Chapter 1

INTRODUCTION

1.1 Computer Graphics

The term computer graphics includes almost everything on computers that is not text or sound. Today almost every computer can do some graphics, and people have even come to expect to control their computer through icons and pictures rather than just by typing.

In our Computer Graphics lab, we think of computer graphics as drawing pictures on computers, also called rendering. The pictures can be photographs, drawings, movies, or simulations -- pictures of things which do not yet exist and maybe could never exist. Or they may be pictures from places we cannot see directly, such as medical images from inside your body.

We spend much of our time improving the way computer pictures can simulate real world scenes. We want images on computers to not just look more realistic, but also to BE more realistic in their colors, the way objects and rooms are lighted, and the way different materials appear. We call this work "realistic image synthesis", and the following series of pictures will show some of our techniques in stages from very simple pictures through very realistic ones.

1.2 OpenGL Technology

OpenGL is a graphics application programming interface (API) which was originally developed by Silicon Graphics. OpenGL is not in itself a programming language, like C++, but functions as an API which can be used as a software development tool for graphics applications. The term Open is significant in that OpenGL is operating system independent. GL refers to graphics language. OpenGL also contains a standard library referred to as the OpenGL Utilities (GLU). GLU contains routines for setting up viewing projection matrices and describing complex objects with line and polygon approximations.

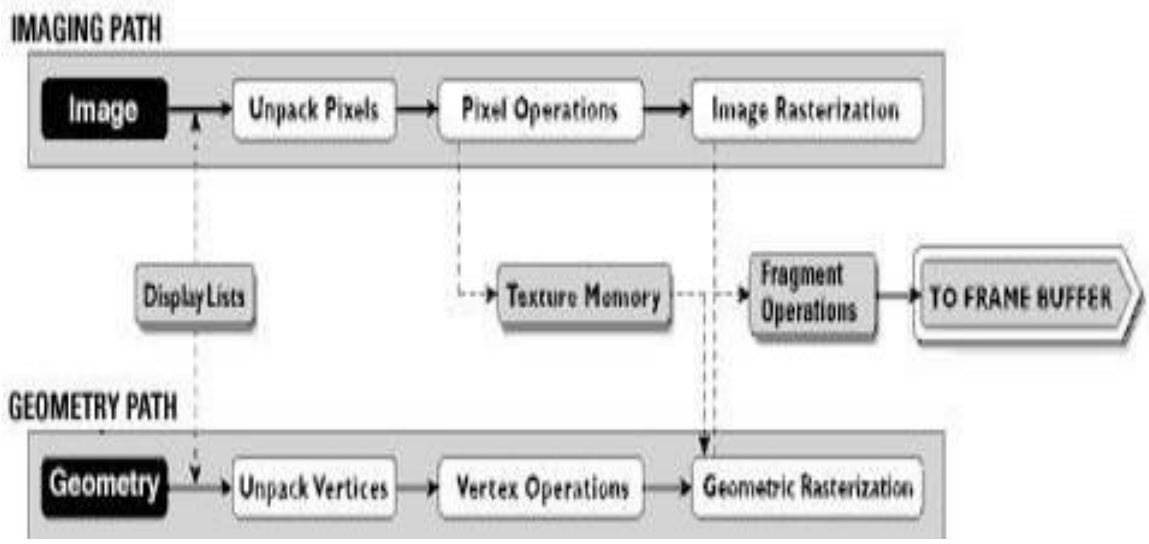
OpenGL gives the programmer an interface with the graphics hardware. OpenGL is a low-level, widely supported modelling and rendering software package, available on all

platforms. It can be used in a range of graphics applications, such as games, CAD design, modelling.

OpenGL is the core graphics rendering option for many 3D games, such as Quake 3. The providing of only low-level rendering routines is fully intentional because this gives the programmer a great control and flexibility in his applications. These routines can easily be used to build high-level rendering and modelling libraries. The OpenGL Utility Library (GLU) does exactly this, and is included in most OpenGL distributions! OpenGL was originally developed in 1992 by Silicon Graphics, Inc, (SGI) as a multi-purpose, platform independent graphics API. Since 1992 all of the development of OpenGL

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive three-dimensional applications.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, you must work through whatever windowing system controls the particular hardware you're using.



fig(1.1) OpenGl Visualization Programming Pipeline

The OpenGL Visualization Programming Pipeline:-

Routines simplify the development of graphics software—from rendering a simple geometric point, line, or filled polygon to the creation of the most complex lighted and texturemapped NURBS curved surface. OpenGL gives software developers access to geometric and image primitives, display lists, modeling transformations, lighting and texturing, anti- aliasing, blending, and many other features. Every conforming OpenGL implementation includes the full complement of OpenGL functions. The well-specified OpenGL standard has language bindings for C, C++, Fortran, Ada, and Java. All licensed OpenGL implementations come from a single specification and language binding document and are required to pass a set of conformance tests. Applications utilizing OpenGL functions are easily portable across a wide array of platforms for maximized programmer productivity and shorter time-to-market.

All elements of the OpenGL state—even the contents of the texture memory and the frame buffer—can be obtained by an OpenGL application. OpenGL also supports visualization applications with 2D images treated as types of primitives that can be manipulated just like 3D geometric objects. As shown in the OpenGL visualization programming pipeline diagram above.

OpenGL (Open Graphics Library) is a cross-platform API (Application Programming Interface) for rendering 2D and 3D vector graphics. It provides developers with a set of functions to interact with graphics hardware, enabling them to create interactive applications ranging from simple 2D games to complex 3D simulations and virtual reality environments. Originally developed by Silicon Graphics Inc. (SGI) in the early 1990s, OpenGL has since evolved into an industry-standard graphics API widely supported across various platforms, including Windows, macOS, Linux, and mobile operating systems.

One of the key strengths of OpenGL lies in its platform independence and broad compatibility. By abstracting the complexities of different graphics hardware, OpenGL allows developers to write code once and deploy it across multiple platforms without significant modifications. This versatility has made it a preferred choice for game developers, scientific visualization, CAD (Computer-Aided Design), and other applications requiring high-performance graphics rendering. OpenGL operates on a state machine model, where various states (such as transformations, rendering modes, and textures) are set and then applied to vertices and primitives (like points, lines, and

polygons). This approach provides flexibility in defining how graphics are processed and displayed. The API's core functionality revolves around defining geometric shapes, applying textures and shaders (programs that run on the GPU to manipulate graphics data), and controlling how objects are rendered in a scene.

Modern versions of OpenGL have evolved significantly, introducing features like shader-based rendering (OpenGL Shading Language, or GLSL), which allows developers to implement complex lighting, shading, and visual effects directly on the GPU. This shift towards programmable pipelines has enabled more realistic and immersive graphics in games and simulations. Additionally, OpenGL's compatibility with other libraries and frameworks, such as GLFW, SDL, and Qt, further extends its capabilities and simplifies application development across different domains. In summary, OpenGL remains a powerful and versatile tool for developers seeking to create visually compelling and interactive applications across a wide range of platforms. Its robust feature set, cross-platform support, and integration with modern graphics technologies continue to make it a cornerstone of graphics programming in both professional and hobbyist contexts.

OpenGL continues to evolve with advancements in graphics hardware and software. Over the years, OpenGL has adapted to support new technologies and features that enhance its capabilities. For instance, extensions like OpenGL ES (OpenGL for Embedded Systems) cater specifically to mobile and embedded platforms, optimizing performance and efficiency for devices with limited resources. This adaptability has ensured OpenGL's relevance in modern computing environments, from high-end gaming PCs to smartphones and embedded devices powering IoT (Internet of Things) applications.

Moreover, OpenGL's open-source nature and widespread adoption have fostered a vibrant community of developers and enthusiasts. This community contributes to the ecosystem by creating libraries, tools, and frameworks that complement OpenGL's core functionalities. Libraries like OpenGL Utility Library (GLU) and OpenGL Extension Wrangler (GLEW) simplify common tasks such as texture loading, matrix operations, and handling extensions, streamlining development and enabling faster prototyping and deployment of graphics-intensive applications. As the graphics landscape continues to evolve with technologies like Vulkan and DirectX, OpenGL remains a crucial foundation for understanding graphics programming concepts and building cross-platform applications that leverage the power of modern GPUs. Whether used in academia, industry, or hobbyist projects, OpenGL continues to empower developers to create visually rich and interactive experiences across diverse computing environments.

1.3 PROJECT DESCRIPTION:

The project is aimed to provide an interactive gaming experience. In this program the paddles can be controlled by the players using the keyboard. Point is updated each time a player loses. And the winner is also announced once a player reaches 5 points. We can control the paddles by using the following keys:

For player 1-

"q": - To move the paddle upwards.

"a": - To move the paddle downwards.

"z": - To serve the ball.

For player 2-

"p": - To move the paddle upwards.

"l": - To move the paddle downwards.

"m": - To serve the ball.

1.4 OpenGl Functions Used:

glutInit(int argc, char **argv): Initializes GLUT and processes command-line arguments passed from main().

glutCreateWindow(char *title): Creates a window with the specified title for displaying OpenGL graphics.

glutInitDisplayMode(unsigned int mode): Sets the initial display mode, including color depth, single/double buffering, and RGBA/RGB color channels.

glutInitWindowSize(int width, int height): Specifies the initial size of the window in pixels.

glutInitWindowPosition(int x, int y): Sets the initial position of the window's top-left corner on the screen.

glutMainLoop(): Enters the GLUT event processing loop, handling user input, window redraws, and other events.

glClear(GLbitfield mask): Clears buffers to preset values, typically used at the start of the render loop.

glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha): Sets the clear color for the color buffer.

glMatrixMode(GLenum mode): Specifies the target matrix stack for subsequent matrix operations.

glLoadIdentity(): Replaces the current matrix with the identity matrix.

glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble nearVal, GLdouble farVal): Defines a 2D orthographic projection matrix.

glViewport(GLint x, GLint y, GLsizei width, GLsizei height): Sets the viewport for mapping OpenGL's rendering coordinates to the window coordinates.

glutDisplayFunc(void (*func)(void)): Sets the display callback function, called when the window needs to be redrawn.

glutReshapeFunc(void (*func)(int width, int height)): Sets the reshape callback function, called when the window is resized.

glutKeyboardFunc(void (*func)(unsigned char key, int x, int y)): Sets the keyboard callback function, handling ASCII keyboard input.

glutSpecialFunc(void (*func)(int key, int x, int y)): Sets the special keyboard callback function, handling special keys like arrow keys.

glutIdleFunc(void (*func)(void)): Sets the idle callback function, continuously called when there are no other events to process.

glutSwapBuffers(): Swaps the front and back buffers of the current window, displaying the rendered image in double-buffered mode.

Chapter 2

SYSTEM REQUIREMENTS

2.1 Hardware requirements:

Pentium or higher processor.

128 MB or more RAM.

A standard keyboard, and Microsoft compatible mouse

VGA monitor.

2.2 Software requirements:

The graphics package has been designed for OpenGL; hence the machine must have Dev C++.

Software installed preferably 6.0 or later versions with mouse driver installed.

GLUT libraries, Glut utility toolkit must be available.

Operating System: Windows

Version of Operating System: Windows XP, Windows NT and Higher

Language: C++

Chapter 3

IMPLEMENTATION

3.1 Inbuilt Functions :

glBegin()

C Specification: void glBegin (Glenum mode); Parameters: mode

Description: Specifies the primitive(s) that will be created from vertices present between glBegin and the subsequent glEnd. Ten symbolic constants are accepted.

glEnd()

C Specification: void glEnd();

Description: glBegin and glEnd delimit the vertices that define a primitive or a group of like primitives. glBegin accepts a single argument that specifies in which often ways the vertices are interpreted. Taking ‘n’ as an integer count starting at one, and ‘N’ as the total number of vertices specified, the interpretations are as follows: GL_LINES

Treats each pair of vertices as an independent line segment. Vertices 2n-1 and 2n define line n. N/2 lines are drawn.

GL_LINE_LOOP

Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and n+1 define line n. The last line, however, is defined by vertices N and N lines are drawn.

GL_QUADS

Treats each group of four vertices as an independent quadrilateral. Vertices 4n-3, 4n-2, 4n- 1 and 4n define quadrilateral n. N/4 quadrilaterals are drawn.

GL_POLYGON

Draws a single, convex polygon. Vertices 1 through N define this polygon.

GlClearColor()

C Specification: void glClearColor(GLClamp red, GLClamp green, GLClamp blue, GLClamp alpha); Parameters: red, green, blue, alpha specifies the red, blue, green and alpha values used when the color buffers are cleared, the initial values are all 10.

Description: glClearColor specifies the red, green, blue, alpha values used by glClear to clear the color buffers. Values specified by glClearColor are clamped to the range [0, 1].

glClear()

C specification: voidglClear(GLbitfield mask);

Parameters: mask bitwise OR of masks that indicate the buffers to be cleared. The four masks are GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_ACCUM_BUFFER_BIT.

glFlush()

C Specification: void glFlush();

Description:glFlush empties all the buffers causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine.

GlLoadIdentity()

C specification: void glLoadIdentity();

Description:glLoadIdentity replaces the current matrix with identity matrix.

glOrtho()

C Specification: void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdoubleznear,GLdoublezfar);

Description: specifies the coordinate system OpenGL assumes as it draws the final image and how the image gets mapped to the screen.

glVertex()

C Specification: void glVertex{234}{sifd}[v](TYPEcoords);

Description: glVertex commands are used within glBegin/glEnd pairs to specify point, line and polygon vertices. It specifies a pointer to an array of two, three or four elements.

glutCreateWindow()

C Specification: intglutCreateWindow(char *name);

Description: glutCreateWindow creates a top-level window. The name will be provided to the window's name.

glutReshapeFunc()

C specification: void glutReshapeFunc(void (*func)(int width, int height));

Description: glutReshapeFunc sets the reshape callback for the current window. The reshape callback is triggered when a window is reshaped.

glutKeyboardFunc()

C Specification:void glutKeyboardFunc(void (*func)(char key, intx,inty));

Description: glutKeyboardFunc sets the keyboard callback for the current window. When a user types into the window, each key press generating an ASCII character will generate a keyboard callback.

3.2 UserDefined Functions :

Initializing Variables

//The provided game class initializes and manages essential variables and parameters for a gaming environment. It begins by defining fundamental properties such as window dimensions (WinWid, WinHei) and viewport settings (OrthoWid, OrthoHei) to configure the graphical display. Other crucial parameters include the dimensions of the playing field (FieldSizeX, FieldSizeY), the thickness of paddles (PThickness), and characteristics of game elements such as the ball (BallSize, BallSpeedX) and midline (MLineT). Scores for each player (ScoreL, ScoreR) are initialized to zero, and positional values (TextPosX, TextPosY) for displaying text elements on the screen are set accordingly. Additionally, speed variables (PSpeedY for paddles) dictate the movement dynamics within the game.

Instances left and right are objects instantiated from the game class, representing the left and right paddles respectively. These instances are likely used to manage paddle movement (move() method) and interaction with the ball (care() method), encompassing functions related to updating positions, handling collisions, and possibly managing game states such as scoring. The class encapsulates the foundational setup and behavior of game components, providing a structured framework for implementing game logic and rendering visuals using OpenGL or a similar graphics library.

class game{ public:int OrthoWid; int OrthoHei; int WinWid; int WinHei; int winXPos; int winYPos;

float FieldSizeX; float FieldSizeY; int delay;

float BallSpeedX; float PSpeedY; game(){

delay = 1;

PThickness = 10;

BallSize = 5;

FieldSizeX = 600;

FieldSizeY = 400;

BorderT = 10;

MLineT = 5;

ScoreL = 0;

ScoreR = 0;

TextPosX = 0;

TextPosY = FieldSizeY + 10; BallSpeedX = 22.5;

PSpeedY = 2;}

Down = false; hold = false;}

void draw(); void move(); void care();

} left,right;

Code for paddle functions

//The KeyControl function adjusts the vertical velocity (vy) of two paddles (left and right) based on keyboard input, allowing them to move up or down at a specified speed (PSpeedY).

void game::KeyControl(){

if((left.Up)&&(!lefThe KeyControl function adjusts the vertical velocity (vy) of two paddles (left and right) based on keyboard input, allowing them to move up or down at a specified speed (PSpeedY).ght.vy = PSpeedY;

if((!right.Up)&&(right.Down))right.vy = -PSpeedY;}

Code to define ball physics

//The care function in the reflector class checks if a condition (hold) is true, then adjusts the ball's horizontal position (x) and vertical velocity (vy) based on collision with a paddle (settings.PThickness).

void reflector::care(){ if(hold){

ball.vx = 0;

if(x < 0)ball.x = x + 2*settings.PThickness;

if(x > 0)ball.x = x - 2*settings.PThickness; ball.vy = vy;

ball.y = y;}}

Code for drawing the field void game::DrawField(){ if(l%2==0){

//The DrawField function alternates the color of the field between random shades of gray and random shades of a color based on the value of l, ensuring a visually dynamic appearance with each call to the function.

glColor3f((rand()%2),(rand()%2),(rand()%2)); l++;}

else if(l%3==0){

glColor3f((rand()%3),(rand()%3),(rand()%3)); l++;}

Code for initializing ball

//The ball class defines attributes (x, y, vx, vy) and methods (move(), reflection(), draw()) for simulating and rendering a ball object in a game environment. This setup encapsulates ball physics and graphical rendering functionalities within a single class instance.

```
class ball{ public:

float x; float y; float vx; float vy;

void move(); void reflection(); void draw();

}ball;
```

Code for initializing paddle

//The reflector class initializes attributes such as position (x, y), vertical velocity (vy), size, and input flags (Up, Down, hold) for simulating a paddle in a game environment.

```
class reflector{ public:float x,y; float vy; float size;

bool Up, Down, hold; reflector(){

vy = 0;

y = 0;

Up = false; else{

glColor3f((rand()%5),(rand()%5),(rand()%5)); l++;

}if(ScoreR >= 5){ glColor3f(0,1,0);

glRasterPos2f(TextPosX+270,TextPosY+50);
glutBitmapCharacter(GLUT_BITMAP_9_BY_15,'W');
glutBitmapCharacter(GLUT_BITMAP_9_BY_15,'I');
glutBitmapCharacter(GLUT_BITMAP_9_BY_15, 'N');

glutBitmapCharacter(GLUT_BITMAP_9_BY_15,'N');
glutBitmapCharacter(GLUT_BITMAP_9_BY_15,'E');
glutBitmapCharacter(GLUT_BITMAP_9_BY_15, 'R');

}}
```

Code for moving the paddles

//The DrawScore function in OpenGL renders player scores (ScoreL and ScoreR), labels ("PLAYER 1" and "PLAYER 2"), and victory messages ("WINNER") with dynamic color changes based on game conditions (r).

```
void reflector::move(){ y += vy;

if(y < -settings.FieldSizeY + size/2){ y = -settings.FieldSizeY + size/2;vy = 0;}

if(y > settings.FieldSizeY - size/2){ y = settings.FieldSizeY - size/2;vy = 0;}}

void reflector::draw(){ glBegin(GL_QUADS); if(m%2==0){glColor3f(1,0,0); m++;}
```

Code to call all draw funcitons

//The DrawScore function uses OpenGL to display player scores, labels, and victory messages with color changes based on the variable r.

```
void draw(){ glClear(GL_COLOR_BUFFER_BIT);

glBegin(GL_QUADS); settings.DrawField();

ball.draw();

glEnd();

left.draw();

right.draw(); settings.DrawScore(); glutSwapBuffers();

}
```

Chapter 4

SNAPSHOTS

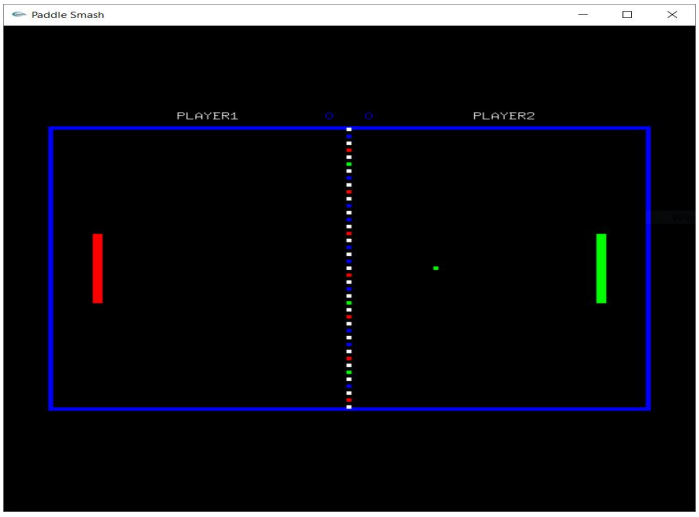


Fig 4.1: Shows the normal gameplay of the program.

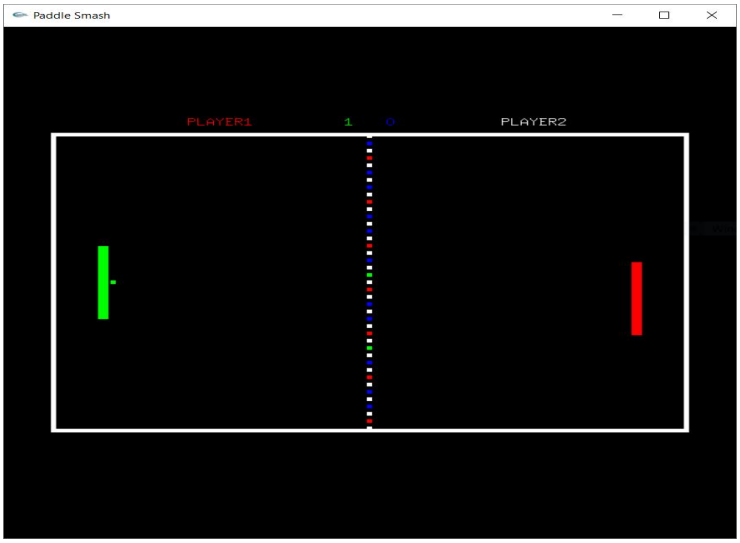


Fig 4.2: Shows the hold position of the ball.

Fig 4.3: Shows the end of the game

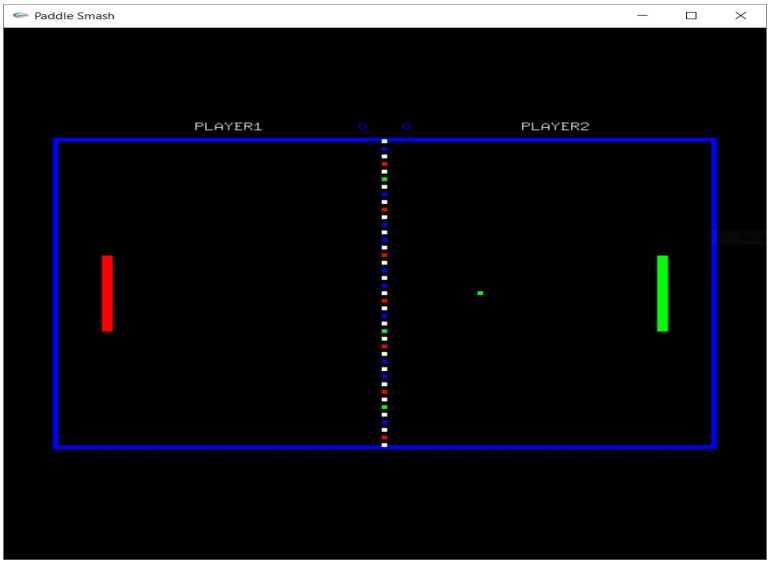


Fig 4.4: Shows the hold position of the ball.

CONCLUSION

Paddle Smash Graphics package has been developed Using OpenGL.The illustration of graphical principles and OpenGl features are included and application program is efficiently developed.

The aim in developing this program was to design a simple program using Open GL application software by applying the skills we learnt in class, and in doing so, to understand the algorithms and the techniques underlying interactive graphics better.

The designed program will incorporate all the basic properties that a simple program must possess.

The program is user friendly as the only skill required in executing this program is the knowledge of graphics.

REFERENCES

Books:

[1]The Red Book –OpenGL Programming Guide,6th edition.

[2]Rost, Randi J. : OpenGL Shading Language, Addison-Wesley

[3]Interactive Computer Graphics-A Top Down Approach Using OpenGL, Edward Angel, Pearson-5th edition.

Websites:

[1]http://www.opengl.org/documentation

[2]http://www.nehe.gamedev.net/lesson.asp?index=01

[3]http://en.wikipedia.org/wiki/OpenGL#Documentation