFC library so that the users would not face the problems of infamous “DLL conflicts”. A *Microsoft Word* document with instructions was also distributed in the self-extracting compressed file. Mainly, two groups of users were targeted:

- *Windows* users without any GUI programming experience
- Experienced MFC developers

A postgraduate student at the University of Strathclyde, Glasgow claimed that GGS helped him to despatch information of sentimental value without incurring expenses. He scanned some photographs, arranged them in a GGS document, marked them with the free-hand curve tool, recorded some brief messages in Cantonese, and sent the final document and a copy of the GGS executable as e-mail attachments to his family members in Hong Kong. He mentioned that he felt very emotional when he managed to highlight the window of a building and let his family members know his new place of residence in Glasgow.

Unfortunately, many users did not try the sound recording features in GGS. Rather they used the graphics tools to draw sketches. Some of them felt that GGS should have more features to deal with graphic objects. However, a few were pleased to see that the printing quality was better than that of *Paint*. *Paint* only deals with bitmaps and comes as part of every copy of *Windows*. *Paint* diagrams contain raster data and cannot take advantage of high resolution printers, and therefore, appear with “jaggies” when printed. The graphic objects in GGS do not suffer from this problem

since they are essentially in vector format and independent of any device resolution. One interesting fact has been revealed in this context. Many users do not have a vector graphics package installed on their PC's. A number of them use *Microsoft Word* and *Excel* but not aware of the fact that these packages contain a basic drawing module for sketches, flowcharts, etc. To some extent, GGS has become helpful to them as a 'cut-down' vector graphics package.

A lecturer in Singapore mentioned that annotation with sounds could be very useful and practical. When he goes through the reports submitted by the students, he often comes up with new ideas in his mind but does not get enough space to write them down on the reports. He also indicated that a lot of people would prefer to express their comments and ideas verbally rather than in writing.

8.3.3 Feedback from MFC Developers

GGS was also distributed among software engineers with professional MFC/C++ experience. Unsurprisingly, the reaction was completely different. Most of them were satisfied with the user interface design and did not find anything unusual or "abrupt". Some of them felt interested to see the source code. The idea of playing sounds and rendering graphic objects separated by timers was appreciated. However, the general impression was that GGS would require a lot more functionality before achieving any commercial flavour. Firstly, GGS supports BMP, WMF, EMF and WAVE which are *Windows* native file formats. GGS has to support many more to stand as a commercial application. Secondly, in their opinion, it would be effective and useful if some form of compression mechanism is introduced in GGS since WAVE and BMP files can be very large and practically without any compression.

One important suggestion come out of this trial. If GGS is converted from a stand-alone application to a full OLE (i.e., Object Linking and Embedding) server, GGS documents could be embedded in OLE containers. OLE is essentially an integration process and GGS could take advantage of powerful features offered by full-blooded commercial applications. For example, a *Microsoft Word* or *Excel* document could contain a GGS document as an OLE item.

8.4 GGS AS AN OLE SERVER - AN AFTERTHOUGHT

OLE is an object-oriented technology that enables development of reusable software components. The OLE component object model paradigm represents a fundamental shift in the way applications are written. Instead of traditional procedural

programming in which each component implements the functionality it requires, the OLE architecture allows applications to use shared objects that provide specific functionality. Things like text documents, charts, spreadsheet tables, mail messages, graphics, all appear as objects to the OLE application.

OLE applications are of two basic types: container applications and server applications. OLE container applications provide users with the ability to create, edit, save, and retrieve compound documents. OLE server applications provide users with the means to create documents and other data representations that can be contained as either links or embeddings in the OLE container applications. An OLE application can be a container or a server or both.

8.4.1 Steps to Provide OLE Server Support After the Fact

GGs was not designed to be a full OLE server. Hence, AppWizard was run again to create a dummy application with the full server option. In the next step, some essential files and resources were copied from the dummy application to the original GGS project. The `CDocument` class implements standard document behaviour in a stand-alone application. When the application runs as an OLE in-place editing server, however, the document must do extra work on behalf of OLE. The bulk of this OLE document support is implemented in the `COleServerDoc` class. Hence, the base class of `CGGSdoc` (i.e., the GGS document class) was changed from `CDocument` to `COleServerDoc` and the document's support for embedded items was added.

The `COleServerItem` class provides the server interface to OLE items. A linked item can represent some or all of a server document. An embedded item always represents an entire server document. The server item's `OnDraw()` member function is called when the server document needs to draw itself as an inactive embedded object inside the container window. In contrast, the view's `OnDraw()` is called when the document is activated in-place inside the container. The `CGGSItem` class was derived from `COleServerItem` and some OLE-specific codes were added. Some application-specific codes were also necessary to complete the conversion.

8.4.2 Embedded GGS Items

At present, GGS only supports embedding but not linking. An embedded item is physically stored in the compound document, along with all the information needed to manage the item. In other words, the embedded item is actually a part of the compound document in which it resides. This arrangement has a couple of

disadvantages. Firstly, a compound document containing embedded items will be larger than one containing the same items as links. Secondly, changes made to the source of an embedded item will not be automatically replicated in the embedded copy.

Still, for certain purposes, embedding offers several advantages over links. Firstly, users can transfer compound documents with embedded items to other computers, or other locations on the same computer, without breaking a link. Secondly, users can edit embedded items without changing the content of the original. Sometimes, this separation is precisely what is required. Thirdly, embedded items can be activated in-place, meaning that the user can edit or otherwise manipulate the item without having to work in a separate window from that of the item's container. Instead, when the item is activated, the container application's user interface changes to expose those tools that are necessary to manage or modify the item. Figure 8.1 is an example where GGS has taken over *Microsoft Word's* menu bar and the toolbar since an embedded GGS item has been activated:

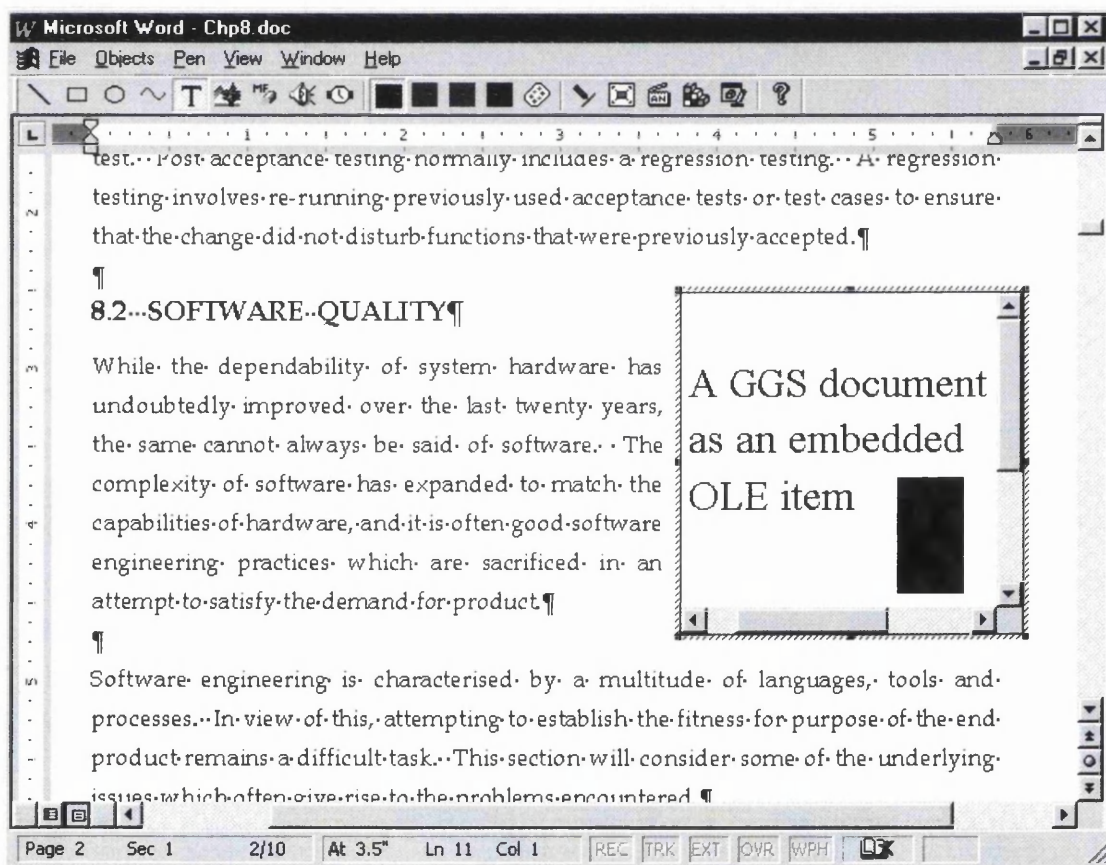


Figure 8.1 An Embedded GGS Item in a Word Processor

8.4.3 Why Not An ActiveX Document Server?

ActiveX and OLE have become synonymous. What people once referred to as “OLE controls (OCX)”, they now refer to as “ActiveX controls”. OLE DocObjects are now “ActiveX documents”. In some cases, Microsoft has updated entire documents on how to implement OLE technologies to be ActiveX technologies, and the only material change was to remove the term “OLE” and replace it with “ActiveX” [LeAr*98]. However, there are some differences as well.

The ActiveX document architecture is an extension of the OLE linking and embedding model and gives the document more control over the container in which the user is hosting the document. The most obvious change is how the menus are presented. A standard OLE document’s menu merges with the container, providing a combined feature set; whereas an ActiveX document takes over the entire menu system, thus presenting the feature set of only the document and not that of both the document and the container. The fact that the feature set of the document is exposed is the premise for all the differences between ActiveX documents and OLE documents. The container is just a hosting mechanism, and the document has all of the control.

Another difference between ActiveX documents and OLE documents is in printing and storage. An OLE document’s designer intends for the document to be a part of the container’s document that is hosting it. Therefore, *Windows* prints and stores the OLE document as a piece of the host container’s document. The operating system expects ActiveX documents to support their own native printing and storage functions and therefore does not integrate them with the container’s document.

ActiveX documents should be used within a uniform presentation architecture, rather than within an embedded document architecture, which is the basis for OLE documents. *Microsoft Internet Explorer* is an example of a uniform presentation architecture that supports ActiveX documents. *Internet Explorer* merely presents the Web pages to the user, but the user views, prints, and stores the pages themselves as a single entity, separate from the host container. On the other hand, *Microsoft Word* and *Microsoft Excel* are examples of the OLE document architecture. If a GGS document is embedded in a *Word* document, it is actually stored with the *Word* document and is an integral part of it. This is more appropriate in the case of GGS. Hence, GGS was not upgraded from a full OLE server to an ActiveX document server, although, the steps involved were straightforward and trivial [MSDN81].

8.5 CONCLUSIONS

A few years ago, the person learning to program *Windows* for the first time had a limited number of programming tools to choose from. C was the language spoken by the *Windows* SDK, and alternative *Windows* programming environment such as Visual Basic had not arrived on the scene. Most *Windows* applications were written in C, and the *Windows* programmer faced the daunting task not only of learning the ins and outs of a new operating system but also of getting acquainted with the hundreds of different API functions that *Windows* supports.

Today most *Windows* programs are written in C++. C++ has rapidly replaced C as the professional *Windows* programmer's language of choice because the sheer complexity of *Windows*, coupled with the wide-ranging scope of the *Windows* API, demands an object-oriented programming language. Many programmers have found that C++ offers a compelling alternative to C because it makes *Windows* programming simpler by abstracting the API and encapsulating the basic behaviour of GUI objects in reusable classes. An overwhelming majority of C++ programmers have settled on MFC as their class library of choice.

Along with many other *Windows* programmers, the author acknowledges the benefits of using MFC and object-oriented programming techniques in C++. In this concluding section, some of these benefits are reassessed based on the lessons learnt in this research project.

8.5.0 Criteria for Evaluating Application Frameworks for Developing Multimedia Applications

There are many ways to evaluate an application framework such as MFC based on several potential evaluation criteria. Properly evaluating application frameworks is not a trivial task. The factors to consider are so numerous and the possible implementations vary so widely that it is easy to become overwhelmed. Criteria such as cost, support, training and contacts with vendor are very important but they are not relevant in the context of this project. It is rather important to look at commercially proven approaches to software development [Walk98] and select criteria from there. Some of the "best practices" are as follows:

- **Develop Software Iteratively:** Given today's sophisticated software systems, it is not possible to first define the entire problem sequentially, design the entire solution, build the software and then test the product at the end. An iterative approach is required that allows an increasing understanding of the

problem through successive refinements, and to grow incrementally an effective solution over multiple iterations.

- **Use Component-Based Architectures:** Components are non-trivial modules, subsystems that fulfil a clear function and promote more effective software reuse.
- **Visually Model Software:** Visual abstractions allow software developers to hide the details and write code using “graphical building blocks” that help them to communicate different aspects of their software.
- **Verify Software Quality:** Poor application performance and poor reliability are common factors which dramatically inhibit the acceptability of today’s software applications. Hence, quality should be reviewed with respect to the requirements based on reliability, functionality, application performance and system performance.
- **Control Changes to Software:** The ability to manage and track changes is essential in an environment in which change is inevitable. Successful iterative development depends heavily on the ability to control, track and monitor changes to software.

It seems, in the light of these commercially proven approaches to software development and the lessons learnt in this research project, the following evaluation criteria are most appropriate:

- Code reusability
- Fast development time
- Rapid software prototyping
- Separation of concerns
- A rich set of widgets
- Serialization
- Escape mechanisms
- Defensive Programming

The remaining sections present an attempt to explain the benefits and deficiencies of using MFC and OOP for building multimedia applications according to the above criteria.

8.5.1 Code Reusability

Newcomers are dismayed when they first learn that the famous “Hello, World”

program for *Windows* requires about 75 lines of code in its simplest form! The problem only gets worse as applications become more sophisticated: windows must be registered, then created; the window procedure (which not uncommonly runs for several pages) must be written. Similar lengthy dialogue procedures are required for all dialogue boxes. There are large applications in which dialogue and window procedures consume literally thousands of lines of “open” source code (i.e., long sequences of program statements with no real structure and no function calls) [Pros96]. Such code is not only difficult to understand and maintain, it can result in needlessly enormous compiled programs.

The reason *Windows* programs require so much effort is that *Windows* fails to encapsulate enough functionality and it leaves much grunt work for each application to perform. To make a hardware analogy: writing a program in *Windows* is like building a computer from individual resistors and transistors, when what is required are integrated circuit, modules that perform high-level functions. The solution to this problem is to write the missing functionality using OOP techniques. But without a good class library to serve as a starting point, OOP does little to reduce the amount of code to be written. In the absence of OOP and MFC, *Windows* programmers followed the “copy-paste-edit” school of reusability. Most of them wrote applications by first pasting an existing *Windows* application into their editor, and then modifying it to suit their purpose.

This error-prone method of developing applications has changed after the introduction of OOP and MFC for *Windows*. Software developers reuse their own code and gain maximum leverage from their own code libraries. Not only does the library approach make maximum use of existing code, the applications are easier to maintain. If there is a bug in the library, the developer fixes it once for all applications. Likewise, if he adds an enhancement to the library, he adds it to all applications simultaneously.

In course of developing GGS, the author has created many general classes that can be used in other applications. The wrapper classes for bitmaps, metafiles and sounds are expected to be especially helpful for other multimedia applications.

8.5.2 Fast Development Time

An MFC project for the first time is not easy. It is not uncommon for beginners to wonder why there are so many files in the AppWizard-generated code. The AppWizard code is liable to be puzzling for the first time because it leans heavily on

the code in the class library. Document/view applications can do powerful things with precious little code but rushing headlong into the documents and views before mastering the more fundamental aspects of MFC is a little like trying to design a computer without knowing what an integrated circuit is.

The simpler classes in a GUI library could be studied in isolation. However, the classes in an application framework form tightly knit clusters. It is essential to study such classes, particularly their patterns of interactions, before instances can be used really effectively in programs. Detailed study is necessary before the classes can be extended through the creation of new subclasses.

The author's experience with the MFC library is no exception in this context. In the beginning, it did present something of a challenge in the form of a steep "learning curve". There was a lot to master before exploiting some essential features of the library. However, the return on this investment of effort has been worthwhile. It has been found that the user interface components of new programs could be created with a fraction of the effort that would have been required using just the low-level graphics primitives and system calls of a host platform. It is also much easier to develop the interfaces with an application framework that supports the expected look-and-feel of a graphical environment.

8.5.3 Rapid Software Prototyping

Prototyping is a process that enables the developer to create a model of the software that must be built. Prototyping begins with requirements gathering. Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the user (e.g., input approaches and output formats). The quick design leads to the construction of a prototype. The prototype is evaluated by the customer/user and is used to refine requirements for the software to be developed. A process of iteration occurs as the prototype is "tuned" to satisfy the needs of the customer while at the same time enabling the developer to better understand what needs to be done.

GGs started as a simple prototype but its first version identified most of the software requirements. Three other versions were implemented based on those requirements and user feedback. The process of developing GGS shed light on important aspects of software prototyping. The effects of OOP and MFC in software prototyping are

found to be different from those achieved by other popular RAD tools such as Visual Basic.

To construct a prototype, a suitable implementation approach must be used. This approach must offer features that satisfy the general requirements of prototyping, such as rapidity and ease of modification. Obviously, an approach that relies upon a complex and lengthy development cycle is unsuitable. Some desirable prototyping features are as follows:

1. An interactive/visual user interface design tool.
2. Easy connection of user interface components to underlying functional behaviour.
3. Easy to learn and use implementation language.
4. Modifications to the resulting software are easy to perform.

Visual Basic is appropriately suited to this form of rapid development. However, C++ is not an easy language and a developer has to ride a steep learning curve before using MFC in his first application. Otherwise, MFC and C++ satisfy other features for rapid prototyping. There are several IDE's (integrated development environments) for MFC/C++ including Microsoft Visual Studio. The wizards in MFC make modifications to the resulting software really easy. The MFC message map mechanism connects user interface components to underlying C++ objects. This is definitely a superior alternative to the `switch` statement used in traditional *Windows* programs to handle messages.

Visual Basic is very good in developing throwaway prototypes. VB applications are executed on an interpretative basis, and they do not rely upon lengthy compilation processes. Although this maybe considered as a limiting factor when developing production quality systems, whose speed and response times are crucial, prototyping does not requires this. In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There may not be any alternative but to start again, and build a redesigned version in which these problems are solved.

When a new system concept or technology is used, one may have to build a system to throw away. In fact, the first prototype of MFC was thrown away [ShWi96]. However, throwaway prototyping can be problematic for the following reasons:

- The customer sees what appears to be a working version of the software, unaware that in the rush to get it working, overall software quality or long-

term maintainability were not considered. When informed that the product must be rebuilt, the customer cries foul and demands that “a few fixes” be applied to make the prototype a working product. Too often, software development management relents.

- The developer often makes implementation compromises in order to get a prototype working quickly. An inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become familiar with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

An MFC application such as GGS can be developed as an evolutionary prototype rather than a throwaway. Once the software requirements are defined by the prototype, the developer can improve upon individual C++ classes used in the project with an eye toward quality and maintainability. MFC provides the infrastructure and OOP lays the basis for later decisions about enhancements.

8.5.4 Separation of Concerns

MFC uses a lot of tricks to make *Windows* objects such as windows, dialogue boxes, and controls behave like C++ objects. For example, it is not possible to implement a reusable list box class using C/SDK that displays a navigable list of drives and directories on the host PC unless it is written as a custom control. This is because clicking an item in the list box sends a notification to the list box’s parent (the window or the dialogue box in which the list box appears), and it is up to the parent to process that notification. In other words, the list box does not control its own destiny; it is the parent’s job to update the list box’s contents when a drive or a directory is changed. However, in an MFC application, windows and dialogues reflect unprocessed notifications back to the controls that sent them. This makes it possible to create a self-contained and highly reusable list box class that responds to its own click notifications by deriving a class from `CListBox` and overriding the list box’s virtual function, `OnChildNotify()`. Inside `OnChildNotify()`, the developer provides handlers for different list box events. The resulting list box class implements its own behaviour and can be ported to another application with little more than a `#include` statement in a source code file.

There are various other examples of OOP reusability and separation of concerns in an MFC application. MFC provides abstractions that go above and beyond what the *Windows* API has to offer. For example, MFC’s document/view architecture builds a

powerful infrastructure on top of the API that separates a program's data from its graphical representations. Such abstractions are totally foreign to the API and do not exist outside the framework of MFC. In addition, the document class when properly implemented, deals with abstract class pointers without knowing much details of individual objects. A multimedia application such as GGS involves lots of similar objects with slightly different behaviour. The document class in a multimedia application should manage the objects in most general fashion and reduce overall complexity in the project.

8.5.5 A Rich Set of Widgets

MFC is renowned for its rich set of widgets to accomplish various design and implementation tasks. The widgets provide features that were once considered to be very difficult to implement and only found in heavyweight commercial applications. For example, the `CToolBar` class in MFC does 99 percent of the work to implement a toolbar that can be docked to different sides of a window or floated in a window of its own. MFC's `CFrameWnd` class has all the intelligence built in to implement a dynamic data exchange (DDE) connection that enables a running instance of an application to open a document whose icon is double-clicked in the operating system shell. Buttons with pictures on them are also easy with MFC's `CBitmapButton` class. Another important example is MFC's OLE support. Only a few developers have the desire or the know-how to write the code for an OLE container or server from scratch. But MFC simplifies the development of OLE-enabled applications by providing the bulk of the code in classes such as `COleDocument`.

8.5.6 Serialization

The MFC serialization mechanism is very interesting. Like many other MFC controls and widgets, the serialization mechanism does a lot of work behind the scenes. The internal details are complex but the author took an attempt to present an in-depth dissection of the MFC serialization in Chapter 3. The discussions and illustrations are believed to be valuable for future developers.

8.5.7 Escape Mechanisms

It is not essential for a software developer to embrace the MFC philosophy. MFC does not limit his freedom to call the *Windows* API functions as and when necessary. This type of escape mechanism might be important to experienced *Windows* programmers new to MFC. GGS was a good learning exercise and its development involved the implementation of standard as well as non-standard features. The API

functions for Media Control Interface (MCI) had been developed before the MFC library was created. The task of encapsulating the MCI API's in GGS and fitting them in the MFC architecture was not straightforward and involved non-standard implementation techniques. In other words, MFC does support certain escape mechanisms but not all.

A new developer starts learning an application framework by writing functions such as `MyDocument::DoMenuCommand()` with no idea of how these functions get invoked. Sometimes, this is exactly what one wants. It is not necessary to redo all the code that responds to a menu selection and eventually calls a menu-handling function; it is only the menu-handling function that is new, all the other bits are standard. However, if any non-standard processing is required in a particular application, the developer must be able to intervene in these interactions and arrange for additional processing. This is often difficult for a new developer to know exactly where and how to intervene. The developer may have to work through the framework code in order to determine where to override and extend the standard behaviour of framework classes.

8.5.8 Defensive Programming

MFC has plenty of macros. Although they serve important purposes, the use of parameterised macros is not regarded as a good programming practice because a macro call may look like a function call but is not semantically equivalent and is not type safe. It is difficult to debug macros and new developers may find them confusing. The practical solution is to avoid user-defined macros and only use well tested macros in MFC.

It is easy to find and explore good features in MFC rather than a few bad ones. The author would like to end his evaluation of MFC and OOP in the construction of multimedia applications with a few words on an important aspect of the MFC design philosophy: defensive programming. The code for MFC is laced with `ASSERT` and `ASSERT_VALID` macros verifying that the class library is getting the results it expects. If an assertion fails, a message box pops up to inform the developer where the error occurred. Better still, the macros compile only in debug builds, so they add no overhead to the retail code. The developer can also follow the MFC design philosophy and sprinkle `ASSERT` and `ASSERT_VALID` macros throughout his code as a first line of defence against bugs.

8.6 LAST WORDS

This research project was motivated by a desire to find new ways of implementing simple animation systems. The author and some other users have found that the GGS documents occupy a lot less space compared to digital video clips, but they could express questions, clarifications, ideas and themes very well if their contents are arranged properly.

Converting GGS from a stand-alone application to a full OLE server was an afterthought. However, it was a useful exercise to check the design and integrity of GGS and understand some reasons behind the ever-changing specifications in the software industry. Users always want new features and extra ways of looking at and manipulating the data. Hence, the software always changes, sometimes in unexpected ways. Therefore, the software architecture of GUI applications should be flexible and robust. It must allow changes to be made easily, and it should be highly decoupled so that when those changes are made they have a minimum effect.

The multimedia prototype developed as part of the project is not complex but the author would like to build more complex software systems in the future based on this experience of analysing, designing and implementing GGS. To emphasise this point and conclude this thesis, the author would like to quote Gall [Gall86]: "A complex system that works is invariably found to have evolved from a simple system that worked... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system."

8.7 REFERENCES

- [Gall86] Gall, J.: "Systemantics: How Systems Really Work and How They Fail", 2nd Edition, The General Systemantics Press, p. 65, 1986.
- [INET81] BPR 9000 Incorporated: "Software Validation", <http://www.bpr9000.com/validation.html>.
- [LeAr*98] Leinecker, R.C., Archer, T., et al.: "Visual C++ 6 Programming Bible", IDG Books Worldwide Inc., 1998.
- [MeTw98] Messer, A.C. and Twomey, B.J.: "Software Based Systems in the Marine Environment", Lloyd's Register Technical Association, Paper No. 5, 1997-98.

- [MSDN81] Microsoft Developer Network Library: "Upgrade to an Active Document Server", Porting and Upgrading, Visual C++ Programmer's Guide, July 1999.
- [Pros96] Proise, J.: "Programming Windows 95 with MFC", Microsoft Press, Washington, 1996.
- [ShWi96] Shepherd, G. and Wingo, S.: "MFC Internals: Inside the Microsoft Foundation Class Architecture", Addison-Wesley Developers Press, 1996.
- [Walk98] Walker, R.: "Software Project Management -A Unified Framework", Addison-Wesley, 1998.

