COMPUTER GRAPHICS – AN OBJECT ORIENTED APPROACH

A Thesis Presented
by
Paul O'Connell B.Sc.

Submitted to the
Dublin City University
Computer Applications
for the degree of
MASTER OF SCIENCE
July 1989

Acknowledgement

To my parents who gave me the opportunity to pursue these studies, providing support and encouragement along the way.

Abstract

Paul O'Connell, B.Sc.
Supervisor: Dr. M. Scott PhD

More and more applications are being found for computer graphics ranging from business graphics to scientific modelling. Packages can be purchased which support these applications but sometimes users find these packages limit the control they have over the final image and are forced to resort to programming in order to overcome these limitations. COOGE is an attempt to support these users by providing a means to generate graphic applications using a set of graphic objects specifically geared towards CAD users. It provides all the power of a programming language while at the same time providing all the comfort of a package. COOGE is device-independent and user extendable. It provides a wide range of transformations and primitive graphic functions in both 2D and 3D. It also allows a user to set up a library of images in a device-independent format which can then in turn be incorporated into future images. This thesis outlines the issues behind designing a graphics library and how object oriented methods can be used to meet these objectives. Finally it attempts to outline the impact object-oriented development will have on the future of computer graphics.

Declaration

No portion of this work has been submitted in support of an application for another degree or qualification in the Dublin City University or any other University or Institute of Learning.

Contents

List of Figures

Chapter 1

COMPUTER GRAPHICS – NOT A PRETTY PICTURE

1.1 Introduction

"Computer graphics is the art or science of producing graphical images with the aid of a computer". No short definition of computer graphics, however, can capture all of its applications. Computer graphics permits the vast quantities of complex interrelations of information to be organized and manipulated in a way that exploits the unique human ability to work with patterns. As it says in the ancient Chinese proverb "A picture is worth a thousand words."

Most people seem to think that there is something magical being worked by people in computer graphics, as all they ever see is the final result such as computer generated colour images of crystal balls and draught sets, or some pictures of the mandelbrot set. However, the fundamentals of computer graphics are really quite simple – all you need is some basic knowledge of computer software and a fundamental understanding of geometry.

More and more applications have been found for computer graphics, such as business graphics which illustrate the current state of the business in terms of bar charts, pie charts and graphs or applications in a scientific research field to model a section of DNA or used in the modelling of the workings of a human organ. It is however, in the Computer Aided Design (CAD) field that this research has been concentrated.

CAD systems are now capable of drawing all the traditional lines and shapes of manual drawing systems. They usually operate under prompting and menu control with a screen cursor being used to designate co-ordinate points and to interact with the drawing. Most systems have some degree of 'intelligence' that partially automates the drawing process, such as the generation of circles, arcs, tangents and so on. CAD systems can, by generating and storing 'standard' shapes, also reduce the work load of the draughtsman.

For example a designer whose job it is to design the interiors of the rooms in a building must show how the furniture will be laid out and how chairs will be arranged. It would be a very time consuming job to draw each item in the room by hand and for this reason most CAD systems allow the user to build up a library of frequently drawn objects such as chairs and tables. The designer can simply call up any of these shapes at any time and reproduce them in his/her drawing with only a couple of commands.

The final drawings can be stored for later use or for transmission to other systems that require them. Sophisticated and specialised CAD systems often have the ability to check the correctness and integrity of the design using preprogrammed knowledge of the subject and high accuracy calculations. CAD systems can also be used in simulation to see how a piece of machinery will work without ever having to build a prototype. For instance, you could draw an engine and use modelling to see graphically how the engine will perform.

1.2 Computer Graphics Packages

A number of different CAD packages or systems are available at the moment, each one geared towards a different type of user. For instance there is Auto-CAD [ACAD] for architectural applications, Micro-CAD for electronic applications. In general these packages are well designed allowing the user a wide range of control over the image being produced. A lot of CAD packages now support 3-D with the more advanced products supporting solid modelling and hidden line removal.

In order for any CAD package to be successful it must be capable of being run on a number of different graphics cards and devices. In order words, the package and the CAD images or pictures themselves must be device-independent.

Using a CAD package has a number of advantages for a user. For instance most of the packages take advantage of pop-up and pull-down menus to provide a user-friendly interface which means that the user is not required to learn a complete programming language in order to produce CAD images. This is especially useful for inexperienced users.

1.2.1 Disadvantages

No CAD image will be of much use to the user unless a number of transformations can be applied to it. For instance, the viewer might want to view the image from a number of different angles or to scale the image up or down to suit a particular requirement. The type of control the user has over the final image depends firstly on how the image is stored. To apply these kind of transformations successfully to the image, the individual elements that went to make up the picture must be stored (the lines, circles, rectangles etc.) rather that just a copy of the final screen. The way in which the image is stored also affects the usefulness of the image. For instance, it is often necessary to transfer an image from one package to another. Unfortunately, no standards currently exist for the transfer of CAD images between products although most packages can import/export using an AutoCad display file format.

The transformation functions provided by the packages also determine the control the user has over the final image and how it will look. If these functions fall short of what the user actually requires then there is no easy solution. It is usually impossible for the user to add new functions to the package or even to enhance the existing ones. In fact the only type of changes that can be made are usually some type of simple customization of the user interface.

3-D Images are relatively easy to represent on a 2-D computer screen once drawn, but drawing a 3-D image on a 2-D screen in the first place is not as simple as it seems. The reason for this is that each point on the physical screen can actually represent an infinite number of points in the 3-D world. When a user draws a line to a point on the screen how do you decide where the point actually is in the 3-D world ? This type of complication means that 3-D drawing of images is only supported by a small number of packages and usually it is a fairly complex procedure to produce the image. One method of overcoming this problem is explained by B. Ozell [BOZE85] in an article written on CAD applications. Other methods of overcoming this problem are outlined by Timothy Johnson in an article on a program for drawing in three dimensions called Sketchpad III [JOHN63].

As users require more and more complex transformations to be applied to the image they generally find that it is the packages which limit them. The only other choice is to move back down the graphical ladder to the basics and resort to writing a program to produce the image.

1.3 Graphic Function Libraries

In the past, virtually every time a graphical display terminal or screen was attached to a computer in a unique configuration, a fresh set of graphics functions had to be written to support it. The design and implementation of this software was an activity that absorbed large amounts of time and energy. Despite this expenditure of human resources, the result was often unsatisfactory, with users finding the system difficult to program or lacking in capability. For such reasons, each fresh graphics hardware configuration tended to spawn not just one new set of graphic functions but two or three in succession.

The more recently developed libraries tend to reverse the situation with each set of functions supporting a variety of hardware configurations. These so called device-independent graphics have become popular with the application programmers who have used them. For instance MetaGraphics [META88] provide a set of graphical routines that can be used with a wide number of graphic display cards. These types of systems have been criticised for failing to cater for the user who needs a high performance system. This is a legitimate criticism, pointing to a need for graphics systems that are suited to both high- and low-performance displays.

Graphics functions, however, need to be more than just device-independent they also need to be sufficiently general purpose to support a wide variety of applications. System designs often fall short in this area because the designers believe that they know what the final applications will be, and that they can optimize the design in their favour. Typically, an unanticipated graphics application will turn up that requires awkward modifications to the system. The designer should always do his best to anticipate these requirements, by providing a general set of functions.

Graphic functions should also be high-level . They should provide a simple yet powerful means of writing graphics applications, and should hide from the programmer the low-level features of the hardware. Ideally, they should make graphic applications as easy to write and maintain as any other type of interactive program.

Furthermore, these graphics libraries should provide a standard interface to allow the user to switch from one set of library routines to another without having to alter his/her program code. Most graphics libraries tend to provide the same graphics functions but the interface to these functions vary considerably from one set of library routines to another. For instance to draw a line from a point 10,10 to 100,100 might require one command in one library or two commands to achieve the same effect in another library.

```
eg.
   _line(10,10,100,100) as opposed to
   _moveto(10,10); _lineto(100,100)
```

With no standard interface to functions being provided, the user is forced to reprogram when changing from one library to another.

A well designed Set of graphics functions allows the user full control over how the final graphical image will look. The major disadvantage of this type of system is that a reasonable knowledge of a programming language is required to use such a system, and the final output might suffer because of shortcomings in the user's programming ability rather than shortcomings in the user's graphical abilities.

In the past, graphics system designers have often shown poor judgement in choosing design criteria and establishing design priorities. They have frequently fallen into the trap of making speed of response their overriding concern. Taken to its logical conclusion, this approach leads them to abandon high-level general-purpose features since these might cause "inefficiency". Instead each system is designed to make the most efficient possible use of the display hardware. Systems of this sort, besides making programs difficult to write and maintain, are inevitably highly device-dependent. This device-dependence means that these systems also have a limited life span for as soon as the display hardware changes the system itself must be reprogrammed to work with the new hardware.

Again, because of this overriding concern for efficiency most of these graphic libraries do not support 3-D graphics because of the complexity of programming and the effect that 3-D support has on the speed of the system. This forces the users to supplement the system with their own set of 3-D transformations and viewing routines.

1.4 GKS – A Graphics Standard

In an attempt to produce some standards for graphical functions, a 2-D Graphics Kernel System (GKS) [DINF89] was designed by the International Standards Organisation (ISO) and became a standard in 1983. GKS was an attempt to make graphical programs portable and device-independent. GKS can be used as a base on which portable CAD systems can be designed and it can be accessed from a number of computer languages, including both Pascal and Fortran.

GKS is a device-independent kernel system with all the graphics commands being stored in a device-independent format in a metafile. GKS then uses device-drivers to convert the device-independent commands into device-specific commands for the input/output devices attached to the system. By changing the device drivers one can change the input and output devices without having to reprogram. This approach, while providing device-independent graphics, tends to be quite slow.

The GKS design objectives were as follows :

- GKS has to include all the capabilities that are essential for the whole spectrum of graphics, from simple passive output to highly interactive applications.
- The whole range of graphic devices, including vector and raster devices, microfilm recorders, storage tube displays, refresh displays and colour displays must be controllable by GKS in a uniform way.

To meet these objectives a large system was required and because of this GKS appears to be very top heavy, being slow to use and slow to respond. In a CAD environment device-independence is a priority – but so too is a reasonable speed of drawing.

GKS as it stands only operates in 2-D, with no support for drawing or inputting 3-D images. A number of attempts are being made to extend GKS to 3-D [HOPG86] to include such features as the definition and display of 3D graphical primitives and mechanisms to control viewing transformations and associated parameters. In fact, this move to 3D has resulted in the GKS standard being extended in two different directions and two new graphics systems emerging, namely, GKS-3D and PHIGS (Programmers Hierarchial Interactive Graphics Standard).

1.5 The Hybrid

The SEILLAC I [GUED76] committee was made up of 25 experts from Northern America and Europe, who met in Seillac France and participated in a workshop on the subject of "Methodology In Computer Graphics". They formulated some principles to be kept in mind when designing Graphic systems. Among the main principles outlined were :

- Portability of Application Programs
- Device Independency
- Portability of picture data
- Portability of Education

These principles were the underlying design criteria used in the formulation of the hybrid object oriented graphical environment called COOGE (CAD Object Oriented Graphics Environment). The concept is simple – to provide a set of graphical functions to a user that are device-independent, simple to use, powerful enough for an experienced user and at the same time user expandable and completely customizable. The COOGE system is geared more towards CAD type users who need complete control over how the final image will appear and who need to apply complex transformations to the images. COOGE hides the underlying graphic routines from the user by object oriented techniques, and so these routines can be replaced with routines for a specific machine or graphics card without altering the user interface, thus eliminating the need to reprogram to suit a particular graphics card or device.

A range of primitive shapes are provided, including points, lines, polygons, circles and rectangles in 2 and 3 dimensions. The user can combine these shapes to make more complex objects, or can invent a new shape and add it to the system without affecting existing shapes. At the same time, it is guaranteed that all routines that manipulate and draw shapes will also support this new shape. Each shape type can be individually drawn, moved, scaled and rotated in any of the three dimensions.

A wide range of image transformations are provided including scaling, translating, viewport (clipping) support and rotation in any of the three dimensions. These transformations work in both 2-D and 3-D. The user has full control of the type of view to present of a 3-D object, ranging from one point perspective to oblique, isometric and orthographic. Even non standard views are supported with the user supplying the viewing angle and projection plane. The mathematics of these operations are completely transparent to the user and if a number of transformations are specified they are combined to insure no loss of speed when drawing the final image.

Support for windowing and simultaneous views of the same object(s) are also provided.

All of this is provided within the programming environment of C^{++}. The design of the COOGE system is such that the user has all the power of a programming language and a powerful set of graphic functions available in a simple user-friendly way.

The COOGE system is fully outlined in the following chapters.

Chapter 2

IN A CLASS OF ITS OWN

2.1 Introduction

The following is a quick description and summary of the object oriented approach to software design. It is included to give the user some insight into the design philosophy behind the system and to give some idea as to why an object oriented design method and language were used rather that the normal functional design methods and standard languages.

Object oriented development is different to the normal software development approach. In object oriented development, the decomposition of a system is based on the concept of a object. An object is an entity whose behaviour is characterized by the actions it suffers and those it requires of other objects.

2.2 Object Oriented vs. Functional Development

In normal program development, well developed systems tend to consist of modules or collections of subprograms. This design method works very well for normal design methods which tend to concentrate on the algorithms and functions that must be applied to data. However, this method of design has some serious drawbacks associated with it, as pointed out by Guttag [GUTT78] "unfortunately, the nature of the abstractions that may be conveniently achieved through the use of subroutines is limited. Subroutines, while well suited to the description of abstract events (operations), are not particularly suited to the description of abstract objects. This is a serious drawback".

The functional development methods suffer from a number of fundamental limitations.

1. They do not effectively address data abstraction and information hiding.
2. They are generally inadequate for problem domains with natural concurrency.
3. They are not responsive to changes in the problem space.

Object oriented development and languages are an attempt to overcome these problems.

In functional development we would examine a system to see what operations were taking place and then would model these operations using procedures and functions. In object oriented development, we look at the system as a set of objects rather that operations, and use these objects to generate the program structure. Object oriented development is an attempt to design programs that closely resemble what happens in the real world, making a direct and natural correspondence between the world and its model, rather than having to shape the problem so that it fits nicely into a functional decomposition technique which only concentrates on the actions of a system and fails to take account of the underlying objects that suffer or create/perform these actions.

Furthermore in functional decomposition a lot of the required data tends to be global, so any small change in the underlying data structure could mean reprogramming of a number of the subprograms. In object-oriented development the effect of changing an object's representation tends to be more localised, only affecting the immediate object – which means that objects can be added or changed in this system without the side-effects normally associated with this process.

The underlying principles and the foundation on which object oriented development is based are data abstraction and information hiding.

Shaw's definition of abstraction is "a simplified description, or specification, of a system that emphasies some of the system's details or properties while suppressing others" [SHAW84]. Parnas [PARN72] suggests in relation to information hiding that we should decompose systems based upon the principle of hiding design decisions about our abstractions. This concept of data abstraction allows an object or class to be defined by a name, a set of proper values, and a set of proper operations, rather than its storage structure, which should be hidden. The object can only be accessed by the proper set of operations provided by that object and the private data of an object can not be seen by any other objects.

Thus when we are designing any program we must first of all look at the problem not as a collection of procedures and functions but as a set of interacting objects. The steps involved in this process as set out by Booch are as follows [BOOC86] :

- Identify the objects and their attributes
- Identify the operations suffered and required of each object
- Establish the visibility of each object in relation to other objects
- Establish the interface of each object
- Finally, implement each object

1. **Identify The Objects And Their Attributes**

   This involves the recognition of the major objects or classes of objects in the problem space, plus their role in our model of reality.

2. **Identify The Operations For Each Object**

   This step serves to characterize the behaviour of each object or class of object. Here we establish the static semantics of the object by determining the operations that may be performed meaningfully on the object or by the object. Also at this time we must establish the dynamic behaviour of each object by identifying the constraints upon time or space that must be observed. For example, in a graphics based system with an object that acts as a graphics window, you can only draw in a window after the window has been opened.

3. **Establish An Object's Visibility**

   We must now try to identify the static dependencies among objects and classes of objects (in other words what objects see and how they are seen by a given object). The purpose of this step is to capture the topology of objects from the model of reality.

4. **Establish The Interface**

   This step establishes the interface between the outside view of an object (its clients) and the inside view of an object (its internal representation).

5. **Implement**

   Finally, you must choose a suitable representation for each object or class of object and implement the interfaces from the previous step.

There are major benefits to be derived from an object oriented approach to system design. As pointed out by Buzzard [BUZZ85], "there are two major goals in developing object-based software. The first is to reduce the total life-cycle software cost by increasing programmer productivity and reducing maintenance costs. The second goal is to implement software systems that resist both accidental and malicious corruption attempts". With regard to maintaining such programs, Meyers [MEYE81] reports that "apart from its elegance, such modular, object-oriented programming yields software products on which modifications and extensions are much easier to perform than with programs structured in a more conventional, procedure-oriented fashion". The reasons for these benefits are that understandability and maintainability are enhanced due to the fact that any changes that have to be made to a particular object are localised and should not affect other objects in the system.

The characteristics of an object are as follows:

An object –

- has a state
- is characterized by the actions it suffers and that it requires of other objects.
- is an instance of some (possibly anonymous) class
- is denoted by a name.
- has restricted visibility of and by other objects
- may be viewed either by its specification or by its implementation

For each object we have constructors that create the object, destructors that destroy the object and a number of operations that the object can perform. Objects can interact with each other without having to know how an object is internally represented. This leaves us free to change the internal representation of an object without affecting other objects in the system.

There is also the concept of hierarchial types which allow one to define general interfaces that can be further refined by providing subordinate types. The concept of information hiding also applies to the hierarchial structures, so there can be hierarchies with/without the base class hidden.

A number of different object oriented languages exist at present, such as ADA, Smalltalk-80, C^{++}, Objective C. Having decided to implement the project in an object oriented language the choice then has to be made as to which language is most suitable for the purpose. A number of languages, such as Smalltalk, only allow a user to operate within that environment, so it is impossible to create a standalone application without carrying the full overhead of the environment. In a graphics environment, the ability to number crunch was a very important factor along with the overall speed of the system. C^{++}'s ability to use C code directly was seen as a major advantage because of C's ability to produce machine efficient code for number crunching. Object oriented languages that relied on message passing (such as Objective C) tended to be slower than those which relied on function or method calls (such as C^{++}).

It seemed that C^{++} had a number of advantages:

- It could handle C code directly
- It was compiled rather than interpreted (faster execution)
- Once a program was compiled it required no further support from the C^{++} environment.
- C^{++} used function calls to invoke actions rather than message passing which tended to be slower.

After weighing up the 'pros' and 'cons' of the various languages available, it was eventually decided to opt for C^{++}, an enhancement to the C language developed at Bell Laboratories in 1983 by Dr. Bjarne Stroustrup. The language itself became commercially available in 1985.

C^{++} was developed to meet the following Goals [STRO86] :

1. Retain the extremely high machine efficiency and portability for which C is famous.

2. Retain compatibility between C^{++} and C.

3. Repair long-standing flaws, particularly C's lax treatment of types. C has long been criticised for its weak type checking, even inside a given function, and no type checking across functions, even inside the same file.

4. Upgrade C in line with modern data-hiding principles.

A brief overview is given of the C^{++} programming language to give the reader an idea of the underlying programming language and its capabilities [BCOX86].

2.3 Classes

Data abstraction is a programming technique in which the programmer can define general-purpose and special types as the basis for applications. These user defined types are convenient for application programmers since they provide local referencing and data hiding. The result is easier debugging, maintenance and improved program organization. In C^{++} these objects or new user-defined data types are defined through the Class statement as shown in the example of class ostream.

```
class ostream {

public:
  FILE *file;
  int nextchar;
  char buf[128];
};
```

The similarity between the class declaration and the C struct statement can be seen. This statement declares a new class called ostream with three members **file**, **nextchar** and **buf**. The keyword **public:** makes these names public and accessible to any program that incorporates this declaration. In general the public interface specifies how a user can create and manipulate objects of a given class. A class's members are private by default, meaning they can only be accessed by that class's associated procedures and functions and thus hiding the actual implementation details of the class from the user. This hiding of the representation of an object is the key to modularity and it allows the representation of an object to be changed without affecting the users of the object. The functions used to access the Classes private data can be of two forms – member functions or friend functions.

2.4 Member Functions

Member functions are similar to standard C functions except they go one step further and are actually linked to the class in the same sense that the data members are, and operate on some instance of the class. Member functions are declared by mentioning their declaration inside the class declaration, alongside the declarations of member variables.

```
class date {
      int day,month,year;       // implementation for dates
public:
      void set(int,int,int)    // interface to  dates
      void next();
      void print();
};
```

Member functions are called by a syntax that reflects their role as operations performed by a specific object, and parallels the way structure members are accessed.

```
myBirthday.print();
today.next();
```

To define a member function, the name of its class must also be provided.

```
void date::next()
{
 day = day+1;
  .....
};
```

Such functions also receive an implicit argument, '**this**', which identifies the object performing the function. In this example, the day field can be referenced as either **day** or **this** $\rightarrow$ **day**.

2.5 Friend Functions

Private data in C^{++} means private and functions cannot access the private data of a class without using some extremely devious type casting. Sometimes it is useful, for reasons of speed or more elegant coding, to allow a function access to the private data of a class. Friend functions provide a means whereby conventional C functions, with no particular connection with the class, can access the private data of a class.

```
class date {
   int day,month,year;          // Private by default

public
  friend void setDate(date*,int,int,int); // Note the argument
  friend void nextDate(date*);            // types accepted by
  friend void nextToday();                // each function
  friend void printDate(date*); };        // must be declared!
```

The private members (day, month & year) are now accessible to the four friend functions only. The class might then be used as follows:

```
date myBirthday, today;

setDate(&myBirthday,24,2,1966);
setDate(&today,8,8,1988);
printDate(&today);
nextDate(&today);
```

2.6 Derived Classes

The ability to define subclasses with inheritance by describing how a new (derived) subclass differs from some older (inherited) class is at the heart of every object oriented language and is implemented in C^{++} as follows. Consider, for example, a graphics system that needs to support a number of shape types such as circles, triangles, squares, lines, etc. For example, a class shape which specifies the general properties of all shapes could be defined as follows:

```
class shape {

      point centre;
      int colour;

public:

   void scale(float,float,float) // scale shape in x,y,z
   void move(vector);            // reposition a shape
   void rotate(int);             // rotate a shape
   virtual void draw();          // Display a shape
};
```

Virtual functions are functions that could not be implemented without knowing the specific shape, and that must therefore be overridden in each subclass like this:

```
class circle : public shape {

      int radius;

public:
    void rotate(int);         // how to rotate a circle
    void draw();              // how to draw a circle
    ....
};
```

The virtual keyword signals that these functions must be dynamically bound, and triggers the compiler to add an invisible member to each instance that explicitly indicates its class at runtime. It is now possible to define subclasses that obey a common protocol, so that any instance can be drawn by:

```
anyInstance.draw();
```

Note that new shapes can be added to the system without modifying any existing code so **anyInstance.draw()** will still work with shapes that were not even thought of when the program was originally compiled.

2.7 Constructors & Destructors

In C^{++} you can specify a constructor function which defines how an instance of a class should be initialized and a destructor function which defines how it should be destroyed. The constructor function can be used to initialise an instance of the class which appears in the initialization section of a program or on the stack during a function call. A definition of a point class is given below which illustrates the use of constructors.

```
class point {
      int x,y,z;

public:
  point() { x=0;y=0;z=0;}                    // constructor
  point(int xc, int yc,int zc)
              { x=xc; y=yc; z = zc;}         // Constructor
  point(int xc) { x = xc; y =0; z=0;}        // Constructor
  point(point p) { x=p.x; y=p.y; z=p.z;}     // Constructor
  friend point operator+(point,point)        // Add 2 points
  friend point operator+(point,int)          // Add int to point
    ....
};
```

This defines a point on a three dimensional integer co-ordinate plane. The first four functions called ‘point’ are the constructors for the class. You can see from the example that you can provide a number of different ways to initialize an object. For instance:

```
point p;
point p1(12,23,34);
point p2(2);
point p3(p1);
```

can all be used to initialise an instance of the point class.

These constructors/destructors provide guaranteed initialization/cleanup for objects of a given class. Since the declaration also includes the implementation of these operators, the implementation will be expanded inline, and no function call overhead will be incurred to initialize vectors. The inline expansion feature applies to any kind of operator that may be declared in a class, not just constructors. These in-line functions are unlike the macros commonly used in

C in that they obey the usual type and scope rules. Using in-line functions can lead to apparent run-time improvements over C. In-line substitution of functions is especially important in the context of data abstraction and object oriented programming. With these styles of programming, very small functions are so common that function-call overhead can become a performance bottleneck.

2.8 Operator Overloading

In C^{++} operators such as +,–,= are treated just like functions and you can define a new implementation of an operator for any class. To do this, simply prefix the operator token with the keyword 'operator' in the class definition. In the previous example of the class **point** the last pair of declarations define how the '+' operator should work for the class point when both sides are points or when the right side is an integer.

These overloaded operators can then be used as if they were a composite part of the language. For example

```
point p(10,10,2), p1(20,30,40),p3;

p3 = p1+p2;                // Add the two points to form new point
```

2.9 Memory Management

C^{++} provides a number of methods for dealing with objects and memory management. A number of these methods are mentioned by Brad Cox [BCOX86] in his book on object oriented programming and are outlined below.

In C^{++} new objects can be allocated dynamically on the heap and then referred to by address. Objects can also be allocated statically and referred to by name, as for the four points shown in the previous example. At run-time, space for objects known by name must be initialized whenever their name enters scope. For example:

1. Objects passed as arguments to functions, and objects declared as local (auto) variables, must be initialized when that function is called.
2. Objects returned from functions must be initialized as that function returns.
3. Objects in the function's call stack (arguments and local variables) must be destroyed when the function returns and the stack collapses.

C^{++} provides a way for the class developer to specify what should happen in these cases. The point example shows how constructor operators are specified. The inverse is a destructor operation, whose name is the name of the class preceded with a t̃oken. For example, a string class might be

```
class string {
      int length;
      char *bytes;

public:

    string(char*);                      // String constructor
    ~string() {delete bytes;}            // String Destructor
    int length() {return length;}
    char *text() {return bytes;}
};


string::string(char *s)
{
   length = strlen(s);
   bytes = new char[length+1];
   strcpy(bytes,s);
}
```

This guarantees that each string instance created when strings are passed to and returned from functions is a unique copy. But it does not handle multiple references created when one string is explicitly assigned to another in an assignment statement. This can be arranged by overriding the assignment operator.

Notice that although this does not replace automatic garbage collection, it can sometimes reduce the need for it. Automatic garbage collection is still desirable when objects are multiply-referenced via pointers. By copying objects each time they are needed, multiple references do not occur, so objects can be disposed of whenever they go out of scope. Of course, this is feasible only for very small objects. For example, the string implementation shown involves allocating and initializing a new copy every time the string is passed to a function, and the overhead could easily become intolerable. C^{++} does not provide automatic garbage collection.

2.10 Separate Compilation

C^{++} because of the way it is compiled retains no memory of past compilations. This means that firstly no check for consistency of information across a number of compilations can be made and secondly that all definitions about external C^{++} classes and functions must be incorporated into the main source file using the standard #include statement. This lack of a consistency check across compilations means that in practice two completely different definitions of the same class might exist in different compilations without the compiler ever being aware of this.

In order to overcome some of these problems two files are usually prepared for each class. The first file contains the class declarations and must be included into any program that wants to use the class. The second file contains the definitions of the methods/functions provided by the class and the compiled version must be combined with the program to provide an executable image.

The first file contains the private and public definitions of a class. This file is in fact public information and means that user has the ability to change the public/private definitions of a class to make private information accessible. This means that the private data of a class is not truly private as the user at any time can change the class definition and actually make it public.

2.11 Conclusion

In conclusion, object oriented design has a number of advantages over the normal standard design methods and standard languages:

- The ability to hide the underlying data structure of an object from the user means that objects structures can be changed without affecting the user.
- The localisation of the effects of changes in the program means that as new concepts and ideas arrive they can easily be incorporated into existing programs with the minimum of recoding.

C^{++} as an object oriented language has the following advantages over standard procedural languages and other object oriented languages according to Stroustrup who may be just a little biased about the merits of C^{++}!

"C^{++} is distinguished among languages that support object-oriented programming such as Smalltalk, by a variety of factors: its emphasis on program structure; the flexibility of encapsulation mechanisms; its smooth support of a range of programming paradigms; the portability of C^{++} implementations; the run-time efficiency (in both time and space) of C^{++} code; and its ability to run without a large run-time system."

Chapter 3

SOME BASIC GRAPHICS

3.1 Introduction

Just as a painter requires canvas or paper on which to draw so a programmer needs a medium to produce a graphics image. The COOGE users are provided with a V.D.U. or graphics screen to display their image or picture. The screen itself can be thought of as consisting of a matrix of cells called pixels, where a pixel is the smallest addressable point on the screen, with the origin (0,0) usually in the bottom left-hand corner. Each of these pixels can be turned on/off to make a point on the screen visible/invisible. By turning on a number of pixels on the screen you can generate a picture or "image" as it is more generally called. In this simplified system each pixel on the screen requires at least one bit of computer memory to tell the computer whether the pixel is on/off.

Using a more complex display architecture you can associate a colour with each pixel and produce a colour image on the screen. The more colours a display has per pixel the more bits are required per pixel (to tell the computer what colour the pixel is) and hence the more memory required for the display. Also the higher the resolution of the display (the greater the number of pixels) the more memory you require. Because of these memory requirements a vast number of different type of graphics cards have evolved offering very high resolution with few colours or low resolution with many colours and various options in between.

3.2 Graphics Functions

Turning on/off individual pixels would be a very slow and complex way for a user to generate an image. For this reason functions are required that can manipulate a number of pixels at a time e.g. to draw a line. These functions save the user from worrying about individual pixels, for instance a user can just specify the start and end points of a line and the function does the rest, working out the best line between the two points and drawing it on the screen. These functions then leave the user free to concentrate on the contents of the image.

The design of the actual graphics functions or language extensions plays a vital part in determining the success or failure of the system. We should look on these functions as a means of providing the programmer with controls over the functions within the system's hardware and software. These controls should be as simple and as powerful as possible, and should not be too numerous: too many controls provide opportunities for meaningless or erroneous operations, against which the user is usually never warned.

In effect, the designer of a set of graphics functions should aim to remove most of the programmers opportunities to make logical mistakes without causing him/her to feel too restricted. Thus it is not enough just to reduce the range of functions, because if the remaining set of functions do not provide the power the user needs, such as structuring or transforming capability, he/she will remedy the problem by writing the missing functions himself/herself.

The provision of a small number of powerful graphics functions with sensibly chosen default values is an ideal way to reduce the likelihood of erroneous combinations.

3.3 Primitive Functions

COOGE not only supports lines but rectangles, squares, cubes, circles, spheres and cylinders as well. In fact, because of the way COOGE was designed, it can support any shape that the user may care to imagine or invent. The way it manages this is discussed in the following chapter. Sometimes it is useful to be able to write text on the screen, for example to label a part of the image. You may want different styles and sizes of text (fonts) so any text functions should support a number of different fonts. In the COOGE system there are a number of pre-defined fonts but the user can design and add new fonts to the system if required. As mentioned previously, colour is a very important part of any image, so functions to change the current foreground and background colours must also be provided. Other general functions are provided to move and control the position of the graphics cursor and to save and restore bitmapped images on the screen. All of these primitive functions in COOGE work in both 2 and 3 dimensions.

3.4 Co-ordinate Systems

It is very important that an image is device-independent, i.e. it appears the same on different graphics screens. For this reason the actual co-ordinate system used to specify lines etc. is very important. Say, for instance you specify your drawings in terms of the physical co-ordinates (pixel resolution) of the screen. This then means that the resulting image is now device-dependent. If you change to a screen with a different resolution the image will appear completely different and it means you will have to re-define the entire drawing to take account of the new screen's resolution. It would be far better if you could define the drawings in terms of a device-independent co-ordinate system which would mean the image would appear the same no matter what the physical resolution of the actual screen actually was.

When you move onto a 3-D co-ordinate system as in COOGE you also have to decide the direction of the Z-axis relative to the viewer. Two 3-D co-ordinate systems are possible, a right-handed system (Z axis comes out of the page) or a left-handed co-ordinate system (Z axis goes into the page) . Both systems work equally well but COOGE was implemented using the right-handed co-ordinate system.

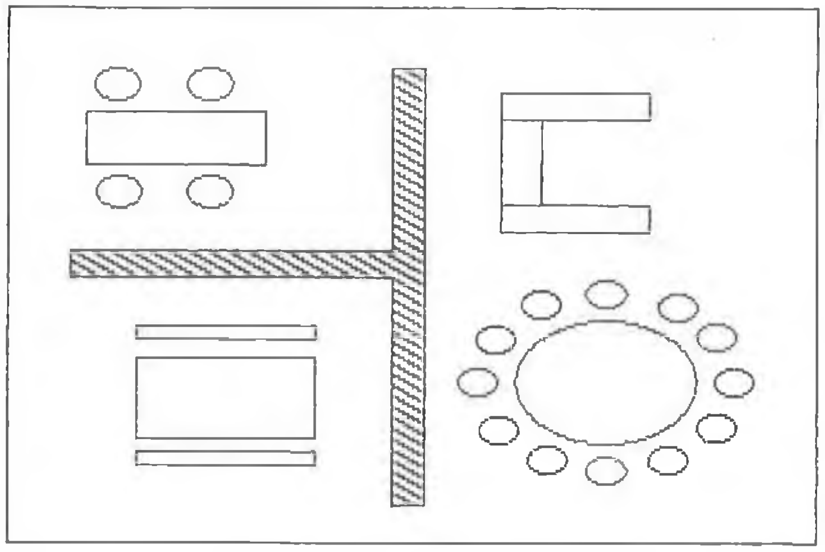
COOGE allows device-independent drawings by allowing users to define their drawing in terms of a world co-ordinate system. The user's world is a virtual 65,536 x 65,536 x 65,536 pixel screen.

3.5 Windowing

When drawing a graphics image it is possible to map the entire virtual screen to the display to produce an image or to specify a 'window' which decides which rectangular part of the world will appear on the screen as an image. The computations involved in this operation, mapping the user's window on to the screen, are called windowing. The user can set the maximum and the minimum co-ordinates (in world co-ordinates) of the window. Any part of the object that does not lie inside the window is made invisible through a process know as 'clipping': any object lying wholly outside the window boundary is not mapped onto the screen, any object lying partially inside and partially outside is cut off (scissored) at the window edge before being mapped onto the screen. The windowing function insures that the same amount of a picture will appear on the screen even if different screens with different resolutions are used.

The window can take in the whole world or it might just take in a particular part of the drawing. For instance in Fig 3.1 we have a floor plan of an office. The user might select a window that displays just one chair in the office (Fig 3.1a) or maybe a desk (Fig 3.1b) or even the entire office. The window is then mapped onto the physical screen. Using this system all drawing processes are device-independent and as such the image is independent of the actual physical resolution of the device on which it is displayed.

By selecting different windows a user can zoom in on a particular object in



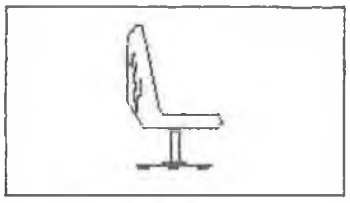


Fig 3.1 (a)

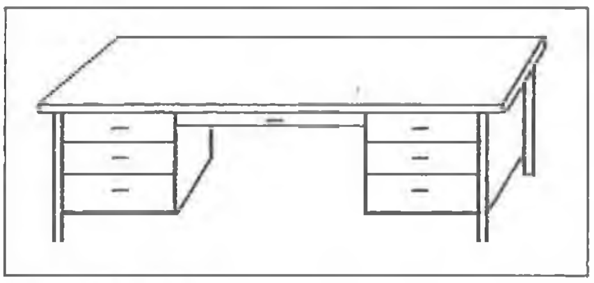


Fig 3.1(b)

Figure 3.1: Floor plan of office

a picture so that the one object fills the entire screen or zoom out so that the entire picture fits on the screen. Using a process called panning a user can also move the window up/down, right/left to move around an image.

3.6 Transformation Functions

Transformation functions are functions which allow the user to manipulate the graphical information generated by the output routines such as the primitive functions mentioned previously. The transformation routines should be both simple to use and efficient in execution. The normal types of transformation routines required are to scale, rotate and translate (move to a different position) graphical images. It should be easy for the user to specify the type of transformation required. These transformation routines should work not just in 2-D but in 3-D as well. Extending the range of transformation functions to handle three-dimensional images adds little to the complexity of the system but will significantly increase its usefulness.

The user should not be limited to a certain set of values for these transformations, such as only being allowed to scale an image by a factor of 2, or only allowed rotate the image by a multiple of 90 degrees. These sorts of limitations imposed on the user will soon force him/her to supply his/her own set of transformation functions. Also the co-ordinates used in these transformations should be in world co-ordinates to insure that the transformations remain device-independent.

3.7 Viewports

Just as a window is used to define how much of the picture should appear on the screen, a rectangular viewport is used to specify where on the screen it should appear. A viewport is a rectangular portion of the screen onto which the window and therefore the window contents are mapped. The default value for the viewport is usually the entire screen. The viewport cannot be defined in device co-ordinates, because the resolution can vary from one screen to another. It is instead defined in terms of normalised device co-ordinates (NDCs). The screen is divided up into real numbers from 0 to 1 in the x and y direction with the origin in the bottom left hand corner of the screen. The viewport is then specified as the portion of the screen in which you want the image to appear. Setting a viewport to (0.0,0.0,0.5,0.5) would make the image appear in the bottom left quadrant of the screen. Drawing the image of the office using this viewport would result in Figure 3.2.

3.8 3-D To 2-D

When drawing a 3-D image on the screen we are faced with the problem of how to represent a 3-D object on a 2-D screen. To achieve this, some type of viewing transformation must be applied to the 3-D image to convert it to 2D so it can be displayed. Firstly, you could just ignore or throw away the Z co-ordinate of each point and draw the resulting image. The result is an orthographic projection

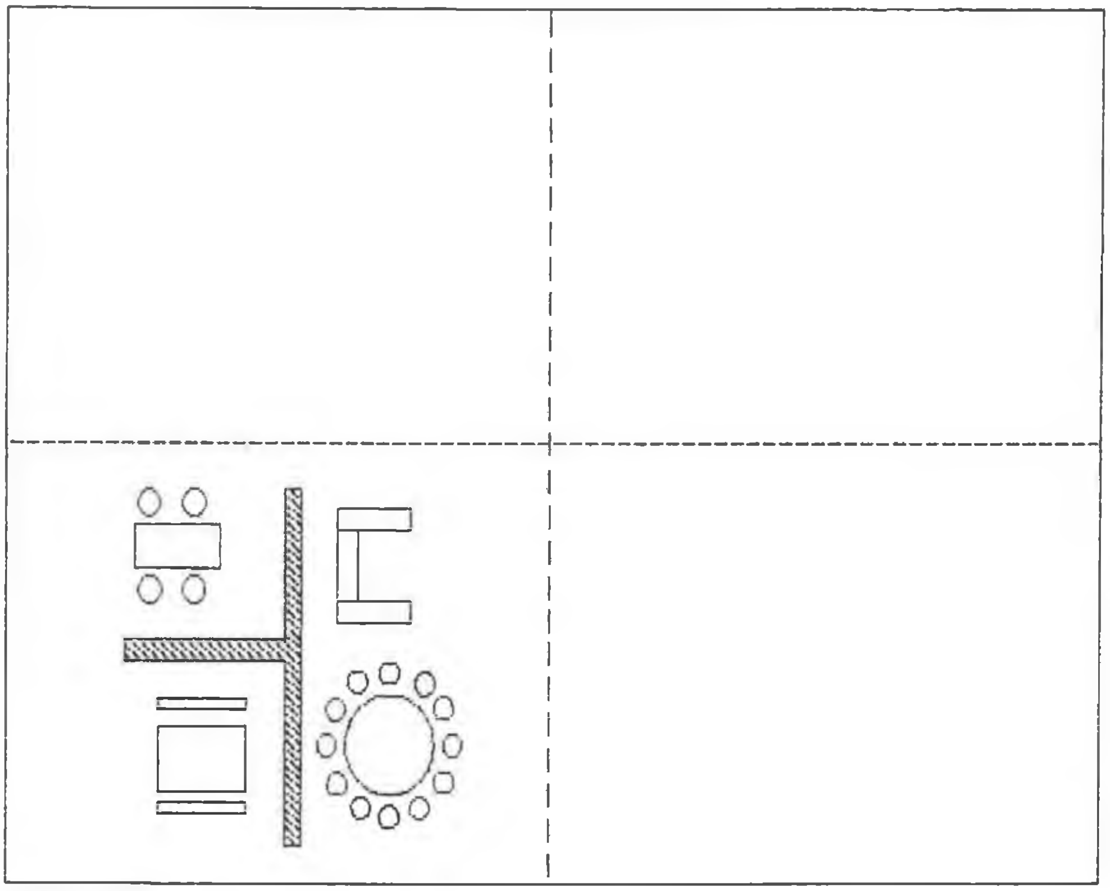


Figure 3.2: Viewport transformation

which is easy to program but unfortunately does not look very realistic see – Fig 3.3. A number of other projections such as isometric, oblique and perspective are normally used to produce a more realistic image. All of these projections are supported by COOGE.

If one of these standard projections is not suitable COOGE allows you to

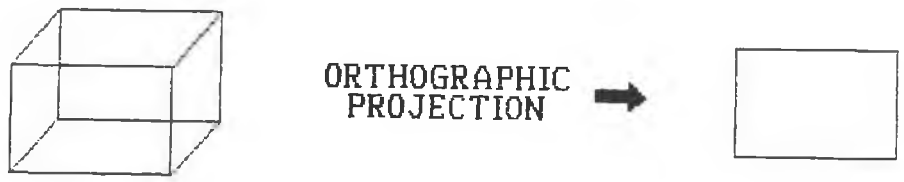


Figure 3.3: Orthographic projection of a cube

define your own by specifying details of the angle at which the viewer is looking at the object (viewing angle) and details of the 2-D plane onto which the object is to be projected.

3.9 Segmentation

Sometimes it is useful to partition a large image into a number of logically related units called segments. Segmentation of an image has a number of advantages for a user.

Localisation : If any part of an image changes then normally the whole image would have to be redrawn. Using segmentation only the segment(s) that contain the changes have to be redrawn.

Re-usability : It is possible to build up a library of commonly used segments which can be included into any drawing. These segments might include commands to draw an item of furniture or even to draw the company logo on the page.

For example in the drawing of a room segmentation might be used so that, for instance, one segment might contain a chair and another a table. Without segmentation, if you wanted to delete the chair from the image you would have to delete it line by line. Using segmentation it is possible to remove the chair from the picture in one operation, temporarily by making the segment invisible, or permanently by actually deleting the segment physically.

3.10 Graphics Pipeline

Each time you want to plot a point or a line on the screen it must pass through a number of stages to convert it from its world co-ordinate system into actual physical points on the screen. The stages through which it must pass for a 2 dimensional system are outlined in Fig 3.4.

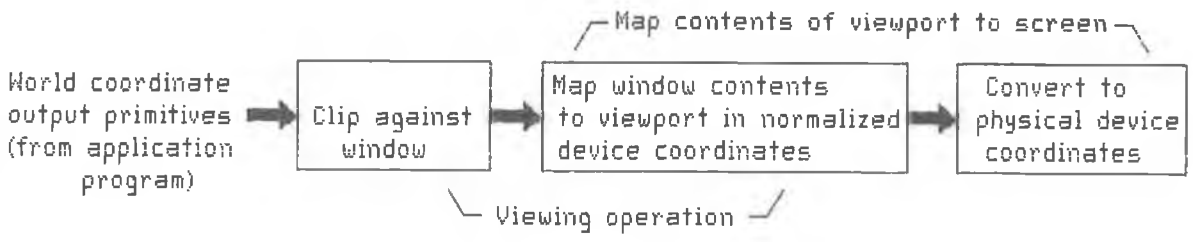

Map contents of viewport to screen
World coordinate output primitives (from application program)
Clip against window
Map window contents to viewport in normalized device coordinates
Convert to physical device coordinates
Viewing operation

Figure 3.4: 2D Graphics pipeline

As you can see the pipeline is relatively simple in 2D. You simply specify a window on the 2D world and a viewport on the 2D view surface. Conceptually, objects in the world are clipped against the window and are then transformed into the viewport for display.

When we move to three dimensions we increase the complexity of the viewing pipeline. For instance the viewport is no longer a rectangle as it is in 2-D, it becomes a View Volume in 3-D which increases the complexity of the clipping operation. You also have to incorporate a projection to transform the image from 3D to a 2D projection plane so it can be displayed on the screen. The final pipeline is shown below as Figure 3.5.

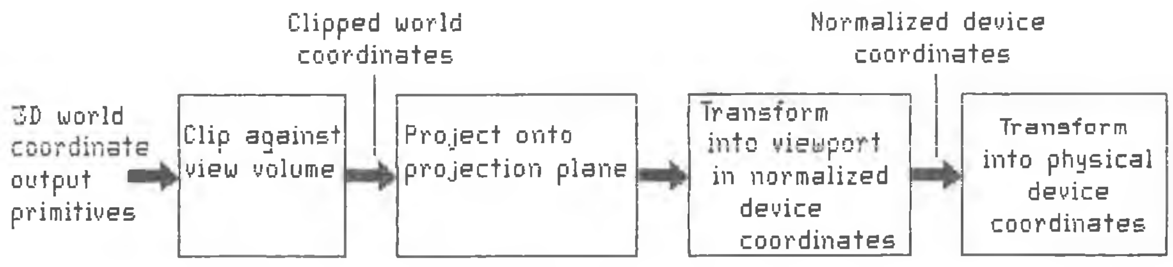

Clipped world coordinates
Normalized device coordinates
3D world coordinate output primitives
Clip against view volume
Project onto projection plane
Transform into viewport in normalized device coordinates
Transform into physical device coordinates

Figure 3.5: 3D Graphics pipeline

Having passed though all these stages the image is finally ready to be displayed on the screen.

3.11 The Mathematics Behind The Picture

As you can see from the graphics pipeline there are a number of stages through which a point must pass before it can actually be displayed on the screen. The transformations that must be performed, such as scaling, rotating, are generally stored in a matrix structure and the point to be transformed must be premultiplied by the matrix. For example, to rotate the point (x,y,z) 90 degrees around the Z-axis the following transformation is performed :-

$$\begin{vmatrix} cos(90) & sin(90) & 0 \\ -sin(90) & cos(90) & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} X \\ Y \\ Z \end{vmatrix} = \begin{vmatrix} X_1 \\ Y_1 \\ Z_1 \end{vmatrix}$$

By applying this transformation to a number of points it is possible to rotate a shape or entire image. Fig 3.6 shows the effect this transformation has on a simple shape.

If there are a number of transformations to be applied to a point e.g. if it is to be scaled, rotated and translated, they must be applied one at a time, which means that the speed a point can pass along the graphics pipeline depends on the number of transformations that must be applied to the point. It would be much faster if the transformations could somehow be combined to form some kind of master transformation that could be applied to the point rather than a number of individual ones.

In order to combine transformations the co-ordinates must be presented in a particular format called homogeneous co-ordinates. Using homogeneous co-

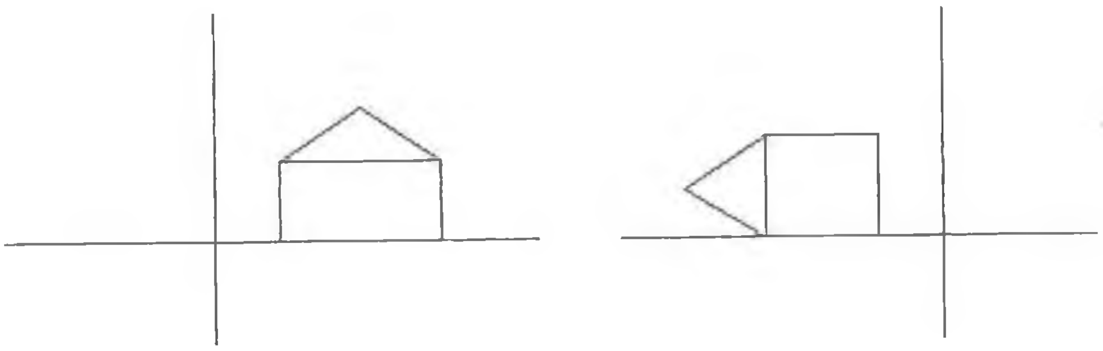


Figure 3.6: Rotation of a shape by 90° about the Z-axis

ordinates transformations can be combined simply by multiplying them together. The order in which they are multiplied determines the order in which the transformations will be applied to the point.

The translation from normal co-ordinates to homogeneous co-ordinates takes place internally within the COOGE system and hence is invisible to the user. For example, the homogeneous version of the point (x,y,z) is (x,y,z,w) where W is the scale factor (usually one for simplicity).

Using this kind of notation, most transformations can be expressed in a matrix. For instance if we want to scale a point by a scaling factor Sx, Sy, and Sz in the X, Y and Z directions, respectively, then the matrix for scaling in X, Y, Z is :

$$\begin{vmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Any point x, y, z can now be scaled by multipying the point by the scaling matrix as shown.

$$\begin{vmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} X \\ Y \\ Z \\ 1 \end{vmatrix} = \begin{vmatrix} X_1 \\ Y_1 \\ Z_1 \\ 1 \end{vmatrix}$$

As already mentioned, transformations can be combined to form a new transformation. This means that if a number of transformations have to be applied to a point, rather than applying them one at a time, they can be combined and only this combined transformation need be applied to the point. This cuts down on the number of mathematical operations that must be performed each time a point has to be processed and thus considerably speeds up the drawing process.

3.11.1 Viewing In 3-D

As mentioned previously, viewing an object in 3D is more complex than viewing an object in 2D. In order to view a 3D object on a 2D surface, such as the screen or V.D.U., we must introduce some sort of projection that maps the 3D object onto a 2D projection plane. There are two main classes of projection that can be applied – these are parallel and perspective projection. The difference in these projections stems from the fact that these projections have a different relationship between the centre of projection and the projection plane. Fig 3.7 shows the difference between the two types of projections. You can also see the effect of perspective fore-shortening, which simply means that, all other things being equal, objects further away from the centre of projection appear smaller. This produces a very realistic effect as this is, in fact, what happens when looking at objects with the human eye. The disadvantage of this realism is, however, that you cannot take accurate measurements from the drawings because of the effect of perspective fore-shortening.

3.11.2 Perspective Projection

The most important thing in a perspective projection is the centre of projection, as this will determine how the final image will appear. In perspective projection, parallel lines converge to a vanishing point unless they happen to be parallel to the projection plane as well. If the set of lines is parallel to one of the the three principal axes, the point is called a principal vanishing point. Therefore the maximum number of principal vanishing points you can have is 3 – corresponding to the three such principal axes being cut by the projection plane. The type of projection that occurs depends on the number of principal axes cut by the projection plane and hence the number of vanishing points. For this reason you can get one, two or three point projections, depending on how many axes

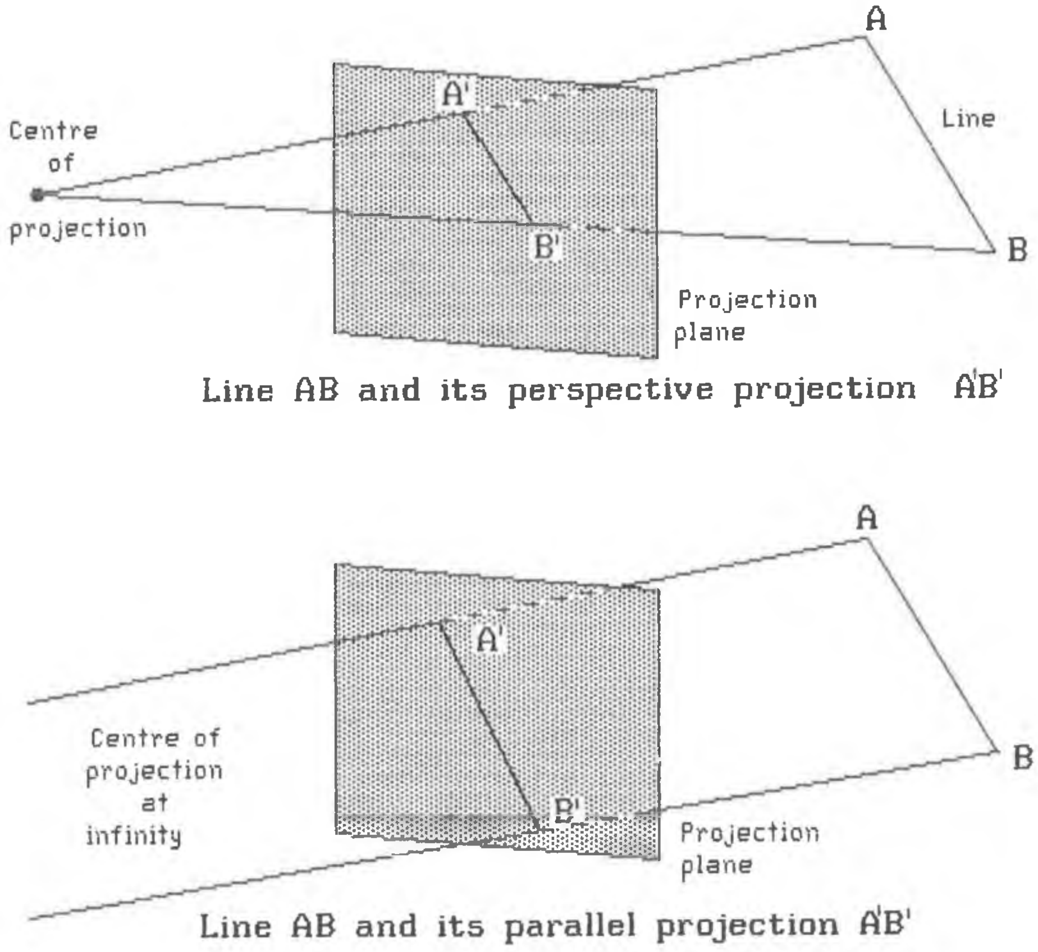

A
A'
Centre of projection
Line
B'
B
Projection plane
Line AB and its perspective projection A'B'
A
A'
Centre of projection at infinity
B
B'
Projection plane
Line AB and its parallel projection A'B'

Figure 3.7: The two basic types of projection

the projection plane cuts. Fig 3.8 shows the effect of a one point perspective projection on a cube and Fig 3.9 shows the effect of a two point perspective projection on a cube.

The mathematics of the perspective projection are in fact quite complicated. For simplicity's sake we will take the projection plane as normal to the Z-axis at a distance **d** from the origin.

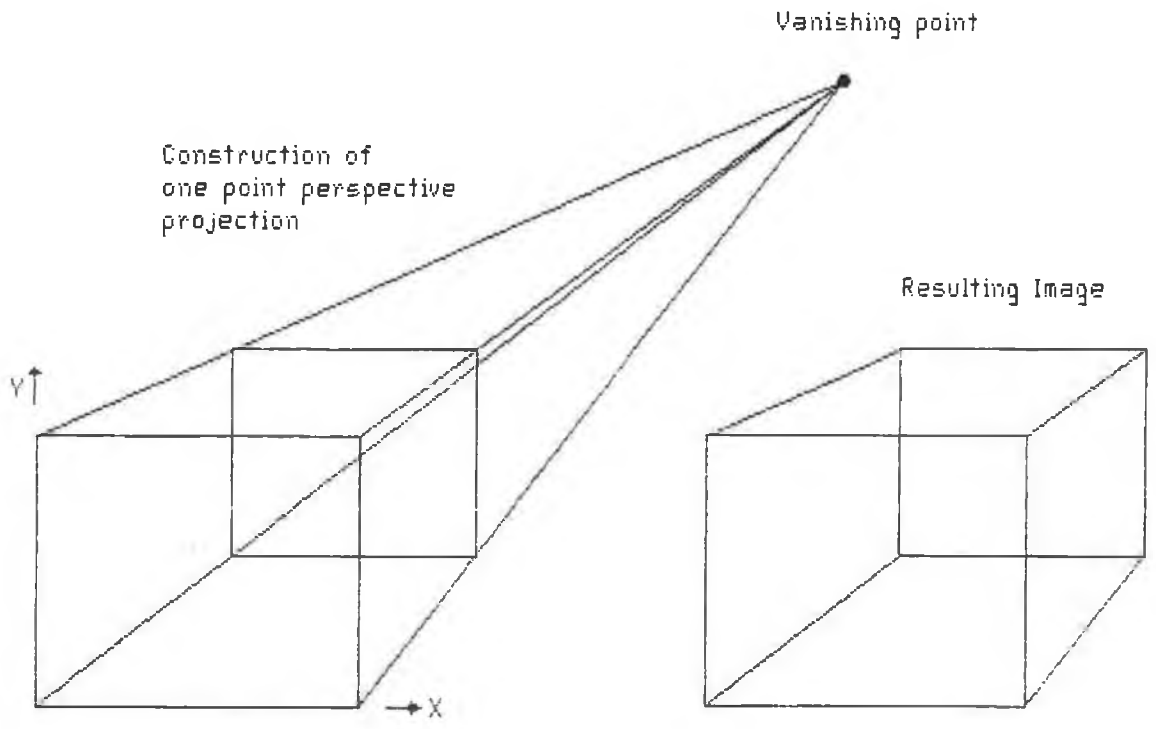


Figure 3.8: One point perspective projection of a cube

Fig 3.10 shows the projection plane at a distance d from the origin and a point P to be projected. To calculate the new point P1 we use similar triangles to yield :

$$\frac{X_p}{d} = \frac{X}{Z} \quad , \quad \frac{Y_p}{d} = \frac{Y}{Z}$$

Multiplying each side by d yields :

$$X_p = \frac{d\,.\,X}{Z} = \frac{X}{Z/d} \quad , \quad Y_p = \frac{d\,.\,Y}{Z} = \frac{Y}{Z/d}$$

Looking at this equation you can see that a point is scaled by its distance from the projection plane (Z/d). This causes the projection of more distant objects to

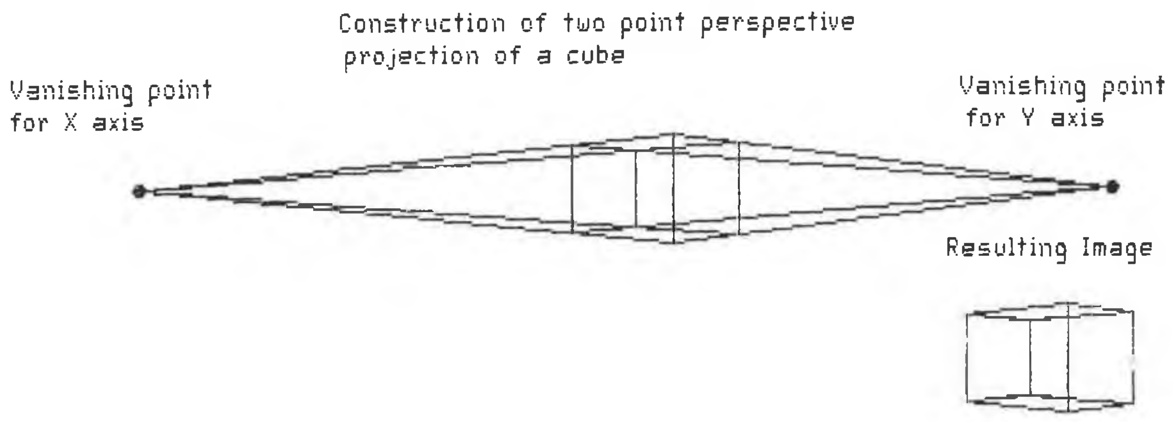

Construction of two point perspective projection of a cube
Vanishing point for X axis
Vanishing point for Y axis
Resulting Image

Figure 3.9: Two point perspective projection of a cube

be smaller than that of closer objects.

Converting to matrix format yields :

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{vmatrix}$$

Applying this to the point [x, y, z, 1] yields [x, y, z, z/d]. The point must then be divided by the scale factor (z/d) before it can be drawn. This increases the complexity of applying a perspective projection rather than a parallel projection and is treated as a special case by the COOGE system. Also it is slightly slower to apply a perspective projection than a parallel projection as you have to apply an extra division to each point before it can be drawn.

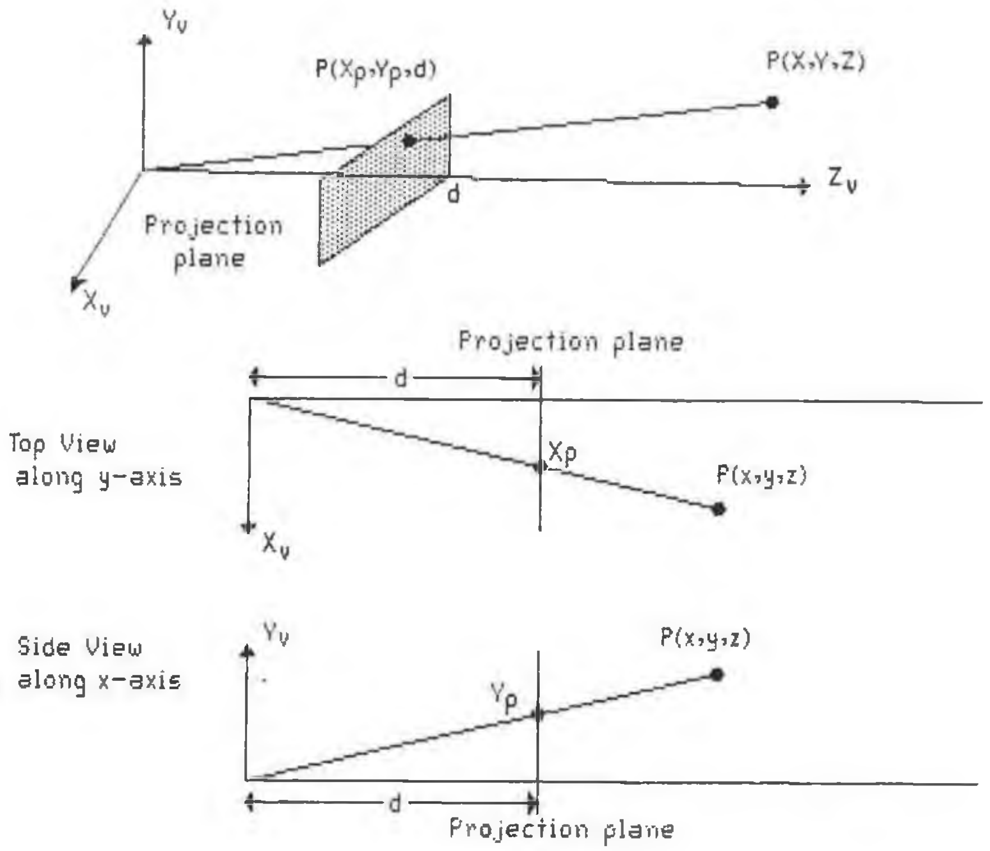


Figure 3.10: Perspective projection

3.11.3 Parallel Projections

There are two main types of parallel projections – orthographic, where the direction of projection is normal to the projection plane, and oblique where it is not.

3.11.4 Orthographic

The three most common cases of orthographic projection are when the projection plane is parallel to one of the principal axes. These allow you to produce front, top and side projection which are commonly used in engineering drawings, as measurements can be taken directly from the projection. The problem as already mentioned previously, is that only one face of the object is projected so no 3D information can be deduced from the drawing as shown in Fig 3.3.

Another common projection used is an isometric projection where the projection plane makes equal angles with all three axes. This gives a more realistic 3D effect as shown in Fig 3.11.

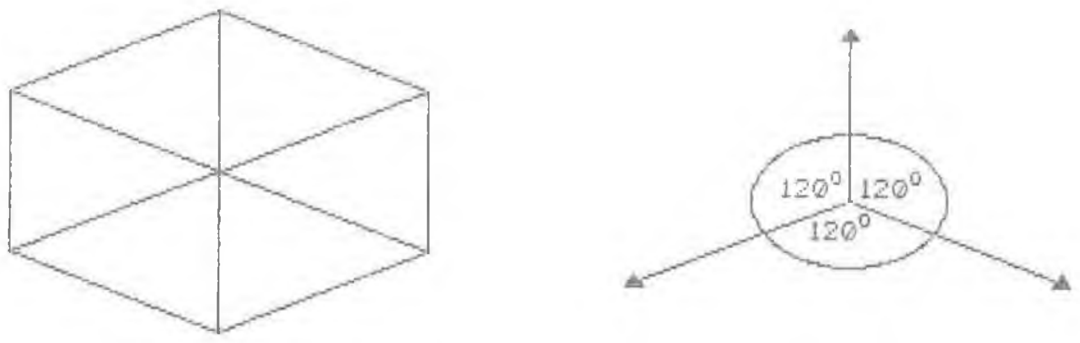


Figure 3.11: Isometric projection of unit cube

3.11.5 Oblique Projections

In oblique projections the projection plane is normal to a principal axis but not however to the direction of projection.

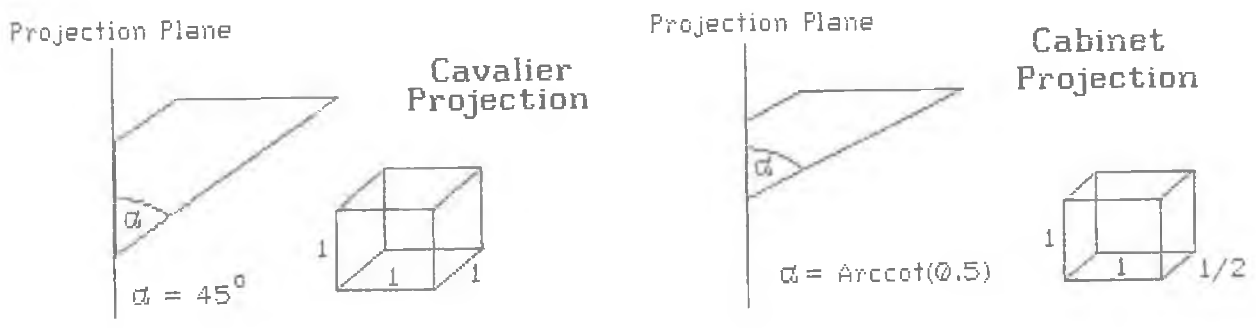


Figure 3.12: Cavalier and Cabinet projection of a cube

The two main projections are Cavalier & Cabinet. A Cavalier projection is where the direction of projection makes a 45 degree angle with the projection plane. Any line that is perpendicular to the projection plane is projected so that the length of the line remains the same. Again, because perpendicular distance is preserved and no fore-shortening takes place, it doesn't look realistic. In order to produce a more realistic image a Cabinet projection is generally used. Here the direction of projection makes an angle of arccot(0.5) with the projection plane. Using this projection any line normal to the projection plane is halved when projected. Fig 3.12 shows the effect of the two different oblique projections on a unit cube.

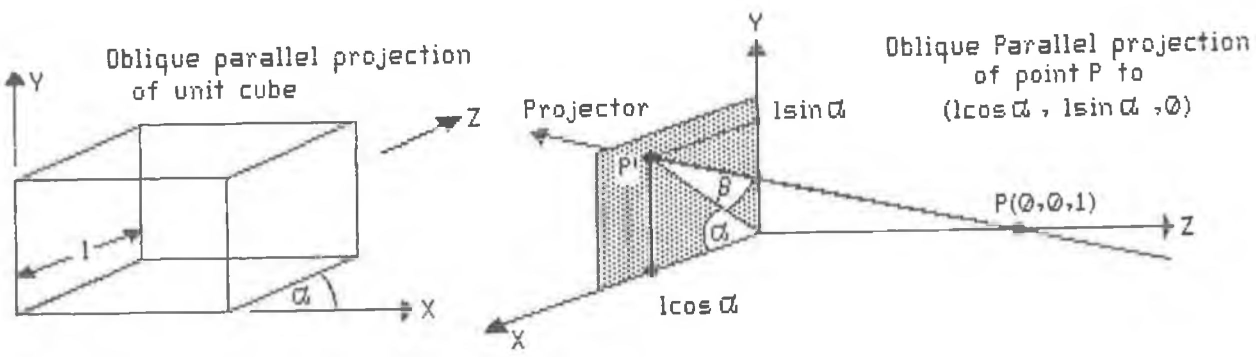

Oblique parallel projection of unit cube
Y
Z
l
α
X
Projector
Y
lsinα
Oblique Parallel projection of point P to (lcosα , lsinα ,0)
P'
β
α
P(0,0,1)
Z
lcosα
X

Figure 3.13: Construction of a parallel projection

Fig 3.13 shows the effect of a parallel projection on a unit cube and on the point (0,0,1). From Fig 3.13 we get the two equations

$$X_p = X + Z(l\,.\,cos\alpha) \quad , \quad Y_p = Y + Z(l\,.\,sin\alpha)$$

Converting this to matrix format :

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ l\,.\,cos\alpha & l\,.\,sin\alpha & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

The required projection can be determined by the value given to l. For the standard projections the value given to l is as follows:

Cavalier projection $l = 1$,

Cabinet projection $l = 1/2$,

Orthographic projection $l = 0$.

3.12 Conclusion

This chapter has outlined some of the functions required from any graphics system and the mathematics required for these functions. It is not enough just to hide the mathematics of these operations from the user if in doing so you also hide some of the power of these transformations. For this reason it is not just the range and type of functions provided that are important but also the user interface to these functions. It is no good providing a function that has an interface that makes it difficult to use, such a function may as well not have been provided in the first place. The following chapter discusses the user interface and how to make these transformation functions as accessible and user friendly as possible.

Chapter 4

THE OBJECT OF IT ALL

4.1 Introduction

The next stage was to design an object oriented graphics library that would be specifically geared towards CAD users. There are a number of issues which must be tackled when designing any graphics packages, regardless of its object oriented nature or its intended use. These issues were raised by Newman & Sproull [WNEW79] in an article on the principle of interactive graphics. These design principles, and how Object-oriented development was used to meet them, are discussed in this chapter.

4.2 The Design Priorities

The basic issues Newman & Sproull considered should be foremost in a designer's mind when designing a graphics system are :

Simplicity : Features that are too complex for the application programmer to understand will not be used. The basic rule is that any feature that you find difficult to explain will be difficult to use.

Consistency : A consistent graphics system is one that behaves in a generally predictable manner. Function names, calling sequences, error handling and co-ordinate systems should all follow simple and consistent patterns (without exception). The user should be able to build up a conceptual model of the graphics system and how it functions.

Completeness : There should be no irritating omissions in the set of functions provided by the system; missing functions will have to be supplied by the application programmer, who may not be in a position to write them. This does not mean that the designer should provide every imaginable graphics function that the user will ever require but rather should provide a reasonably small set of functions that can conveniently handle a wide range of applications.

Robustness : Application programmers are capable of extraordinary misuse of graphics systems, usually because of a lack of understanding of the system. These errors should only cause termination of execution in extreme circumstances as this will generally cause the user to lose valuable results.

Performance : A system's reponse and performance should not rely on the use of fast hardware or on a user having to resort to 'special tricks' to improve the performance of the system.

Economy : Graphics systems should be small and economical so that adding graphics to an existing application program can always be considered.

4.3 Meeting These Priorities

The next stage in the design process was to design a set of objects that would be used to implement the graphics system while at the same time being bound by all these design principles. The design principles themselves were met in the following ways:

Simplicity : One of the main considerations when using a graphics system is that each of the functions provided by the system must be easy and simple to use. For instance some of the functions require a number of optional parameters to provide full control. However, a user might only be interested in a simplified version of the function. The user need supply only the required parameters and COOGE will supply the default optional parameters.

Consistency : Consistency in the way functions are provided by the graphics system is very important. For this reason COOGE uses inheritance to ensure consistency in the treatment of functions. By using inheritance, one object can inherit a complete set of functions from another. If any of the underlying functions are changed, all objects in the system will inherit the new functions, which ensures consistent use of functions throughout the system.

Completeness : COOGE in no way attempts to supply every graphics function that a user will ever require. Instead it supplies a small set of graphic functions in line with what most users will need. The system is also designed in such a way that it is easy for users to design and add their own functions to the system without affecting existing code.

Robustness : In order to improve the robustness of the system a number of objects are designed to operate in the background without the user ever being aware of them. These objects are created as soon as the program is run. For instance, a SCREEN object was designed which looks after the screen informing other objects if the user changes the graphics mode etc.

When drawing on the screen, the COOGE system makes intelligent guesses, if the user makes a mistake with a particular command, to try to ensure that the drawing will actually appear on the screen. For instance if a window is opened on the screen the COOGE system will direct all future graphic operations to the window, automatically scaling and translating images so they appear in the window.

Performance : COOGE is designed in such a way that it does not rely on the use of any particular hardware, either graphical or mathematical, to run but it will take advantage of any special hardware it finds.

Economy : Despite the power of COOGE it is relatively compact in terms of memory and disk requirements and adds little overhead compared to using a standard library.

4.4 The Objects Required

With these objectives in mind the graphics process was examined to determine the objects that would be required in a graphics system. However, before any the graphics objects are designed, there are a number of objects that must be dealt with first. These are what I termed service objects – objects that provide functions that most of the higher objects will require.

4.4.1 Service Objects

A lot of the objects in the system require the use of lists to keep track of various pieces of information, for instance, you might want to keep a list of shapes that have to be drawn on the screen. For this reason a list object was designed that provides a number of functions that you would use on any list, and that can be included into any other object.

```
class list
{
     pointer to head of list
     pointer to current position in list

public :
       void insert(base*) // add at head of list
       void append(base*) // add to end of list
       base* get()        // remove object from list
       void clear()       // empty the list
       void reset()       // reset the current position in the
                          // list
       base* next()       // return pointer to next item in list
       void remove(base*) // delete an itme from the list
};
```

The list object is further refined by the system to yield a static list object which can be used to keep a static list of objects in the system.

4.4.2 The BASE Object

There are a number of functions that every object in the system will be required to perform. For instance you must be able to save and load an object to/from disk. Because of the way C^{++} is compiled, as explained in Chapter 2, no record is kept of the internal structure of objects in the system once compiled. This means that C^{++} doesn't inherently support loading/saving of objects. For this reason, you need some method for providing this function and ensuring that all other objects can inherit it. It was decided to provide a BASE object which all other objects will inherit. It is called BASE as it is the basis of all other objects in the system. Another function required by most objects is the ability to draw itself on the screen, and because this was such a widely used function it was decided to also include this in the BASE object.

```
class base {

public :
  virtual base* getobj(IMAGE*,
                       base*)   // load an object from disk
  virtual putobj(IMAGE*)        // save an image to disk
  virtual void draw()           // force objects to support a
                                // draw function
};
```

4.4.3 Saving Objects

The IMAGE referred to in the base class is an object which looks after the saving and loading of other objects to and from disk. As each object is stored to disk the IMAGE object labels the object just stored so that it can be reloaded in the future. Sometimes when writing a list of objects to disk it is useful to write a NULL object to disk to indicate that the list is finished. So the IMAGE object supports a NULL object. To load an object from disk, simply tell the IMAGE what object you want to load and it returns a pointer to the loaded object if it can load it or a NULL pointer if not.

```
class image {

 file*       // disk file to save/load object to/from
 filename
 label       // object just saved/loaded

public :

   void open(char*)  // open image file
   void rewrite()    // rewrite the image file
   int match(char*)  // check if object to be loaded matches next
                     // one on disk retun true /false

  base* loadobject(object name)  // load an object from disk

  void write(object address)     // save object to disk
}
```

This loading and saving of objects appears relatively simple. This is deceptive. When saving an object to disk you must also save those parts of that object that were inherited from other objects. This means that you must keep track of all the ancestors of an object i.e. all the objects it inherited and they inherited in turn etc. As already stated C^{++} provides no support for keeping track of the inheritance tree once compiled, so when an object is saved to disk COOGE ensures that any of its inherited parts are also saved by calling the save function for each inherited part of the object.

COOGE allows any image to be saved as a collection of objects. This provides a device-independent means of saving any image to disk and provides a standard means for image transfer between programs. For instance

```
image img;

  img.open('test.dat');        // save image to file test.dat
  scr.putobj(&img);            // save the screens contents
```

will save to disk any objects appearing on the screen in object format.

This ability also allows you to build up a library of commonly used objects which can then be included into any drawing. For instance you could build up a library of commonly used components in electronic circuits and use them in designing new circuits.

4.4.4 Loading Objects

Loading a previously saved object from disk is a more complex operation. The reason for this is that one object cannot access the private data of another object except in the special case when it is declared as a friend. This means that if you want to load a LINE, for example, from disk the only object that can check if it is a line, and then load it, is another line object. The reason for this is that only a LINE object has access to the internal representation and storage of its private data. For this reason a static list is set up in the system which keeps a copy of every type of object in the system. Any type of object can be added to the list by using the activate command. For example

```
   Activate(cylinder);
```

will allow instances of the cylinder object to be loaded from disk. The activate command works on a complete class rather than an instance of a class and need only be called once to ensure all instances of the class can be loaded from disk. Activating an object means that the system has to set memory aside to handle the loading and saving of instances of the object. For this reason the user can activate only those objects he/she will require. Some objects in the system, such as segments and windows, are automatically activated by the system to ensure that they can be loaded from disk at any time.

If an object is to be loaded from disk the image object asks each object on the list in turn if it is an instance of the object stored on disk. If it is, then the object is asked to load the object from disk and return a pointer to the new object. If an object has inherited a number of other objects, COOGE will ask each inherited part to also load itself from disk before returning a pointer to the new object.

4.4.5 Bitmaps

Another object that is required by a number of the main graphical objects is a BITMAP . This object takes a section on the screen specified by the user and saves it so that it can be restored at some later time to its original position – or even a different position on the screen.

```
class bitmap {
  pointer to saved image
  co-ords of stored area

public:

  int save(int,int,int,int)  // save a rectangular area of screen
  int restore()             // restore it to its original position
  int restore(int,int,int,int) // restore it to a different pos.
}
```

4.5 The Viewing Operation

The next task was to examine the graphics pipeline and to isolate the objects required to provide the functions required. As stated above, an object is defined by the actions it performs and requires of others. It would have been possible in theory to assign an object to each stage of the graphics pipeline and pass information from one object to another till the information was eventually drawn on the screen, but it was decided to treat the complete pipeline as one object which accepted a point or a line in world co-ordinates, then clipped, transformed and projected the point or line onto the screen.

The name given to this object was the 'VIEWSET' as it contained the set of transformations and viewing operations to be performed on a point or line.

4.5.1 The VIEWSET

The viewset contains information on the current rotation, scaling and translations being applied, as well as the viewport and the current projection. The advantage of this is that the complete viewing process was isolated in one object. This firstly meant that the mathematics and complications of the viewing operation were concentrated in one place and the internal mathematics themselves hidden from the user who only has access to the object through its interface. This means that new views can be added without affecting the user's own code. Secondly because all the mathematics are concentrated in the one object, the speed of drawing is improved as you do not have functions calls between a number of objects, as the VIEWSET processes the entire object.

A simplified version of the viewset is as follows

```
Class viewset : public base {

 Viewport co-ords
 Window co-ords
 Viewing matrix
 Rotation matrix                  // contains current projection
 Translation and Scaling factors
 Current transformation matrix    // matrix of combined
                                  // transformations

public :

 void scale(sx,sy,sz);        // set scaling factors in x,y,z dir
 void rotateabout(point,      //  rotate about a point in the Z dir
                  angle)
 void rotateaboutx(point,     // "         "   "    "    "   "  X dir
                   angle)
 void rotateabouty(point,     // "         "   "    "    "   "  Y dir
                  angle)
 void setvport(float,float,     // Set the viewport co-ordinates
              float,float);     // in NDC's

 void moveto(point)         // moveto a point on the screen
 void lineto(point)         // draw a line to the point
 void drawpoint(point)      // draw a point on the screen

 void setoblique(float,float);    // set viewing projections etc.
 void setcavalier(float);
 void setcabinet(float);
 void setortho();
};
```

As new transformations are added, or as existing transformations are changed, the current transformation matrix is updated immediately. This ensures that there is no loss of speed when applying a number of transformations to a graphics object.

4.5.2 The SCREEN

The VIEWSET object was not intended to be used by itself but rather to be incorporated/inherited into other objects that require these graphical operations. For example, it was decided to treat the screen itself as a distinct object. An instance of this object, called SCR, is created by the system before any of the user objects. This means that the SCREEN object can take control of the graphics system initialising all the parameters. For instance, one of the functions provided by the SCREEN is the ability to change from one graphics mode to another. The SCREEN can then inform other objects in the system that such a change has taken place and make any adjustments that are necessary. For instance, the function that maps the world co-ordinates to physical co-ordinates has to be informed about the change in resolution. It keeps details of the actual physical resolution of the screen so the viewset can map the world co-ordinates correctly to the physical screen co-ordinates. In this system the user can assign graphics objects to windows, segments etc. so it is drawn when the window opens or when the segment is drawn. If an object is not assigned, it uses the default SCREEN settings when being drawn.

```
class screen :
            public viewset {  // includes all viewset functions

   current graphics mode;
   colour information;

public :
  void init(device mode);  // set the graphics mode
  void refresh();          // refresh the screen
  void setbkcolor(color)   // set the background color
};
```

4.5.3 Sensitive Objects

As mentioned before, the screen must inform other objects in the system if the user changes the current graphics mode. In order to meet this goal, it was decided to set up a static list of objects in the system which must be informed when such a change takes place. A sensitize function is provided by the system for this end.

For example

```
object.sensitize()
```

will make an object sensitive to any changes in the graphics environment and it will be immediately informed when such changes take place. Some objects in the system, because of the nature of their use, are automatically sensitized when created, such as segments and windows. You can remove an object from this list by using the desensitize function.

For example

```
object.desensitize()
```

4.5.4 SHAPES

The most obvious kinds of objects that are required in a graphics system are fundamental SHAPE objects such as lines, squares, boxes, circles and polygons. Using these as a type of building block, more complex objects can be built. The basic requirement of a shape is that you should be able to draw it on the screen, but more advanced facilities are provided to allow you to rotate, scale and move shapes. Another consideration is colour. Each shape must also be able to hold the colour required if colour is supported by the system. The current SHAPE objects do not provide any support for deciding the edges of a shape or its faces, because hidden line removal is not supported. All shape objects are made from the generic SHAPE object which provides support for initialising the shape data structure (in this case an array of points) and for drawing the shape.

Lines, points and circles were implemented as distinct graphical objects, all derived from a base class called SHAPE which outlined the operations required for any graphical shape. Every shape in the system is capable of being scaled, rotated, translated and you can set its colour. A number of predefined shapes in 2D and 3D exist within the system and the user is free to create new shapes of his/her own and add them to the system. With regard to the circle object there are two choices available for drawing the circle – an adaptation of the standard Bresenham's algorithm [BRES77] which produces a very accurate result but takes time to draw, or a polygon approximation of the circle which draws very quickly but does not draw as accurately. In practice the polygon approximation would be used when to provide fast response when drawing, such as when rotating a picture, and the Bresenham algorithm can be used to provide the final output. The sphere drawing routine also allows you to choose between the speed of drawing and the accuracy required.

The default shape is

```
class shape : public base {

public:
        points                 // Co-ords of the shape
        color                 // color of the shape

 void draw()                // draw the shape
 void scale(sx,sy,sz)       // scale the shape in X,Y,Z direction
 void rotateabout(point,       //  rotate about a point in the Z dir
                   angle)
 void rotateaboutx(point,      //  "        "   "    "   "  "  X dir
                    angle)
 void rotateabouty(point,      //  "        "   "    "   "  "  Y dir
                    angle)
 void move(int,int,int)  // move shape relative
 void moveto(point);     // move the shape to a particular point
 void setcolor(color)    // set the shape's colour
};
```

As mentioned this shape is then used to create usable shapes such as lines :

```
class line : public shape {

public :
  virtual base* getobj(IMAGE*,
                         base*)    // load an object from disk
  virtual putobj(IMAGE*)          // save an image to disk
  void draw()             // draw the line
  line(point,point)       // initialize the line
  set(point,point)        // set the lines co-ords
};
```

4.5.5 SEGMENTS & SCENES

Sometimes it is useful to group a number of graphics objects into segments as mentioned in Chapter 3. Any number of graphical objects can be grouped together into a segment. It is possible to assign a name/id to a segment so that you can for instance define a segment called ‘table’ which draws a table. The segment also can be make invisible or visible on the screen by setting the segments ‘visibility’ setting.

```
class segment : public viewset {

 visible         // whether segment is visible/invisible
 id              // name/ id of the segment
 shape list      // list of the shapes added to the segment

public:

 void append(shape) // add a shape into the segment
 void clear()       // clear all shapes from segement
 void draw()        // draw the segment contents
 void setvisibility(switch); // set visibility on/off
};
```

It is also possible to group a number of segments together into a SCENE. You can then manipulate the SCENE as if it were just one big segment, as any operation you can apply to a segment can be applied to a SCENE.

4.5.6 WINDOWS

The next object to be designed was a window object, which is an object to allow the user to use part or all of the screen as a drawing surface. The window can pop-up, saving what was currently on the screen and restore the previous contents of the screen when it is closed. The screen can be divided into any number of windows (depending on the memory capacity of the user's machine) but you can only draw in the most recently opened window. If a window is open on the screen then all draw commands use this window for their output by default.

```
class window : public segment

     id           // id of the opened window
     save area    // data saved when the window is opened
     bitmap       // saved contents of the screen
```

```
public :
     void append(shape)   // add shapes/segments to the window
     void append(segment)
     void open()           // open/pop-up the window
     void close()          // close window + restore screen
};
```

Each window is assigned an ID to insure that you cannot draw in a window that was covered by a more recently opened window. When a window is opened the SCREEN is informed so that all unassigned commands to draw shapes are now assigned to be drawn in the current window.

4.5.7 FONTS

COOGE has a font object that is used to control the loading of fonts into the system by the user. A predefined Roman stroke font was designed to test the font object but the user can design his or her own fonts and add them into the system.

```
class font {

 font id
 static list of fonts already loaded


public:

     void load(font id)      // load a font from disk
     void draw(point,char*)  // draw string at point
     void setscale(sx,sy)    // set scaling factors for the font
};
```

In order to prevent a font being loaded twice, a static list is set up by COOGE that keeps track of all fonts as they are loaded by the user. When loading a font, the font object checks this list to see that the font has not already been loaded.

The font class itself uses another service class called FONTREC to actually load the font from disk.

4.5.8 Drawing Text

To actually draw a string on the screen you must use a TXT object which is derived from the shape class.

```
class txt : public shape {

 char* strng                // text to be written
 font id                    // font to be used

public :

 void draw()  // draw the text string
 void settxt(font id,point,char*)    // set the text to be drawn

}
```

Using this design for the text object meant that the text is treated just like any other shape in the system, which means it can be freely incorporated into drawings, segments, windows, etc. , without the need for any special commands or instructions.

When drawing a TXT object, the font list is checked to find the font required and the string to be written is then passed to the required font object so that it can be drawn on the screen.

The overall list of objects in the system and their hierarchy is as shown in Figure 4.1.

4.6 Conclusion

This chapter outlined the design objectives that must be kept in mind when designing any graphics system. COOGE was designed to meet these objectives using object oriented techniques. All objects in the system can communicate to inform each other if any changes take place in the graphics environment. A lot of these objects are designed to operate in the background without the user ever being aware of them. These objects are designed to help the user rather than hinder him/her and attempt to protect the graphics environment from accidental damage and loss of information. The main graphics objects themselves attempt to provide all the power of a CAD graphics library in an easy to use format. The actual implementation of the system itself is discussed in the following chapter.

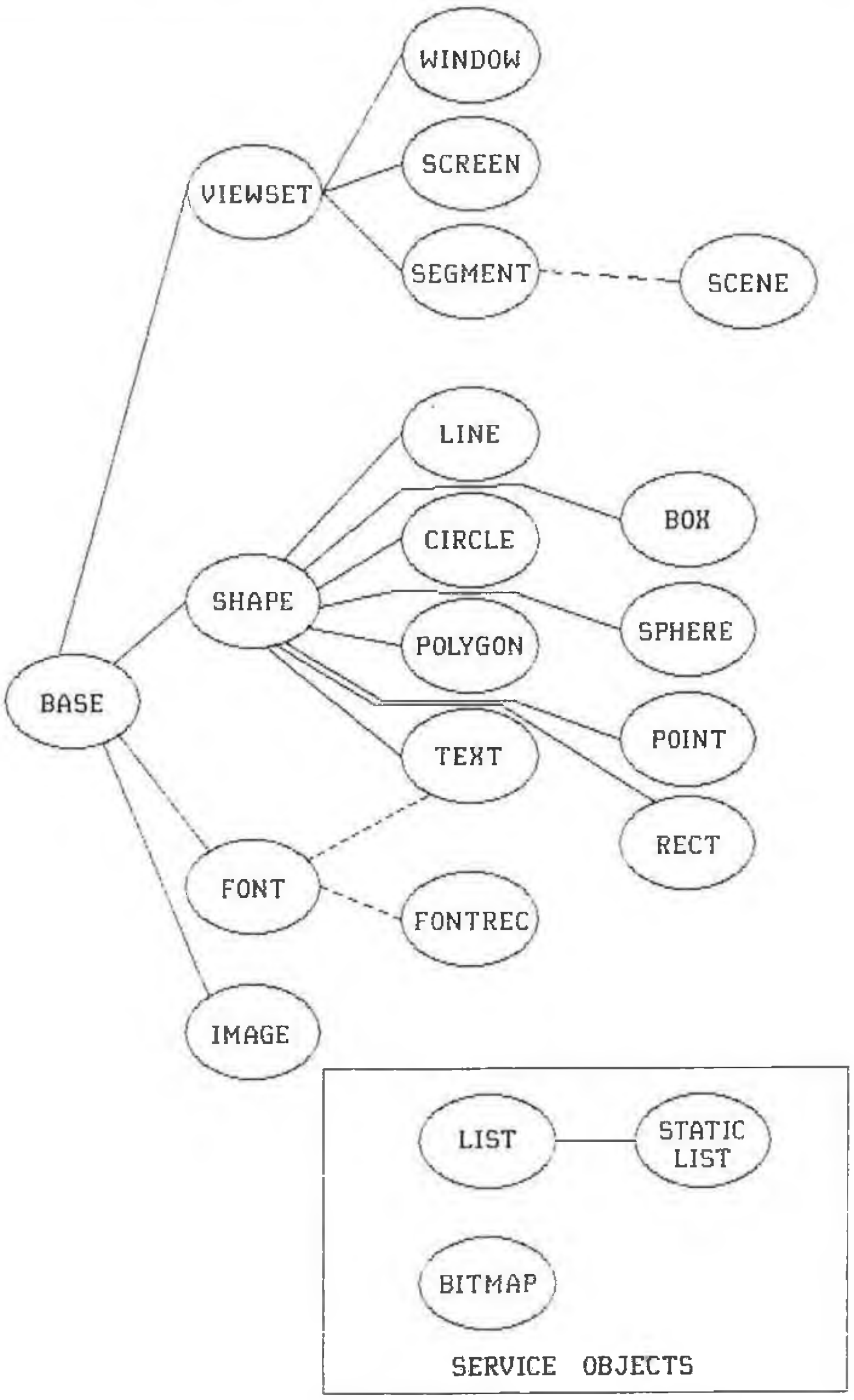


Figure 4.1: System hierarchy of objects

Chapter 5

IMPLEMENTATION

5.1 Introduction

Having designed the objects that were required by the graphics system and the functions they were required to perform, the final task was to actually program and implement the system. As stated, the system was implemented using C^{++}. This chapter outlines some of the implementation problems associated with the system and gives a few simple examples of the system in use.

5.2 The Compiler

The actual version of C^{++} used was a version supplied by Glockenspiel and was originally run on an 8086 based machine. It was then moved to an 80386 based machine, an IBM PS2 system 80, to improve the speed of program compilation. Even with this changeover, RAM disks had to be used to provide a reasonable speed of compilation. The C^{++} compiler is not actually a compiler as such, rather it is a preprocessor. It takes the C^{++} program code and converts it to Microsoft C compatible code. This code must then in turn be processed by the Microsoft C compiler before an executable version of the program can be produced. During this implementation stage a number of problems occurred which required a re-design of some parts of the system.

5.2.1 Implementation Problems

One of the main problems in using the C^{++} compiler was its need for vast quantities of memory even to compile small programs. For instance, a lot of the functions that the objects provide had been designed as inline functions in order to improve the speed performance of the system. However, it soon became clear that the compiler used up so much memory in compiling inline functions that it would run out of memory if more than a handful of inline functions were included in a program. Thus functions calls had to be substituted for inline functions and in so doing slowed down the overall performance of the system. Even without inline functions all memory resident programs including keyboard drivers had to be removed from memory in order to give the compiler enough memory to compile the program, even after it had been broken up into a number of small individually compilable modules. Some of the code was in fact broken into small modules for no logical reason except to ensure that the number of lines in each module was small enough to be processed.

Another problem was that the original design of the objects took advantage of multiple inheritance in C^{++} to create some of the objects but the version of the compiler used did not support multiple inheritance. This meant that some of the objects had to be redesigned to simulate multiple inheritance so that the user is presented with the original intended interface. The only disadvantage that the user will notice is that it took a lot more code to simulate multiple inheritance than if multiple inheritance had been used, and ended up with a less efficient program.

Finally, the actual version of C^{++} used to produce the code was a beta test version which meant that when an error occurred it took a while to figure out whether the preprocessor or the code itself was incorrect. Also a lot of the error messages were obscure to say the least. For example during one compilation the following error message appeared

```
''Error in token 128 Ooops error in the error handler !''
```

It took a while to get to the bottom of that particular problem, which was in fact a missing semi-colon.

5.3 Which Graphics Library ?

Having designed the objects required to perform the graphics functions, the next stage was to find a graphics library that the objects could 'sit on top of' to provide an object oriented graphics environment. A number of existing graphics libraries were examined including Dr. Halo and Metagraphics. After some research I decided to use the standard graphics library provided by Microsoft C with version 4.00 of the compiler.

The routines required by COOGE from the library and the objects that require them are as follows :-

The Viewset

MoveTo(x,y)	– Move the cursor to a position x,y on the screen
LineTo(x,y)	– Draw a line from the graphics cursor to the position x,y
Viewport(x1,y1,x2,y2)	– Set the viewport to the rectangle specified
SetColor(color)	– Set the current drawing colour

The Screen

InitDevice(device)	– Set the graphics mode
SetBkColor(color)	– Set the background colour

The Bitmap

GetImage(x1,y1,x2,y2)	– Save a copy of part of the screen
PutImage(x1,y1,x2,y2)	– Restore an image to the screen

These routines are used internally by COOGE and are not used directly by the user. This means that the underlying graphics library can be replaced at any time without affecting the user's own code, which in turn means it is relatively simple to port COOGE from one machine to another.

As an example to illustrate this, COOGE was originally programmed using a graphics library called MetaWINDOW supplied by MetaGraphics [META88]. Halfway through the implementation stage, is was decided to move to the standard Microsoft C graphics library. The change from one library to another took a matter of hours rather than days, with only three objects in the system being affected. Also it was only the private implementation of this objects which had to be changed with the public interface remaining the same and hence the user's code was unaffected.

5.4 Modularization

The program code itself is broken into two parts – the header files, which the user requires to define the objects used in the system, and the private code which contains the actual implementation of the objects. The COOGE system was broken up into a number of smaller modules which means the user only has to include those parts of the system he/she requires. The system is broken up into the following modules:

STDINC.HXX	–	Standard basic COOGE system (must be included) such as SCREEN and service objects.
SEGLIST.HXX	–	Contains segment, scene and window definitions.
SHAPES.HXX	–	Standard graphics shapes defined by the system.
IMAGE.HXX	–	Provides support for loading/saving objects.
FONT.HXX	–	Supports loading and using of character fonts.
TEXT.HXX	–	Graphics text object definition.

5.5 The System In Action

In order to give a clearer example of how the system works I will use an example program that draws a box on the screen in 3D and then rotates the box by 90 degrees about the X, Y and Z axis, respectively, and then finally the rotated box is drawn on the screen. The program to accomplish this is as follows :

```
#include <stdinc.hxx>  // standard include files
#include <seglist.hxx>
#include <shapes.hxx>

main()
{
  box bx(point(300,500),
         point(600,700),100);  // define box to be drawn

  scr.init(_HRESBW);           // set the screen to 640 x 200
                               // mode

  scr.setwindow(0,0,800,800);  // set an 800 x800 window
  scr.setcabinet();            // set a cabinet projection for
                               // the screen

  bx.draw();                   // Draw the box
  getch();                     // wait for key to be pressed

  bx.rotateaboutx(400,0,90);   // rotate the box about the X axis
  bx.rotateabouty(400,0,90);   //   "     "   "    "    " Y axis
  bx.rotateabout(400,400,90);  //   "     "   "    "    " Z axis

  scr.refresh();               // Clear the screen

  bx.draw();                   // draw box in new orientation

  getch();                     // wait for key to be pressed
}
```

The first thing to notice is the fact that you can specify details of an object including its co-ordinates at the time it is created, as in the box object.

As mentioned before in Chapter 4, there is a default screen object call SCR created before all other objects in the system so that this object can take control of the graphics system. The user can create another screen object to override the system screen if desired. The first thing SCR does is to interrogate the computer to find out what mode the screen is currently in, the screen resolution and colour information. It passes this information on to any other objects that then require it. If the user then changes the screen mode, as in the example, then all objects in the system are notified of the change and thus react accordingly.

In the screen object or any other object that uses a window, if no window is specified by the user, then the resolution of the screen is used as the default window. Similarly for the viewport, if no viewport is specified the complete screen is used for the viewport by default. In this example we define a 800 x 800 window and we take a quarter of the screen centred on the middle as the viewport. Graphics object such as segments, windows, etc., can each have their own individual viewports and windows.

As you can see, specifying the 3D view required is relatively simple. In the example program, a cabinet projection was specified but it is just as easy to specify an isometric or oblique projection or even a particular projection of the user's own making. All objects in the system that are now drawn on the screen will be drawn using a cabinet projection unless they specify their own projection.

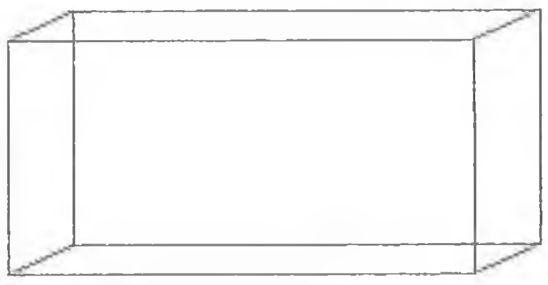


Figure 5.1: Cube generated

The box is now drawn on the screen using the draw command which works with all shapes and graphical objects. See Fig 5.1

To rotate an object is relatively easy as shown in the examples. There are two ways of rotating an object. You can rotate the object relative to the previous rotations as we do in our example. So the box is rotated by 90 degrees in the x, y and z plane. You can also specify an absolute rotation so that it is rotated to that absolute position. For example,

```
bx.rotateaboutto(400,400,90);
```

would rotate the box to 90 degrees off the horizontal in the Z direction. You can also specify negative rotations. Alternatively, rather than rotating the box, you could have specified a rotation for the screen, and all objects drawn on the screen would be drawn with the specified rotation.

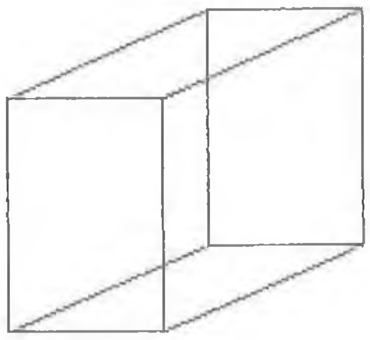


Figure 5.2: Rotated cube generated

The screen is refreshed to clear the first box from the screen and the rotated box is now drawn on the screen using the draw command. See Fig 5.2

It is possible to add the box to the screen so that every time the screen is being refreshed the box will be drawn. Using the command

```
scr.append(&bx);
```

adds the box to the screen's refresh cycle.

Another thing to note is that each object also has a destructor which looks after the object when it is being deleted. The screen object which operates in the background is one of the last objects destroyed as the program is terminating. The screen object destructor restores the screen to its original mode and colour. This means the user does not have to worry about restoring the screen to its original mode as it is all done automatically.

5.5.1 The Use Of Segments

Sometimes, as mentioned before, it is useful to segment the display so a number of objects can be displayed simultaneously. For instance, you might want to display a number of different views of the one object. The following example displays a box using two different views and two different rotations for each view.

```
#include <stdinc.hxx>  // standard include files
#include <seglist.hxx> // header file for segments
#include <shapes.hxx>  // header file for shapes

main()
{

  segment seg1,seg2,seg3,seg4;   // segments to be used
  scene scn;
  box bx(point(200,120),
         point(600,400),150);  // define box to be drawn

  scr.init(_HRESBW);           // set the screen to 640 x 200
                               // mode


  seg1.setcabinet();                // Set first segment
  seg1.append(&bx);                 // add box to segment
  seg1.setvport(0.0,0.0,0.5,0.5); // set the viewport

  seg2.setcabinet();               // use cabinet projection for
  seg2.append(&bx);                // first two segments
  seg2.setvport(0.0,0.5,0.5,1.0);
  seg2.rotateabout(320,100,45);

  seg3.setcavalier();               // use cavalier projection for
  seg3.append(&bx);                 // last two segments.
  seg3.setvport(0.5,0.0,1.0,0.5);

  seg4.setcavalier();
  seg4.append(&bx);
  seg4.setvport(0.5,0.5,1.0,1.0);
  seg4.rotateabout(320,100,45);

  scn.append(&seg1);       // add segments to the refresh cycle
  scn.append(&seg2);
  scn.append(&seg3);
  scn.append(&seg4);

  scn.refresh();               // draw the segments

  getch();
}
```

The program starts by defining the box to be drawn and initializing the segments, their viewports, rotations and contents.

The segment has to be informed that the box is to be part of the segment and to be drawn each time the segment is refreshed. To add any object to a segment, window or screen and put it on the object's refresh list (the list of objects that must be redrawn if the object is refreshed) you simply append it as shown in the example. The object can be removed from the list by using the delete command. For example seg.delete(&bx) will remove the box from the segments refresh list.

We finally add the segments themselves to the scene's refresh cycle so that each time the scene is refreshed the segments will also be drawn.

Finally by refreshing the scene we get the segments and their contents to appear on the screen as shown in Fig 5.3.

This example is a relatively trivial one with only one box in each segment.

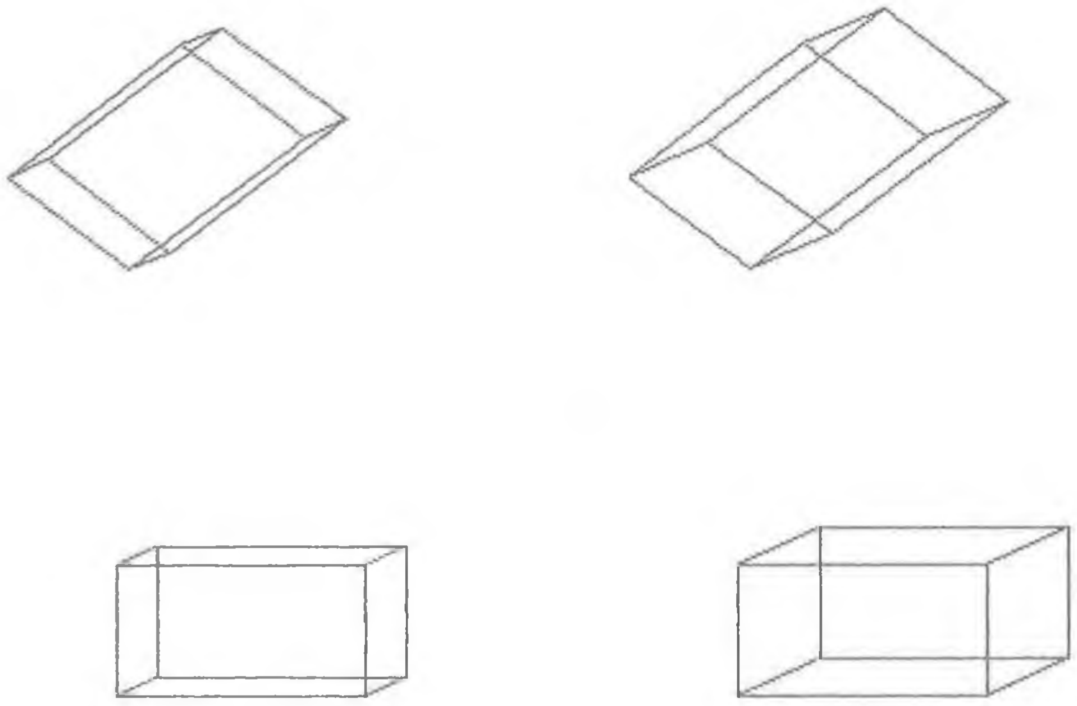


Figure 5.3: Generated image of cubes

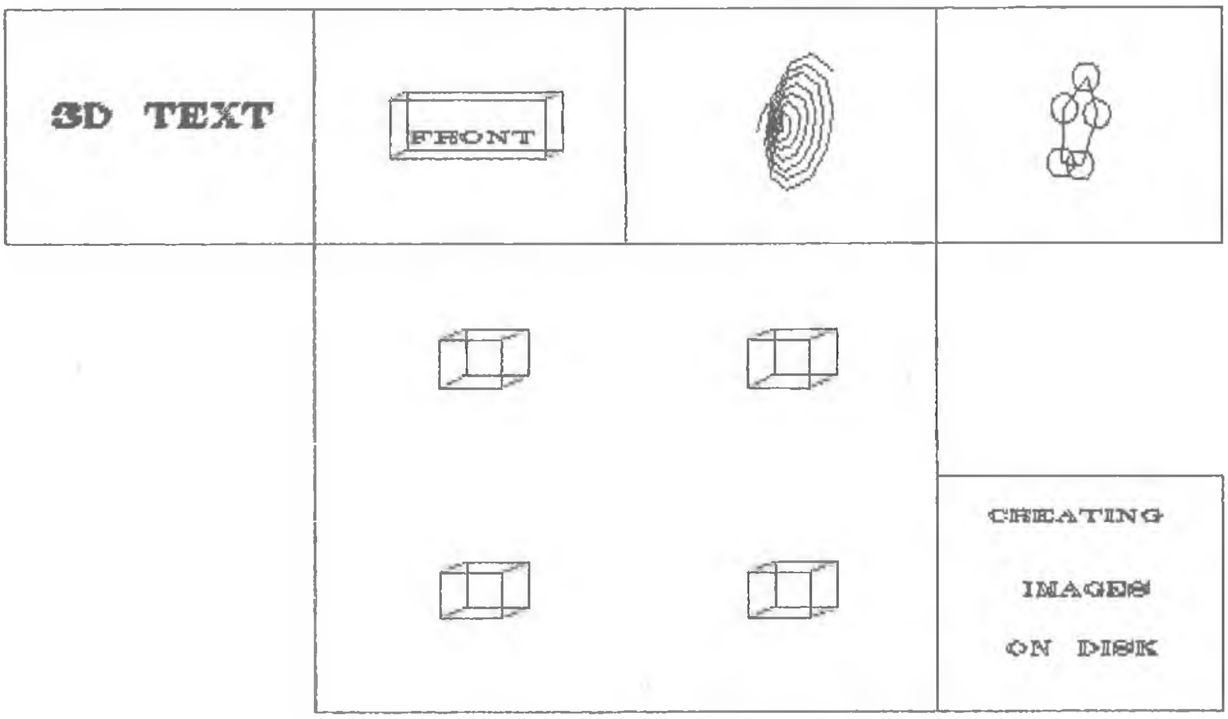


Figure 5.4: Sample screen from demonstration program

Each segment could in fact contain a very complicated drawing such as a house and you could use various segments to simultaneously view the house from a number of different angles and viewpoints.

A number of example programs are included in the appendix to demonstrate the use of windows, fonts, loading and saving objects to and from disk, viewing objects from different angles and how to use the various projections available. Fig 5.4 shows an example screen from the demonstration program written using the COOGE objects.

5.6 Conclusion

This chapter is designed to give some idea of how the actual objects in the system can be used to create graphical images. These graphics objects and their interfaces are intended to give the user the comfort of using a package while at the same time providing the freedom of a programming language. Hopefully from the two examples the reader can get some idea of how the system can actually be used. The last chapter takes this one step further and examines the further development of object oriented programming and future enhancements to the COOGE system itself.

Chapter 6

THE FUTURE

6.1 Introduction

COOGE provides a set of objects that provide primitive graphic functions and a range of transformation functions that can be used to manipulate a graphical image. The image itself can be stored in device-independent object format on disk and finally the user can use the library support to build up a library of commonly used images that can be incorporated into future images. COOGE was an attempt to show that such a library of routines/objects was possible and that it could in fact be used as the basis for a device-independent CAD system. This chapter examines possible enhancements to the system and what impact current trends will have on object oriented graphics.

6.2 Towards Object Oriented Graphics

Object oriented development is being used more and more in the design of graphics libraries. Most graphics libraries in fact mention in their advertising literature the fact that they used object oriented development in their design such as Metagraphics MetaWINDOWS [META88]. This internal use of object oriented methods within the library has yet however to be reflected in their use in an object oriented way by the end users.

6.3 COOGE Enhancements

COOGE in no way attempts to supply the ultimate graphics library. Instead it attempts to provide a small set of powerful functions that would be of use to any user. There are a number of enhancements which could be made to the COOGE system to improve it usability and flexibility, and they are outlined below.

6.3.1 The Shape

The shape object currently uses a set of points to define a given shape. The shape object could be improved in a number of ways, firstly by providing hidden line removal and secondly by supporting solid modelling. In order to meet these goals it would be necessary to incorporate the idea of 'faces' and vertices into the shape object. These changes would be internal to the shape object and would not affect the user's end code. The actual implementation of hidden line removal would also affect the viewset object which is in charge of the graphics pipeline. This in turn would require a closer relationship between the viewset and shape objects.

6.3.2 Windows

Currently it is only possible to draw in the most recently opened window but a more complex window system could be implemented which would allow a user to draw in any uncovered part of any window on the screen. Scroll bars and pop-up menus could also be employed adding to the usability and flexibility of the system.

6.3.3 Input Objects

As it stands COOGE provides no support for input into the system except for the standard input routines provided by 'C' itself. It would be possible to design an object that supports the keyboard and another object to support mouse input. The usefulness and portability of the system would be improved by using these device-independent input objects, with the system not having to rely on any particular mouse or keyboard design for input.

6.3.4 Multitasking

In the future as more multitasking hardware becomes available and concurrent object oriented programming languages become more widespread it should be possible to redesign COOGE to take advantage of this without too much difficulty. This would significantly increase the speed of the system. As it stands most of the program's drawing time is used up not in actually drawing a point but in just calculating where a point in the 3D space will actually appear on the screen. By examining the drawing process you will discover that each point needs at least 21 mathematical operations performed on it before it can actually be drawn, which is extremely time consuming. If hidden line removal and solid modelling were supported then the amount of mathematical operations would increase still further.

A concurrent version of C^{++} could be used to clear this bottleneck by re-designing the VIEWSET so it can process a number of points simultaneously and then pass the transformed points to the display hardware so they can actually be drawn. This would significantly increase the drawing speed of the system.

6.4 Similar Systems

HOOPS [HOOP89] (Hierarchial Object-Oriented Picture System) is a library of routines that provides support for three-dimensional imaging in both C and Fortran. HOOPS provides a set of routines that allows you to store three-dimensional objects in a database. Objects in the database can then be displayed on the screen (or printer) in one or more viewports, from any point of view, using orthographic or perspective projection. HOOPS also allows you to display the object as a wireframe or shaded solid.

HOOPS supports 4 basic graphics entities, polylines, polygons, pixel arrays, and text strings. The user can build his/her own shapes using these basic building blocks. Segmentation is also supported with segments containing graphical objects or further segments. All graphical information is organised in a hierarchial format with each child segment inheriting all the attributes of its parent such as colour, scale and orientation.

This product is being used as a base from which CAD system designers can build three-dimensional, device and machine independent CAD packages.

A similar system to COOGE was developed for the Macintosh computer in America but for a rather different reason.

It was found that when designing graphics programs for the Macintosh that a lot of the code was quite similar in these programs but it was not reusable so the programs had to be coded from scratch each time. In order to improve this situation an object oriented front end was designed for the Macintosh's standard graphics library. This met two goals, firstly providing a means of reusing existing code, and secondly providing a simple and user friendly interface to the complicated world of Macintosh graphics. The resulting product was called MACAPP [MACA88], programmed in objective pascal, which was designed to help in the production of Macintosh application programs. MACAPP automatically handles standard Macintosh user-interface features such a menus, desk accessories, scrolling, resizing windows and printing. MacApp also includes a high level debugger to help the applications programmer.

This example only shows one area where object oriented development can benefit a designer – namely in the reusability of code. More and more applications for object oriented development are being found and people are beginning to turn to these methods to find solutions to a number of programming problems.

6.5 The Language

Object oriented languages are still in their infancy and as such are constantly changing and improving. Only in recent years has the required support you need for any language become available. These include symbolic debuggers and program development aids.

For instance as already mentioned the C^{++} compiler actually preprocesses the code and converts to Microsoft code. At present no debugger is available for the C^{++} code and you actually have to debug the 'C' code itself. Just to give you some idea of the enormity of this task the following example shows two lines of C^{++} code and the resulting 'C' code.

C^{++} Code

```
Void chartab::clear()
{
  for(int i=0; i<127; i++)
    if (entry[i] != (fontrec*)NULL) delete entry[i];
}
```

```
Resulting C Code

void pascal _chartab_clear(struct chartab *_au0_this)
{ int _au1_i;
for(_au1_i=0;_au1_i<127;_au1_i++)
if((_au0_this->_chartab_entry[_au1_i])!=(((struct fontrec *)0L)))
(((_au0_this->_chartab_entry[_au1_i])?((((_au0_this->
_chartab_entry[_au1_i])->_fontrec_next!=(((struct fontrec*)0L)))
? (_fontrec__dtor((_au0_this->_chartab_entry[_au1_i])->
_fontrec_next,1),0):0),((_au0_this->_chartab_entry[_au1_i]) ?
(_delete((void*)(au0_this->_chartab_entry[_au1_i]))):0)):0));
```

Other compilers have attempted to accelerate the speed of compilation by compiling the C^{++} code direct to an executable format but these too are still in their infancy and will require further improvements before they become more widely accepted.

When using object oriented development techniques you can create or design objects which could be potentially used by a number of different users. A number of commercial companies have now been set up to produce libraries of objects that can be sold to users. These objects provide support for intercomputer communications, printers, the keyboard and many more. For example, a library of objects called PforCe++ [PFOR87] is available which supports communications, text windows, databases and pop-up and pull-down menus along with many others. The commercial availability of these objects means that firstly users do not have to design their own, and secondly the development process is speeded up as the user only has to concentrate on those objects that are specific to the current system.

6.6 Conclusion

Object oriented languages are the programming languages of the future. COOGE was an attempt to show the impact that these languages and development methods can have on a graphics environment. COOGE exploits all the power of C^{++} to create a set of graphics objects which provide the user with a powerful graphics library of routines. This chapter discussed how this could be further enhanced by improving the objects within the system and by improving the language and its supporting components.

Appendix A

SOURCE CODE

BLIST.HXX

```
//*************
// BLIST.HXX
//
// DEFINES A STATIC LIST OBJECT
//
// ***************

class blist {
  slink* last;                    // last->next is head of list
  slink* vptr;                    // current position in scanning list;

public:
    void remove(base*);                  // remove element from list
    void init();                         // Init the list;
    void append(base* a);                // add at tail of list
    void clear();                       // remove all links
    void reset() { vptr = 0;}
    base* next();
};
```

```
//****************************
// Prog : BLIST.cxx
//
// Creates a static list object
//
// Provides functions to
//
//    add to the begining/ end of the list
//    step through the list
//    delete from the list
//    initialise the list
//    set error handlers for the list
//    clear the list
//
//***************************


class base;

void blist::init()
{
  last = (slink*) NULL;
  vptr = (slink*) NULL;
}

base* blist::next()
{
 slink* ll;
        if (vptr == (slink*) NULL)
           ll = vptr =(last) ? last->next : (slink*) NULL;
        else {
            vptr = vptr->next;
            ll = (vptr==last->next) ? (slink*) NULL : vptr;
        }
        return ll ? (base*)ll->e : (base*) NULL;
};

void blist::append(base* a)
{
   if (last)
      last = last->next = new slink(a,last->next);
   else {
      last = new slink(a,(slink*) NULL);
      last->next = last;
   }
}

void blist::clear()
{
   slink* l = last;
   if (l == (slink*) NULL) return;
   do {
       slink* ll = l;
       l = l->next;
       delete ll;
    } while ( l !=last );
  last = vptr = (slink*) NULL;
}

void blist::remove(base* fp)
{
 slink* p,*ll= last;
 int found = 0;

 if (last)
 {
   p = last->next;
   while ((p!=(slink*) NULL) && (!found))
   {
     if (p->e == fp)
     {
       if (ll != p)
       {
           ll-> next = p->next;
           if (p == last)
              last = ll;
       }
       else
       {
           last = (slink*) NULL;
       }
       found = 1;
       delete p;
     }
     else
     {
       ll = p;
       p = (p->next == last->next) ? ((slink*) NULL) : p->next;
     }
   }
 }
}
```

CIRCLE.CXX

```
//*****************************
// Prog : circle.cxx
//
// Draws a circle on the screen
// using conny's version on the Bresenham algorithm
//
//****************************

#include "stdinc.hxx"

static int ychange;
static point pt[8];
static int old[8][2];
static int i;

//#pragma check_stack(off)

void circlepoint()
{
     if (ychange) {
     pt[0].inc(1,-1);
     pt[1].inc(1,1);
     pt[2].inc(-1,-1);
     pt[3].inc(-1,1);
     pt[4].inc(-1,1);
     pt[5].inc(-1,-1);
     pt[6].inc(1,1);
     pt[7].inc(1,-1);

     }
     else
     {

     pt[0].inc(1);
     pt[1].inc(1);
     pt[2].inc(-1);
     pt[3].inc(-1);
     pt[4].inc(0,1);
     pt[5].inc(0,-1);
     pt[6].inc(0,1);
     pt[7].inc(0,-1);
     }
     for (i=0; i<8; i++)
     {
        _moveto(old[i][0],old[i][1]);
        _lineto(old[i][0] = shape::viewptr->windowx(pt[i]),
                old[i][1] = -shape::viewptr->windowy(pt[i]));
    }
}

//#pragma check_stack()

void circledraw( point p ,int r)
{
 int x1,y1;
  register int d;
  y1 = r;

  d  = 3-(2*r);
  x1 = 0;
    pt[0].set(p.x-1,p.y+y1,p.z);
    pt[1].set(p.x-1,p.y-y1,p.z);
    pt[2].set(p.x+1,p.y+y1,p.z);
    pt[3].set(p.x+1,p.y-y1,p.z);

    pt[4].set(p.x+y1,p.y-1,p.z);
    pt[5].set(p.x+y1,p.y+1,p.z);
    pt[6].set(p.x-y1,p.y-1,p.z);
    pt[7].set(p.x-y1,p.y+1,p.z);

    for(i=0;i<8; i++)
     {old[i][0] = shape::viewptr->windowx(pt[i]);
```

```
      old[i][1] = -shape::viewptr->windowy(pt[i]);}

   ychange =0;
   while (x1<y1) {
      circlepoint();
      if (d < 0) {
          d+=(4*x1)+6;
          ychange = 0;
      }
      else {
            d+= (4*(x1-y1))+10;
            y1--;
            ychange =1;
      }
      x1++;
   }
       ychange =1;
       circlepoint();
}

#pragma check_stack(off)

void circlezpoint()
{
     if (ychange) {             // z change in this case
     pt[0].inc(1,0,-1);
     pt[1].inc(1,0,1);
     pt[2].inc(-1,0,-1);
     pt[3].inc(-1,0,1);
     pt[4].inc(-1,0,1);
     pt[5].inc(-1,0,-1);
     pt[6].inc(1,0,1);
     pt[7].inc(1,0,-1);

     }
     else
     {

     pt[0].inc(1);
     pt[1].inc(1);
     pt[2].inc(-1);
     pt[3].inc(-1);
     pt[4].inc(0,0,1);
     pt[5].inc(0,0,-1);
     pt[6].inc(0,0,1);
     pt[7].inc(0,0,-1);
     }
     for (i=0; i<8; i++)
     {
        _moveto(old[i][0],old[i][1]);
        _lineto(old[i][0] = shape::viewptr->windowx(pt[i]),
                old[i][1] = -shape::viewptr->windowy(pt[i]));
    }
}

#pragma check_stack()

void circlezdraw( point p ,int r)
{
 int x1,y1;
  register int d;
  y1 = r;

  d  = 3-(2*r);
  x1 = 0;
    pt[0].set(p.x-1,p.y,p.z+y1);
    pt[1].set(p.x-1,p.y,p.z-y1);
    pt[2].set(p.x+1,p.y,p.z+y1);
    pt[3].set(p.x+1,p.y,p.z-y1);

    pt[4].set(p.x+y1,p.y,p.z-1);
    pt[5].set(p.x+y1,p.y,p.z+1);
    pt[6].set(p.x-y1,p.y,p.z-1);
    pt[7].set(p.x-y1,p.y,p.z+1);

    for(i=0;i<8; i++)
     {old[i][0] = shape::viewptr->windowx(pt[i]);
      old[i][1] = -shape::viewptr->windowy(pt[i]);}

   ychange =0;
   while (x1<y1) {
      circlezpoint();
      if (d < 0) {
          d+=(4*x1)+6;
          ychange = 0;
      }
      else {
            d+= (4*(x1-y1))+10;
            y1--;
            ychange =1;
      }
      x1++;
   }
       ychange =1;
       circlezpoint();
}
```

```
//******************************
// Prog : CIRCLE1.cxx
//
// Draws a circle using a 40 sided polygon
// as an approximation
//***************************



#include"stdinc.hxx"


#define sides 40
#define iter (sides/2+1)

void circledraw(point cen , int r)
{
 point p1;
 int i;
 int old[2][2];   // stored co-ordinates
 float p;

 int x,y;                    // general temp variables

 p1.set(r+cen.x,cen.y,cen.z);
 old[0][0]=old[1][0] =shape::viewptr->windowx(p1);
 old[0][1]=old[1][1] = -shape::viewptr->windowy(p1);

 for (i=0,p=0; i<iter; i++,p+=3.1415926*2/sides)
 {
   p1.set( x=((int)(cos(p)*r)+cen.x),(y=(int)(sin(p)*r))+cen.y,cen.z); // set up circle array
   _moveto(old[0][0],old[0][1]);
   _lineto(old[0][0]=shape::viewptr->windowx(p1),old[0][1]=-shape::viewptr->windowy(p1));
    p1.set( x,cen.y-y,cen.z); // set up circle array
   _moveto(old[1][0],old[1][1]);
   _lineto(old[1][0]=shape::viewptr->windowx(p1),old[1][1]=-shape::viewptr->windowy(p1));

  }
}
```

DEMO.CXX

```
//****************************
// Prog : DEMO.cxx
//
// This program provides a demonstration of some
// of the objects in COOGE. It must be linked with
// dem, dem1 & dem2 to provided an exe version
//
//***************************



#include "stdinc.hxx"
#include "seglist.hxx"
#include "shapes.hxx"
#include "image.hxx"
#include "text.hxx"
window w;

extern void showtext();
extern void textcube();
extern void showpoly();
extern void showcircles();
extern void savefile();
extern void showwindows();

void showsphere()
{
  sphere s;
  float p;
  segment seg;
   int i;
  s.set(point(320,240),200);
    seg.setcabinet();
    seg.append(&s);
    seg.sensitize();
    w.open();
    seg.refresh();
    getch();
    seg.setvport(0.75,0.5,0.98,0.75);
    seg.setframe(on);

    w.refresh();
    w.close();
    seg.refresh();
    seg.desensitize();

    getch();

}



 void loading()
{
   image img;
   scene* scp,*scp1= NULL;
   window w1;
   txt tmp,tmp1,tmp2;
   segment sg;
   tmp.setscale(4,8);
   tmp1.setscale(4,8);
   tmp2.setscale(4,8);
   tmp.settxt(ROMAN,point(200,650),"LOADING");
   tmp2.settxt(ROMAN,point(250,400),"IMAGES");
   tmp1.settxt(ROMAN,point(150,200),"FROM DISK");

   sg.setcabinet();
   sg.setvport(0.75,0.02,0.98,0.25);
   sg.setwindow(0,0,800,800);
   sg.setframe(on);
   sg.append(&tmp);
   sg.append(&tmp1);
   sg.append(&tmp2);
   sg.refresh();

  w1.setvport(0.25,0.25,0.75,0.75);
  w1.setframe(on);
  w1.setautosave(off);
  w1.open();
    img.open("test");
    scp = (scene*) img.loadobject("SCENE");
    while (scp !=  (scene*) NULL)
    {
     scp->refresh();
     if (scp1 != (scene*) NULL)
     {
       scp1->wipe();
       delete scp1;
     }
     scp1 = scp;
     scp = (scene*) img.loadobject("scene");

    }
    getch();
    w1.refresh();
    w1.close();
    w1.setvport(0.75,0.02,0.98,0.25);
    w1.open();
    w1.refresh();
    scp1->refresh();
    scp1->wipe();
    delete scp1;
    w1.close();
    getch();

}

#include "font.hxx"
font ft;

main()
{
  txt tmp;
  ft.load();
  rect rb;
  scr.init(_ERESNOCOLOR); //_VRES16COLOR);
  w.setvport(0.25,0.25,0.75,0.75);
  w.setframe(on);
  w.setautosave(off);
  showtext();
  textcube();
  showpoly();
  showcircles();
  showsphere();
  savefile();
  loading();
  showwindows();
    getch();
}
```

DEM.CXX

```
#include "stdinc.hxx"
#include "seglist.hxx"
#include "shapes.hxx"
#include "font.hxx"
#include "text.hxx"

extern window w;
extern font ft;

void showtext()
{
int i;
segment sg;
txt  tmp;
float r=0;
  tmp.setscale(6,4);
  sg.setcabinet();
  sg.setvport(0.0,0.75,0.25,1.0);
  sg.setframe(on);
  sg.setwindow(0,0,639,199);
  tmp.setscale(4,8);
  tmp.settxt(ROMAN,point(200,100),"TEXT");
  sg.append(&tmp);
  sg.refresh();
  sg.setframe(off);
  tmp.setscale(4,4);
  sg.setvport(0.0,0.0,1.0,1.0);
  tmp.settxt(ROMAN,point(100,100,0),"2D TEXT");
//  sg.sensitize();
  sg.setwindow(0,0,639,199);
  w.open();
  for (r=0;r<4; r+=0.1)
  {
  tmp.setscale(r,r);
  sg.refresh();
  }

  for (i=180;i>=0;i-=4)
  {
  sg.rotateaboutxto(100,0,i);
  sg.refresh();
  }

  for (i=60;i>=0;i-=2)
  {
  sg.rotateaboutto(250,100,i);
  sg.refresh();
  }
  ft.set3d();
  tmp.settxt(ROMAN,point(100,100,0),"3D TEXT");

  for (i=180;i>=0;i-=2)
  {
  sg.rotateaboutyto(150,0,i);
  sg.refresh();
  }

  getch();

  sg.setvport(0.0,0.75,0.25,1.0);
  sg.setframe(on);

  w.refresh();
  w.close();
  sg.refresh();
  sg.desensitize();
  ft.set3d(off);
  getch();

};

void textcube()
```

```
{
   int xt;
   box* r1;
   txt t;
   segment seg;

   r1 = new box;

    r1->set(point(200,300),point(600,500),100);
    t.settxt(ROMAN,point(250,350),"FRONT");
    t.setscale(4,8);
     seg.setcabinet();
     seg.append(r1);
     seg.append(&t);
     seg.setwindow(0,0,800,800);
//     seg.sensitize();
     w.open();
     seg.refresh();

     for (xt=4; xt<360; xt+=4)
     {
       seg.rotateaboutxto(400,50,xt);
       seg.refresh();
     }
     seg.rotateaboutxto(400,50,0);
     seg.refresh();
     for (xt=4; xt<360; xt+=4)
     {
       seg.rotateaboutyto(400,50,xt);
       seg.refresh();
     }
     seg.rotateaboutyto(400,50,0);
     seg.refresh();
     for (xt=4; xt<360; xt+=4)
     {
       seg.rotateaboutto(400,400,xt);
       seg.refresh();
     }
     seg.rotateaboutto(400,50,0);
     seg.refresh();
     getch();
    seg.setvport(0.25,0.75,0.5,1.0);
    seg.setframe(on);

    w.refresh();
    w.close();
    seg.refresh();
    seg.desensitize();

    getch();

}


void showpoly()
{
  polygon* py;
  float p;
  segment seg;
   int i;

  py = new polygon(100);
 for (i=0,p=0; i<100; i++,p+=3.1415926*2/10)
 {
   py->append(point((int)(cos(p)*(100-i*4))+500,(int)(sin(p)*(100-i*4))+500,i));                         // set up
  }
    seg.setcabinet();
    seg.append(py);
//    seg.sensitize();
    w.open();
    for(i=1; i<450; i+=4)
    {
      seg.rotateaboutyto(500,50,i);
      seg.refresh();
    }
    seg.setvport(0.5,0.75,0.75,1.0);
    seg.setframe(on);

    w.refresh();
    w.close();
    seg.refresh();
    seg.desensitize();

    getch();

}

void showcircles()
{
  float p;
  circle* c;
  int i;
  polygon py;
  point pt;
  segment seg;
 for (i=0,p=0; i<5; i++,p+=3.1415926*2/5)
  {
   c = new circle;
   c->set(pt = point(320,(int)(sin(p)*(100-i*2))+240,(int)(cos(p)*(100-i*2))),30);                         // set
   py.append(pt);
   seg.append(c);
  }
  seg.append(&py);
  py.setclosed(on);

    seg.setcabinet();
    seg.setwindow(0,0,639,479);
//    seg.sensitize();
    w.open();
      seg.refresh();
  for (i=0; i<360; i+=4)
  {
    seg.rotateaboutyto(320,0,i);
    seg.refresh();
  }
  for (i=0; i<360; i+=4)
  {
    seg.rotateaboutxto(240,0,i);
    seg.refresh();
  }
  for (i=0; i<360; i+=4)
  {
    seg.rotateaboutto(320,240,i);
    seg.refresh();
  }
    seg.setvport(0.75,0.75,0.98,1.0);
    seg.setframe(on);

    w.refresh();
    w.close();
    seg.refresh();
    seg.desensitize();

    getch();


}
```

DEM1.CXX

```
#include "stdinc.hxx"
#include "seglist.hxx"
#include "shapes.hxx"
#include "image.hxx"
#include "text.hxx"
extern window w;


void savefile()
{
   int xt;
   image img;
   box* r1;
   window w1;
   scene sc;
   segment seg,seg2,seg3,seg4,sg;
   txt tmp,tmp1,tmp2;
   tmp.setscale(4,8);
   tmp1.setscale(4,8);
   tmp2.setscale(4,8);
   tmp.settxt(ROMAN,point(150,650),"CREATING");
   tmp2.settxt(ROMAN,point(250,400),"IMAGES");
   tmp1.settxt(ROMAN,point(200,200),"ON DISK");

   sg.setcabinet();
   sg.setvport(0.75,0.25,0.98,0.5);
   sg.setwindow(0,0,800,800);
   sg.setframe(on);
   sg.append(&tmp);
   sg.append(&tmp1);
   sg.append(&tmp2);
   sg.refresh();


    r1 = new box;
     r1->set(point(800,800),point(1200,1200),400);
     r1->setcolor(1);
     seg.append(r1);
     seg.setcabinet();
     seg.setwindow(0,0,2000,2000);
     seg.setvport(0.0,0.0,0.5,0.5);
     seg.sensitize();
     sc.append(&seg);
//     r1 = new box;
//     r1->set(point(800,800),point(1200,1200),400);
//     r1->setcolor(2);
     seg2.append(r1);
     seg2.setcabinet();
     seg2.setwindow(0,0,2000,2000);
     seg2.setvport(0.0,0.5,0.5,1.0);
     seg2.sensitize();

     sc.append(&seg2);
//     r1 = new box;
//     r1->set(point(800,800),point(1200,1200),400);
//     r1->setcolor(3);
     seg3.append(r1);
     seg3.setcabinet();
     seg3.setwindow(0,0,2000,2000);
     seg3.setvport(0.5,0.5,1.0,1.0);
     seg3.sensitize();

     sc.append(&seg3);
//     r1 = new box;
//     r1->set(point(800,800),point(1200,1200),400);
//     r1->setcolor(4);
     seg4.append(r1);
     seg4.setcabinet();
     seg4.setwindow(0,0,2000,2000);
     seg4.setvport(0.5,0.0,1.0,0.5);
     seg4.sensitize();
     sc.append(&seg4);

     r1->set(point(800,800),point(1200,1200),400);

     w1.setvport(0.25,0.25,0.75,0.75);
     w1.setframe(on);
    w1.setautosave(off);

     w1.open();
     sc.refresh();

     img.open("test");
     img.rewrite();
     for (xt = 0; xt<46; ++xt) {
        sc.rotateabout(900,1000,4);
        sc.putobj(&img);
     }
      for (xt = 0; xt<46; ++xt) {
        sc.rotateaboutx(900,200,4);
        sc.putobj(&img);
     }


      for (xt = 0; xt<46; ++xt) {
        sc.rotateabouty(1000,200,4);
        sc.putobj(&img);
     }
     img.close();
     w1.refresh();
    w1.close();
    w1.setvport(0.75,0.25,0.98,0.5);
    w1.open();
    w1.refresh();
    sc.refresh();
    w1.close();
    getch();

}
```

DEM2.CXX

```
#include "stdinc.hxx"
#include "seglist.hxx"
#include "shapes.hxx"
#include "image.hxx"
#include "text.hxx"
extern window w;

void openwindow(float x,float y,float x1,float y1)
{
 window w2;
 txt tmp;
 tmp.setscale(4,8);
 tmp.settxt(ROMAN,point(200,100),"WINDOW");

 w2.setcabinet();
 w2.setvport(x,y,x1,y1);
// w2.setwindow(0,0,800,800);
 w2.setframe(on);
 w2.append(&tmp);
 w2.setautosave(off);
 w2.open();
 w2.refresh();
 getch();
  tmp.draw();
  getch();
 w2.close();
 getch();
}
void showwindows()
{
   int xt;
   _moveto(0,0);
   _lineto(400,-150);
   getch();
   openwindow(0.10,0.10,0.400,0.5);
}
```

FONT.HXX

```
// ******************
// FONT HEADER FILE
//
// DEFINES THE FONT OBJECT AND A NUMBER OF SERVICE
// OBJECTS THAT ARE USED IN THE DRAWING OF TEXT ON
// THE SCREEN
//
//***************
enum fonttype { mve,drw,chardef,chardraw }; // Font rec types

class font;
class fontrec;

class chartab {
friend class fontrec;
      fontrec* entry[127];             // Chartable of pointers to routine to
                                       // draw characters
public:
      chartab() { init();}
      ~chartab() {clear();}

      void init();
      void clear();
      void load(FILE*);
      void draw( char, font*);
};


class fontrec {
     // char width = 125 pixels with proportional spacing
     // char height = 76 pixels with       "          "

      static float sx,sy;                        // Scale factor for font
      static int cx,cy,cz;                       // Starting position
      static int tx,ty;                          // Current Offset from start pos
      static int old[2][2];                      // old x & y positions
      char typ;                                  // Type of record
      union { char x; char fc; char fd; };       // Record data
      char y;
      fontrec* next;                             // pointer to next record

public :
     fontrec() {next = NULL;}
     ~fontrec() { if (next != (fontrec*) NULL)
                      delete next;}
     int read(FILE*);
     void write(FILE*);
     void setrec(fonttype,char,char);
     void draw(font*);
     void draw3d(font*);
//     void operator=(fontrec&);
     void setscale(float,float);
     void setstartpoint(point);
     void set3dstartpoint(point);
     void notrelative();
     void loadtable(chartab*, FILE*);
};



class font {
friend class fontrec;
friend void wrchar(int,point,char*,float,float);

    chartab* table;
    float sx, sy;                          // Current Scaling factors
    int fontid;                            // Number of the font
    int d3;                                // whether font is 3d or not
    static blist fontlist;                 // list of fonts
    void reset();                          // Clear font record
public :

   font() { table = new chartab; d3 = off;
            fontlist.append((base*) this); }
    ~font() {  delete table;                 // Need to dealocate table store
               fontlist.remove((base*) this);}
    void load(char* = "FONT");
    void draw(char,int=0);
    void draw(point,char*);
    void draw(point,char);

    void setscale(float =1, float =1);       // Set scaling factors
    void set3d(int = on);                    // set 3d on/off
};
```

```
#include "stdinc.hxx"
#include "font.hxx"
#define depth 10              // default z value for 3d font

/*********************************************************************/
/*                                                                   */
/*             CLASS FONTREC PROCEDURES AND FUNCTIONS                */
/*                                                                   */
/*********************************************************************/

void fontrec::setrec(fonttype t, char x1, char y1)
{
 typ = (char) t; x = x1; y = y1;
};

int fontrec::read(FILE* f)
{
 if (!fread((char*)&typ,sizeof(char),1,f))
   {
     return false;
    }
    else
     {
       fread((char*)&x,sizeof(char),1,f);
       fread((char*)&y,sizeof(char),1,f);
       return true;
           }
};

void fontrec::write(FILE* f)   /// REM  PUT IN SOMETHING FOR BAD FILE
{

  fwrite((char*)&typ,sizeof(char),1,f);
  fwrite((char*)&x,sizeof(char),1,f);
  fwrite((char*)&y,sizeof(char),1,f);

};

void fontrec::draw3d(font* ft)
{
   fontrec* tmp;
   int dx,dy;
   point p;
   tmp = this;
   while (tmp != (fontrec*) NULL)
   {
   switch ((fonttype)tmp->typ) {
    case mve     : tx+=int(tmp->x);ty+=int(tmp->y);
```

```
                   p.set((cx+int(tx*sx)),(cy+int(ty*sy)),depth);
                   old[1][0]=shape::viewptr->windowx(p);old[1][1] =-shape::viewptr->windowy(p);
                   p.inc(0,0,-depth);
                   old[0][0]=shape::viewptr->windowx(p);old[0][1]=-shape::viewptr->windowy(p);
                   break;
    case drw     : tx+=int(tmp->x);ty+=int(tmp->y);
                   p.set((cx+int(tx*sx)),(cy+int(ty*sy)),0);
                   _moveto(dx=old[0][0],dy=old[0][1]);
                   _lineto(old[0][0]=shape::viewptr->windowx(p),old[0][1]=-shape::viewptr->windowy(p));
                   _moveto(dx,dy);
                   _lineto(old[1][0],old[1][1]);
                    p.inc(0,0,depth);
                   _lineto(old[1][0]=shape::viewptr->windowx(p),old[1][1] =-shape::viewptr->windowy(p));

                   break;
    case chardraw: ft->draw(tmp->fd,relative);break;
    case chardef : break;
   }
    tmp = tmp->next;
  };
}

void fontrec::draw(font* ft)
{
   fontrec* tmp;
   tmp = this;
   if (ft->d3)
     draw3d(ft);
   else

   while (tmp != (fontrec*) NULL)
   {
     switch ((fonttype)tmp->typ) {
      case mve     : tx+=int(tmp->x);ty+=int(tmp->y);
                     shape::viewptr->moveto(point((cx+int(tx*sx)),(cy+int(ty*sy))));
                     break;
      case drw     : tx+=int(tmp->x);ty+=int(tmp->y);
                     shape::viewptr->lineto(point((cx+int(tx*sx)),(cy+int(ty*sy))));
                     break;
      case chardraw: ft->draw(tmp->fd,relative);break;
      case chardef : break;
     }
    tmp = tmp->next;
  };
}

// void fontrec::operator=(fontrec& ft)
// {
//  typ = ft.typ; x = ft.x; y = ft.y;
// }

void fontrec::loadtable(chartab* tb, FILE* fp)
{
  fontrec* last = (fontrec*) NULL;
  fontrec* tmp;

  tmp = new fontrec;
  tmp->next = NULL;
   while (tmp->read(fp))
  {
     switch (tmp->typ) {
     case chardef:
          tb->entry[tmp->fc] = tmp;
          if (last != (fontrec*) NULL)
             last->next = NULL;
          last = NULL;
          break;
     case mve:
     case drw:
     case chardraw:
          if (last != (fontrec*)NULL)
                last->next = tmp;
          last = tmp;
          tmp = new fontrec;
          tmp->next = NULL;
          break;
     }

   }
   delete tmp;
}

inline void fontrec::setscale(float scx, float scy)
{
 sx = scx; sy = scy;
}

inline void fontrec::setstartpoint(point p)
{
 cx = p.x; cy = p.y; cz = p.z; tx=ty=0;

}

inline void fontrec::set3dstartpoint(point p)
{
 cx = p.x; cy = p.y; cz = p.z; tx=ty=0;

 old[0][0] = shape::viewptr->windowx(p);
 old[0][1] = -shape::viewptr->windowy(p);
 p.inc(0,0,depth);
 old[1][0] = shape::viewptr->windowx(p);
 old[1][1] = -shape::viewptr->windowy(p);
}

inline void fontrec::notrelative()
{
  tx = ty =0;
}

/**********************************************************************/
/*                                                                    */
/*              CLASS CHARTAB PROCEDURES AND FUNCTIONS                */
/*                                                                    */
/**********************************************************************/

void chartab::init()
{
 for (int i=0; i<127; i++)
     entry[i] = (fontrec*) NULL;
}

void chartab::clear()
{
 for (int i=0; i<127; i++)
    if (entry[i] !=(fontrec*)NULL)
       delete entry[i];
}

inline void chartab::load(FILE* fp)
{
  fontrec fr;
  fr.loadtable(this,fp);
}

inline void chartab::draw(char ch, font* fp)
{
  if (entry[ch] != (fontrec*)NULL)
     entry[ch]->draw(fp);
}
/**********************************************************************/
/*                                                                    */
/*              CLASS FONT PROCEDURES AND FUNCTIONS                   */
/*                                                                    */
/**********************************************************************/
```

FONT.CXX

```
inline void font::setscale(float scx, float scy)
{
  fontrec fr;
  sx = 640.0*scx/(6080.0);     // 6080 = (80 chars across * 76 char width)
  sy = 200.0*scy/(3125.0);     // 3125 = (25 rows * 125 char height)
  fr.setscale(sx,sy);
}

void font::set3d(int d=on)
{
 d3 = d;
}
void font::reset()
{
  table->clear();
  setscale(1,1);
}


void font::load(char* fname)
{
 FILE* stream;
 stream = fopen(fname,"r+b");
 if (stream!=0)
 {
  reset();
  fread(&fontid,sizeof(int),1,stream);
  table->load(stream);
 }
 fclose(stream);
};

void font::draw(char a,int relat)
{
 fontrec fr;
 if (!relat)
    fr.notrelative();
 table->draw(a,this);
};


void font::draw(point p,char* txt)
{
  int i=0;
  fontrec fr;
 if (d3)
 {
  fr.set3dstartpoint(p);
 }
 else
 {
  fr.setstartpoint(p);
  shape::viewptr->moveto(p);
 }
   while ( txt[i])
   { table->draw(txt[i],this);
     i++;
   }
};

void font::draw(point p,char c)
{
 fontrec fr;
 if (d3)
 {
  fr.set3dstartpoint(p);
 }
 else
 {
  fr.setstartpoint(p);
  shape::viewptr->moveto(p);
 }
```

```
  font::draw(c,relative);
};



void wrchar(int id, point p, char* txt, float ssx, float ssy)
{
  font* fp;
  int found =false;

  font::fontlist.reset();
  while ((fp=(font*) font::fontlist.next()) && (!found))
  {
   if (fp->fontid==id )
      {
       fp->setscale(ssx,ssy);
       fp->draw(p,txt);
       found = true;
      }
  }
}
```

IMAGE.HXX

```
// **********
// IMAGE HEADER FILE
//
// DEFINES THE IMAGE OBJECT THAT STORE AND LOADS
// OTHER OBJECTS TO/FROM DISK
//
//*****************

#define MAXOBJS 30                  // max number of unique objects to store
#define OBJNAME 15                  // max length of the object's name
#define IDINCREASE 10               // Amount to increas table when full

class imageids {
    char** tab;                         // table of names;
    int top;                           // highest entry in the table;
    int sz;                            // current size of the table;

public:
    void init();                    // set up the table;
    void destroy();                 // destroy the table;

      imageids() {init();}            // Set up the table
      ~imageids() { destroy();}       // destroy the table
      unsigned char add(char*);       // add name if doesn't exist
      char* find(int);                // find name -> 0 = doesn't exist
      void save(char*);               // save the table ids to file;
      void load(char*);               // load the table ids from file.
};

class image {
    FILE* stream;             // pointer to file
    char* fname;              // file name;
    char lbl[OBJNAME];        // Name of the current object;
    imageids* id;             // Pointer to Id table

    void readlabel();         // read the label of the next object
public :
          image() {id = new imageids;
                    id->init();}
          image(char* n) { open(n);}
          ~image()      { close(); delete id;}

     void label(char[OBJNAME]);  // label record with details;
     void write(void *,long );   // put to file address, size;
     void writenull();           // writes a NULL class to the file
     int  nullobject();          // returns true if object is NULL object
     void readnull();            // clears the NULL object from input;
     void read(void *, long );   // read in from file - address,size
     void open(char*);           // open image file;
     void rewrite();             // rewrite the file;
     void close();               // close the file;
     void objectread();          // called if object is successfully loaded
     int match(char*);           // check if object matches label
     base* loadobject(char*);    // load next object on disk
};
```

```
//*****************************
// Prog : IMAGE.cxx
//
// Creates a image object which looks after the
// storing and retrieving of objects to/from disk.
//
//****************************
```

```
#include "stdinc.hxx"
#include "image.hxx"

#define FileNameLength 100              // max length of a file name


/***************************************************************/
/*                                                             */
/*                    CLASS IMAGEIDS                           */
/*                                                             */
/* This class contains a lookup table for each class of object */
/*  written to a file and returns an index to that class name  */
/*  in the table.                                              */
/*                                                             */
/***************************************************************/
void imageids::init()
{
 tab = new char* [MAXOBJS];
 for (int i=0; i<MAXOBJS; i++)
      tab[i] = (char*) NULL;              // Set the lookup table to blank
 top = 0;                                 // Set the top of table to 1st
 sz = MAXOBJS;                            // Set the size of the table
}

void imageids::destroy()
{
   for(int i =0 ; i<top; i++)             // Clear the table
       delete tab[i];
   top =0;                                // reset the top
}

unsigned char imageids::add(char* name)
{
 int i=0;
 char tmp[OBJNAME];
 char** tmptab;

 strcpy(tmp,name);
 tmp[OBJNAME-1] = '\0';                // limit string to OBJNAME characters
 while ((i <top) && (!(stricmp(tab[i],tmp)==0)))  i++;
 if (i == top)
  {
    if (top >= sz ) {
       tmptab = new char* [sz+IDINCREASE];     // set up new table
       memcpy(tmptab,tab,sizeof(char*)*sz);    // Copy old table to new table
       delete tab;                             // remove old table
       tab = tmptab;
       sz += IDINCREASE;
       for (int x=top; x<top+IDINCREASE; x++)
           tab[x] = (char*) NULL;             // initialize extra bit of table
    }
    tab[top]= strdup(tmp);                    // set up copy of key
    top++;
  }
  return (unsigned char) i;
}

char* imageids::find(int i)
{
  return tab[i];
}

void imageids::save(char* fname)
{
  FILE* stream;
  char temp[OBJNAME];
  char* filename;

  filename = new char[FileNameLength];
  strcpy(filename,fname);
  stream = fopen(strcat(filename,".id"),"w+b");
  delete filename;
  if (fwrite(&top,sizeof(int),1,stream))
  {
    for (int i =0; i<top; i++)
    {
      strncpy(temp,tab[i],OBJNAME);
      temp[OBJNAME-1] = '\0';
      fwrite(temp,sizeof(char),OBJNAME,stream);
    }
  }
  fclose(stream);
}
void imageids::load(char* fname)
{
  FILE* stream;
  char* filename;
  destroy();
  filename = new char[FileNameLength];

  strcpy(filename,fname);
  stream = fopen(strcat(filename,".id"),"rb");
  delete filename;
  if (stream != (FILE*) NULL)
  {
    if (fread(&top,sizeof(int),1,stream))
    {
      for (int i =0; i<top; i++)
      {
       tab[i] = new char[OBJNAME];
       fread(tab[i],sizeof(char),OBJNAME,stream);
      }
    }
    else
    {
      top =0;
    }
  }
  fclose(stream);
}


/****************************************************************/
/*                                                              */
/*                         CLASS IMAGE                          */
/*                                                              */
/****************************************************************/

int image::match(char*p)
{
  return (stricmp(lbl,p)==0) ? (true) : (false);
};


void image::readlabel()
{
 unsigned char i;
  if (!fread(&i,sizeof(char),1,stream))
  {
       lbl[0] = '\0';
  }
  else
  {
   strncpy(lbl,id->find(i),OBJNAME);
  }
};

void image::objectread()
{
  readlabel();
};

void image::label(char lb[OBJNAME])
```

```
{
  unsigned char i;
  i=id->add(lb);
  fwrite(&i,sizeof(char),1,stream);
};

void image::write(void * addr,long size)
{
  fwrite(addr,size,1,stream);
}

void image::writenull()
{
  label("NULL");
}

int image::nullobject()
{
 return match("NULL");
}

void image::readnull()
{
 readlabel();
}

void image::read(void* addr, long size)
{
  fread(addr,size,1,stream);
};

void image::open(char* name)
{
  char* p;
  fname = strdup(name);
  p = new char[FileNameLength];

  strcpy(p,name);
  stream = fopen(strcat(p,".img"),"rb");
  delete p;
  id->load(fname);
  readlabel();
};

void image::rewrite()
{
  char* p;

  fclose(stream);
  p = new char[FileNameLength];
  strcpy(p,fname);
  stream = fopen(strcat(p,".img"),"w+b");
  delete p;
  lbl[0]='\0';
  id->destroy();
};

void image::close()
{
  id->save(fname);
  fclose(stream);
};


base* image::loadobject(char* name)
{
 base* t = (base*) NULL;
 base* obj = (base*) NULL;
 if ((match(name)) || (nullobject()) || (stricmp(name,"")==0))
 {
    base::objectlist.reset();
    while ((t = base::objectlist.next()) && (obj == (base*) NULL))
    {
      obj = t->getobj(this,obj);
    }
 }

// if (obj == (base*) NULL)
// {
//   printf("Failed to load from (%s) - label -> %s     obj -> %s \n",fname,lbl,name);
//   getch();
//   exit(1);
// }
 return obj;
};
```

POINT.HXX

```
// ****************
// POINT.HXX
//
// DEFINES THE POINT OBJECT TO BE USED BY THE SYSTEM
//
// ********************************

struct point {                              // Definition of a point
       int x,y,z;
       point() {}
       point( int a, int b, int c =0)
                 { x=a;y=b;z=c;}
       void set( int a=0, int b=0, int c=0)
                  { x=a; y = b; z = c;}
       void inc(int a=0,int b=0,int c=0)       // increment the point
                  { x+=a;y+=b;z+=c;}

};
```

```
// ***********
// SEGLIST.HXX
//
// CONTAINS THE DEFINITIONS OF THE SCREEN, SEGMENT, SCENE
// AND WINDOW OBJECTS
//
// ************


/**************************************************************************/
/*                              CLASS SCREEN                              */
/**************************************************************************/

class screen : public viewset {
friend class segment;
friend class viewset;

       slist scr_list;           // pointer to list of associated segments
       int devmode;              // device mode of the screen
       long color;               // Screen background color
public :
       screen() { shape::viewptr = this;     // Screen Creator
                  devmode = _DEFAULTMODE;    // Set video mode to default mode
                  color = _BLACK;            // Set background color to black
                 }

//       ~screen() { _setvideomode(_DEFAULTMODE);}    // Reset the screen
      base* getobj(image*, base* );                // load an screen from disk;
      void putobj(image*);                         // save a screen to disk;


       void init(int dev= _HRESBW);                    // init the screen
       void review() { viewset::cur_screen = this;  // reset the viewport etc.
                       shape::viewptr = this;
                       setvport();      }
       void refresh();                         // refresh the screen
       void save();                            // save the current screen
       void restore();                         // restore the saved screen
       void setbkcolor(long c=_BLACK);         // Set the background color

};


/**************************************************************************/
/*                              CLASS SCENE                               */
/**************************************************************************/

class scene : public base {
friend class screen;
friend class segment;

       slist scn_list;                   // pointer to list of associated seg

public :

    base* getobj(image*, base* );                // load an scene from disk;
    void putobj(image*);                         // save a scene to disk;

    void insert(segment* p)
              {scn_list.insert((base*) p);}       // add segment* at head of list
    void append(segment* p)
             {scn_list.append((base*) p);}       // add segment* at tail of list
    segment*  get()
             { return (segment*) scn_list.get(); } // return and remove segment* at head of list
    void clear()  {scn_list.clear();}    // remove all links
    void refresh();                      // redraw the segment
    void wipe();                         // delete all segments
    void scale(float,float,float);         // Main scaling

    void scalex(float fx)                  // Scale up/down x Values
            { scale(fx,1.0,1.0); }
    void scaley(float fy)                  // Scale up/down y values
```

```
        { scale(1.0,fy,1.0); }
    void scalez(float fz)           // Scale up/down z values
        { scale(1.0,1.0,fz); }

     void rotateabout(int,int,int,int=relative);    // Rotate about point in Z Dir
     void rotateaboutto( int a,int b,int c)         // Rotate absolute about
            { rotateabout(a,b,c,!relative); }       // point in Z dir
     void rotate(int a)                             // Rotate about origin
            { rotateabout(0,0,a);}                  // in Z Dir
     void rotateto(int a)                           // Rotate absolute about
            { rotateabout(0,0,a,!relative); }       // Origin in Z dir
     void rotateaboutx(int,int,int,int = relative); // Rotate about point in X Dir
     void rotateaboutxto(int a,int b, int c)        // Rotate absolute about
            { rotateaboutx(a,b,c,!relative); }      // point in X dir
     void rotatex(int a)                            // Rotate about origin
            { rotateaboutx(0,0,a);}                 // in X dir
     void rotatexto(int a)                          // Rotate Absolute about
            { rotateaboutx(0,0,a,!relative); }      // origin in X direction
     void rotateabouty(int,int,int,int = relative); // Rotate about point in Y Dir
     void rotateaboutyto(int a,int b,int c)         // Rotate absolute about
            { rotateabouty(a,b,c,!relative); }      // point in Y dir
     void rotatey(int a)                            // Rotate about origin
            { rotateabouty(0,0,a);}                 // in Y dir
     void rotateyto(int a)                          // Rotate absolute about
            { rotateabouty(0,0,a,!relative); }      // origin in Y dir


    void movevport(float,float);          // move scene relative
    void movewindow( int,int);

  /////// maybe put in set views etc eg perspective , oblique-> <-
};



/*****************************************************************************/
/*                              CLASS SEGMENT                                */
/*****************************************************************************/

class segment : public viewset{
friend class screen;
friend class window;
    static blist segelist;    // list of all the segments created
    int visible;             // boolean - whether segment is visible or not
    int autoclear;           // boolean - whether to clear viewport on refresh
    int frame;               // boolean - Whether segment is to be framed
    int copy;                // boolean - whether a copy or not
    int locked;              // boolean - whether segment is locked

    slist shapelist;        // list of shapes in the segment
    char* id;               // Id name of the segment

public:
      segment()       {  visible=autoclear=on;         // segment creator
                          frame=off; id=(char*)NULL;
                     }

      segment(screen * p)
        { p->scr_list.append(this);                   // segment creator for a screen
                           segment(); }
      segment(scene * p) { p->scn_list.append(this); // segment creator for a screen
                           segment(); }


      ~segment()      { shapelist.clear(); }            // segment destructor;

      base* getobj(image*, base* );                  // load a segment from disk;
      void putobj(image*);                           // save a segment to disk;

    void insert(shape* p) {shapelist.insert(p);}      // add shape* at head of list
    void append(shape* p) {shapelist.append(p);}      // add shape* at tail of list
    shape*  get() { return (shape*) shapelist.get(); }        // return and remove shape* at head of list
    void clear()  {shapelist.clear();}              // remove all links
    void refresh();                                // redraw the segment
    void draw();                                   // draw the segment
    void setvisibility(int a) { visible = a; }    // set seg visibility on/off
    void setautoclear(int a) { autoclear = a;}     // set auto clear on/off
    void setframe(int a) { frame = a;}             // Set frame on/off

    void save() ;                                  // save the current segment
    void restore();                                // restore the saved segment
    void setlock(int);                             // set segment lock on/off
    void setid(char*);                             // set the id name of the segment

 };


/*************************************************************************/
/*                              CLASS WINDOW                             */
/*************************************************************************/

class window : public segment {
      int autosave;                            // whether to save underneath;
      int id;                                  // Id number of the window
      viewset* sav;                            // Save old viewset pointer
      bitmap imptr;                            // Saved screen
      int tscrxmax,tscrymax,tscrxmin,tscrymin;  // saved screen co-ords

public :
    window()      {   id =0;
                      setvisibility(on);       // window creator
                      setautoclear(on);
                      autosave= on;
                   }

    ~window()    { if (id!=0)                  // Window Destructor
                        close();
                 }
    void append(segment* p) {shapelist.append(p);} // insert segment
    void append(shape*p) {shapelist.append(p);}
    void refresh();
    void open();                               // Pop-Up window
    void close();                              // Close the window;
    void setautosave(int);                     // set autosave on/off
    void setvport(float,float,float,float);    // Make sure none of the vport
    void movevport(float,float);               // procedures are used when the
    void movevportto(float,float);             // window is open
};


/*********************************************************************/
/*                     GENERAL VARIABLE DEFINITIONS                  */
/*********************************************************************/

extern screen scr;








#include "stdinc.hxx"
#include "seglist.hxx"
#include "image.hxx"

/********************************************************************/
/*            Degree to radian conversion function                  */
/********************************************************************/

float radian(float a)
```

```
{
 return (a*0.0174532925199432957692);
}

/**********************************************************************/
/*                 BITMAP CLASS FUNCTIONS AND PROCEDURES             */
/**********************************************************************/

ACTIVATE(bitmap);                 // add bitmap to list of storeable objects

int bitmap::save(int x1,int y1, int x2, int y2)
{
  if (sav == (char far*) NULL) {
         xmin = x1; ymin=y1;     // Negate the y co-ords for MSC V5 Graphs
         xmax = x2; ymax=y2;     // Package (origin top left !)

         sav = (char far*) new unsigned int[size=_imagesize(xmin,-ymin,xmax,-ymax)];
         if (sav != (char far*) NULL) {
             _getimage(xmin,-ymin,xmax,-ymax,sav);
         }
         else
         {
             size =0;
          }

   }
   return (sav != (char far *) NULL);
}

void bitmap::restore()
{
 if (sav != (char far*) NULL){
        _putimage(xmin,-ymax,sav,_GPSET);  // Dodgey line !!!
        size =0;
        delete sav;
        sav = (char far*) NULL;
  }

}

void bitmap::restore( int x1,int y1,int x2,int y2)
{
     xmin = x1; ymin= y1; xmax = x2; ymax = y2;
     restore();
}

void bitmap::putobj(image* ip)
{
  ip->label("BITMAP");
  ip->write(&xmin,sizeof(int)*4);
  ip->write(&size,sizeof(long));
  if ( size !=0)
      ip->write((void*) sav,sizeof(int)*size);
};

base* bitmap::getobj(image* ip, base* obj)
{
  if (ip->match("BITMAP"))
    {
     if (obj == (base*) NULL)
          obj = (base*) new bitmap;
     ip->read(&((bitmap*) obj)->xmin,sizeof(int)*4);
     ip->read(&((bitmap*) obj)->size,sizeof(long));
     if ( ((bitmap*) obj)->size !=0)
        {
          ((bitmap*) obj)->sav = (char far*) new unsigned int[((bitmap*) obj)->size];
          ip->read( (void*)((bitmap*) obj)->sav,sizeof(int)*((bitmap*) obj)->size);
        }
     ip->objectread();
     }
  return obj;
};
```

```
/************************************************************************/
/*                VIEWSET CLASS FUNCTIONS AND PROCEDURES              */
/************************************************************************/

ACTIVATE(viewset);

void viewset::putobj(image* ip)
{
  ip->label("VIEWSET");
  ip->write(&vxmin,sizeof(float)*4);
  ip->write(&wxmin,sizeof(int)*4);
  ip->write(&mat[0][0],sizeof(float)*12);
  ip->write(&vmat[0][0],sizeof(float)*12);
  ip->write(&rmat[0][0],sizeof(float)*12);
  ip->write(&xfac,sizeof(float)*3);
  ip->write(&tx,sizeof(float)*3);
  ip->write(&sstx,sizeof(float)*3);
  ip->write(&d3,sizeof(int));
  ip->write(&pz,sizeof(int));
  ip->write(&sensitized, sizeof(char));
};

base* viewset::getobj(image* ip, base* obj)
{
  if (ip->match("VIEWSET"))
    {
     if (obj == (base*) NULL)
          obj = (base*) new viewset;
     ip->read(&((viewset*)obj)->vxmin,sizeof(float)*4);
     ip->read(&((viewset*)obj)->wxmin,sizeof(int)*4);
     ip->read(&((viewset*)obj)->mat[0][0],sizeof(float)*12);
     ip->read(&((viewset*)obj)->vmat[0][0],sizeof(float)*12);
     ip->read(&((viewset*)obj)->rmat[0][0],sizeof(float)*12);
     ip->read(&((viewset*)obj)->xfac,sizeof(float)*3);
     ip->read(&((viewset*)obj)->tx,sizeof(float)*3);
     ip->read(&((viewset*)obj)->sstx,sizeof(float)*3);
     ip->read(&((viewset*)obj)->d3,sizeof(int));
     ip->read(&((viewset*)obj)->pz,sizeof(int));
     ip->read(&((viewset*)obj)->sensitized,sizeof(char));

     ((viewset*)obj)->initfactors();              // reset factors in case
                                                 // screen mode changed
     if (((viewset*)obj)->sensitized)
        ((viewset*)obj)->sensitize();
     ip->objectread();
     }
  return obj;
};


/**************************************************************************/
/*                              SETMAT                                    */
/*                                                                        */
/* This function sets the contents of mat1 to the contents of mat2.       */
/*                                                                        */
/**************************************************************************/

inline void viewset::setmat(float mat1[4][3], float mat2[4][3])
{
   memcpy((void *) mat1,(void *) mat2, sizeof(float)*12);  // size of float * 12 members
                                                          // in array
}


/**************************************************************************/
/*                              SETIMAT                                   */
/*                                                                        */
/* This function sets the contents of mat1 to the Identity Matrix         */
/*                                                                        */
/**************************************************************************/
```

```
void viewset::setimat(float mat1[4][3])
{
  static float imat[4][3] = {
                { 1 , 0 , 0 },
                { 0 , 1 , 0 },
                { 0 , 0 , 1 },
                };                             // last row init to 0 by default
  memcpy((void *) mat1,(void *) imat, sizeof(float)*12);  // Size of float * 12

}


/***************************************************************************/
/*                              MATMULTI                                   */
/*                                                                         */
/* This function multiplies matrix 1 by matrix 2 leaving the result in     */
/*  matrix 1.                                                              */
/***************************************************************************/



void viewset::matmulti(float mat1[4][3], float mat2[4][3])
{
  float mat3[4][3];
  int x;
  int y;


  for ( x=0; x<3; x++)
     for( y=0; y<3; y++)
          mat3[x][y] = mat1[x][0]*mat2[0][y] + mat1[x][1]*mat2[1][y] +
                       mat1[x][2]*mat2[2][y];

  for( y=0; y<3; y++)
          mat3[3][y] = mat1[3][0]*mat2[0][y] + mat1[3][1]*mat2[1][y] +
                       mat1[3][2]*mat2[2][y] + mat2[3][y];

   setmat(mat1,mat3);
}

/***************************************************************************/
/*                              PREMULTI                                   */
/*                                                                         */
/* This function premultiplies matrix 1 by matrix 2 leaving the result     */
/*  in matrix 1.                                                           */
/***************************************************************************/

void viewset::premulti(float mat1[4][3], float mat2[4][3])
{
  float mat3[4][3];
  int x;
  int y;


  for ( x=0; x<3; x++)
     for( y=0; y<3; y++)
          mat3[x][y] = mat2[x][0]*mat1[0][y] + mat2[x][1]*mat1[1][y] +
                       mat2[x][2]*mat1[2][y];

  for( y=0; y<3; y++)
          mat3[3][y] = mat2[3][0]*mat1[0][y] + mat2[3][1]*mat1[1][y] +
                       mat2[3][2]*mat1[2][y] + mat1[3][y];

   setmat(mat1,mat3);
}


void viewset::viewsetinit()
{
  setimat(rmat);
  setimat(vmat);
  setimat(mat);
  vxmax=vymax=xfac=yfac=zfac = 1.0;
  sstx=ssty=sstz=tx=ty=tz = 0.0;
  vxmin=vymin=wxmin=wymin=tvxmin=tvymin= 0;    // Init Viewport + window Mins
  wxmax=wymax=1000;                            // set window to 1000 x 1000

   if (cur_screen== (screen*) NULL) {
     scrxmax=tvxmax=639; scrymax=tvymax=199;      // Set viewport and window
                                                  // prevent Div by Zero
  }
  else
  {
     tvxmax= absscrxmax;        // Set viewport and window
     tvymax= absscrymax;        // co-ord to default Screen Values
  }

  d3 = off;
  sensitized = false;
  sensitize();                          // sensitize the viewset to changes
                                        // in the graphics environment
}

void viewset::setfactors(int txx=0,int txy=0, int txz=0)
{
     float sxfac,syfac,szfac;
     float stx,sty,stz;
     int i;

     tvxmin = int(vxmin*(scrxmax-scrxmin))+scrxmin;
     tvymin = int(vymin*(scrymax-scrymin))+scrymin;
     tvxmax = int(vxmax*(scrxmax-scrxmin))+scrxmin;
     tvymax = int(vymax*(scrymax-scrymin))+scrymin;

     sxfac = xfac;    // Save old values in temp variables
     syfac = yfac;
     szfac = zfac;
     stx   = sstx;
     sty   = ssty;
     stz   = sstz;
     xfac = ((tvxmax-tvxmin)*1.0)/(wxmax-wxmin);  // set z factors in future
     yfac = ((tvymax-tvymin)*1.0)/(wymax-wymin);
     zfac = 1.0;
     tx   += txx;
     sstx = tx+(-xfac*wxmin) + tvxmin;
     ty   += txy;
     ssty =ty+(-yfac*wymin) + tvymin;
     tz   += txz;

      sxfac = (xfac / sxfac);                   // Calculate change in scale
      syfac = (yfac / syfac);
      szfac = (zfac / szfac);

     for(i = 0; i<3; i++) {                     // Fix new matrices
       vmat[i][0] *= sxfac;
        mat[i][0] *= sxfac;
       vmat[i][1] *= syfac;
        mat[i][1] *= syfac;
       vmat[i][2] *= szfac;
        mat[i][2] *= szfac;
    }
    vmat[3][0] = (vmat[3][0] -stx) *sxfac + sstx;
     mat[3][0] = ( mat[3][0] -stx) *sxfac + sstx;
    vmat[3][1] = (vmat[3][1] -sty) *syfac + ssty;
     mat[3][1] = ( mat[3][1] -sty) *syfac + ssty;
    vmat[3][2] = (vmat[3][2] -stz) *szfac + sstz;
     mat[3][2] = ( mat[3][2] -stz) *szfac + sstz;

}

void viewset::initfactors()
{

     tvxmin = int(vxmin*(scrxmax-scrxmin))+scrxmin;
     tvymin = int(vymin*(scrymax-scrymin))+scrymin;
     tvxmax = int(vxmax*(scrxmax-scrxmin))+scrxmin;
     tvymax = int(vymax*(scrymax-scrymin))+scrymin;

     xfac = ((tvxmax-tvxmin)*1.0)/(wxmax-wxmin);  // set z factors in future
     yfac = ((tvymax-tvymin)*1.0)/(wymax-wymin);
     zfac = 1.0;
     sstx = tx+(-xfac*wxmin) + tvxmin;
     ssty =ty+(-yfac*wymin) + tvymin;

     setimat(vmat);
     vmat[0][0] = xfac;
     vmat[1][1] = yfac;
     vmat[2][2] = zfac;
     vmat[3][0] = sstx;
     vmat[3][1] = ssty;
     vmat[3][2] = sstz;

}

void viewset::sensitize()
{
  if (!sensitized)
  {
     views.append(this);
     sensitized= true;
  }
}

void viewset::desensitize()
{
 if (sensitized)
    views.remove(this);
}

void viewset::checkvport()   // checks viewport co-ords to prevent illegal co-ords
{
     vxmax = (vxmax >1.0) ? (1.0) : (vxmax);    // prevent division by 0
     vymax = (vymax >1.0) ? (1.0) : (vymax);
     vxmin = (vxmin <0) ? (0) : (vxmin);    // prevent division by 0
     vymin = (vymin <0) ? (0) : (vymin);
}

void viewset::checkwindow() // check window co-ords to prevent /0;
{
   wxmax = (wxmax - wxmin) ? (wxmax) : (wxmax+1);
   wymax = (wymax - wymin) ? (wymax) : (wymax+1);
}

void viewset::setvport(float a,float b, float c, float d)

{
     vxmin = a;
     vymin = b;
     vxmax = c;
     vymax = d;

     checkvport();
     setfactors();
     setvport();
}

void viewset::movevportto(float x,float y)

{
     vxmax += x - vxmin;
     vxmin = x;

     vymax += y -vymax;
     vymin = y;
     checkvport();
     setfactors();
    setvport();
}

void viewset::movevport(float x,float y)

{
    vxmax += x;
    vxmin += x;

    vymax += y;
    vymin += y;

    checkvport();
    setfactors();
    setvport();
}


void viewset::updatevports()
{
 viewset* p;
 views.reset();
 while ( p= (viewset*) views.next() ) p->setfactors();
}

void viewset::setvport()
{
  _setcliprgn(tvxmin,absscrymax-tvymin,tvxmax,absscrymax-tvymax);  // MSC V5 inverted co-ord system
}

void viewset::framevport()
{
  _rectangle(_GBORDER,tvxmin,-tvymin,tvxmax,-tvymax);  // MSC V5
}

void viewset::erasevport()
{
 _clearscreen(_GVIEWPORT);
}

void viewset::setwindow(int a,int b,int c,int d)
{
   wxmin = a; wymin=b; wxmax=c; wymax=d;

   checkwindow();
   setfactors();
}

void viewset::movewindowto(int a,int b)
{
   wxmax += (a-wxmin);
   wxmin = a;
   wymax +=(b-wymin);
   wymin  = b;
   checkwindow();
   setfactors();
}

void viewset::scale( float fx,float fy=1.0,float fz=1.0 )
{
   wxmax = (fx) ? (int)((wxmax/fx)) : (wxmax);
   wymax = (fy) ? (int)((wymax/fy)) : (wymax);
   checkwindow();
   setfactors();
}

/*****************************************************************/
/*                      ROTATEABOUT                              */
/*                                                               */
/* This function sets a rotation matrix mat for a rotation of   */
/* d degrees about the point (x1,y1)   and Z axis               */
/*                                                               */
/* ( cos0, sin0, 0, 0)   Matrix For Rotation about a point     */
/* (-sin0, cos0, 0, 0)          in the Z direction             */
/* (    0,    0, 1, 0)  xfac = (x1*(1-cos0)) + y1*sin0         */
/* ( xfac, yfac, 0, 1)  yfac = (y1*(1-cos0)) - x1*sin0         */
/****************************************************************/

void viewset::rotateabout(int x1,int y1,int d,int rel)
{

     float dg;
     float cosdg,sindg;
     float xfac,yfac;
     int y;

     dg = radian(d);                    // Calculations for Matrix
     cosdg = float(cos(dg));
     sindg = float(sin(dg));
     xfac = (x1*(1-cosdg)) + y1*sindg;
     yfac = (y1*(1-cosdg)) - x1*sindg;

     if (!rel) {                        // If its not a relative rotation
         setmat(mat,vmat);              // reset the rotation matrix
         setimat(rmat);

         rmat[0][0] = rmat[1][1] = cosdg;        // Set rotation Matrix
         rmat[0][1] = sindg;
         rmat[1][0] = -sindg;
         rmat[3][0] = xfac;
         rmat[3][1] = yfac;

         for( y=0; y<3; y++) {
             mat[0][y] = vmat[0][y]*cosdg + vmat[1][y] * sindg;
             mat[1][y] = vmat[0][y]*-sindg + vmat[1][y] * cosdg;
             mat[3][y] = vmat[0][y]*xfac + vmat[1][y] * yfac + vmat[3][y];
         }
    }
    else
    {
          float tmat[4][3];             // Relative rotation
          float tmat1[4][3];            // Relative rotation

          setmat(tmat,rmat);            // copy matrices
          setmat(tmat1,mat);

          for( y=0; y<3; y++) {
             rmat[0][y] = tmat[0][y]*cosdg + tmat[1][y] * sindg;
             rmat[1][y] = tmat[0][y]*-sindg + tmat[1][y] * cosdg;
             rmat[3][y] = tmat[0][y]*xfac + tmat[1][y] * yfac + tmat[3][y];
             mat[0][y] = tmat1[0][y]*cosdg + tmat1[1][y] * sindg;
             mat[1][y] = tmat1[0][y]*-sindg + tmat1[1][y] * cosdg;
             mat[3][y] = tmat1[0][y]*xfac + tmat1[1][y] * yfac + tmat1[3][y];

           }
    }

}


/****************************************************************/
/*                        ROTATEABOUTX                          */
/*                                                              */
/* This function sets a rotation matrix mat for a rotation of  */
/* d degrees about the point (y1,z1)   and X axis              */
/*                                                              */
/* ( 1,    0,    0, 0)     Matrix For Rotation  about a point  */
/* ( 0, cos0, sin0, 0)          in the X direction             */
/* ( 0,-sin0, cos0, 0)     yfac = (y1*(1-cos0)) + z1*sin0      */
/* ( 0, yfac, zfac, 1)     zfac = (z1*(1-cos0)) - y1*sin0      */
/****************************************************************/

void viewset::rotateaboutx(int y1,int z1,int d,int rel)
{
    float dg;
    float cosdg,sindg;
    float yfac,zfac;
    int y;

    dg = radian(d);
    cosdg = float(cos(dg));
    sindg = float(sin(dg));

    yfac = (y1*(1-cosdg)) + z1*sindg;
    zfac = (z1*(1-cosdg)) - y1*sindg;

    if (!rel) {
        setmat(mat,vmat);
        setimat(rmat);

        rmat[1][1] = rmat[2][2] = cosdg;        // Set rotation Matrix
        rmat[1][2] = sindg;
        rmat[2][1] = -sindg;
        rmat[3][1] = yfac;
        rmat[3][2] = zfac;

        for( y=0; y<3; y++) {
           mat[1][y] = vmat[1][y]*cosdg + vmat[2][y] * sindg;
           mat[2][y] = vmat[1][y]*-sindg + vmat[2][y] * cosdg;
           mat[3][y] = vmat[1][y]*yfac + vmat[2][y] * zfac + vmat[3][y];
        }
    }
    else
    {
         float tmat[4][3];
         float tmat1[4][3];

         setmat(tmat,rmat);
         setmat(tmat1,mat);

         for( y=0; y<3; y++) {
            rmat[1][y] = tmat[1][y]*cosdg + tmat[2][y] * sindg;
            rmat[2][y] = tmat[1][y]*-sindg + tmat[2][y] * cosdg;
            rmat[3][y] = tmat[1][y]*yfac + tmat[2][y] * zfac + tmat[3][y];
            mat[1][y] = tmat1[1][y]*cosdg + tmat1[2][y] * sindg;
            mat[2][y] = tmat1[1][y]*-sindg + tmat1[2][y] * cosdg;
            mat[3][y] = tmat1[1][y]*yfac + tmat1[2][y] * zfac + tmat1[3][y];
         }
    }
}


/*****************************************************************/
/*                      ROTATEABOUTY                             */
/*                                                               */
/* This function sets a rotation matrix mat for a rotation of   */
/* d degrees about the point (x1,z1)   and Y axis               */
/*                                                               */
/* ( cos0, 0,-sin0, 0)   Matrix For Rotation about a point      */
/* (    0, 1,    0, 0)        in the Y Direction                */
/* ( sin0, 0, cos0, 0)   xfac = (x1*(1-cos0)) - z1*sin0         */
/* ( xfac, 0, yfac, 1)   yfac = (z1*(1-cos0)) + x1*sin0         */
/*****************************************************************/

void viewset::rotateabouty(int x1,int z1,int d,int rel)
{

    float dg;
    float cosdg,sindg;
    float xfac,zfac;
    int y;

    dg = radian(d);
    cosdg = float(cos(dg));
    sindg = float(sin(dg));
```

```
     xfac = (x1*(1-cosdg)) - z1*sindg;
     zfac = (z1*(1-cosdg)) + x1*sindg;

     if (!rel) {

         setmat(mat,vmat);
         setimat(rmat);

          rmat[0][0] = rmat[2][2] = cosdg;    // Set rotation matrix
          rmat[2][0] = sindg;
          rmat[1][2] = -sindg;
          rmat[3][0] = xfac;
          rmat[3][2] = zfac;

          for( y=0; y<3; y++) {
             mat[0][y] = vmat[0][y]*cosdg + vmat[2][y] * -sindg;
             mat[2][y] = vmat[0][y]*sindg + vmat[2][y] * cosdg;
             mat[3][y] = vmat[0][y]*xfac + vmat[2][y] * zfac + vmat[3][y];
          }
    }
    else
    {
          float tmat[4][3];
          float tmat1[4][3];

          setmat(tmat,rmat);
          setmat(tmat1,mat);

          for( y=0; y<3; y++) {
             rmat[0][y] = tmat[0][y]*cosdg + tmat[2][y] * -sindg;
             rmat[2][y] = tmat[0][y]*sindg + tmat[2][y] * cosdg;
             rmat[3][y] = tmat[0][y]*xfac + tmat[2][y] * zfac + tmat[3][y];
             mat[0][y] = tmat1[0][y]*cosdg + tmat1[2][y] * -sindg;
             mat[2][y] = tmat1[0][y]*sindg + tmat1[2][y] * cosdg;
             mat[3][y] = tmat1[0][y]*xfac + tmat1[2][y] * zfac + tmat1[3][y];
          }
    }

}

#pragma check_stack(off)              // TURN OFF STACK CHECKING

point viewset::setpoint(point p)
{
 return point(int(p.x*mat[0][0]+p.y*mat[1][0]+mat[2][0]*p.z + mat[3][0]),
              int(p.x*mat[0][1]+p.y*mat[1][1]+mat[2][1]*p.z + mat[3][1]),
              int(p.x*mat[0][2]+p.y*mat[1][2]+mat[2][2]*p.z + mat[3][2]));
}
inline int viewset::windx( point p)
{
    return int(p.x*mat[0][0]+p.y*mat[1][0]+mat[2][0]*p.z +mat[3][0]);
}

inline int viewset::windy(point p)
{
   return int(p.x*mat[0][1]+p.y*mat[1][1]+mat[2][1]*p.z+mat[3][1]);
}


int viewset::windowx( point p)
{
    return (d3) ?
        (int(p.x*mat[0][0]+p.y*mat[1][0]+mat[2][0]*p.z +mat[3][0])) :
        (int(p.x*mat[0][0]+p.y*mat[1][0]+mat[3][0]));
}

int viewset::windowy(point p)
{
   return (d3) ?
      (int(p.x*mat[0][1]+p.y*mat[1][1]+mat[2][1]*p.z+mat[3][1])) :
      (int(p.x*mat[0][1]+p.y*mat[1][1]+mat[3][1]));
```

```
}


double viewset::absolutex(int a)           // THIS NEEDS A BIT OF WORK !!!!!!!!
{
   return ((a-wxmin)*((tvxmax-tvxmin)*1.0/(wxmax-wxmin)));
}



void viewset::moveto(point p)
{
  _moveto(windx(p),-windy(p));       // MSC V5 - graphics
}
void viewset::lineto(point p)
{
  _lineto(windx(p),-windy(p));       // MSC V5 - graphics
}

void viewset::drawpoint(point p)
{
 _setpixel(windx(p),-windy(p));     // MSC V5 - Graphics

}

#pragma check_stack()            // TURN STACK CHECKING BACK ON

/***************************************************************/
/*                          SETOBLIQUE                         */
/* This procedure allows you to set an oblique projection for */
/* an object.                                                  */
/*                                                             */
/* (      1,    0, 0, 0)                                       */
/* (      0,    1, 0, 0)                                       */
/* ( lcosb,lsinb, 0, 0)                                        */
/* (      0,    0, 0, 1)                                       */
/***************************************************************/

void viewset::setoblique(float l,float b)
{
  float dg;

  initfactors();                     // reset viewing matrix
  dg = radian(b);
//  lcosdg = l*float(cos(dg));
//  lsindg = l*float(sin(dg));


  vmat[2][0] = l*float(cos(dg))*xfac; //+ lsindg*smat[1][0];
  vmat[2][1] = l*float(sin(dg))*yfac; // lcosdg*smat[0][1] +
  // vmat[2][2] =  lcosdg*smat[0][2] + lsindg*smat[1][2];

  setmat(mat,vmat);                        // set ctm to vmat
  premulti(mat,rmat);                      //

  d3 = on;
}

/***************************************************************/
/*                        SETPERSPECTIVE                       */
/* This procedure allows you to set a one point perspective   */
/* projection for an object.                                   */
/*                                                             */
/* (      1,      0, 0,    0)    Conny's Algorthim             */
/* (      0,      1, 0,    0)                                  */
/* ( -Xc/Zc, -Yc/Zc, 0, 1/Zc)                                  */
/* (      0,      0, 0,    1)                                  */
/***************************************************************/

void viewset::setperspective(int d)
{
  float tmat[4][3];
  setimat(vmat);                              // Reset viewing matrix
  vmat[0][0] = xfac;
  vmat[1][1] = yfac;
  vmat[2][2] = zfac;
  vmat[3][0] = tx;
  vmat[3][1] = ty;
  vmat[3][2] = tz;
/*
  setimat(tmat);

  tmat[0][0] = tmat[2][2]=-1;

  premulti(vmat,tmat);
*/
  setmat(mat,vmat);                            // set ctm to vmat
  premulti(mat,rmat);                          //
  pz = d;
  d3 = on;
}

void viewset::positionat(int x,int y)
{
  setfactors(x,y);
}

void segment::draw()
{
  base* p;
  float smat[4][3];
  shapelist.reset();
  setmat(smat,shape::viewptr->mat);     // save current viewing matrix
  premulti(shape::viewptr->mat,mat);    // add in segments matrix
  while (p = shapelist.next()) p->draw(); // draw the shapes
  setmat(shape::viewptr->mat,smat);     // reset the viewing matrix
}
```

SEGS.CXX

```
#include "stdinc.hxx"
#include "seglist.hxx"
#include "image.hxx"


/**************************************************************************/
/*                  SCREEN SCENE PROCEDURES AND FUNCTIONS                  */
/**************************************************************************/

ACTIVATE(scene);

void scene::putobj(image* ip)
{
  segment* p;

  ip->label("SCENE");
  scn_list.reset();
  while(p=(segment*) scn_list.next()) p->putobj(ip);
  ip->writenull();

};

base* scene::getobj(image* ip, base* obj)
{
  if (ip->match("SCENE"))
    {
     if (obj == (base*) NULL)
          obj = (base*) new scene;
     ip->objectread();

     while( !ip->nullobject())
     {
       ((scene*)obj)->scn_list.append((segment*) ip->loadobject("SEGMENT"));
     }
     ip->readnull();
     }
  return obj;
};


void scene::scale(float fx,float fy, float fz)
{
 segment* p;

 scn_list.reset();
 while ( p = (segment*) scn_list.next() )   p->scale(fx,fy,fz);


}

void scene::wipe()
{
 segment* p;

 scn_list.reset();
 while ( p = (segment*) scn_list.next() ) { p->desensitize();  delete p; }


}

void scene::rotateabout(int xa,int ya,int d,int rel)
{
 segment* p;

 scn_list.reset();
 while ( p = (segment*) scn_list.next() ) p->rotateabout(xa,ya,d,rel);

}

void scene::rotateaboutx(int ya,int za,int d,int rel)
{
 segment* p;
 scn_list.reset();
 while ( p = (segment*) scn_list.next() ) p->rotateaboutx(ya,za,d,rel);

}

void scene::rotateabouty(int xa,int za,int d,int rel)
{
 segment* p;

 scn_list.reset();
 while ( p = (segment*) scn_list.next() ) p->rotateabouty(xa,za,d,rel);

}

void scene::movevport(float dx, float dy)
{
 segment* p;

 scn_list.reset();
 while ( p = (segment*) scn_list.next() ) p->movevport(dx,dy);

}

void scene::movewindow(int dx,int dy)
{
 segment* p;

 scn_list.reset();
 while ( p = (segment*) scn_list.next() ) p->movewindow(dx,dy);

}

void scene::refresh()
{
 segment* p;

 scn_list.reset();
 while(p = (segment*) scn_list.next()) p->refresh();

};
/*********************************************************************/
/*               SEGMENT CLASS PROCEDURES AND FUNCTIONS              */
/*********************************************************************/

ACTIVATE(segment);

void segment::putobj(image* ip)
{
   shape* p;
   int idlen;                          // Length of the ID for the segment
   ip->label("SEGMENT");
   idlen =strlen(id)+1;
   ip->write(&idlen,sizeof(int));
   if (idlen !=1) {
      ip->write(id,sizeof(char)*idlen);
   }

   ip->write(&visible,sizeof(int));
   ip->write(&autoclear,sizeof(int));
   ip->write(&frame,sizeof(int));
   shapelist.reset();
   while(p=(shape*) shapelist.next()) p->putobj(ip);
   ip->writenull();
   viewset::putobj(ip);
};

base* segment::getobj(image* ip, base* obj)
{
   int idlen;
   if (ip->match("SEGMENT"))
     {
```

```
    if (obj == (base*) NULL)
         obj = (base*) new segment;
    ip->read(&idlen,sizeof(int));
    if (idlen !=0)
    {
     ((segment*)obj)->id = new char[idlen];
     ip->read(((segment*)obj)->id,sizeof(char)*idlen); // check it reads into right place

    }
    ip->read(&((segment*)obj)->visible,sizeof(int));
    ip->read(&((segment*)obj)->autoclear,sizeof(int));
    ip->read(&((segment*)obj)->frame,sizeof(int));
    ip->objectread();

    while( !ip->nullobject())
    {
      ((segment*)obj)->shapelist.append((shape*) ip->loadobject(""));
    }
    ip->readnull();
    viewset::getobj(ip,obj);
    }
  return obj;
};

void segment::refresh()
{
  shape* p;

  if (visible)
  {
        shape::viewptr = this;
        shapelist.reset();
        viewset::setvport();

        if (autoclear)
            viewset::erasevport();

        if (frame)
            viewset::framevport();

        while (p =(shape*) shapelist.next()) p->draw();

        viewset::cur_screen->review();

   }

}

void segment::setid(char* st)
{
 if (id == (char*) NULL)
 {
  delete id;
  }
  id = strdup(st);

}
/***********************************************************************/
/*                  SCREEN CLASS PROCEDURES AND FUNCTIONS              */
/***********************************************************************/

ACTIVATE(screen);
screen scr;                   // default screen object

void screen::putobj(image* ip)
{
  segment* p;
  ip->label("SCREEN");
  ip->write(&devmode,sizeof(int));
  ip->write(&color,sizeof(long));
  scr_list.reset();
  while(p=(segment*)scr_list.next()) p->putobj(ip);
  ip->writenull();
  viewset::putobj(ip);
};

base* screen::getobj(image* ip, base* obj)
{

  if (ip->match("SCREEN"))
    {
     if (obj == (base*) NULL)
          obj = (base*) new screen;
     ip->read(&((screen*)obj)->devmode,sizeof(int));
     ip->read(&((screen*)obj)->color,sizeof(long));
     ip->objectread();

     while( !ip->nullobject())
     {
       ((screen*)obj)->scr_list.append((segment*) ip->loadobject("SEGMENT"));
     }
     ip->readnull();
     viewset::getobj(ip,obj);
     }
  return obj;
};


void screen::init(int dev)
{
    segment* p;
    struct videoconfig config;
    if (!windowopen)
    {
        desensitize();
        viewset::cur_screen = this;      // Set screen Pointer
        shape::viewptr = this;           // Set Viewing Pointer
        _setvideomode(devmode=dev);      // set and save the device mode
        setbkcolor(color);               // set the background color
        _getvideoconfig(&config);
        absscrxmax=scrxmax= config.numxpixels;
        absscrymax=scrymax= config.numypixels;
        scrxmin=scrymin=0;
        _setlogorg(0,scrymax);

        setwindow(0,0,scrxmax,scrymax);

        setvport(0.0,0.0,1.0,1.0);
        updatevports();                  // inform other objects of change
                                         // in screen resolution
   }
}

void screen::refresh()
{
    segment* pt;
    if (!windowopen)
    {
        review();
        viewset::erasevport();
        scr_list.reset();
        while ( pt = (segment*) scr_list.next() ) pt->refresh();
   }
}


void screen::setbkcolor(long c)
{
  color = c;
  setbackcolor(color);
}
```

SHAPES.HXX

```
// **************
// SHAPES HEADER FILE
//
// DEFINES A LIST OF SHAPES THAT CAN BE USED
//
//

class dot  : public shape {

public :
      void draw();
      dot()
           { initshape(1);
             set(0,0,0);}
//      dot( int x,int y, int z=0)
//           { set(x,y,z);}
      void set(int x,int y,int z =0)
            { p[0].x = x; p[0].y = y; p[0].z = z;}
};


class line : public shape {

public :

      base* getobj(image*,base*);
      void putobj(image*);
      void draw();
//      line (int a, int b, int c,int d)
//           { line();
//             set(point(a,b),point(c,d)); }
      line()
           { initshape(2);}
      line(point p1, point p2)
           { initshape(2);
             p[0]=p1;p[1]= p2; }
      void set(int x1,int y1,int x2,int y2)
            { p[0]=point(x1,y1); p[1]=point(x2,y2);}
      void set(point a, point b)
            { p[0] = a; p[1] = b;}
};


class polygon : public shape {
      int curpos;                     // Current pos in polygon
      int closed;                     // Whether closed or not;
public :

      base* getobj(image*,base*);
      void putobj(image*);
      void draw();
      polygon()
           { initshape(10);
             curpos = 0; closed = off;}             // delfault size
      polygon(int i)
           { initshape(i);
              curpos = 0; closed = off;}
      void append(point);                  // Add a point to the polygon;
      void setclosed(int a=on) { closed = a;} // set closed on/off
};


/************************************************************************/
/*                           CLASS RECTANGLE                            */
/*                                                                      */
/*   p[0] = min;  p[1] = max; p[2] = lh1; p[3] = rh1;                   */
/*                                                                      */
/*    p[2]--------p[1]         Point Diagram of rectangle               */
/*      |           |                                                   */
/*      |           |                                                   */
/*      |           |                                                   */
/*    p[0]--------p[3]                                                  */
```

```
/***********************************************************************/

class rect : public shape {

public :
      base* getobj(image*,base*);
      void putobj(image*);

      void draw();
      rect(int a,int b,int c,int d)
             { rect();
               set(point(a,b),point(c,d)); }
      rect()
             { initshape(4);}
      void set(point,point);
      void set(int a ,int b,int c,int d)
              { set(point(a,b),point(c,d));}

};


/***********************************************************************/
/*                           CLASS BOX                                  */
/*                                                                      */
/*   p[0] = min1;  p[1] = max1; p[2] = lh1; p[3] = rh1;                 */
/*   p[4] = min1;  p[5] = max1; p[6] = lh2; p[7] = rh2;                 */
/*                                                                      */
/*       p[6]--------p[5]                                               */
/*        /|         / |                                                */
/*    p[2]--------p[1] |        Point Diagram of Box                    */
/*      |  |        |  |                                                */
/*      |p[4]-------|p[7]                                               */
/*      | /         | /                                                 */
/*    p[0]--------p[3]                                                  */
/***********************************************************************/

class box : public shape {

public :
      base* getobj(image*,base*);
      void putobj(image*);

      void draw();
      box(point a,point b, int d) {
               box();
               set(a,b,d); }
      box()
         { initshape(8);}

      void set(point a ,point b,int d);

};


/***********************************************************************/
/*                        CLASS CIRCLE                                  */
/*          -----                                                       */
/*        /       \                                                     */
/*       / r       \                                                    */
/*      |<-- p[0]   |          Point Diagram of Circle                  */
/*       \         /                                                    */
/*        \       /                                                     */
/*          ------                                                      */
/***********************************************************************/

class circle : public shape {

      int r;

public :
      base* getobj(image*,base*);
      void putobj(image*);
      void draw();
      circle(int a,int b,int c)
         { initshape(1);
           set(point(a,b),c); }
      circle() { initshape(1);
                }
      void set( int a, int b,int c)
          { set(point(a,b),c); }
      void set( point a,int c)
           { p[0] = a; r=c;}

     void scalex(float fx)               // Scale up/down x Values
            { scale(fx,1.0,1.0); }
     void scaley(float fy)               // Scale up/down y values
            { scale(1.0,fy,1.0); }
     void scalez(float fz)               // Scale up/down z values
            { scale(1.0,1.0,fz); }

      void scale(float,float = 1.0, float = 1.0);
};


class cyclinder : public shape {

      int r;
      int h;

public :
      base* getobj(image*,base*);
      void putobj(image*);

      void draw();
      cyclinder()
         { initshape(1);
            r = h =0;}
      void set( point a, int r1, int h1)
           { p[0] = a; r=r1; h = h1;}

     void scalex(float fx)               // Scale up/down x Values
            { scale(fx,1.0,1.0); }
     void scaley(float fy)               // Scale up/down y values
            { scale(1.0,fy,1.0); }
     void scalez(float fz)               // Scale up/down z values
            { scale(1.0,1.0,fz); }

      void scale(float,float = 1.0, float = 1.0);


};


class sphere : public shape {

      int r;

public :
      base* getobj(image*,base*);
      void putobj(image*);

      void draw();
      sphere()
         { initshape(1);
            r =0; }
      void set( point a, int r1)
           { p[0] = a; r=r1; }

     void scalex(float fx)               // Scale up/down x Values
            { scale(fx,1.0,1.0); }
     void scaley(float fy)               // Scale up/down y values
```

```
            { scale(1.0,fy,1.0); }
     void scalez(float fz)                  // Scale up/down z values
            { scale(1.0,1.0,fz); }

      void scale(float,float = 1.0, float = 1.0);

};
```

```
//****************************
// Prog : SHAPES.cxx
//
// Creates a numer of shape objects
//  and their supporting functions
//
//***************************

#include "stdinc.hxx"     // Include Definitions
#include "shapes.hxx"
#include "image.hxx"

extern void circledraw(point,int);

/*********************************************************************/
/*                CLASS SHAPE FUNCTIONS AND PROCEDURES               */
/*********************************************************************/

ACTIVATE(shape);

void shape::initshape( int s = 0)
{
  p = new point[s];
  sz = s;
  color = 1;
}

void shape::expandshape(int nsz)
{
  point* p1;
  if (nsz > sz) {
     p1 = new point[nsz];
     memcpy(p1,p,sizeof(point)*sz);
     delete p;
     p = p1;
     sz = nsz;
  }
};

void shape::scale(float fx,float fy, float fz)
{
  for (int i = 1; i<sz; i++ )
       p[i] = point(p[i].x+int((p[i].x-p[0].x)*fx),
                    p[i].y+int((p[i].y-p[0].y)*fy),
                    p[i].z+int((p[i].z-p[0].z)*fz));
}

void shape::rotateabout(int xa,int ya,int d)
{
  viewset v;
  v.rotateabout(xa,ya,d);
  for (int i = 0; i<sz; i++ )
       p[i] = v.setpoint(p[i]);

}
```

```
void shape::rotateaboutx(int ya,int za,int d)
{
  viewset v;

  v.rotateaboutx(ya,za,d);
  for (int i = 0; i<sz; i++ )
       p[i] = v.setpoint(p[i]);

}

void shape::rotateabouty(int xa,int za,int d)
{
  viewset v;

  v.rotateabouty(xa,za,d);
  for (int i = 0; i<sz; i++ )
       p[i] = v.setpoint(p[i]);

}

void shape::move(int dx,int dy, int dz)
{
  for (int i = 0; i<sz; i++ )
       p[i] = point(p[i].x+dx,p[i].y+dy,p[i].z+dz);
}

void shape::moveto(int x,int y, int z)
{
  int dx,dy,dz;
  dx = x-p[0].x;
  dy = y-p[0].y;
  dz = z-p[0].z;

  p[0] = point(x,y,z);
  for (int i = 1; i<sz; i++ )
       p[i] = point(p[i].x+dx,p[i].y+dy,p[i].z+dz);
}

/************************************************************/
/*             CLASS DOT PROCEDURES AND FUNCTIONS         */
/************************************************************/


ACTIVATE(dot);                              //Make dot active

void dot::draw()
{
  shape::viewptr->drawpoint(p[0]);
}

void shape::putobj(image* ip)
{
  ip->label("SHAPE");
  ip->write(&sz, sizeof(char));
  ip->write(&color, sizeof(int));
  if (sz != 0)
     ip->write(p,sizeof(point)*sz);
}

base* shape::getobj(image* ip, base* obj)
{
  if (ip->match("SHAPE"))
   {
      if (obj == (base*) NULL)
         obj = (base*) new shape;
      ip->read(&((shape*) obj)->sz,sizeof(char));
      ip->read(&((shape*) obj)->color,sizeof(int));
      if ( ((shape*) obj)->sz !=0)
      {
          ((shape*) obj)->p = new point[((shape*) obj)->sz];
          ip->read(((shape*) obj)->p,sizeof(point)*((shape*) obj)->sz);
      }
      ip->objectread();
   }
   return obj;
}

/********************************************************************/
/*                CLASS LINE FUNCTIONS AND PROCEDURES             */
/********************************************************************/

ACTIVATE(line);

void line::putobj(image* ip)
{
  ip->label("LINE");
  shape::putobj(ip);
}

base* line::getobj(image* ip, base* obj)

{
  if (ip->match("LINE"))
   {
      if (obj == (base*) NULL)
         obj = (base*) new line;
      ip->objectread();
      shape::getobj(ip,obj);
   }
   return obj;
}


void line::draw()
{
 setlinecolor(color);
 viewptr->moveto(p[0]);
 viewptr->lineto(p[1]);
}

/********************************************************************/
/*              CLASS RECTANGLE FUNCTIONS AND PROCEDURES          */
/********************************************************************/

ACTIVATE(rect);

void rect::putobj(image* ip)
{
  ip->label("RECT");
  shape::putobj(ip);
}

base* rect::getobj(image* ip, base* obj)

{
  if (ip->match("RECT"))
   {
      if (obj == (base*) NULL)
         obj = (base*) new rect;
      ip->objectread();
      shape::getobj(ip,obj);
   }
   return obj;
}

void rect::draw()
{
 point pt;

 setlinecolor(color);
 viewptr->moveto(p[0]);
 viewptr->lineto(p[2]);
 viewptr->lineto(p[1]);
 viewptr->lineto(p[3]);
 viewptr->lineto(p[0]);
}

void rect::set(point min,point max)
{
   p[0]= min; p[1]= max; p[2].x = p[0].x;
   p[3].x = p[1].x; p[3].z = p[0].z;
   p[2].y = p[1].y; p[3].y = p[0].y; p[2].z = p[1].z;
}

/*********************************************************************/
/*                  CLASS BOX FUNCTIONS AND PROCEDURES               */
/*********************************************************************/

ACTIVATE(box);

void box::putobj(image* ip)
{
  ip->label("BOX");
  shape::putobj(ip);
}

base* box::getobj(image* ip, base* obj)

{
  if (ip->match("BOX"))
   {
      if (obj == (base*) NULL)
         obj = (base*) new box;
      ip->objectread();
      shape::getobj(ip,obj);
   }
   return obj;
}


void box::draw()
{
 setlinecolor(color);
 viewptr->moveto(p[0]);
 viewptr->lineto(p[2]);
 viewptr->lineto(p[1]);
 viewptr->lineto(p[3]);
 viewptr->lineto(p[0]);

 viewptr->lineto(p[4]);
 viewptr->lineto(p[6]);
 viewptr->lineto(p[5]);
 viewptr->lineto(p[7]);
 viewptr->lineto(p[4]);

 viewptr->moveto(p[3]);
 viewptr->lineto(p[7]);
 viewptr->moveto(p[2]);
 viewptr->lineto(p[6]);

 viewptr->moveto(p[1]);
 viewptr->lineto(p[5]);

}

void box::set(point a ,point b,int d)
{
  p[0] = a; p[1] = b;
  p[4] = p[0]; p[4].z = p[0].z+d;
  p[5] = p[1]; p[5].z = p[1].z + d;
  p[2].x = p[0].x; p[3].x = p[1].x; p[3].z = p[1].z;
  p[2].y = p[1].y; p[3].y = p[0].y; p[2].z = p[0].z;
  p[6] = p[2]; p[6].z = p[2].z +d;
  p[7] = p[3]; p[7].z =p[3].z + d;
}


/**********************************************************************/
/*              CLASS CIRCLE FUNCTIONS AND PROCEDURES                 */
/**********************************************************************/

ACTIVATE(circle);

void circle::putobj(image* ip)
{
  ip->label("CIRCLE");
  ip->write(&r,sizeof(int));
  shape::putobj(ip);
}

base* circle::getobj(image* ip, base* obj)

{
  if (ip->match("CIRCLE"))
   {
     if (obj == (base*) NULL)
        obj = (base*) new circle;
     ip->read(&((circle*) obj)->r,sizeof(int));
     ip->objectread();
     shape::getobj(ip,obj);
   }
   return obj;
}


void circle::draw()
{
 circledraw(p[0],r);
}

void circle::scale(float fx,float fy, float fz)
{
  if (fx != 1.0)
     r = r + int(r*fx);
  else
     if (fy != 1.0)
       r = r + int(r*fy);
 else
     if (fz != 1.0)
       r = r + int(r*fz);
}

/**********************************************************************/
/*              CLASS POLYGON FUNCTIONS AND PROCEDURES                */
/**********************************************************************/

ACTIVATE(polygon);

void polygon::append(point p1)
{
 if (curpos >=sz)
    expandshape(sz+10);
  p[curpos] = p1;
  curpos++;
}

void polygon::putobj(image* ip)
{
  ip->label("POLYGON");
  ip->write(&curpos,sizeof(int));
  ip->write(&closed,sizeof(int));
  shape::putobj(ip);
}

base* polygon::getobj(image* ip, base* obj)
```

```
{
  if (ip->match("POLYGON"))
   {
      if (obj == (base*) NULL)
         obj = (base*) new polygon;
      ip->read(&((polygon*) obj)->curpos,sizeof(int));
      ip->read(&((polygon*) obj)->closed,sizeof(int));
      ip->objectread();
      shape::getobj(ip,obj);
   }
   return obj;
}



void polygon::draw()
{
 if (curpos > 0)
 {
   setlinecolor(color);
   viewptr->moveto(p[0]);
   for (int i =1; i<curpos; i++)
       viewptr->lineto(p[i]);
  if (closed) {
    viewptr->lineto(p[0]);
  }
  }

}

/*********************************************************************/
/*              CLASS CYCLINDER FUNCTIONS AND PROCEDURES             */
/*********************************************************************/

ACTIVATE(cyclinder);

void cyclinder::putobj(image* ip)
{
  ip->label("CYCLINDER");
  ip->write(&r,sizeof(int));
  ip->write(&h,sizeof(int));
  shape::putobj(ip);
}

base* cyclinder::getobj(image* ip, base* obj)

{
  if (ip->match("CYCLINDER"))
   {
      if (obj == (base*) NULL)
         obj = (base*) new cyclinder;
      ip->read(&((cyclinder*) obj)->r,sizeof(int));
      ip->read(&((cyclinder*) obj)->h,sizeof(int));
      ip->objectread();
      shape::getobj(ip,obj);
   }
   return obj;
}


void cyclinder::draw()
{
 circledraw(p[0],r);
 circledraw(point(p[0].x,p[0].y,p[0].z+h),r);
 viewptr->moveto(point(p[0].x+r,p[0].y,p[0].z));
 viewptr->lineto(point(p[0].x+r,p[0].y,p[0].z+h));
 viewptr->moveto(point(p[0].x-r,p[0].y,p[0].z));
 viewptr->lineto(point(p[0].x-r,p[0].y,p[0].z+h));
 viewptr->moveto(point(p[0].x,p[0].y+r,p[0].z));
 viewptr->lineto(point(p[0].x,p[0].y+r,p[0].z+h));
 viewptr->moveto(point(p[0].x,p[0].y-r,p[0].z));
 viewptr->lineto(point(p[0].x,p[0].y-r,p[0].z+h));

}

void cyclinder::scale(float fx,float fy, float fz)
{
  if (fx != 1.0)
     r = r + int(r*fx);
  else
     if (fy != 1.0)
       r = r + int(r*fy);
 else
     if (fz != 1.0)
       h = h + int(h*fz);
}

/*********************************************************************/
/*              CLASS SPHERE FUNCTIONS AND PROCEDURES                */
/*********************************************************************/

ACTIVATE(sphere);

void sphere::putobj(image* ip)
{
  ip->label("SPHERE");
  ip->write(&r,sizeof(int));
  shape::putobj(ip);
}

base* sphere::getobj(image* ip, base* obj)

{
  if (ip->match("SPHERE"))
   {
      if (obj == (base*) NULL)
         obj = (base*) new sphere;
      ip->read(&((sphere*) obj)->r,sizeof(int));
      ip->objectread();
      shape::getobj(ip,obj);
   }
   return obj;
}

extern spheredraw(point,int);

void sphere::draw()
{
 spheredraw(p[0],r);
}

void sphere::scale(float fx,float fy, float fz)
{
  if (fx != 1.0)
     r = r + int(r*fx);
  else
     if (fy != 1.0)
       r = r + int(r*fy);
 else
     if (fz != 1.0)
       r = r + int(r*fz);
}
```

SLIST.HXX

```
// *********
// SLIST HEADER FILE
///
// HEADER FILE FOR THE LIST OBJECT
//
// ******************

class slist;
class base;


class slink {
friend class slist;
friend class blist;

    slink* next;
    base* e;
    slink(base* a, slink* p) { e=a; next =p; }
};

class slist {
  slink* last;                    // last->next is head of list
  slink* vptr;                    // current position in scanning list;

public:

    void insert(base* a);              // add at head of list
    void append(base* a);              // add at tail of list
    base* get();                      // return and remove head of list
    void clear();                     // remove all links
    void remove(base*);               // remove an item from the list
    void init();                      // initialises pointers
    void reset() { vptr = 0;}
    base* next();
    slist()      { last=0; }
    slist(base* a) { last=new slink(a,0); last->next=last; }
    ~slist()     { clear(); }
};


typedef void (*PFC)(char*);              // pointer to function type
extern PFC slist_handler;
extern PFC set_slist_handler(PFC);
extern void default_error(char*);
```

```
//******************************
// Prog : SLIST.cxx
//
// Creates a list object
//
// Provides functions to
//
//    add to the begining/ end of the list
//    step through the list
//    delete from the list
//    initialise the list
//    set error handlers for the list
//    clear the list
//
//****************************

void default_error(char* s)
{
}

PFC slist_handler = default_error;


PFC set_slist_handler(PFC handler)
{
   PFC rr = slist_handler;
   slist_handler = handler;
   return rr;
}

void slist::init()
{
  last = (slink*) NULL;
  vptr = (slink*) NULL;
}

base* slist::next() {
        slink* ll;
        if (vptr == (slink*) NULL)
           ll = vptr =(last) ? last->next : (slink*) NULL;
        else {
            vptr = vptr->next;
            ll = (vptr==last->next) ? (slink*) NULL : vptr;
        }
        return ll ? ll->e : (base*) NULL;
};

void slist::insert(base* a)
{
   if (last)
      last->next = new slink(a,last->next);
   else {
       last = new slink(a,(slink*) NULL);
       last->next = last;
  }
}

void slist::append(base* a)
{
   if (last)
      last = last->next = new slink(a,last->next);
   else {
      last = new slink(a,(slink*) NULL);
      last->next = last;
   }
}

base* slist::get()
{
   if (last == (slink*) NULL) (*slist_handler)("get from empty slist"); //NOTE:
       // Contrary to BS p. 205, function ptr MUST be dereferenced
   slink* f = last->next;
   base* r = f->e;
   last = (f==last) ? (slink*) NULL : f->next;
   delete f;
   return r;
}

void slist::clear()
{
   slink* l = last;
   if (l == (slink*) NULL) return;
   do {
       slink* ll = l;
       l = l->next;
       delete ll;
    } while ( l !=last );
}

void slist::remove(base* fp)
```

```
{
 slink* p,*ll= last;
 int found = 0;

 if (last)
 {
   p = last->next;
   while ((p!=(slink*) NULL) && (!found))
   {
     if (p->e == fp)
     {
       if (ll != p)
       {
           ll-> next = p->next;
           if (p == last)
              last = ll;
       }
       else
       {
           last = (slink*) NULL;
       }
       found = 1;
       delete p;
     }
     else
     {
       ll = p;
       p = (p->next == last->next) ? ((slink*) NULL) : p->next;
     }
   }
 }
}
```

```
//****************************
// Prog : Sphere.cxx
//
// Draws a sphere on the screen
//
// Setting the sides and iterations allows you
// to determine the accuracy or speed of the drawing process
//
//***************************

#include "stdinc.hxx"


#define sides 40                // number of sides in approximating polygon
#define iter (sides/2+1)        // number of iterations

void spheredraw(point cen , int r)
{
 point p1;
 float v[iter][2],p;
 int i,oldz,zfac;
 float inc=1.0;
 float fac, oldfac;


 int old[4][2];                     // save screen co-ords
 int x,y,x1,y1;                     // general temp variables


 for (i=0,p=0; i<iter; i++,p+=3.1415926*2/sides)
 {
   v[i][0] = cos(p);                        // set up circle array
   v[i][1] = sin(p);
  }


  oldfac=fac = r;
  oldz=zfac= 0;

 while (inc >0.0)
 {
   p1.set((int)(v[0][0]*fac)+cen.x,y=(int)(v[0][1]*fac)+cen.y,zfac+cen.z);  // set starting points

   old[1][0]=old[0][0] = shape::viewptr->windowx(p1);         // front circle start x,y
   old[1][1]=old[0][1] = -shape::viewptr->windowy(p1);

   p1.set(p1.x,p1.y,cen.z-zfac);                           // back circle start x,y
   old[3][0]=old[2][0] = shape::viewptr->windowx(p1);
   old[3][1]=old[2][1] = -shape::viewptr->windowy(p1);        // Draw front circles

  for (i=1; i<iter; i++)
  {
              // Front circle top
    _moveto( old[0][0],old[0][1]);                // Draw front circle
    p1.set( x=(int)(v[i][0]*fac)+cen.x,(y=(int)(v[i][1]*fac))+cen.y,cen.z+zfac);
    _lineto(old[0][0]=shape::viewptr->windowx(p1),old[0][1]=-shape::viewptr->windowy(p1));
    p1.set(x1=((int)(v[i][0]*oldfac))+cen.x,(y1=(int)(v[i][1]*oldfac))+cen.y,cen.z+oldz);
    shape::viewptr->lineto(p1);                                // vertex
              // front circle bottom
    _moveto( old[1][0],old[1][1]);         // Draw front circles
    p1.set( x,cen.y-y,cen.z+zfac);
    _lineto(old[1][0]=shape::viewptr->windowx(p1),old[1][1]=-shape::viewptr->windowy(p1));
    p1.set(x1,cen.y-y1,cen.z+oldz);                     // vertex
    shape::viewptr->lineto(p1);

                // back face top
    _moveto( old[2][0],old[2][1]);            // back face
    p1.set(x,cen.y+y,cen.z-zfac);
    _lineto(old[2][0]=shape::viewptr->windowx(p1),old[2][1]=-shape::viewptr->windowy(p1));
    p1.set(x1,cen.y+y1,cen.z-oldz);                   // vertex
    shape::viewptr->lineto(p1);
               // back face bottom
    _moveto( old[3][0],old[3][1]);          // Draw back circles
    p1.set( x,cen.y-y,cen.z-zfac);
    _lineto(old[3][0]=shape::viewptr->windowx(p1),old[3][1]=-shape::viewptr->windowy(p1));
    p1.set(x1,cen.y-y1,cen.z-oldz);                                  // vertex
    shape::viewptr->lineto(p1);
   }
   inc-= 0.05;
   oldfac= fac;
   fac = r*inc;
   oldz = zfac;
   zfac= (int)(sqrt(1.0-(inc*inc))*r);

 }
}
```

```
// **************
// STDINC.HXX
//
// DEFINES THE STANDARD OBJECTS AND MACROS USED BY THE SYSTEM
//
// ***************

#define ACTIVATE(bp) base1 zzz##bp##((base*) new bp );

#define true   1
#define false  0
#define on     1
#define off    0
#define ADD    1
#define relative  1

extern float  radian(float);
extern double cos(double);
extern double sin(double);
extern double sqrt(double);

class segment;
class screen;
class viewset;
class base;


#include "stdinc.h"
#include "slist.hxx"                    // List Of Include FILES !!
#include "blist.hxx"                    // static list file
#include "point.hxx"

class image;
class base;
/***************************************************************************/
/*                                                                         */
/*                               CLASS BASE                                */
/*                                                                         */
/***************************************************************************/

class base {
friend image;
friend class base1;
  static blist objectlist;                // list of all objects created


public :

  virtual base* getobj(image*, base* )
                 { return (base*) NULL; }    // load an object from disk;
  virtual  void putobj(image*) {}            // save an object to disk;
  virtual  void draw() { }                   // Draw an object on the screen
  void append(base* bp)
           { objectlist.append(bp); }        // append an object to the list

};

class base1 : public base
{

  public:
  base1(base* bp) { objectlist.append(bp); }    // append classes to list
  base1()         { base::objectlist.init();}   // init the object list
  ~base1()        { base::objectlist.clear();}  // delete the list

};

// static base1 tyhghdsucsjhja;               // Set up object management list
                                            // in memory

/***************************************************************************/
/*                                                                         */
/*                              CLASS BITMAP                              */
/*                                                                        */
/**************************************************************************/

class bitmap : public base {              // Bitmap Class to store image
  char far * sav;                         // Pointer to storage area
  int xmin,ymin,xmax,ymax;                // Stored rectangle area
  unsigned long size;                     // size in bytes of area;

public :
  bitmap() { sav = (char far*) NULL;
             size = 0;
             xmin=ymin=xmax=ymax=0; }
  ~bitmap() { if (sav != (char far*) NULL)
                   delete sav; }
  base* getobj(image*,base*);
  void putobj(image*);
  int save(int,int,int,int);
  void restore(int,int,int,int);
  void restore();
};


/**************************************************************************/
/*                              CLASS VIEWSET                             */
/**************************************************************************/

// NOTE : The viewing matix VMAT is stored multiplied by the scaling
//        matrix to save multiplication time in rotations.

class viewset: public base {

friend class screen;
friend class segment;
friend class window;

    int tvxmin,tvymin,tvxmax,tvymax;   // Viewport Co-ords
    int wxmin,wymin,wxmax,wymax;       // window co-ords
    float vxmin,vymin,vxmax,vymax;     // viewport in ndc's
    float mat[4][3];                   // general clipping matrix
    float vmat[4][3];                  // Contains viewing matrix
    float rmat[4][3];                  // Contains rotation matrix

    float xfac,yfac,zfac;              // Scaling factors
    float tx,ty,tz;                    // Scaling translations
    float sstx,ssty,sstz;              // Temporary translation values.

    int d3;                            // whether 3-d is on/off
    int pz;                            // z centre of perspective proj;
    char sensitized;                   // whether viewset is sensitive to change
    static screen* cur_screen;         // Pointer to current screen
    static int windowopen;             // Number of open windows
    static int scrxmax,scrymax;        // logical max x & y co-ords of screen
    static int scrxmin,scrymin;        // logical min x & y co-ords of screen
    static int absscrxmax,absscrymax;  // physical max screen co-ords
    static blist views;                // list of viewsets created

    void setimat(float a[4][3]);                 // Set a = Identity matrix
    void matmulti(float a[4][3],float b[4][3]); // Multiply matrix a by b
    void premulti(float a[4][3],float b[4][3]); // Pre-Multi matrix a by b
    void setmat(float a[4][3],float b[4][3]);   // Set matrix a = b
    void setfactors(int=0,int=0,int=0);          // calculate scaling factors
    void initfactors();                          // init the scaling & viewing matrix
    void checkvport();                           // check the viewport co-ords
    void checkwindow();                          // check the window co-ords
    int windx(point);                            // returns clipped & scaled xcord
    int windy(point);                            // returns clipped & scaled ycord

public :
        void viewsetinit();                        //  Init default values
        void updatevports();                       // update any vports
        void sensitize();                          // make viewset sensitive
        void desensitize();                        // desensitize viewset

        viewset() { viewsetinit();}
        ~viewset() { if (sensitized)             // remove viewset from list
                          desensitize(); }
        base* getobj(image*,base*);              // load viewset from disk
        void putobj(image*);                     // save viewset to disk

    int windowx(point);                          // returns clipped & scaled xcord
    int windowy(point);                          // returns clipped & scaled ycord

    float* savemats();                           // save the current matrix
    void retmats(float*);                        // restore current matrices

    double absolutex(int);                          // returns absloute scaled value
    void positionat(int,int);                       // position viewport at point
    int windowx(int a,int b ,int c=0)
              { return windowx(point(a,b,c));}
    int windowy(int a,int b,int c=0)
              { return windowy(point(a,b,c)); }

    void scale(float,float,float);               // Main scaling

    void scalex(float fx)                        // Scale up/down x Values
           { scale(fx,1.0,1.0); }
    void scaley(float fy)                        // Scale up/down y values
           { scale(1.0,fy,1.0); }
    void scalez(float fz)                        // Scale up/down z values
           { scale(1.0,1.0,fz); }
    void framevport();                           // frame the current viewport
    void erasevport();                           // clear the current viewport
    void setvport(float,float,float,float);      // Set viewport co-ords

    void movevport(float,float);                 // Move viewport relative
    void movevportto(float,float);               // Move viewport Absolute

    void setvport();                             // set viewport to viewsets co-ords
    void setwindow(int,int,int,int);             // set window co-ords

    void movewindowto(int,int);                  // Move window to co-ords
    void movewindow(int x,int y)
      { movewindowto(int(wxmin+x),int(wymin+y)); } // move window by offsets

    void rotateabout(int,int,int,int=relative);     // Rotate about point in Z Dir
    void rotateaboutto( int a,int b,int c)          // Rotate absolute about
            { rotateabout(a,b,c,!relative); }       // point in Z dir
    void rotate(int a)                              // Rotate about origin
            { rotateabout(0,0,a);}                  // in Z Dir
    void rotateto(int a)                            // Rotate absolute about
            { rotateabout(0,0,a,!relative); }       // Origin in Z dir
    void rotateaboutx(int,int,int,int = relative); // Rotate about point in X Dir
    void rotateaboutxto(int a,int b, int c)         // Rotate absolute about
            { rotateaboutx(a,b,c,!relative); }      // point in X dir
    void rotatex(int a)                             // Rotate about origin
            { rotateaboutx(0,0,a);}                 // in X dir
    void rotatexto(int a)                           // Rotate Absolute about
            { rotateaboutx(0,0,a,!relative); }      // origin in X direction
    void rotateabouty(int,int,int,int = relative); // Rotate about point in Y Dir
    void rotateaboutyto(int a,int b,int c)          // Rotate absolute about
            { rotateabouty(a,b,c,!relative); }      // point in Y dir
    void rotatey(int a)                             // Rotate about origin
            { rotateabouty(0,0,a);}                 // in Y dir
    void rotateyto(int a)                           // Rotate absolute about
            { rotateabouty(0,0,a,!relative); }      // origin in Y dir

    void set3d() { d3 = on; }
    point setpoint(point);

    void moveto(point);                             // Move to a point
    void lineto(point);                             // line
    void drawpoint(point);                          // draw a point
```

```
     void setperspective(int);                   // Set Perspective
     void setoblique(float,float);               // set oblique l,b projection
     void setcavalier(float c = 30)              // Cavalier projection
                         { setoblique(1,c); }
     void setcabinet(float c = 30)               // Cabinet Projection
                         { setoblique(0.5,c); }
     void setortho() {   setoblique(0,90);       // Orthographic Projection
                               }                 // to be worked ON !!!!!!

};


/*************************************************************************/
/*                                                                       */
/*                              CLASS SHAPE                              */
/*                                                                       */
/*              DRAW command allready included in base object.           */
/*************************************************************************/

class shape : public base {                // general shape structure

public :
     unsigned char sz;                     // contains No. of points

     static viewset* viewptr;              // pointer to current viewset
     point* p;                             // array of points
     int color;                            // color of the shape

     shape()                               // shape creator
          { p = (point*) NULL;
            sz = 0; }
     ~shape() {delete p;}
     void initshape(int=0);                // shape initializer
     void expandshape(int);                // expand the shape;
     void setcolor(int c) { color = c;}    // set the shapes color
     base* getobj(image*,base*);           // load shape from disk
     void putobj(image*);                  // save shape to disk


     void scale(float,float=1.0,float=1.0); // Main scaling

     void scalex(float fx)                 // Scale up/down x Values
            { scale(fx,1.0,1.0); }
     void scaley(float fy)                 // Scale up/down y values
            { scale(1.0,fy,1.0); }
     void scalez(float fz)                 // Scale up/down z values
            { scale(1.0,1.0,fz); }
     void rotateabout(int,int,int);        // Rotate about point in Z Dir
     void rotate(int a)                    // Rotate about origin
             { rotateabout(0,0,a);}        // in Z Dir
     void rotateaboutx(int,int,int);       // Rotate about point in X Dir
     void rotatex(int a)                   // Rotate about origin
             { rotateaboutx(0,0,a);}       // in X dir
     void rotateabouty(int,int,int);       // Rotate about point in Y Dir
     void rotatey(int a)                   // Rotate about origin
             { rotateabouty(0,0,a);}       // in Y dir
     void move(int,int,int=0);             // move shape relative
     void moveto(int,int,int=0);           // move absolute


};
```

TEXT.HXX

```
// **********
// TEXT.HXX
//
// DEFINES THE TEXT OBJECT USED BY THE USER TO DRAW
// TEXT ON THE SCREEN
//
// ************

#include "text.def"
/*****************************************/
/*                                       */
/*          TEXT CLASS                   */
/*                                       */
/*****************************************/

class txt : public shape {
     char*   strng;                    // text string to be written
     int   fontid;                     // font to be used
     float sx,sy;                      // scaling factors
public :
      void draw();
    txt()
           { initshape(1);
             p[0].set(0,0,0);
             strng =(char*) NULL;
             sx=sy=1.0;
             fontid =0;}
/*      txt( int x=0,int y=0, int z=0)
           { initshape(1);
             p[0].set(x,y,z);
             strng = (char*) NULL;
             sx=sy=1.0;
             fontid = 0;}
*/
      ~txt() { if (!(strng==(char*)NULL)) delete strng;}
      void settxt(int,point , char* );
      void setscale(float,float);
};
```

```
#include "stdinc.hxx"     // Include Definitions
#include "text.hxx"

extern void wrchar(int,point,char*,float,float);

void txt::settxt(int fname,point pt, char* str)
{
  p[0] = pt;
  if (!(strng==(char*)NULL))
      delete strng;
  strng= strdup(str);
  fontid= fname;
}

void txt::setscale(float ssx, float ssy)
{
 sx= ssx; sy= ssy;
}


void txt::draw()
{
  wrchar(fontid,p[0],strng,sx,sy);
}
```

W.CXX

```
#include "stdinc.hxx"
#include "seglist.hxx"
#include "image.hxx"

float* viewset::savemats()
{
 float* p1,*p;
 p1= p = new float[36];
 memcpy((void *) p1,(void *) mat, 48);      // size of float * 12 members
 p1+=12;
 memcpy((void *) p1,(void *) rmat, 48);      // size of float * 12 members
 p1+=12;
 memcpy((void *) p1,(void *) vmat, 48);      // size of float * 12 members
 return p;
}

void viewset::retmats(float* p)
{
 float* p1;
 p1= p ;
 memcpy((void *) mat,(void *) p1, 48);      // size of float * 12 members
 p1+=12;
 memcpy((void *) rmat,(void *) p1, 48);      // size of float * 12 members
 p1+=12;
 memcpy((void *) vmat,(void *) p1, 48);      // size of float * 12 members
 delete p;
}

/*********************************************************************/
/*               WINDOW CLASS PROCEDURES AND FUNCTIONS               */
/*********************************************************************/

ACTIVATE(window);

void window::setautosave(int o)
{
  autosave = o;
}
void window::open()
{
  if (id == 0) {

     if ((autosave == off ) || (imptr.save(tvxmin,tvymin,tvxmax,tvymax))) {
         desensitize();
         tscrxmax=scrxmax;
         tscrymax=scrymax;
         tscrxmin=scrxmin;
         tscrymin=scrymin;
         scrxmax=tvxmax-1;              // reduce by one in case the
         scrymax=tvymax-1;              // viewport is framed
         scrymin=tvymin+1;
         scrxmin=tvxmin+1;

         windowopen ++;
         id = windowopen;
         sav = shape::viewptr;       // Save old pointer
         shape::viewptr = this;      // All shape operations directed to window
         updatevports();             // update any vports necessary
         refresh();                  // Draw anything in the window
         sensitize();
     }

  }
}

void window::close()
{
  if (id ==windowopen && id !=0) {
         scrxmax=tscrxmax;
         scrymax=tscrymax;
         scrxmin=tscrxmin;
         scrymin=tscrymin;
```

```
        viewset::setvport();
        if ( autosave == on)
           imptr.restore();
        windowopen--;
        shape::viewptr = sav;
        id = 0;
        updatevports();
  }
}

void window::refresh()
{
  window * t;

  t = this;

  if (visible && (windowopen == id) && (id !=0)) {
        shape::viewptr = this;
        shapelist.reset();
        viewset::setvport();
        if (autoclear)
            viewset::erasevport();
        if (frame)
            viewset::framevport();

        viewset::cur_screen->review();
   }
}

void window::setvport(float a,float b, float c, float d)
{
 if (id ==0) {
        viewset::setvport(a,b,c,d);
 }
}

void window::movevport(float a,float b)
{
  if (id ==0) {
      viewset::movevport(a,b);
  }
}
void window::movevportto(float a,float b)
{
  if (id ==0) {
      viewset::movevportto(a,b);
   }
}
```

Bibliography

[ACAD] *AutoCAD — a computer aided design package*
AutoDesk Incorporated, 2320 Marinship Way, Sausalito, CA 94965

[BCOX86] *Object oriented programming – An evolutionary Approach*
Brad J. Cox
Addison-Wesley,1986

[BOOC86] *Object Orientated Development*
G. Booch
IEEE Transactions on software engineering,
Vol. SE-12, No. 2, February 1986

[BOZE85] *A Geometric Modeller for Turbomachinery Applications*
B. Ozell & R. Camarero
CAD center, Ecole Polytechnique, C.P. 6079, Succursale A, Montreal, H3C 3A7, Canada.

[BROO86] *No silver bullet – essence and accidents of software engineering*
F.P. Brooks
Information processing 86, Elsevier Science Publishers B.V. (North-Holland) ,1986

[BUZZ85] *Object-based computing and the Ada programming language*
G. Buzzard and T. Mudge
Computer, Vol 18,No. 3,pg 12,1985

[DINF79] *Proposal of Standard DIN 66 252 Information Processing Graphical Kernel System (GKS), Function Description (1979)*
Deutsches Instuit Fur Normung

[FOLE84] *Fundamentals of interactive computer graphics*
J.D. Foley, A. Van Dam
Addison-Welsey Publishing Company,1986

[GUED76] *Methodology in Computer Graphics*
R.A. Guedj, H.A. Tuckers (EDS.)
Proc IFIP WG 5.2 Workshop Seillac I, May 1976, North-Holland, Amsterdam (1979).

[GUTT78] *The Design of Data Type Specification (Current Trends in Programming Methodology. Vol. 4)*
J. Guttag, E. Horowitz and D. Musser
Englewood Cliffs, NJ: Prentice Hall, 1978, pg. 200

[HOOP89] *HOOP 2.03 (Hierarchial Object-Oriented Picture Systems)*
ITHACA SOFTWARE
902 WEST SENECA ST., ITHACA, NEW YORK 14850

[HOPG86] *Geometric Modelling and Computer Graphics (techniques and applications)*
F.R.A HOPGOOD
RUTHERFORD APPLETON LABORATORY, UNICOM TECHNICAL PRESS, 1986

[JOHN63] T.E. JOHNSON ,MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Sketchpad III, A Computer Program For Drawing In Three Dimensions
Interactive Computer Graphics, IEEE Computer Society

[META88] *MetaWINDOW - Graphics Functions Library*
MetaGraphics Software Corporation, Scotts Valley, CA, 1986

[MACA88] *Object-Oriented Programming for the Macintosh*
K. SCHUMUCKER
HAYDEN BOOKS, 1988

[MEYE81] *Towards a two-dimensional programming environment*
B. MEYER
READINGS IN ARTIFICIAL INTELLIGENCE. PALO ALTO, CA: TIOGA, 1981 , PG. 178

[MYERSW] *Interactive Computer Graphics*
W. MYERS

[NEWM79] *Principles of interactive computer graphics*
W.N. NEWMAN, R.F. SPROULL
MCGRAW-HILL BOOK COMPANY,1979

[PARN72] *On the criteria to be used in decomposing systems into modules*
D.L. PARNSA
COMMUNICATIONS ACM, DECEMBER 1972

[PFOR87] *PforCe++ - libraries for C++*
NOVUM. ORGANUM INC.
PHOENIX TECHNOLOGIES LTD., 1987

[SHAW84] *Abstraction techniques in modern programming languages*
M.SHAW
IEEE SOFTWARE, VOL. 1.,NO. 4, PG 10, OCT 1984

[SPROUL] *An Approach to graphics system design*
W.M. Newman & R.F. Sproull

[STRa86] *A better C ?*
Bjarne Stroustrup
Byte August 1988, McGraw-Hill, pg. 215-216

[STRO86] *The C++ reference manual*
Bjarne Stroustrup
Addison-Welsey 1986

[SUTHER] *Computer Displays*
I.E. Sutherland

[WNEW79] *Principles Of Interactive Computer Graphics*
W.M. Newman, R. F. Sproull
McGraw-Hill Book Company, 1979 ,pg 80