SECTION 2 COMPUTER GRAPHICS AND MULTIMEDIA – LAB

Structure **Page Nos.**

2.0 INTRODUCTION

In this section of the lab course, you will have hands on experience of Computer graphics programming C language and visualisation of these graphics using OpenGL graphics software. This section is based on course MCS–053: *Computer Graphics and Multimedia*. Before getting into problem solving using C you will learn with the use of the following:

- What is OpenGL?
- The working of OpenGL.
- The commands needed to work with OpenGL.
- Drawing a picture with the use of OpenGL.

A list of programming problems is given session wise at the end of this section.

2.1 OBJECTIVES

After completing this lab section, you should be able to:

- get and install OpenGL software;
- create a project in VC++ 6.0 and runs an OpenGL application;
- familiarise yourself with OpenGL graphics picture software;
- list various primitives of the OpenGL;
- describe and list the OpenGL function for 3-D transformation and viewing;
- implement transformation and 3-D viewing using OpenGL Library functions primitives for the suggested problems;
- familiarisation yourself with VRML, and
- familiarisation yourself with various features of VRML.

2.2 INSTALLING OPENGL

It is a straightforward and cost free process to obtain the OpenGL software for almost any computer platform. Here, we will discuss how to access the software for each of the major platforms. On the Internet we can get enough information about OpenGL. The Internet site *http://www.opengl.org/*gives us a rich source of information about OpenGL and a starting point for downloading software. Some on-line manuals are also available at Silicon Graphics' site: *http://www.sgi.com/software/opengl/manual.html*. The book OpenGL Programming Guide, by Mason Woo, Jacki Neider, and Tom Davis (1999, currently in its 3rd edition), is one of the essential sources of information about using OpenGL, and gives pointers on obtaining and installing the software.

Need for OpenGL

With any system, you start with a C\C++ compiler and install appropriate OpenGL header files (.h) and libraries. Three Libraries associated with header files are required for the use of OpenGL:

- OpenGL (The basic API tool)
- GLU (the OpenGL Utility Library)
- GLUT (the OpenGL Utility Toolkit, a windowing tool kit that handles window system operations).

  Typically, files associated with each library are:

  - Header files (.h):
  - Library files: Static library files (.lib) and Dynamic linked library (.dll).

Adding Header Files*:* We place (add) the 3 header files: *Gl.h, Glu.h,* and *Glut.h* in a *gl* subdirectory of the include directory of your compiler. Thus, in each application program we write the following include statements:

```
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/Glut.h>
```

Adding Library Files*:* Now, you have to add the appropriate OpenGL library (.lib and .dll) files to your project so that the linker can find them, in order to run your program successfully.

MS Visual C++ (5.0 or later version) is a suitable environment for using OpenGL. If you are working with MS Visual C++ i.e., "win32 console application", there is no need to build a Graphical User Interface (GUI): as the input from the keyboard and output of text take place through the separate console window.

The library files are available on the Microsoft ftp site:
ftp://ftp.microsoft.com/softlib/mslfiles/opengl95.exe , which includes gl.h, glu.h, glu32.lib, opengl32.lib, glu32.dll and opengl32.dll.

The glut library files is available on the site:
http://reality.sgi.com/opengl/glut3/glutdlls.zip which includes glut.h, glut32.lib, and glut32.dll.

When these files have been downloaded, place all the three (.lib) files and all the three (.dll) files in some suitable directory (if they are not already there). For example, if we

are using MS Visual C++, add all (.lib) files in "c:\ Program Files\Microsoft Visual studio/VC98/lib" then, place the all (.dll) files in "c:\WinNT\System32 directory".

Thus, to run an OpenGL Application on your system, you need the following:

1) MS Visual C++ (5.0 or later version)
2) Library files: Static(.lib) and Dynamic (.dll) and
3) Header files (.h)

Static Libraries:

- **glu32.lib** : Available in VC98 installation folder Microsoft Visual Studio\VC98\Lib.
- **opengl32.lib** : Available in VC98 installation folder Microsoft Visual Studio\VC98\Lib
- **glut32.lib** : Available at OpenGL web sites, copy this file to the above folder.

Dynamic Libraries:

- **glu32.dll** : Available in WINDOWS\system32.
- **opengl32.dll** : Available in WINDOWS\system32.
- **glut32.dll** : Available at OpenGL web sites, copy this file to the above folder.

Header Files:

- **gl.h** : Available in VC98 installation folder Microsoft Visual Studio\VC98\Include\GL.
- **glu.h** :Available in VC98 installation folder Microsoft Visual Studio\VC98\Include\GL.
- **glut.h** : Available at OpenGL web sites, copy this file to the above folder.

2.3 CREATING A PROJECT IN VC++ 6.0 AND RUNNING AN OPENGL APPLICATION

i) Open VC++ 6.0
ii) Choose: ***File → new***
iii) In the Projects tab choose: ***Win32 Console Application***
iv) Enter **Project name** and **Location**: eg: OpenGLExample, C:\PROGRAM FILES\MICROSOFT VISUAL STUDIO\MY PROJECTS\ OpenGLExample.
v) Click **OK**.
vi) Choose option "***An empty project***".
vii) Click **finish** then **OK**. *Now a workspace is created.*
viii) Now go to **Project →Settings**
ix) Go to **Link** tab menu.

x) In the "***Object/Library modules***" enter the names of the three .lib files ***glu32.lib, glut32.lib and opengl32.lib***.
xi) Press **OK.**
xii) Now goto **File→ new**
xiii) Choose *C++ Source file.*
xiv) Name it ***GLExample*** (for example).
xv) Press **OK.**
xvi) Type your code in that source file.
xvii) Press **F7** to build.
xviii) Press **Ctrl+F5** to run

Let us consider the step-by-step procedure (as describe above) to execute an example program in VC++ 6.0.

Step: 1-2: After Opening VC++ 6.0, and choosing File menu, you will see the following window:

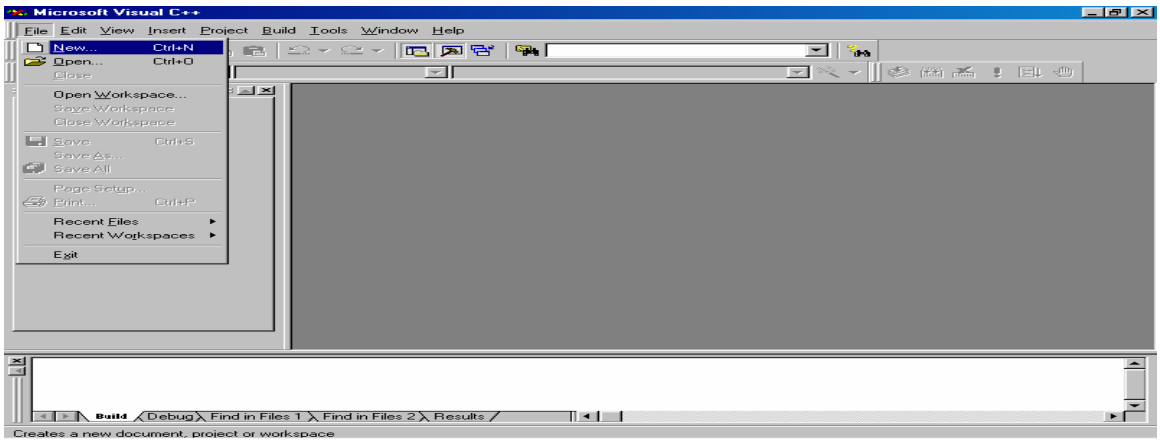

Figure 1: After Opening VC++ 6.0 and Choosing New from File Menu

Step:3-5: In the Projects tab choose: ***Win32 Console Application*.** Type your **Project name (For** example **OpenGLExample).** Click **OK** ***Application.***

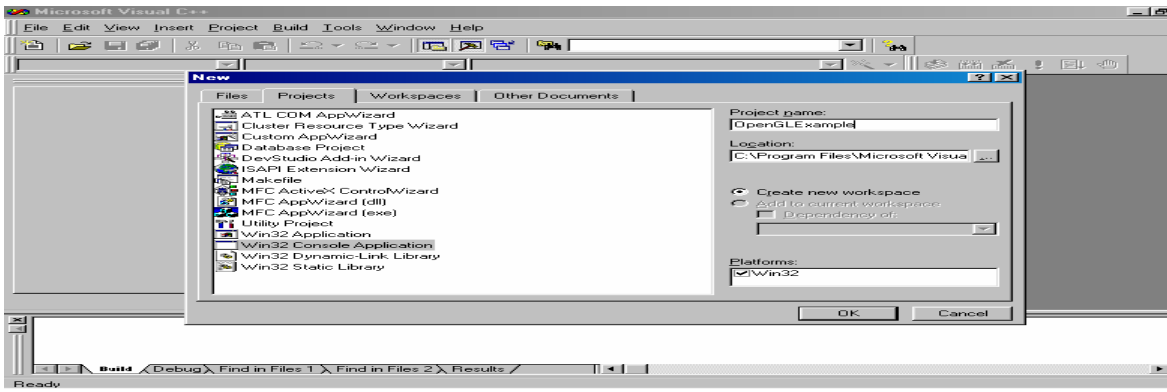

Figure 2: Selecting Project Tab Menu from Title Bar and Choosing *Win32 Console*

Now type your *Project name* (For example **OpenGLExample).**
Step:6-7: Choose an Option "**An empty Project**", then Click **Finish** and **OK**. Now a Work space is created.
Step 8: Now go to **Project →Settings**.
Step 9-11: Go to **Link** tab menu. In the "***Object/Library modules***" enter the names of the three .lib
files. ***glu32.lib glut32.lib opengl32.lib***. Press OK.

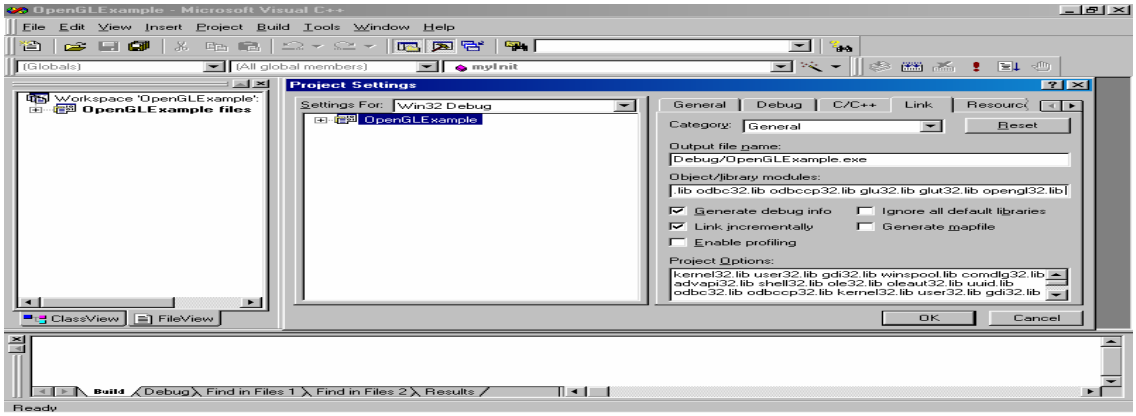

Figure 3: After Typing the Names of the Three .lib files. *glu32.lib glut32.lib opengl32.lib* in the *Object/Library modules*.

Step:12-15: Go to **file→new.** Now choose **C++ source file.** Give **file name** as **GLExample** (for example)

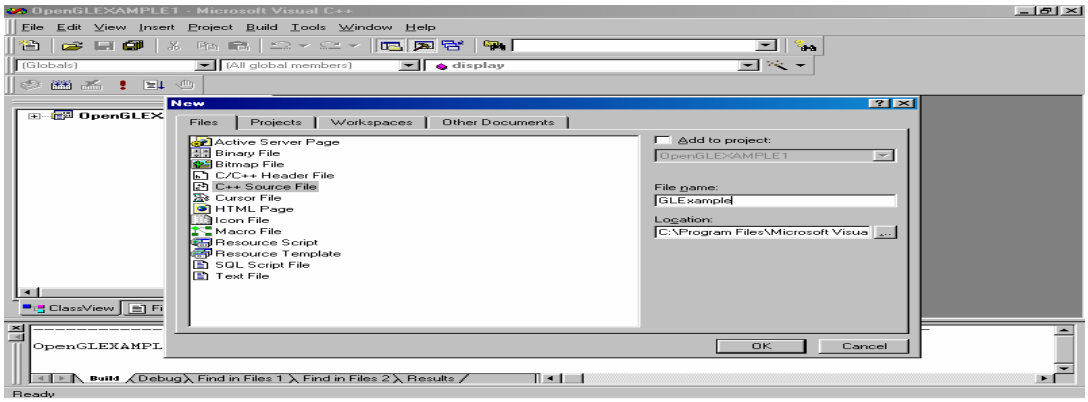

Figure 4: After selecting File Menu Tab & Select New. Now Type your File Name in the *File Name* Title Bar (say GLExample).

Step-16: Type your source code in that source file.

Step17: Press **F7** to build your file.

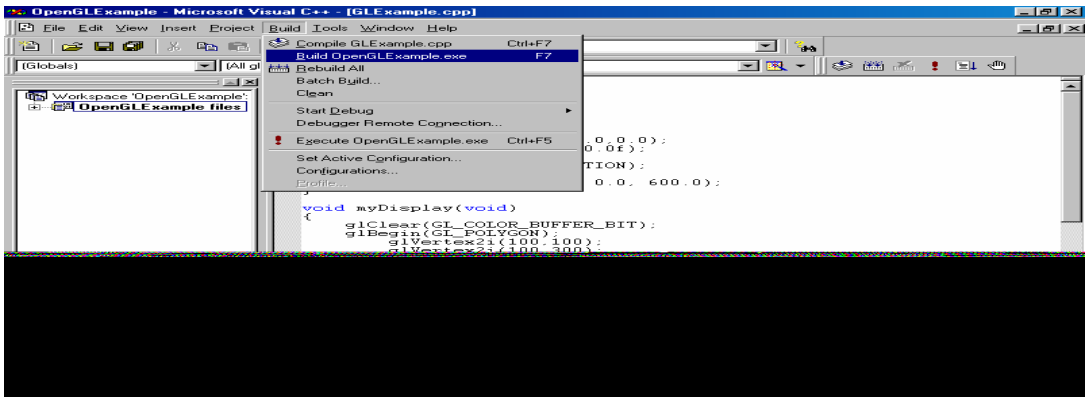

Figure 5: After writing your code in the work space provided, & press F7 to build your file.

Step18: Press **Ctrl+F5** to run your file **OpenGLExample.cpp**

Figure 6: The Output of the program

The following is a complete program written on Step:16-17, to draw a **filled polygon**:

```
#include <windows.h>   // use as needed for your system
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>
    // *************** myInit() **************
void myInit(void)
{
        glClearColour(1.0,1.0,1.0,0.0);      // set white background colour
        glColour3f(0.0f, 0.0f, 0.0f);          // set the drawing colour
        glPointSize(4.0);                    // One 'dot' is a 4 by 4 pixel
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, 800.0, 0.0, 600.0);
}
  // ********************** myDisplay() ***********
void myDisplay(void)
{
        glClear(GL_COLOUR_BUFFER_BIT);         // clear the screen
        glBegin(GL_POLYGON);       //Draw the polygon with the following vertex
point
        glVertex2i(100,100);
            glVertex2i(100,300);
            glVertex2i(400,300);
            glVertex2i(600,150);
           glVertex2i(400,100);
        glEnd();
        glFlush();                          // send all output to the display
}
   // ****************** main function ******************
void main(int argc, char** argv)
{
glutInit(&argc, argv);  // initialise the toolkit
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);  // set display mode
glutInitWindowSize(800,600);        // set window size
glutInitWindowPosition(100,100);   // set window position on your screen
glutCreateWindow("OpenGL Window");   // Open the screen window with this name
glutDisplayFunc(myDisplay);            // register redraw function
myInit();
glutMainLoop();                      // go into a perpetual loop
}
```

2.4 BASICS OF OPENGL

OpenGL emerged from Silicon Graphics Inc. (SGI) in 1992 and has become a widely adopted graphics Application Programming Interface (API). An industry consortium, Architecture Review Board (ARB), is responsible for guiding the evolution of this software. OpenGL provides a set of drawing tools through a collection of functions that are known as within an application. OpenGL is a library of functions for fast 3D rendering. That means you provide the data in an OpenGL-useable form and OpenGL will show this data on the screen (render it).

It is developed by many companies and it is free to use. You can develop OpenGL-applications without licensing. It is easy to install and use as we have already discussed in section 2.2. It is easily available (usually through free downloads over the Internet) for all types of computer systems.

One basic advantage of using OpenGL for a Computer Graphics course is its hardware- and "system-independent" interface. An OpenGL-application will work on every platform, as long as there is an installed implementation.

The OpenGL API is the premier environment for developing portable, interactive 2D and 3D graphics applications. A low-level, vendor-neutral software interface, the OpenGL API is sometimes called the "assembler language" of computer graphics. In addition to providing enormous flexibility and functionality, Applications in markets such as CAD, content creation, energy, entertainment, game development, manufacturing, medical, and VRML have benefited from the characteristics of platform accessibility and depth of functionality of the OpenGL API.

Because OpenGL is system independent, there are no functions to create windows etc., but there are helper functions for each platform. A very useful thing is GLUT.

What is GLUT?

An OpenGL Utility Toolkit (GLUT) is a complete API written by Mark Kilgard which lets you create windows and handle messages. It exists for several platforms, that means that a program which uses GLUT can be compiled on many platforms without (or at least with very few) changes in the code. Thus, GLUT does include functions to open a window on whatever system you are using.

The following code by using OpenGL Utility Toolkit can open the initial window for drawing:

```
#include <windows.h>   // use as needed for your system
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>
void main(int argc, char** argv)
{
    glutInit(&argc, argv);  // initialise the toolkit
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);  // set display mode
    glutInitWindowSize(640,480);        // set window size
    glutInitWindowPosition(100,100);   // set window position on your screen
    glutCreateWindow("OpenGL Window");   // Open the screen window
    glutDisplayFunc(myDisplay);            // register redraw function
    myInit();                                  // additional initialisations as necessary
    glutMainLoop();                        // go into a perpetual loop
}
```

The first five functions of main () use the OpenGL Utility Toolkit to initialise and display the screen window in which your program will produce graphics. You can copy these functions as your first graphics program and run, it will open the initial window for drawing. A brief description of these five functions is as follows:

- glutInit(&argc,argv): This function initialises the OpenGL Utility Toolkit. Its arguments are the standard ones for passing information of command lines.

- glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB); : This function specifies how the display should be initialised. The built-in constants GLUT_SINGLE and GLUT_RGB, which are ORed together, indicate that a single display buffer should be allocated and that colours are specified using desired amounts of red, green and blue.

- glutInitWindowSize(640,480); : This function specifies that the screen window should initially be 640 pixels wide and 480 pixels high. The user can resize the window as desired.

- glutInitWindowPosition(100,100); : This function specifies that the window's upper left corner should be positioned on the screen 100 pixels over from the left edge and 100 pixels down from the top. When the program is running, the user can move this window whenever desired.

- glutCreateWindow("OpenGL Window"); : This function actually opens and displays the screen window, putting the title " OpenGL Window" in the title bar.

How does OpenGL work?

OpenGL keeps track of many *state variables,* such as the current size of a point, the current colour of drawing, the current background colour, etc. The value of the state variable remains active until a new value is given. Each state variable has a default value. The values of the state variables, whether set by default or by the programmer, remain in effect until changed. At any point, the programmer can query the system for any variable's current value. Typically, one of the four following commands is used to do this depending upon the desired data type of the answer: *glGetBooleanv(), glGetDoublev(), glGetFloatv(),* or *glGetIntegerv()*. For example the current size of a point, line width, colour of a drawing, the background colour etc. can be set as follows:

Point Size

To control (or set) the size of a rendered point use the command glPointSize(GLfloat *size*) where *size* is the desired width in pixels for rendered points.The *size* must be greater than 0.0 and by default is 1.0. If the size is 4.0, the point is usually drawn as a square, four pixels on a side.

Line Width

To control the width of lines use the command glLineWidth(GLfloat *width*) where *width* is the desired width in pixels for rendered lines.The *width* must be greater than 0.0 and by default is 1.0.

Colour

The most commonly used variant of the command to set RGBA colours *is glColour3f(GLfloat* red*, GLfloat* green*, GLfloat* blue*)* where *red*, *green*, and *blue* are

values between 0.0 and 1.0, inclusive. The value 0.0 corresponds to the minimum amount of that colour while 1.0 corresponds to the maximum amount of that colour. A common correspondence between colour value and displayed colour are listed in the following table:

Table 1: A correspondence between colour-value and displayed colour. The last column of this table indicates the command used for corresponding colour.

Colour Value	Displayed	Command used
0,0,0	**Black**	**glColour3f**(0.0, 0.0, 0.0);
0,0,1	**Blue**	**glColour3f**(0.0, 0.0, 1.0);
0,1,0	**Green**	**glColour3f**(0.0, 1.0, 0.0);
0,1,1	**Cyan**	**glColour3f**(0.0, 1.0, 1.0);
1,0,0	**Red**	**glColour3f**(1.0, 0.0, 0.0);
1,0,1	**Magenta**	**glColour3f**(1.0, 0.0, 1.0);
1,1,0	**Yellow**	**glColour3f**(1.0, 1.0, 0.0);
1,1,1	**White**	**glColour3f**(1.0, 1.0, 1.0);

The pixel value (0,1,1) means that the Red component is "off", but both green and blue are "on". In most displays, the contributions from each component are added together, so (1,1,0) would repersent the adddition of red and green light, that produces a Yellow colour. As expected, equal amount of red, green, and blue, (1,1,1), produce white.

OpenGL Data Types

To become more independent of the platform and programming language, OpenGL provides it's own data types. In the following list you can find a summary of the most important data types and their equivalent in standard C:

Table 2: Command suffixes and argument data types.

OpenGL Data Type	Internal Representation	Defined as C Type	C Literal Suffix
Glbyte	8-bit integer	Signed char	b
Glshort	16-bit integer	Short	s
GLint, GLsizei	32-bit integer	Long	I
Glfloat, GLclampf	32-bit floating point	Float	f
GLdouble, GLclampd	64-bit floating point	Double	d
GLubyte, GLboolean	8-bit unsigned integer	Unsigned char	ub
GLushort	16-bit unsigned integer	Unsigned short	us
GLuint, GLenum, GLbitfield	32-bit unsigned integer	Unsigned long	ui

Thus, All the OpenGL datatypes begin with "GL". For example GLfloat, GLint and so on. There are also many symbolic constants, they all begin with "GL_", like GL_POINTS, GL_LINES, GL_POLYGON etc. To draw such objects in OpenGL, we pass it a list of vertices. The list occurs between the two OpenGL function calls glBegin() and glEnd(). The argument of glBegin() determines which object is drawn. For example, the following command sequence draw a line.

```
glBegin(GL_LINES);        // use constant GL_LINES here
glVertex2i(200,200);   //  Draw 1st vertical line
glVertex2i(200,400);
glEnd();
```

A function using the suffix i expects a 32-bit integer, but your system might translate int as a 16-bit integer. Thus, when you compile your program on a new system, Glint, Glfloat, etc. will be associated with the appropriate C/C++ types. Thus, it is safer to write same code, written above, as a generic function *drawLineInt( )* :

```
Void drawLineInt(Glint x1, Glint y1,Glint x2, Glint y2)
  {
     glBegin(GL_LINES);          // use constant GL_LINES here
     glVertex2i(200,200);    //  Draw 1st vertical line
     glVertex2i(200,400);
     glEnd();
  }
```

Commands Needed to Work With OpenGL

Now, let us see the commands you need in "reshape" and at the beginning of "display".

There are two (ok, in reality there are three) matrices. The current one is controlled by "*glMatrixMode()"*. You can call it either with GL_PROJECTION or GL_MODELVIEW. In Reshape, you specify a new projection matrix. After turning it current, you load an identity matrix into it, that means "deleting" the former projection matrix. Now, you can apply a new field of view by using a glu* command: *gluPerspective*. You pass the y-angle of view, the aspect ratio (the windows witdth / height), and a near and far clipping distance. The objects or parts of objects, which are not between those distances, will not be visible. You should be careful in this process, don't take a too small near clipping distance, otherwise you could get wrong-looking results. After this, you make the modelview matrix currrent and also load an identity matrix.

Display has to clear the colour buffer. Therefore you have to:
1.*glClear(GL_COLOUR_BUFFER_BIT)*.
2.After the initialization of GLUT, you should have defined a clear colour using *glClearColour()*. It takes four arguments, red, green and blue and an alpha value. Set alpha = 0.0.To get clear red, you would call *glClearColour(1.0,0.0,0.0,0.0);*
Drawing the object.
3.Last but not least you define the "viewport", the area on the window where OpenGL shall paint to. Therefore you use *glViewport(0,0,x,y);* with x and y as the size of your window: You get them from the reshape parameters. Here is how to define the function headers of the two routines:
void Reshape (int x, int y);
void Display (void);

A simple OpenGL Program

The following program will open a window and render a triangle in 3-space from a particular viewpoint:
After clearing the colourbuffer, you can pass some data to OpenGL. It is very simple: You specify some vertices, which are used in a certain way. The way is defined by *glBegin()*. Now we will use GL_POLYGON as a parameter which tells OpenGL to use the following vertices as corners of one polygon.
After passing the data to OpenGL, you have to call *glEnd()*.

To provide a vertex , you can call one of some *glVertex*()* routines. * stands for a combination of the number of arguments (two-dimensional, three-dimensional or four-dimensional) + the data type (f for float, d for double and i for int).
To specify a 3d vertex with integer coordinates, you call *glVertex3i()*.

To finish rendering, you must call *glFlush()* at the end of your display-function. Just call glVertex*() three times between *glBegin* and *glEnd*.

To change the current colour in RGB(A)-mode, you call *glColour*()*. Again * stands for the number of arguments (three for RGB and four for RGBA)+ the data type. If you take a floating-point type as parameters, note that 1.0 is full intensity for the colour component. Also note, that colours are blended, when they are different for one or more vertices in the polygon.If you want to add some vertices to our polygon, you must note that OpenGL cannot render concave polygons (for performance reasons).

The following program merely opens a window and draws a rectangle in it.

```
/*****************************************/
/* A Very Simple OpenGL Example!         */
/*****************************************/

#include <windows.h> /* obviously you would need to change this to your native library
                        if you're compiling under unix */
#include <gl\gl.h>
#include<gl\glu.h>
#include <gl\glut.h>

void init(void);
void display(void);

int main (int argc, char **argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowSize(250, 250);
        glutInitWindowPosition(100, 100);
        glutCreateWindow("My First OpenGL Application");
        init();
        glutDisplayFunc(display);
        glutMainLoop();
        return 0;
}

void init(void)
{
        glClearColour(0.0, 0.0, 0.0, 0.0);
        glColour3f(0.0, 0.0, 1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);
}

void display(void)
{
        glClear(GL_COLOUR_BUFFER_BIT);
        glRectf(-5.0, 5.0, 5.0, -5.0);
        glutSwapBuffers();
}
/*************************************************/
```

Now, Let us go through this program step-by-step:

1. First, you called the glutInit function, which provides general initialisation for the GLUT library:

```
int main (int argc, char **argv)
{
        glutInit(&argc, argv);
```

2. Next you initialise the display mode:

```
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

This tells OpenGL that you want a double-buffered window, with RGB colour.

3. The next few lines create the window:

```
        glutInitWindowSize(250, 250);
        glutInitWindowPosition(100, 100);
        glutCreateWindow("My First OpenGL Application");
```

As you can see, with GLUT this is fairly straightforward.

4. Now it is time to initialise OpenGL. Here, 's the code:

```
void init(void)
  {
   glClearColour(0.0, 0.0, 0.0, 0.0); /* set the background (clearing) colour to
                        RGB(0,0,0) -- black */
  glColour3f(0.0, 0.0, 1.0); /*set the foreground colour to blue */
  glMatrixMode(GL_PROJECTION);  /* Initialise the matrix state */
  glLoadIdentity();
  glOrtho(-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);
 }
```

In OpenGL, matrices are used to manage the view. There are two matrix modelprojection and modelview. Projection is used to set up the viewport and clipping boundry, while modelview is used to rotate, translate and scale objects quickly.

Let us take a look at these two lines:

```
glLoadIdentity();
glOrtho(-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);
```

glLoadIdentity() loads the identity matrix into the current matrix state (in this case the projection matrix). You can consider this the resetting matrix. It resets everything back to zero.

5. Next comes the call to glOrtho.

This function sets up a clipping volume.You can think of a clipping volume as a box in which your drawing commands are rendered. As the viewer, we are positioned outside the box, looking in the front. What we see is whatever is inside the box, projected onto the flat surface that is the side.
Anything outside the box is invisible.The glOrtho function creates an orthographic view.

The arguments for glOrtho are as follows:
void glOrtho(double left, double right, double bottom, double top,double near, double far);

6. Now, let us continue with the application:

```
        glutDisplayFunc(display);
        glutMainLoop();
```

The first function sets the function that GLUT will use whenever it needs to update the view. We then call glutMainLoop() which actually runs the program. From this point on, our work is done; GLUT will handle the details of managing the window and calling our painting function to display it.

7. Here is the display function again:

```
void display(void)
{
        glClear(GL_COLOUR_BUFFER_BIT);
        glRectf(-5.0, 5.0, 5.0, -5.0);
        glutSwapBuffers();
}
```

The first thing we do is called glClear with GL_COLOUR_BUFFER_BIT parameter. This will clear the window with the colour we specified earlier using glClearColour. Next, we actually draw our rectangle, using the glRectf function.

8. The next function, glutSwapBuffers(), swaps the back buffer to the screen buffer. In double-buffered mode, the drawing commands do not draw to the screen. Rather, they draw to an offscreen buffer. Once you are done drawing, you copy the entire back buffer to the screen at once, thus, producing smooth animation. In this simple example there is no animation, but without a double buffer even moving the window around the screen will cause the rectangle to flicker.

Now, you can copy this program into an editor, compile it, and run it.

Note: You will have to link with whatever libs your version of OpenGL has provided, as well as the GLUT library.

2.5 INTRODUCTION TO OPENGL PRIMITIVES

Geometric primitives are described by **glBegin()**...**glEnd()** pairs.
glBegin(): Marks the beginning of a vertex-data list that describes a geometric primitive.
glEnd(): Marks the end of a vertex data list.

For example, to draw a triangle you would use the following statements:

```
glBegin(GL_POLYGON);
glVertex2i(...);      // 1st vertex of a Polygon
glVertex2i(...);      // 2nd vertex of a Polygon
glVertex2i(...);      // 3rd vertex of a Polygon
glVertex2i(...);      // 4th vertex of a Polygon
glVertex2i(...);      // 5th vertex of a Polygon
glEnd();
```

If, here you had used GL_POINTS instead of GL_POLYGON, the primitive would have been simply draw the five points. The following is a complete program in VC++ to draw a filled polygon or a set of points, as explained above.

A program in VC++ to draw a filled polygon or a set of points

```
#include <windows.h>   // use as needed for your system
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>
        // ************** myInit() **************
```

```
void myInit(void)
{
        glClearColour(1.0,1.0,1.0,0.0);      // set white background colour
        glColour3f(0.0f, 0.0f, 0.0f);         // setting the drawing colour
        glPointSize(4.0);                    // One 'dot' is a 4 by 4 pixel
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, 800.0, 0.0, 600.0);
}
  // ********************** myDisplay() ***********
void myDisplay(void)
{
glClear(GL_COLOUR_BUFFER_BIT);         // clear the screen
glBegin(GL_POLYGON);
glVertex2i(100,100);       // Draw these five points as a polygon vertices.
    glVertex2i(100,300);
    glVertex2i(400,300);
    glVertex2i(600,150);
    glVertex2i(400,100);
    glEnd();
    glFlush();                         // send all output to the display
}
   // ****************** main function ********************
void main(int argc, char** argv)
{
glutInit(&argc, argv);  // initialise the toolkit
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);  // set display mode
glutInitWindowSize(800,600);        // set window size
glutInitWindowPosition(100,100);   // set window position on your screen
glutCreateWindow("OpenGL Window");   // Open the screen window
glutDisplayFunc(myDisplay);            // register redraw function
myInit();
glutMainLoop();                        // go into a perpetual loop
}
```

If, here you had used GL_POINTS instead of GL_POLYGON as an argument to the glBegin(), the primitive would have been simply draw the five points. The results of running the program appear in following figure:

Figure (a) GL_POLYGON **Figure (b) GL_POINTS**

Figure 7: Drawing a Polygon or a Set of Points

The argument to glBegin() describes what type of primitive you want to draw. There are ten possible arguments:

Table 3: An OpenGL Geometric Primitives Names and Meanings.

Value	Meaning
GL_POINTS	Individual points
GL_LINES	Pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	Series of connected line segments
GL_LINE_LOOP	Same as above, with a segment added between last and first vertices
GL_TRIANGLES	Triples of vertices interpreted as triangle
GL_TRIANGLE_STRIP	Linked strip of triangle
GL_TRIANGLE_FAN	Linked fan of triangles
GL_QUADS	Quadruples of vertices interpreted as four- sided polygons
GL_QUAD_STRIP	Linked strip of quadrilaterals
GL_POLYGON	Boundary of a simple, convex polygon

glEnd() indicates when you wish to stop drawing that type of primitive. This is necessary because there can be an arbitrary number of vertices for a type of primitive. Assume that n vertices (v0,v1,v2,v3,…….vn-1) are described between a **glbegin()** and **glEnd().**

1) **GL_POINTS**: Takes the listed vertices one at a time and draws a separate point for each of the n vertices.

2) **GL_LINES**: Takes the listed vertices two at a time and draws a separate line for each.

3) **GL_LINE_STRIP**: Draw a series of lines based on pairs of vertices: v0,v1, then v1,v2 then v2,v3, and so on, finally drawing the segment from vn-2 to vn-1. Thus, a total of (n-1) line segments are drawn. Thus to draw a line 'n' should be >1.

4) **GL_LINE_LOOPS:** Same as GL_LINE_STRIP, but it also connect the last vertex v_{n-1} with the first vertex v_0 to construct a loop.

5) **GL_POLYGON:** Draws a polygon based on the given list of points v0,v1,..vn-1 as vertices. 'n' must be at least 3, or nothing is drawn.

6) **GL_QUADS**: Draws a series of quadrilaterals (four-sided polygon) using vertices v0,v1,v2,v3, then v4,v5,v6,v7, and so on. If 'n' is not a multiple of 4, the final one, two, or three vertices are ignored.

7) **GL_QUAD_STRIP**: Draws a series of quadrilaterals (four-sided polygon) beginning with vertices: v_0, v_1, v_3, v_2 then v_2, v_3, v_5, v_4, then v_4, v_5, v_7, v_6, etc. 'n' must be at least 4 before anything is drawn. If n is odd, the final vertex is ignored.

8) **GL_TRIANGLES**: Takes the listed vertices three at a time and draws a separate triangle for each. If 'n' is not a multiple of 3, the final one, or two vertices are ignored.

9) **GL_TRIANGLE_STRIP**: Draws a series of triangles (three-sided polygon) based on triplets of vertices: first v_0, v_1, v_2 then v_2, v_1, v_3, then v_2, v_3, v_4, etc. (in an order such that all triangles are "traversed" in the same way, e.g. , counterclockwise). 'n' must be at least 3, or nothing is drawn.

10) **GL_TRIANGLE_FAN:** Draws a series of connected triangles based on triplets of vertices: first v_0, v_1, v_2, then v_0, v_2, v_3, then v_0, v_3, v_4, etc.

Each geometric object is described by a set of vertices and the type of the primitive to be drawn. The primitive type determines whether and how the vertices are connected.

Specifying Vertices

With OpenGL, all geometric objects are ultimately described as an ordered set of vertices. The command *glVertex*()* is used to specify a vertex.
void **glVertex**{234}{sifd}[v](TYPE coords);

This command is used to specifying a vertex for describing a geometric object. You can supply upto 4 coordinates (x,y,z,w) for a particular vertex or at least two (x,y) by selecting the appropriate version of the command. If you use a version that does not explicitly specify z or w, z is understood to be 0 and w as 1. This command are effective only between **glBegin()** and **glEnd()**.

Here are some example of uses of *glVertex*()*:

```
glVertex2s(1, 2);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 2.0, -4.2, 2.0);

GLdouble vector[3] = {3.0, 10.0, 2033.0};
glVertex3dv(vector);
```

The first example represents a vertex with 3-D coordinates (1,2,0).(Remember that, the z-coordinate is understood to be 0).
The coordinates in the second example are (0.0, 0.0, 3.1415926535898) (double-precision floating-point number).

The third example represents a vertex with 3-D coordinates (1.15,1.0,-2.1) as a homogeneous coordinate.
(Note that x,y, and z-coordinates are divided by w, in a homogeneous coordinate system).
In the final example, dvect is a pointer to an array of three double-precision floating point numbers.

It is necessary to call the command ***glFlush()*** to ensure that all previously issued commands are executed. ***glFlush()*** is generally called at the end of a sequence of drawing commands to ensure all objects in the scene are drawn.

The following example illustrates the different results of some of the primitive types applied to the set of vertices.

Making Point Drawing: To draw a point we use GL_POINTS as the argument to GLBegin( ). Thus to draw a two points, we use the following commands:

```
glBegin(GL_POINTS)
{
  glVertex2d(100,130);
 glVertex2d(100,200);
glEnd(); }
```

Making Line Drawing: To draw a line we use GL_LINES as the argument to GLBegin( ). Thus to draw a four lines, two horizontal and two vertival (tic-tac-toe, we use the following commands:

```
glBegin(GL_LINES)
  glVertex2i(20,40);    // 1st horizontal line
 glVertex2d(80,40);
```

```
  glVertex2i(40,20);    // 1st vertical line
  glVertex2d(40,80);
// four more calls to glVertex2i(…), for the other two lines
glEnd();
glFlush();
```

2.6 EXAMPLE PROGRAMS

```
/*****************************************/
/* Example Program - Drawing in 3D       */
/*****************************************/
#include <windows.h>
#include <gl\gl.h>
#include <gl\glut.h>
void init(void);
void display(void);
void keyboard(unsigned char, int, int);

int main (int argc, char **argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowSize(600, 600);
        glutInitWindowPosition(50, 50);
        glutCreateWindow("A 3D Object");
        init();
        glutDisplayFunc(display);
        glutKeyboardFunc(keyboard);  /* set keyboard handler */
        glutMainLoop();
        return 0;
}

void init(void)
{
        glClearColour(0.0, 0.0, 0.0, 0.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(-15.0, 15.0, -15.0, 15.0, -15.0, 15.0);
}

void display(void)
{
        glClear(GL_COLOUR_BUFFER_BIT);
        glBegin(GL_QUADS);
                glColour3f(0.0, 0.0, 1.0);    /* center square */
                glVertex3f(-3.0, -3.0, 0.0);
                glVertex3f(3.0, -3.0, 0.0);
                glVertex3f(3.0, 3.0, 0.0);
                glVertex3f(-3.0, 3.0, 0.0);
        glEnd();
        glBegin(GL_TRIANGLES);
                glColour3f(1.0, 0.0, 0.0);  /* now draw the four triangles */
                glVertex3f(0.0, 6.5, 0.0);
                glColour3f(0.0, 0.0, 0.9f);
                glVertex3f(-3.0, 3.0, 0.0);
                glVertex3f(3.0, 3.0, 0.0);
```

```
                glColour3f(0.0, 0.0, 0.9f);
                glVertex3f(-3.0, -3.0, 0.0);
                glVertex3f(-3.0, 3.0, 0.0);
                glColour3f(1.0, 0.0, 0.0);
                glVertex3f(-6.5, 0.0, 0.0);

                glColour3f(1.0, 0.0, 0.0);
                glVertex3f(0.0, -6.5, 0.0);
                glColour3f(0.0, 0.0, 0.9f);
                glVertex3f(3.0, -3.0, 0.0);
                glVertex3f(-3.0, -3.0, 0.0);

                glColour3f(1.0, 0.0, 0.0);
                glVertex3f(6.5, 0.0, 0.0);
                glColour3f(0.0, 0.0, 0.9f);
                glVertex3f(3.0, 3.0, 0.0);
                glVertex3f(3.0, -3.0, 0.0);
        glEnd();
        glutSwapBuffers();
}

void keyboard(unsigned char key, int x, int y)
{
        /* this is the keyboard event handler
           the x and y parameters are the mouse
           coordintes when the key was struck */
        switch (key)
        {
        case 'u':
        case 'U':
                glRotatef(3.0, 1.0, 0.0, 0.0); /* rotate up */
                break;
        case 'd':
        case 'D':
                glRotatef(-3.0, 1.0, 0.0, 0.0); /* rotate down */
                break;
        case 'l':
        case 'L':
                glRotatef(3.0, 0.0, 1.0, 0.0); /* rotate left */
                break;
        case 'r':
        case 'R':
                glRotatef(-3.0, 0.0, 1.0, 0.0); /* rotate right */
        }
        display(); /* repaint the window */
}
/*****************************************************************
**/
```

Note that, the glRotate function is used to rotate the view when the u, d, l, or r keys are pressed. Output of this program is shown in *Figure 8* and *Figure 9*.

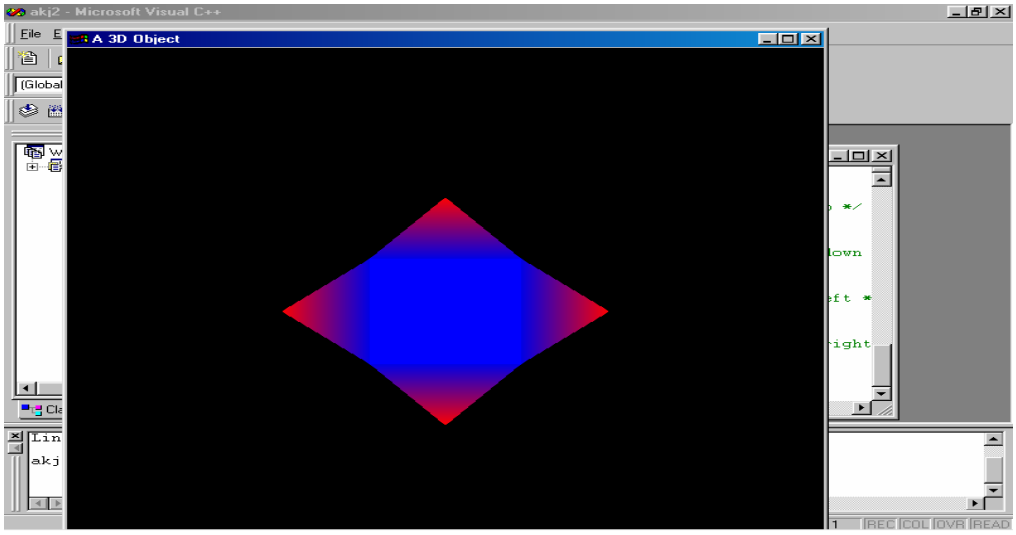

Figure 8: The Output of the Program, which draw a centered square with a triangle on each side of a square

when rotate left using keyboard letter 'L'

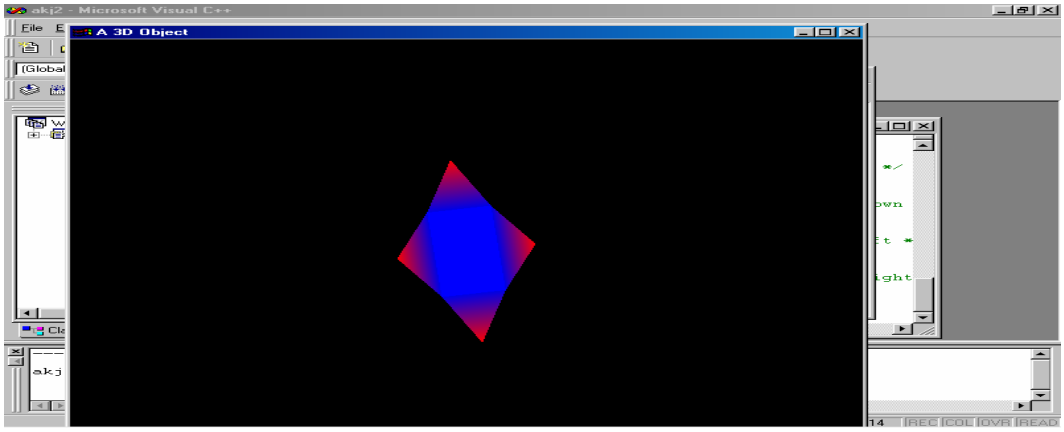

Figure 9: The Output of the Program, when rotate left using keyboard letter 'L'

More about Displaying Points, Lines and Polygons

By default, a point is drawn as a single pixel on the screen, a line is drawn solid with 1 pixel wide, and polygon is drawn solidly filled in. Here you will learn how to change these default display mode.

Displaying Points:

To control the size of a rendered point, you can use ***glPointSize()*** and supply the desired size in pixels as the argument.
Void **glPointSize()**(Glfloat *size*);

This command sets the width in pixels for rendered points; *size* must be greater than 0.0 and by default is 1.0.

The pixels on the screen that are drawn for various point widths depends on whether antialiasing is enabled (antialiasing is a technique for smoothing points and lines as they are rendered). If antialiasing is disabled (by default), fractional widths are rounded to integer widths, and a screen-aligned square region of pixels is drawn. Thus, if the width is 1.0, the square is 1 pixel by 1 pixel. If the width is 2.0, the square is 2 pixels by 2 pixels, and so on.

Displaying Lines:
With OpenGL, you can specify lines with different widths. That is, lines can be drawn with alternating dots and dashes, and so on. This is known as stippled (dotted or dashed) lines.
Void **glLineWidth()**(Glfloat *width*);

This command sets the width in pixels for rendered lines; *width* must be greater than 0.0 and by default is 1.0.

The actual rendering of lines is affected if either antialiasing or multisampling is enabled. Without antialiasing, widths of 1, 2, and 3 draw lines 1, 2, and 3 pixels wide. With aliasing enabled, noninteger line widths are possible.

You can obtain the range of supported aliased line width by using GL_ALIASED_WIDTH_RANGE with **glGetFloatv().** To determine the supported minimum and maximum sizes of antialiased line widths, and what granularity your implementation supports, call **glGetFloatv(),** with
GL_SMOOTH_LINE_WIDTH_RANGE and
GL_SMOOTH_LINE_WIDTH_GRANULARITY

Stippled Lines:
Suppose one wants a line to be drawn with a dot-dash pattern (stippled lines). To make stippled lines, you use the command **glLineStipple()** to define pattern.
Void **glLineStipple(**Glint *factor,* Glushort *pattern*);

This command sets the current stippling pattern for lines. The *pattern* argument is a 16-bit series of 0's and 1's, and it's repeated as necessary to stipple a given line. A 1 indicates that drawing occurs, and a 0 indicates that drawing does not occur, on a pixel-by-pixel basis. A pattern is scanned from the low-order bit up to high-order bit. The pattern can be stretched out by using *factor,* which multiplies each subseries of consecutive 1s and 0s. Thus, if three consecutive 1s appear in the pattern, they are stretched to six if *factor* is 2. *Factor* lies between 1 and 256. Line stippling must be enabled by passing GL_LINE_STIPPLE to **glEnable();** and it is disabled by passing the same argument to **glDisable().**

For example, consider the following sequence of command:
glLineStiple(2,0x3F07);
glEnable(GL_LINE_STIPPLE);

Here, pattern is given in Hexadecimal notation, which is equivalent to 0011111100000111 in binary. The variable *factor* specify how much to "enlarge" *pattern*. Each bit in the pattern is repeated *factor* times (remember low-order bit is used first for drawing). Thus, the pattern 0x3f07 with factor 2 yields 00001111111111110000000000111111. That is a line would be drawn with 6 pixel on; then 10 pixel off; 12 on and 4 off. The following table shows a results of stippled line for various patterns and factors.

Table 4:Stippled lines for a given pattern

PATTERN	FACTOR	RESULT
0x3F07	2	------ ----------
0xAAAA	1	- - - - - - - - - - - - - - - - - - - -
0xAAAA	2	— — — — — — — — — —

In drawing a stippled line with *glBegin(GL_LINE_STRIP); glVertex*(); glVertex*(); glVertex*();..... glEnd();* , the pattern continues from the end of one line segment to the beginning of the next, until glEnd() is executed, which reset the pattern.

The following program illustrates the result of drawing for a different stippled pattern and line widths.

```
// ************************************************************
// Program:3: To draw a stippled line with different Stippled pattern and line width
// ************************************************************

#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>

#define drawOneLine(x1,y1,x2,y2)  glBegin(GL_LINES);  \
        glVertex2f((x1),(y1)); glVertex2f((x2),(y2)); glEnd();

void init(void)
{
        glClearColour(0.0,0.0,0.0,0.0);
        glShadeModel(GL_FLAT);
}

void display(void)
{
        int i;
        glClear(GL_COLOUR_BUFFER_BIT);
        glColour3f(1.0,1.0,1.0);
        glEnable(GL_LINE_STIPPLE);
       //** 1st row-line in the output,3 line, each with different stipple
        glLineStipple(1,0x00FF);              // ****dashed line
        drawOneLine(50.0,125.0,150.0,125.0);
        glLineStipple(1,0x0101);              // *** dotted line
        drawOneLine(150.0,125.0,250.0,125.0);
        glLineStipple(1,0xAAAA);              // *** dash/dot/dash
        drawOneLine(250.0,125.0,350.0,125.0);
       //** 2nd line in the output,3 wide line, each with different stipple
        glLineWidth(8.0);
        glLineStipple(1,0x00FF);          // ****dashed line
        drawOneLine(50.0,100.0,150.0,100.0);
        glLineStipple(1,0x0101);          // *** dotted line
        drawOneLine(150.0,100.0,250.0,100.0);
        glLineStipple(1,0xAAAA);        // *** dash/dot/dash
        drawOneLine(250.0,100.0,350.0,100.0);
        glDisable(GL_LINE_STIPPLE);
        glFlush();
}

void reshape(int w, int h)
{
        glViewport(0,0,(GLsizei) w, (GLsizei) h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,(GLdouble) w, 0.0, (GLdouble) h);
```

```
}

int main(int argc, char** argv)
{
        glutInit(&argc, argv);  // initialise the toolkit
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);  // set display mode
        glutInitWindowSize(400,150);        // set window size
        glutInitWindowPosition(100,100);   // set window position on your screen
        glutCreateWindow(argv[0]);// Open the screen window
        init();
       glutDisplayFunc(display);            // register redraw function
       glutReshapeFunc(reshape);
        glutMainLoop();                     // go into a perpetual loop
   return 0;
}
```

The result of running this program-3 appears in the *Figure 10:*

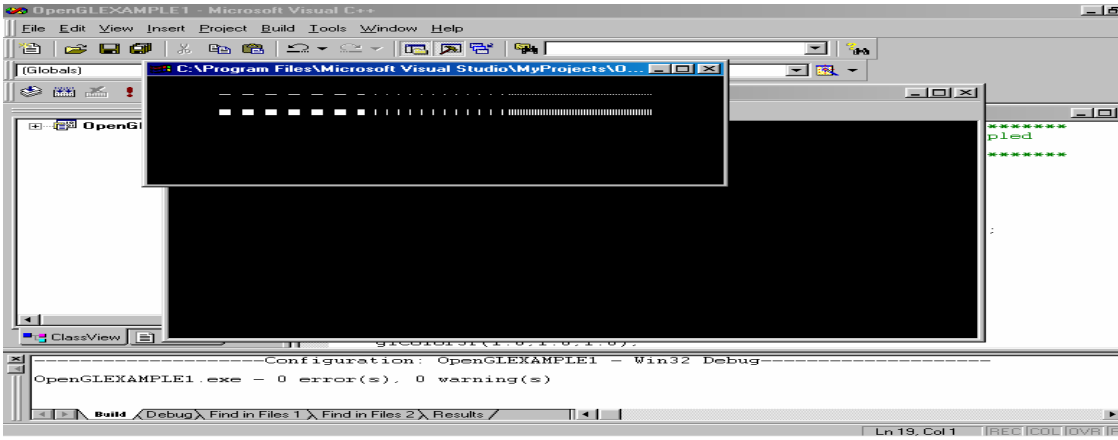

Figure 10: The Output of the Program as a Wide Stippled Lines.

2.7 IMPLEMENTATION OF LINE CLIPPING ALGORITHM

Clipping is used in Computer Graphics to display only those portions of a picture that are within the window area. Everything outside of the window is discarded. Generally, any procedure that identifies those portions of a picture that are either inside or outside of a specified region of space is referred to as a clipping algorithm or simply *Clipping*.

Line Clipping

A line-clipping algorithm involves several parts:

1) We test a given line segment to determine whether it lies completely inside the clipping window. If it does not, we try to determine whether it lies completely outside the window.
2) Finally, if we cannot identify a line as completely inside or outside the window, we must perform intersection calculations with one/more clipping boundaries. We process the "inside-outside" tests by checking the line endpoints.

3) If both endpoints of a line is inside the clipping boundaries, then this line is totally visible (i.e. line P_1P_2).
4) If both endpoints of a line are outside the clipping boundaries, then this line may be totally invisible (i.e. line P_3P_4).
5) All other lines cross one/more clipping boundaries, and may require calculation of multiple intersection points.

Cohen-Sutherland Line Clipping Algorithm

To test, whether a given line is totally visible or totally invisible, this technique divides the regions including the window into 9 regions and assign a 4-bit region code to indicate which of the 9 regions contain the end points of a line. The bits are set to 1(i.e., TRUE) or 0 (i.e., FALSE), starting from the Left-most-bit, based on the following scheme:

1st-bit set→ if the end point is to the TOP of the window.
2nd-bit set→ if the end point is to the BOTTOM of the window.
3rd-bit set→ if the end point is to the RIGHT of the window.
4th-bit set→ if the end point is to the LEFT of the window.
Otherwise the bit is set to 0.

1001	1000	1010
0001	Window 0000	0010
0101	0100	0110

Figure 11 :Inside-Outside codes for a point

Now, we use these endpoints code to determine whether the line segment lies completely inside the clipping window or outside the window edge. In this algorithm, we divide the line clipping process into two phases:

1) Identify those lines which intersect the clipping window and so need to be clipped, and
2) Perform the clipping.

All lines fall into one of the following clipping categories:

Case-1: (*Visible*):- If both 4-bit codes of the endpoints are (0000), then the line lies completely inside the clip window (i.e. line P_1P_2).

Case-2: (*invisible*):- If the bitwise logical AND of the endpoints code is not equal to (0000), then the line lies completely outside the clip window, so are trivially rejected (i.e. line P_3P_4).

Case-3:(*Clipping candidate*):- If the bitwise logical AND of the endpoints code is equal to (0000), then the line is to be clipped with one/more window edges (i.e. line PQ).

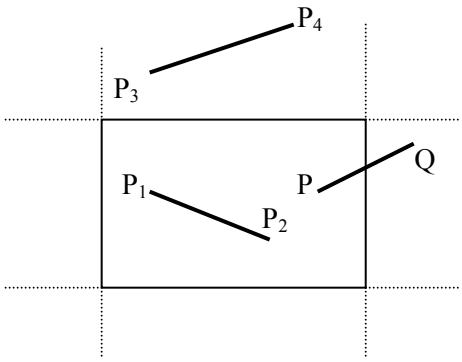

Figure 12: The line segments having end points P_1P_2 (i.e *Visible*), P_3P_4 (i.e. *invisible*) and PQ(i.e.*Clipping candidate*) against a rectangular clipping window.

Following *Figure-11* shows how "inside-outside" test can be carried out, to check how a point 'P' is positioned relative to the window. A single 8-bit word **code** is used: four of its bits are used to capture the four pieces of information. Point P is tested against each window boundary, if P lies outside the boundary, the proper bit of **code** is set to 1 to represent TRUe. The first **code** is initialised to 0, and then its individual bits are set as appropriate using a bitwise OR operation. The values 8, 4, 2, and 1 are simple masks. For instance, since 8 is 00001000 in binary, bitwise OR-ing a value with 8, sets the fourth bit from the right end to 1.

```
/*************************************
  unsigned char code=0;      /* Initially all bits are set to '0'
    ……….
   If(P.x< Window.l) code |= 8;     // set bit 3
  If(P.y< Window.b) code |= 4;     // set bit 2
   If(P.x< Window.r) code |= 2;     // set bit 1
   If(P.y< Window.t) code |= 1;     // set bit 0
/*************************************************
```

The following is a pseudocode for Cohen-Sutherland line clipping algorithm. The point 2 represents 2D point and RealRect holds a rectangular window. (See Figure13).

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

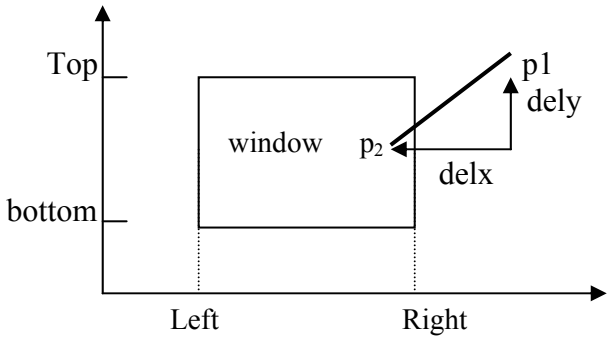

Figure 13: Clipping a Segment Against an Edge

```
int  clipSegment (Point2& pl , Point2& p2 , RealRect W)
{
   do{
         if (trivial accept) return   1 ;  //   some portion survives
       if (trivial reject) return   0 ;  //   no portion survives

         if (pl is out side)
          {
```

```
            if (p1 is to the left) chop against the left edge
              else if (pl is to the right) chop against the right edge
              else if  (pl is below) chop against the bottom edge
              else if  (pl is above) chop against the top edge
           }
         else            //  p2  is   outside
          {
            if (p2 is to the left) chop against the left edge
              else if (p2 is to the right) chop against the right edge
              else if  (p2 is below) chop against the bottom edge
              else if  (p2 is above) chop against the top edge
           }
      }while (1);
  }
/*************************************************
```

Here, each time the do loop is executed, the code for each end point is recomputed and tested. When trivial acceptance and trivial rejection fail, the algorithm tests whether p1 is outside the window, and if it is, that end of the segment is clipped to a window boundary. If p1 is inside the window, then p2 must be outside, so p2 is clipped to a window boundary.

2.8 OPENGL FUNCTION FOR 3-D TRANSFORMATIONS AND VIEWING

In this section, we discuss some OpenGL commands that you might find useful as you specify desired transformations. Before using a Transformation command (glTranslate(), glRatate(),etc.), you need to state whether you want to modify the modelview or projection matrix.

Void glMatrixMode(GLenum *mode*)
This command is used to change the value of the **GL_MATRIX_MODE** state variable. The possible values for the mode argument are **GL_MODELVIEW**, **GL_PROJECTION**, and **GL_TEXTURE**. When *glMatrixMode()* is called, all subsequent transformation commands affect the specified matrix.

> **Note** that, only one matix can be modified at a time. By default, the modelview matrix is the one that is modifiable, and all three matrices contain the identity matrix.

You use the glLoadIdentity() command to clear the currently modifiable matrix for future transformation commands, as these commands modify the current matrix. Typically, you always call this command before specifying projection or viewing transformations.
void **glLoadIdentity(**void**)**
This command is used to set the currently modifiable matrix to the 4x4 identity matrix.

If, you want to specify any matrix to be loaded as the current matrix, you can use:
glLoadMatrix*() or **glLoadTransposeMatrix*()**
Similarly, you can use **glMultMatrix*()** or **glMultTransposeMatrix*()** to multiply the current matrix by the matrix passed in as an argument.

Void **glLoadMatrix{**fd**}(**const TYPE*m**);**
This command sets the 16 values of the current matrix to those specified by m.

Void **glMultMatrix{**fd**}(** const TYPE*m**)**
Multiplies the matrix specified by the 16 values pointed to, by m by the current matrix and stores the result as the current matrix.

In OpenGL, all matrix multiplication occurs as follows: if **C** is the current matrix and **M** is the matrix specified by **glMultMatrix*(),** then after multiplication, the final matrix is always **C.M**.
The argument for **glLoadMatrix*()** and **glMulMatrix*()** is a vector of 16 values (m1,m2,…,m16) that specifies a matrix **M** stored in column-major order as follows:

$$M= \begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix}$$

If, you are programming in C and you declare a matrix as m[4][4], then the element m[i][j] reperesents ith column and jth row of the OpenGL transformation matrix. One way to avoid confusion between column and row is to declare your matrix as m[16]. You can also avoid this confusion by calling the OpenGL command **glLoadTransposeMatrix*()** and **glMultTransposeMatrix*(),** which use row-major(the standard C convension) matrices as arguments.
void **glLoadTransposeMatrix**{fd}(const TYPE *m);
This command sets the 16 values of the current matrix to those specified by m, whose values are stored in row-major order. **glLoadTransposeMatrix*(m)** has the same effect as **glLoadMatrix***(m^T).

void **glMultTransposeMatrix**{fd}(const TYPE *m);
Multiplies the matrix specified by the 16 values pointed to by m by the current matrix and stores the result as the current matrix. **glMultTransposeMatrix*(m)** has the same effect as **glMultMatrix***(m^T).

There are 3 OpenGL commands for modeling transformations: **glTranslate*()**, **glRotate*()**, and **glScale*()**. All these commands are equivalent to producing an appropriate translation, rotation, or scaling matrix, and then calling glMultMatrix*() with that matrix as the argument.

void **glTranslate**{fd}(TYPE x, TYPE y, TYPE z);
This command multiplies the current matrix by a matrix that moves (translates) an object by the given *x*, *y*, and *z* values (or moves the local coordinate system by the same amounts). Note that **glTranslatef**(0.0,0.0,0.0) implies an identity operation- that is , it has no effect on an object or its local coordinate system.

void **glRotate**{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
This command multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counter clockwise direction about the ray from the origin through the point (*x*, *y*, *z*). The *angle* parameter specifies the angle of rotation in degrees. For example: **glRotatef**(45.0,0.0,0.0,1.0) rotates a given object by 45 degrees about the z-axis. And if the angle amount is zero, the **glRotate*()** command has no effect.

void **glScale**{fd}(TYPE x, TYPE y, TYPE z);
This command multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each *x*, *y*, and *z* coordinate of every point in the object is multiplied by the corresponding argument *x*, *y*, or *z*. With the local coordinate system approach, the local coordinate axes are stretched, shrunk, or reflected by the *x*, *y*, and *z* factors, and the associated object is stretched by them.

Out of these three modeling transformation, only glScale*() changes the size of an object: scaling with value greater than 1.0 stretches an object, and the value less than 1.0 shrinks it. Scaling with a –1.0 value reflects an object across the axis. **glScale**(1.0,1.0,1.0) indicates an identity operation.

Viewport Transformation

Viewport transformation is analogous to choosing the size of the developed photograph. By default, OpenGL sets the viewport to be the entire pixel rectangle of the initial program window. The command **glViewport** (GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*) is used to change the size of the drawing region. The *x* and *y* parameters specify the upper left corner of the viewport, and *width* and *height* are the size of the viewport rectangle. This command is often used for the function to handle window reshape events. By default the initial viewport values are (0,0,winWidth, winHeight), where winWidth and winHeight specify the size of the window.

Projection Transformation:

Before using any of the transformation command, you have to call
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
so that the commands affect the projection matrix, rather than the model-view matrix, so that you can avoid compound projection transformations. The purpose of the projection transformation is to defrine a *viewing volume*, which is used in two ways. The viewing volume determines how an object is projected onto the screen (by using a perspective or an orthographic projection), and it defines which objects or portions of objects are clipped out of the final image.

Perspective Projection:

In perspective projections, the further an object is from the camera, the smallest looks and the nearer object looks bigger. This occurs because the viewing volume for a perspective projection is a frustum of a pyramid. The command to define a frustum, **glFrustum(),** calculates a matrix that accomplishes perspective projection and multiplies the current projection matrix (typically the identity matrix) by it.

Void **glFrustum**(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, Gldouble
near, Gldouble far);
This command creates a matrix for a perspective-view frustum and multiplies the current matrix by it. The frustum's viewing volume is defined by the parameters: (left, bottom, -near) and (right, top, -near) specifies the (x,y,z) coordinates of the lower-left and upper-right corners, respectively of the near clipping plane; near and far give the distances from the viewpoint to the near and far clipping planes. Both distances must be positive.

Orthographic Projection

With an orthographic projection, the viewing volume is shaped like a box. Unlike perspective projection, the size of the viewing volume does not change from one end to the other. Therefore, distance from the "camera" does not affect how large an object appears.
The command to create an orthographic viewing volume is
void **glOrtho**(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far***)**.
The arguments *left* and *right* specify the coordinates for the left and right vertical clipping planes. The arguments *bottom* and *top* specifies the coordinates for the

bottom and top horizontal clipping planes. The arguments *near* and *far* specifies the distances to the nearer and farther depth clipping planes. These distances are negative if the plane is to be behind the viewer.

2.9 IMPLEMENTATION OF RAY-TRACING ALGORITHM

In the very simplest terms, ray tracing is a method for producing views of a virtual 3-dimensional scene on a computer. In ray tracing, the light path is traced from the pixel (screen) to the surface (scene) to determine the contributions to the pixel intensity due to reflection and transmission of light ray. A ray tracing is used to obtain global, specular reflection and transmission effects. Pixel rays are traced through a scene moving from object to object while accumulating intensity contributions.

In ray tracing, a ray of light is traced in a backwards direction. That is, we start from the eye or camera and traces the ray through a pixel in the image plane into the scene and determines what it hits. The pixel is then set to the colour values returned by the ray. Each ray that hits an object can spawn other rays (reflected ray and refracted ray), as shown in *Figure 14*.

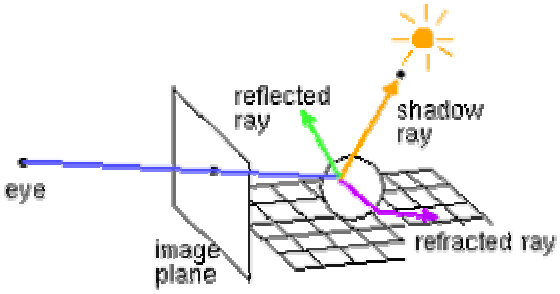

Figure 14: Ray Tracing Involving Reflection and Transmissions.

One of the main problems in the rendering of 3-D scenes is the elimination of hidden surfaces, i.e., surfaces that are not visible from the position of the eye. This problem is tackled in various ways. For example, in the z-buffer method, a depth value is associated with each pixel on the screen. If the depth of the point under consideration from the view plane (the projection screen) is less than the stored depth at that pixel, the pixel is assigned the colour of the point and the depth value is updated to the new value. The process continues in this way.

Ray tracing is an extension of, ray-casting, a common hidden-surface removal method. It is not only looking for the visible surface for each pixel but also by collecting intensity contributions as the ray intersects the surfaces, as shown in *Figure 14*. It tries to mimic actual physical effects associated with the propagation of light.

In ray casting, a ray of light is followed from the eye position through the point on the view plane corresponding to the pixel. For each surface in the scene, the distance the ray must travel before it intersects that surface is calculated, using elementary or advanced mathematical methods. The surface with the shortest distance is the nearest one, therefore, it blocks the others and is the visible surface. Ray casting is therefore, a hidden surface removal method. Here, we will discuss the basic ray tracing algorithm, which is mainly concern with visible surface detection, shadow effects, transparency and multiple light source elimination.

Basic Ray-tracing Algorithm

We first set up the coordinate system, called reference frame with pixel positions specified in the xy-plane, which is screen area centered on viewing coordinate system origin. A light ray from the surface point should pass through the pixel position in the projection plane to reach the centre of the projection for that point to be visible. Since, we do not know which ray from the surface is visible, we trace a light path backwards from the pixel to the surface (or scene) to determine the contributions to the pixel intensity due to reflection and transmission of light ray.

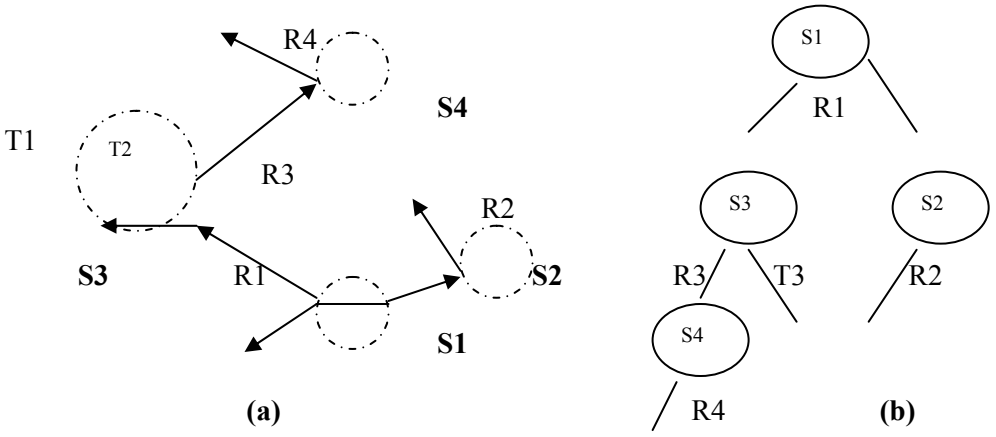

Figure 15: (a) Reflection R and Refraction T path through a scene (b) Binary Tracing tree

The basic ray-tracing algorithm is concerned with one ray per pixel. At the first level, we need to test each surface in the scene to determine whether it is intersected by the ray or not. If the surface is intersected, we calculate the distance from the pixel to the surface-intersection point. The surface yielding the smallest intersection distance is identified as the visible surface for that pixel. The ray can be reflected off the visible surface along a specular path (if the angle of reflection is equal to the angle of incidence). If the surface is transparent, the ray can be transmitted through the surface in the direction of refraction. Reflection and refraction rays are referred to as secondary rays.

The above procedure is repeated for each secondary ray. At the second level, objects are tested for intersection with the secondary ray and the nearest surface along a secondary ray path is used to produce the next level of reflection refraction paths.

As the ray from a pixel moves through the scene, the intersected surfaces at each level are added to form a binary tracing tree, as shown in *Figure15b*. In this *Figure,* the left branches indicate the reflection paths and the right branches indicate the transmission paths. A path in the tree is terminated if, it reaches the maximum depth value of the tree set by the user or if, the ray strikes a light source. The intensity to be assigned to a pixel is determined by combining the intensity contributions starting at the bottom (terminal nodes) of its ray-tracing tree. Surface intensity calculated at each node in the tree is attenuated by the distance of the node from its parent node (surface at the next high level) and is added to the parent surface. Pixel intensity is the sum of the attenuated intensities at the root node of the binary ray-tracing tree. If, the pixel does not intersect any surface, then the pixel is assigned to the background intensity. If, a pixel intersects a non-reflecting light source, the pixel is assigned the intensity of the source.

2.10 INTRODUCTION TO VRML

VRML stand for Virtual Reality Modeling Language (usually pronounced as **vermal**) is a standard file format for representing 3-dimensional (3D) interactive vector graphics, which is particularly designed for the World Wide Web. That is VRML is a file format that defines the layout and content of a 3Dworld with links to more information.

➢ It is a simple text language for describing 3-D shapes, colours, textures, and sounds to produce interactive environments for "virtual world". It is simply a scene description language that describes how three-dimensional environments are represented on the Web.

➢ Unlike other programming languages such as C/C++, VRML does not have to be compiled and run. Simply VRML files is parsed and displayed on the user's monitor. The creation of VRML files is much simpler than programming.

➢ VRML is an interpreted language that is command written in text are interpreted by the browser and displayed on the user's monitor.

Using VRML

➢ You can get VRML Software from the VRML site: **http:// vrml.sdsc.edu** which maintains up-to-date information and links for:

Table 5: Various information on the VRML site

Browser software	Sound libraries
World builder software	Object libraries
File translators	Specifications
Image editors	Tutorials
Java authoring tools	Books
Texture libraries	*and more...*

• First, you have to obtain a tool for VRML. There are freely downloadable versions of products such as Caligari worldSpace, Intervista WorldView and TGS WebSpace. Some allow you to browse in 3D, while others allow for various levels of 3D creation and VRML authoring.

• After you have installed and configured your VRML application, you can load a VRML file the same way you access an HTML file: either by clicking on a link or typing a URL and hitting return. If, you typed the URL into your VRML application, then the file will be loaded. Well-structured VRML files will allow your VRML browser to load the file in pieces, which has the advantage that you can start exploring right away while the browser fetches more detailed objects and those that are not currently in your view.

HTML and VRML

As we know WWW is based on HTTP (Hyper Text Transfer Protocol). The HTML (Hyper Text Markup Language) is used to describe how the text, images, sound and video are represented across different types of computers that are connected to the Internet. HTML allows text and graphics to be displayed together using a two dimensional pages (X and Y planes only). Whereas VRML is a format that describes how three-dimensional environments can be explored and created on the www. Thus,

- VRML is 3D (i.e., using X, Y and Z space) whereas HTML is 2D (i.e., using X and Y planes only). Since, 2D is a subset of 3D, any 2D object can be easily represented in 3D environment. Thus, VRML allows 2D home pages to expand into 3D home worlds.

- VRML allows much more interaction than HTML. When viewing two-dimensional home pages, your options are basically limited to jumping from page to page and looking at images from a fixed, pre-determined perspective. When visiting VRML worlds, however, you can freely choose the perspective from which to view the world.

VRML FILE FORMATE

VRML is a text file format where, e.g., *vertices* and *edges* for a 3D polygon can be specified along with the surface colour, image-mapped textures, shininess, transparency, and so on. URLs can be associated with graphical components so that a web browser might fetch a web-page or a new VRML file from the Internet when the user clicks on the specific graphical component.

- VRML text files use a **.wrl** extension. Text files format may often be compressed using **gzip** so that they transfer over the Internet more quickly.
- You can view VRML files using a VRML browser:
  - A VRML helper-application, and
  - a VRML plug-in to an HTML browser.
- You can view VRML files from your local hard disk, or from the Internet
- You can construct VRML files using:
  - A text editor,
  - a world builder application
  - a 3D modeler and format translator, and
  - a shape generator (like, a Perl script).

- Advantages of using text editor
  - No new software to be bought
  - access to all VRML features, and
  - detailed control of world efficiency

- Disadvantages of using text editor:
  - Hard to author complex 3D shapes, and
  - requires knowledge of VRML syntax

- Advantages of using world builder:
  - Easy 3-D drawing and animating user interface, and
  - little need to learn VRML syntax

- Disadvantages of using world builder:
  - May not support all VRML features, and
  - may not produce most efficient VRML

- Advantages of using a 3D modeler and format translator:
  - Very powerful drawing and animating features, and
  - can make photo-realistic images too

- Disadvantages of using a 3D modeler and format translator:
  - May not support all VRML features,
  - may not produce most efficient VRML,
  - not designed for VRML,

- often a one-way path from 3D modeler into VRML, and
- easy to make shapes that are too complex.

Structure of VRML file

- VRML files contain:
  - The ***file header***
  - ***Comments*** - notes to yourself
  - ***Nodes*** - nuggets of scene information
  - ***Fields -*** node attributes you can change
  - ***Values*** - attribute values
  - Something more. . .

The following shows a sample of VRML file structure:

```
#VRML V2.0 utf8
# A Cylinder
Shape {
   appearance Appearance {
      material Material { }
   }
   geometry Cylinder {
      height 2.0
      radius 1.5
   }
}
```

File Header

In the above VRML file, **#VRML V2.0 utf8,** is a file header, where
#VRML: File contains VRML text
V2.0 : Text conforms to version 2.0 syntax
utf8 : Text uses UTF8 character set.

utf8 is an international character set standard, which stands for UCS (Universal Character Set) Transformation Format, 8-bit.

comments
In the above VRML file, **# A Cylinder,** is a comment. Comments start with a number-sign (#) and extend to the end of the line.

Nodes

Nodes describe shapes, lights, sounds, etc. For example:

Cylinder { }

Every node has:
- A node type (Shape, Cylinder, etc.)
- A pair of curly-braces
- Zero or more fields inside the curly-braces

➢ **node type names**
  - Node type names are *case sensitive*
  - Each word starts with an upper-case character
  - The rest of the word is lower-case

- Some examples: **Appearance, Cylinder, Material, Shape, FontStyle.**

➢ **fields and values:** For example:

```
Cylinder {
            height 2.0
            radius 1.5
          }
```

- Fields describe node attributes
- Every field has: a field name (**height**, **radius**, etc.), a data type (float, integer, etc.) and a default value.
- Field names are *case sensitive*. The first word starts with lower-case character and each added word starts with upper-case character. The rest of the word is lower-case. Some examples are: **appearance, height, radius, fontStyle etc.**
- Different node types have different fields. Fields are optional. A default value is used if a field is not given. Fields can be listed in any order. The order doesn't affect the node.
- Thus, in brief we may conclude that the file header gives the version and provides the encoding. Nodes describe scene content. Fields and values specify node attributes. Everything is case sensitive.

2.11 LAB ASSIGNMENTS

Session 1:

Exercise 1. Open the official openGL website (i) http://www.openGL.org/ (ii) and the microsoft ftp site: *ftp://ftp.microsoft.com/softlib/mslfiles/opengl95.exe* and (3) ***http://reality.sgi.com/opengl/glut3/glutdlls.zip***
(i) Download (copy) all the three header files: **Gl.h, Glu.h** and **Glut.h.**
(ii) Download (copy) , the Static library files: **glu32.lib, opengl32.lib** and **glut32.lib**
(iii) Download (copy), the dynamic libraries: **glu32.dll, opengl32.dll** and **glut32.dll**

Exercise 2. Add all these three header files, static libraries and dynamic libraries in appropriate subdirectories of your compiler to run any OpenGL application program.

Exercise 3. Perform step-1 to step-18, given in section-4.4 , to create a project in VC++ and running an OpenGL application.

Session 2:

Exercise 4. WAP in C/C++ using OpenGL to draw a yellow rectangle on a black-background.

Exercise 5. WAP in C/C++ using OpenGL to draw a red rectangle on a black-background.

Exercise 6. WAP in C/C++ using OpenGL to draw basic primitives of OpenGL such as GL_LINES, GL_QUAD, GL_TRIANGLES etc.

Session 3:

Exercise 7. WAP in C/C++ to implement a DDA Line-drawing algorithm (without using inbuilt line() function).

Exercise 8. WAP in C/C++ to implement a Bresenham's Circle generation algorithm (without using inbuilt circle() function).

Exercise 9. WAP in C/C++ to draw the convex Polygon (a polygon is convex if a line connecting any two points of the polygon lies entirely within it) and fill a color in it.

Session 4:

Exercise 10. WAP in C/C++ using OpenGL to draw a tic-toc-tie board as shown in the following figure.

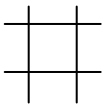

Figure-1

Exercise 11. Modify your program-2 for thick lines and stippled lines as shown in *Figure 2(a), 2(b).*

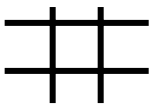

Figure 2(a)

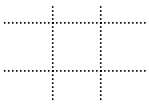

Figure 2(b)

Exercise 12. WAP in C/C++ using OpenGL to draw a hard wire house as shown in *Figure3*. Use basic primitive of openGL.

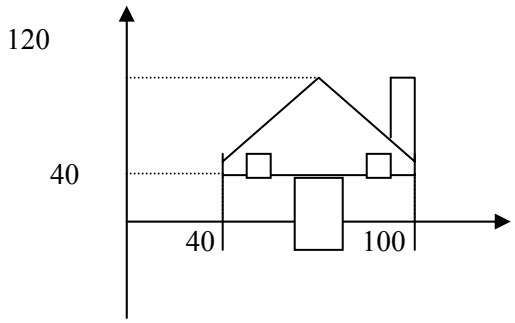

Figure-3

Session 5:

Exercise 13.WAP in C/C++ using OpenGL to draw a checkerboard by writing a routing checkerboard (int size) that draws the checkerboard as shown in *Figure 4.* Place the checkerboard with its lower-left corner at (0,0) and each of the 64 square, where W and B represents white and black colour, respectively.

W	B	W	B	W	B	W	B
B	W	B	W	B	W	B	W
W	B	W	B	W	B	W	B
B	W	B	W	B	W	B	W
W	B	W	B	W	B	W	B
B	W	B	W	B	W	B	W
W	B	W	B	W	B	W	B
B	W	B	W	B	W	B	W

Figure 4

Exercise 14. WAP in C/C++ using OpenGL to draw a stippled line with different stipped pattern and line Width. You can use various pattern and factors as follows:

Pattern	**Factor**
0x7733	1
0x5555	1
0xFF00	1
0xFF00	2
0x3333	3

Exercise 15. WAP in C/C++ using OpenGL to implement Cohen-Sutherland line clipping algorithm. To Implement this consider all the three cases of a line: totally visible, totally invisible and clipping candidate, against the rectangular clipping window.

Session 6:

Exercise 16. Run your program in exercise no. 14 (Cohen-Sutherland line clipping algorithm) for the folowing input, to find the visible portion of the line P(40, 80), Q(120, 30) inside the window, the window is defined as ABCD: A(20, 20), B(60, 20), C(60, 40) & D(20, 40).

Note: You can increase (with same scale) the coordinates of P, Q, A, B, C and D according to your system coordinates.

Exercise 17. WAP in C/C++ using OpenGL to perform a 3-Dimensional transformation, such as
translation, rotation and reflection, on a given triangle.

Exercise 18. Show that two successive reflections about either of the coordinate axes is equivalent to a single rotation about the coordinate origin. Run your program for the following object, given in the first octant.

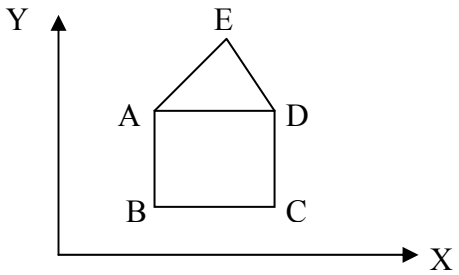

Figure 5

Session 7:

Exercise 19. WAP in C/C++ using OpenGL commands to perform a 3-Dimensional combined transformation, such as translation, rotation and reflection, on a given triangle.

Exercise 20. WAP in C/C++ using OpenGL commands to perform the following sequence of transformation:

i. Scale the given image (say Triangle ABC) x direction to be one-half as large and then rotate counterclockwise by 90^0 about the origin.
ii. Rotate counterclockwise about the origin by 90^0 and then Scale the x direction to be one-half as large.
iii. Translate down ½ unit, right ½ unit, and then rotate counterclockwise by 45^0.

Exercise 21. A square ABCD is given with vertices A(x1,y1),B(x2,y2),C(x3,y3), and D(x4,y4).
WAP in C/C++ to illustrate the effect of a) x-shear b) y-shear c) xy-shear on the given square, when x-shearing factor, a=2 and y-shearing factor, b=3.

Session 8:

Exercise 22. WAP in C/C++ to show that a reflection about the y=x is equivalent to a reflection relative the x-axis followed by a counterclockwise rotation of 90^0.

Exercise 23. Familiarize yourself with various OpenGL commands for parallel & Perspective projections.

Session 9:

Exercise 24. WAP in C/C++ to generate Fractal images (for example you can use Koch curve or Hilbert curves).

Exercise 25. WAP in C/C++ using OpenGL commands to perform a ray-tracing algorithm.

Exercise 26. Familiarise yourself with 3-D viewing using OpenGL library functions primitives.

Session 10:

Exercise 27. Download the VRML plug in on your system and familiarise your self with various features of VRML.

Example 28. Download the samples of VRML. For example, a cube, and experience its rotations. Now, edit this file and change the modeling parameters. Observe the effects of these changes.

2.12 FURTHER READINGS

1. The Internet site http://www.opengl.org
2. The Internet site: http://www.sgi.com/software/opengl/manual.html
3. "OpenGL Programming Guide"-The official guide to learning OpenGL, Version 1.4 (4^{th} edition), by Dave Shreiner, Jackie Neider, Tom Davis (Pearson Education)
4. "Computer Graphics-using OPENGL" (2^{nd} Edition), by F.S. Hill, J.R. (Pearson Edition)
5. "Computer Graphics" Hearn and Baker (PHI).