CHAPTER 1

INTRODUCTION

1.1 Overview

Computer Graphics involves technology to access. The Process transforms and presents information in a visual form. The role of computer graphics insensible. In today life, computer graphics has now become a common element in user interfaces, T.V. commercial motion pictures.

Computer Graphics is the creation of pictures with the help of a computer. The end product of the computer graphics is a picture it may be a business graph, drawing, and engineering.

In computer graphics, two or three-dimensional pictures can be created that are used for research. Many hardware devices algorithm has been developing for improving the speed of picture generation with the passes of time. It includes the creation storage of models and image of objects. These models for various fields like engineering, mathematical and so on.

Today computer graphics is entirely different from the earlier one. It is not possible. It is an interactive user can control the structure of an object of various input devices.

Types of computer graphics :

- **Raster Graphics**: In raster,graphics pixels are used for an image to be drawn.It is also known as bitmap image in which a sequence of images is into smaller pixels.Basically,a bitmap indicates large number of pixels together.
- **Vector Graphics**: In vector graphics, mathematical formulae are used to draw different types of shapes,lines,objects, and so on.

1.2 Problem Statement

The aim of this application is to show a basic implementation of a game involves shooting of arrows one after the other

1.3 About OpenGL

Most of our application will be designed to access OpenGL directly through functions in three libraries. Functions in the main GL (or OpenGL in windows) library have names that begin with the letters gl and are stored in a library usually referred to as GL (or OpenGL in windows).

The second is the OpenGL Utility Library (GLU). This library uses only GL functions but contains code for creating common objects and simplifying viewing. All functions in GLU can be created from the core GL library but application programmers prefer not to write the code repeatedly.

The GLU library is available in all OpenGL implementations; functions in the GLUlibrary begin with letters glu.

To interface with the window system and to get input from external devices into our programs, we need at least one more system-specific library that provides the "glue" between the window system and OpenGL. For the X window system, this library is functionality that should be expected in any modern windowing system.

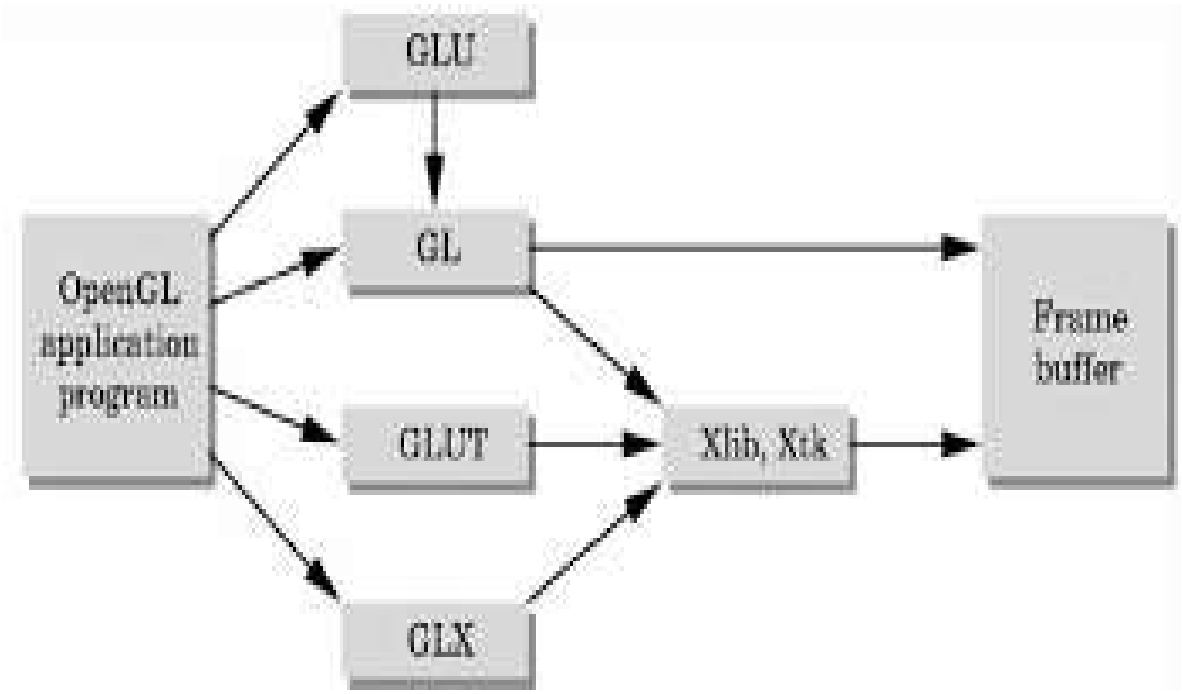


Fig 1.3.1 Library organization

Fig1.3.1 shows the organization of the libraries for an X Window System environment. For this window system, GLUT will use GLX and the X libraries. The application program, however, can use only GLUT functions and thus can be recompiled with the GLUT library for other window system.

1.4 OPENGL ARCHITECTURE:

1.4.1 Pipeline Architectures

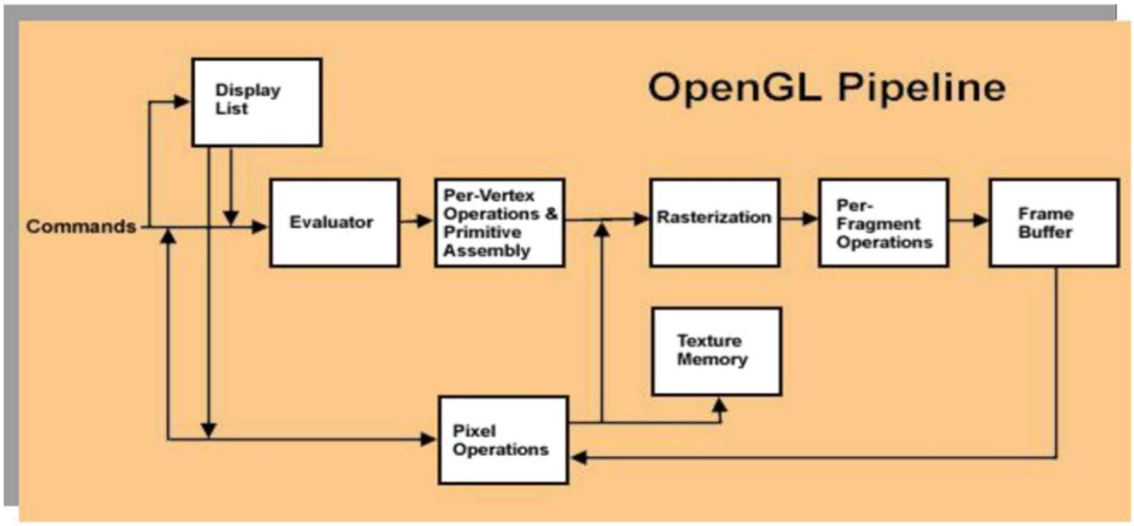


FIG:1.4.1 OPENGL PIPELINE ARCHITECTURE

Display Lists:

All data, whether it describes geometry or pixels, can be saved in a display list for current or later use. (1 alternative to retaining data in a displaylist is processing the data immediately - also known as immediate mode.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode.

Evaluators:

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points.

Per-Vertex Operations:

For vertex data, next is the "per-vertex operations" stage, which converts. The vertices into primitives. Some vertex data (for example, spatial coordinates) are transformed by 4 x 4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen. If advanced features are enabled, this stage is even busier. If texturing is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a color value.

Primitive Assembly:

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped.

In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then view port and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines. The results or this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

Pixel Operations:

While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step. If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory. There are special pixel copy operations to copy data in the frame buffer to other parts of the frame buffer or to the texture memory. A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the frame buffer

Texture Assembly:

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that you can easily switch among them. Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource.

Rasterization:

Rasterization is the conversion of both geometric and pixel data into fragments. Each fragment square corresponds to a pixel in the frame buffer. Line and polygon stipples, line width, point size, shading model, and coverage calculations to support ant aliasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square.

1.4.2 OpenGL Engine And Drivers:

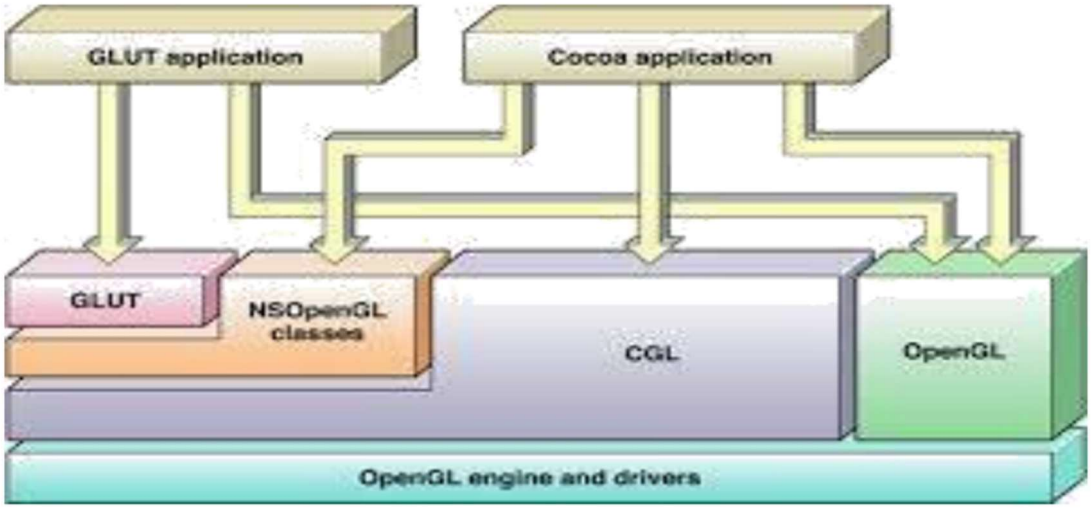
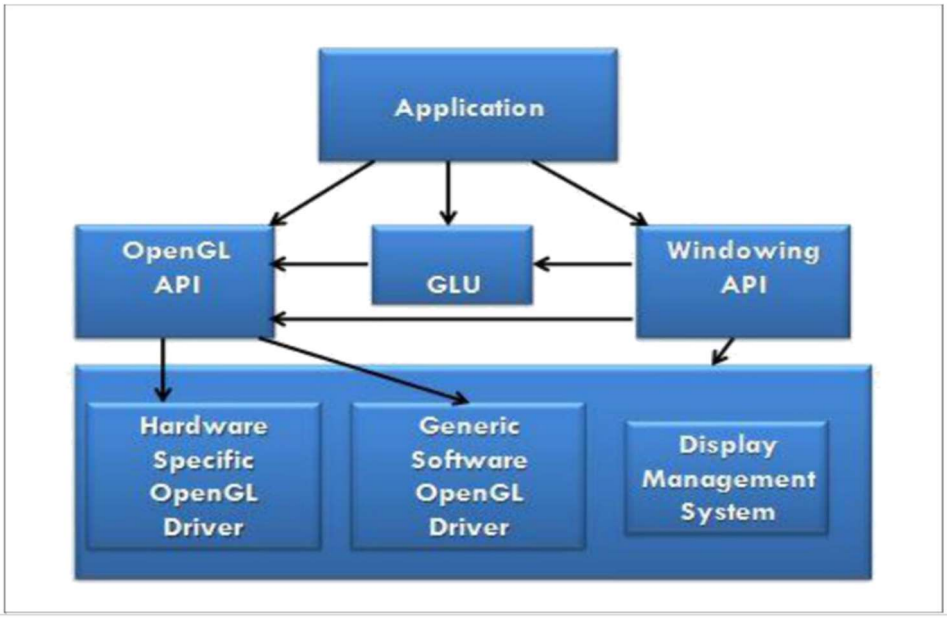


FIG:1.4.2 OPENGL ENGINE AND DRIVERS

1.4.3 Application Development-API's



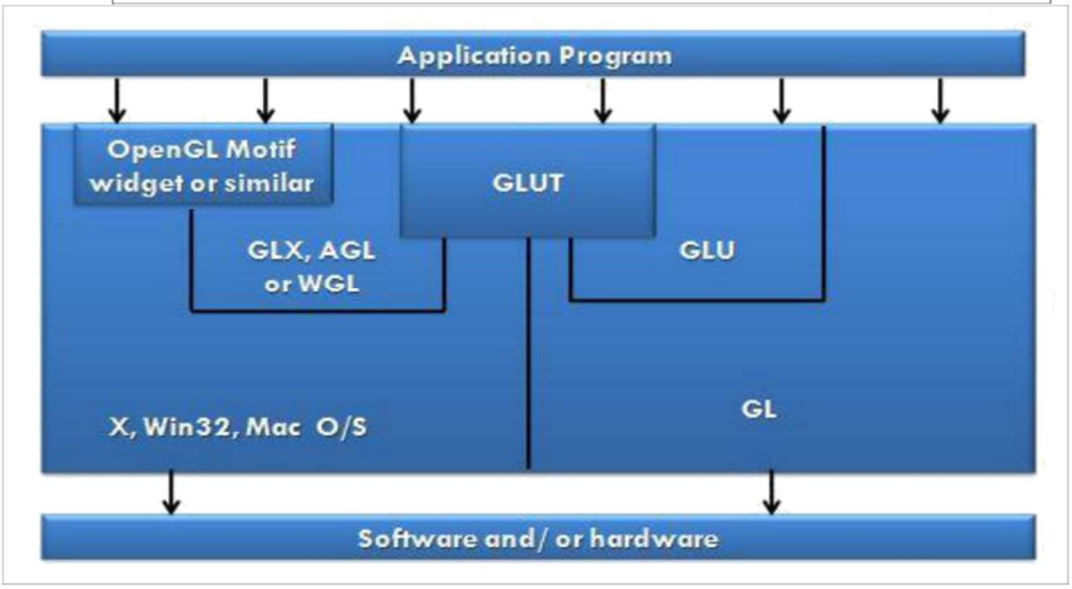


Fig 1.4.3 Application Development API's

The above diagram illustrates the relationship of the various libraries and window system components. Generally, applications which require more user interface support will use a library designed to support those types of features (i.e. buttons, menu and scroll bars, etc.) such as Motif or the Win32 API. Prototype applications, or ones which do not require all the bells and whistles of a full GUI, may choose to use GLUT instead because of its simplified programming model and window system independence.

CHAPTER 2

REQUIREMENTS ANALYSIS

The requirements can be broken down into 2 major categories namely hardware and software requirements. The former specifies the minimal hardware facilities expected in a system in which the project has to be run. The latter specifies the essential software needed to build and run the project.

2.1 Hardware Requirements

The hardware requirement is minimal and the software can run with minimal requirements.

The basic requirements are as enlisted below:

1. Processor: Intel 486/ Pentium processor or a processor with higher specifications
2. Processor speed: 500MHz or above.
3. RAM : 64MB or above
4. Storage space : 2MB or above
5. Monitor resolution: A color monitor with a minimum resolution of 640 * 480.

2.2 Software Requirements

1. An MS-DOS based operating system like Windows 98, Windows 2000 or Windows XP is the platform required to develop the 3D simulation.
2. A C/C++ (integrated with OPEN GL) compiler like codeblocks is required for compiling the source code to make the executable file which can then be directly executed.
3. A built in graphics library; glut.h is required for drawing the layout of the game.
4. glut32.dll for running the application.

CHAPTER 3

SOFTWARE DESIGN

3.1 Overview

This project has been created using the OpenGL interface along with the GLUT(OpenGL Library Toolkit), using the platform Eclipse as the compiling tool. This game has been designed in a simple and lucid manner so that further developments can be made, and run on many platforms with a few changes in the code.

We begin our game by providing a simple menu option upon right click of the mouse to display 2 levels of the game which the user can choose, and a third option to exit the game.

Upon selecting the first level, a red ball is continuously produced and moves upwards at the right side of the screen. Hitting the 'f' key on the keyboard produces an arrow, which the user must aim to hit the balls with. On successful contact with the ball, the game registers it as a 'HIT' and increments a counter variable. A count of 3 (i.e 3 hits) takes the player to the next level of the game.

Level 2 increases the difficulty level by producing2 colours of balls- a blue one and a red one. While the red is as before produced continuously and moves from bottom to top, the blue one moves from top to bottom at irregular intervals to confuse the player. The player needs to aim correctly and target only the red ones. Player needs to again score a minimum of 3 'hits' to be a Winner.

Level 2 can be directly entered in the beginning itself (without playing level 1). We provide this option so that the player can choose his difficulty level.

The underlying logic of the game is Collision detection. The bounds of the ball are detected with the bounds, or the continuously changing position of the arrow head. 2 variables 'pos' (for the arrowhead) and 'up' (for the ball) are kept tab of, and are updated with their changing values.

If the given position vector does intersect with the current value of the 'up', we infer that a collision has been detected and register it as a hit, and increment 'counter'. Other wise the function returns a NULL.

Once a 'hit' takes place, we call a Sleep() timer (as buffers are too fast to see the collision in our game), show the ball burst (without particle effect or engines) and reset the ball to its initial position and the cycle starts all over again.

Careful observation of these values have been made while developing the game.

To model the objects as 3D entities we also use the effects of lighting in our game via 4 different mechanisms:

AMBIENT - light that comes from all directions equally and is scattered in all directions equally by the polygon.

DIFFUSE - light that comes from a point source and hits surfaces with an intensity that depends on whether they face towards the light or away from it.

SPECULAR - light is reflected more in the manner of a mirror where most of the light bounces off in a particular direction defined by the surface shape.

EMISSION - the light is actually emitted by the polygon - equally in all directions.

Texture is yet another feature we have implemented to show the material properties of the object, in whichever small way we could

CHAPTER 4

IMPLEMENTATION

4.1 Module Description

Choose( ):

This function gives a switch case to choose from different menu options. Choosing case 2 (Level 1) calls the function Display1( ) and choosing case 3 (Level 2) calls Display2( ). Choosing quit case 1 will exit from the game at any point in the game.

Display( ):

sets the background colour and clears the color buffer bit, and depth buffer bit (for hidden surface removal and Z buffer test), and it is called from the main()

Display1():

- It plays the Level 1 of the game. It initially sets the the red ball to a position given by 'up' variable, and using the Translatef() function it displaces only the y-coordinate of the ball upwards

- glutSolidSphere() renders a 3D sphere with radius, slices and stack as parameters.

- While the 'f' key is pressed, the function creates an arrow head, and associates a variable 'pos' with it, to translate the arrow towards the right in a single direction. This variable is incremented continually everytime and called with the Translatef() to redraw the arrow at new positions.

- If the condition for bounds satisfy, that means collision has occurred, and counter1 is incremented to register a hit. The flag 'bang' is set to 1, so that when encountered during the next iteration the following changes can take place: position of sphere is reset to 0, bang is reset to 0 (to prepare for next hit), position of arrow is reset.

- 'up' variable is continually incremented to keep the ball moving upwards for a large number of iterations by calling glutPostRedisplay() everytime. It marks the normal plane of current window as needing to be redisplayed with the same specifications.

- The counter1 value is checked for every iteration. Once it has reached a value of 3, Display2() is called to play Level 2 of the game.

Display2( ):

- It enters the Level 2 of the game. It initially sets the the red ball to a position given by 'up' variable, and using the Translatef() function it displaces only the y-coordinate of the ball upwards.
- Also, it sets the blue balls position and using the Translatef() function it displaces only the y-coordinate of the ball downwards.

- glutSolidSphere() renders a 3D sphere with radius, slices and stack as parameters.

- While the 'f' key is pressed, the function creates an arrow head, and associates a variable 'pos' with it, to translate the arrow towards the right in a single direction. This variable is incremented continually everytime and called with the Translatef() to redraw the arrow at new positions.

- If the condition for bounds satisfy, that means collision has occurred, and counter1 is incremented to register a hit. The flag 'bang' is set to 1, so that when encountered during the next iteration the following changes can take place: position of sphere is reset to 0, bang is reset to 0 (to prepare for next hit), position of arrow is reset.

- 'up' variable is continually incremented to keep the ball moving upwards for a large number of iterations by calling glutPostRedisplay() everytime. It marks the normal plane of current window as needing to be redisplayed with the same specifications. The counter2 value is checked for every iteration. Once it has reached a value of 3, myHit() is called to display that the player is a winner.

MyHit( ):

This function is primarily to display text on screen. glBlendFunc defines the operation of blending when it is enabled. We also set the antialiasing for lines and set the line width to prepare to draw our text as Stroke Characters.

Drawhit( ):

It draws text in Stroke Character font. A stroke font is a 3D font. As opposed to bitmap fonts these can be rotated, scaled, and translated.
The GLUT_STROKE_ROMAN font is used here. The text can be placed anywhere on the screen using Translatef() and scaled to any size as needed using Scalef().

Instructions( ):

It creates a separate instruction page where the instructions are displayed on another window using StrokeCharacter font, and translated to appropriate postions on the screen.

Draw instruct( ):

It draws text in Stroke Character font. A stroke font is a 3D font. As opposed to bitmap fonts these can be rotated, scaled, and translated. The GLUT_STROKE_ROMAN font is used here. The text can be placed anywhere on the screen using Translatef() and scaled to any size as needed using Scalef().

Keyboard( ):

Keyboard is a keyboard callback function which is used to make our program interactive. It makes the shoot flag = 1 in our program everytime key 'f' is pressed, by recognizing it as an ASCII character.

Main():

The main function performs the required initializations and starts the event processing loop. All the functions in GLUT have the prefix glut, and those which perform some kind of initialization have the prefix glutInit.

- glutInit(int *argc, char **argv) - parameters are pointers to the unmodified argc and argv variables from the main function.
- We establish the window's position by using the function glutInitWindowPosition.
- We choose the window size using the function glutInitWindowSize.
- We define the display mode using the function glutInitDisplayMode. GLUT_RGB - selects a RGBA window. This is the default color mode. GLUT_SINGLE – selects a single buffer window.
- Each window can be created with glutCreateWindow. The return value of glutCreateWindow is the window identifier.
- glutDisplayFunc() passes the name of the function to be called when the window needs to be redrawn.
- glutKeyboardFunc- is notify the windows system which function(s) will perform the required processing when a key is pressed. This function is to register a callback for keyboard events that occur when you press a key.
- Creating a menu: glutCreateMenu creates a menu table on a default right click of mouse. glutAddMenuEntry adds a menu entry to this menu created.

- When we are ready to get in the application event processing loop we enter glutMainLoop. It gets the application in a never ending loop, always waiting for the next event to process.

Init( ):

Sets the background color for the game and enables light source to provide the following lighting effects:

- AMBIENT - light that comes from all directions equally and is scattered in all directions equally by the polygon.
- DIFFUSE - light that comes from a point source and hits surfaces with an intensity that depends on whether they face towards the light or away from it.
- SPECULAR - light is reflected more in the manner of a mirror where most of the light bounces off in a particular direction defined by the surface shape.
- EMISSION - the light is actually emitted by the polygon - equally in all directions It also sets the Material for the object and enables this option to provide different surface textures to our objects.

4.2 User implementation

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<GL/glut.h>
#include<time.h>
#include<math.h>
//#include<windows.h>
static GLfloat up=-0.2;
static GLfloat pos=-0.2;
int shoot=0,bang=0;
int counter1=0,counter2=0,count=0;
int game,instruct;
char tmp_str[40];
void display2();
void displost();
void init(void)
{
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess[] = { 50.0 };
GLfloat mat_diffuse[]={ 1.0,1.0,1.0,1.0};
GLfloat mat_ambient[]={0.0,0.0,0.0,1.0};
GLfloat light_position[] = { 1.0, 1.0, 0.0, 0.0 };
```

```
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_SMOOTH);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_DEPTH_TEST);
glEnable(GL_COLOR_MATERIAL);
}
void drawhit(const char * message, int x, int y)
{
glPushMatrix();
glScalef(0.3,0.2,0.15);
glTranslatef(x,y,0);
while (*message)
{
glutStrokeCharacter(GLUT_STROKE_ROMAN,*message++);
}


glPopMatrix();
}

void myHit()
{
glClear(GL_COLOR_BUFFER_BIT);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,200,0,200);
glMatrixMode(GL_MODELVIEW);
glClearColor(1.0,0.0,0.5,1.0);
glColor3f(0.0,0.8,0.80);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
glEnable(GL_LINE_SMOOTH);
glLineWidth(4.0);
drawhit("WINNER!!",70,550);
}
void draw_instruct(const char *message, int x, int y)
{
int j;
glPushMatrix();
glScalef(0.1,0.1,0.0);
glTranslatef(x,y,0);
while (*message)
{
glutStrokeCharacter(GLUT_STROKE_ROMAN,*message++);
```

```
}
for(j=0;j<10000;j++);
glPopMatrix();
}
void instructions()
{
glClear(GL_COLOR_BUFFER_BIT);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,200,0,200);
glMatrixMode(GL_MODELVIEW);
glClearColor(1.0,0.7,0.0,1.0);
glColor3f(1.0,0.5,0.1);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
glEnable(GL_LINE_SMOOTH);
glLineWidth(4.0);
draw_instruct("Instructions",600,1850); // change1
draw_instruct("Right click on mouse",300,1700);
draw_instruct("to play",300,1500);
draw_instruct("Press f to shoot",300,1300);


glFlush();
}

void Write(char *string)
{
glScalef(0.02,0.02,0.0);
while(*string)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, *string++);
}
void display1()
{
int i;
if(counter1==3)
{
display2();
glFlush();
}
else
{ int j;
for(j=0;j<10000;j++);
glClearColor(1.0,0.7,0.0,1.0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glColor3f(1, 1, 0);
glRasterPos2f(-0.9, 0.9);
sprintf(tmp_str, "Arrow count: %d", count);
Write(tmp_str);
glPopMatrix();
```

```
if(count>=30)
glutDisplayFunc(displost);
glPushMatrix();
glColor3f(1, 1, 0);
glRasterPos2f(-0.2, 0.9);
sprintf(tmp_str, "Score: %d", counter1);
Write(tmp_str);
glPopMatrix();
glPushMatrix();
glColor3f(1.0,0.0,0.0);
glLoadIdentity();
glTranslatef(0.8,-0.869+up,0.0);
glutSolidSphere(0.15,20,16);
if(shoot==1)
{
glPushMatrix();
glLoadIdentity();
glTranslatef(-0.8+pos,0.0,0.0);
glColor3f(0.0,0.0,0.0);
glLineWidth(2.0);


glBegin(GL_LINES);
glVertex3f(-0.2,0.0,0.0);
glVertex3f(0.1,0.0,0.0);
glVertex3f(0.1,0.0,0.0);
glVertex3f(0.03,0.05,0.0);
glVertex3f(0.1,0.0,0.0);
glVertex3f(0.03,-0.05,0.0);
glEnd();
glPopMatrix();
}
if(bang==1)
{
bang=0;pos=-0.2;
glPushMatrix();
glLoadIdentity();
up=0;
glColor3f(1.0,0.0,0.0);
glutSolidSphere(1,20,16);
glPopMatrix();
}
glPopMatrix();
for( i=0;i<200;i=i+20)
{
if(pos>=1.74 && up>0.825 && up<0.975)
//collision detection
{
counter1 ++;
for(j=0;j<10000;j++);
shoot=0;
```

```
pos=-0.2;
bang=1;
}
if(counter1==3)
count=0;
up=(up+0.005);
if(up>2)
up=0;
if(shoot==1)
{
pos=pos+0.009;
if(pos>2)
{
pos=-0.2;
shoot=0;
}
}
glutPostRedisplay();
}


glFlush();
}
}
void display2()
{int i;
if(counter2==3)
{ myHit();
glFlush();
}
else
{int j;
for(j=0;j<10000;j++);
glClearColor(1.0,0.7,0.0,1.0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
glPushMatrix();
glColor3f(1, 1, 0);
glRasterPos2f(-0.9, 0.9);
sprintf(tmp_str, "Arrow count: %d", count);
Write(tmp_str);
glPopMatrix();
if(count>=20)
glutDisplayFunc(displost);
glPushMatrix();
glColor3f(1, 1, 0);
glRasterPos2f(-0.2, 0.9);
sprintf(tmp_str, "Score: %d", counter2);
Write(tmp_str);
glPopMatrix();
glPushMatrix();
```