CHAPTER 1

INTRODUCTION

Computer Graphics is concerned with all aspects of producing pictures or images using a computer- A particular graphics software system called OpenGL, which has become a widely accepted standard for developing graphics applications .

The applications of computer graphics in some of the major areas are as follows

l. Display of information.

2. Design.

3. Simulation and Animation.

4. User interfaces.

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that are used to specify the objects and operations needed to produce interactive applications.

This project named "ROCKET SHOOTER" uses OpenGL software interface and develops a game.

1.1 PROBLEM STATEMENT

Computer graphics is no longer a rarity. It is an integral part of all computer user Interfaces and is indispensable for visualizing 2D, 3D and higher dimensional objects.

In the proposed system, the OpenGL is an graphic software system 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.

OpenGL doesn't provide high-level commands for describing models of three dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, you must build up your desired model from a small set of geometric primitives - points, lines, and polygons.

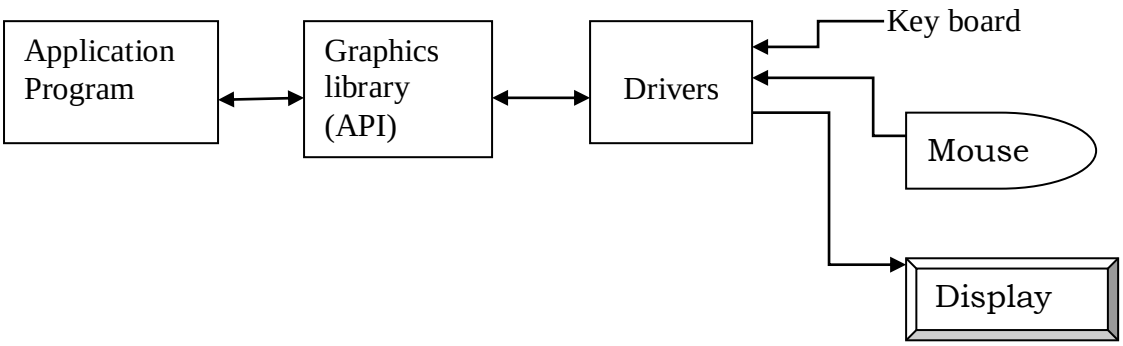


Fig1.1 Application programmers model of graphics system

The interface between an application program and a graphics system can be specified through a set of functions the resides in a graphics library .These specification are called the application programmer's interface (API).The application programmer see only the API and is thus shielded from the details of both the hardware and the software implementation of the graphics library. The software drivers are responsible for interpreting the output of the API and converting this data to a form that is understood by the particular hardware.

1.2 OBJECTIVES OF THE PROJECT

The objective of this project is to demonstrate how this program interacts with the user through keyboard functions, mouse functions and renders the graphic design of shooting rocket and collision functions.

The other objectives of this project are:

- It makes use of interactive programming.
- It provides passive motion callbacks respectively for the *current window.*
- It makes use collision functions to detect the collision between the rockets.

1.3 SCOPE OF THE PROJECT

The application program developed can be used in various fields as follows:

- In the field of gaming by simulating concepts and real world scenarios.
- For modeling space programs to get clear depiction.

1.4 SUMMARY

This chapter deals with the first phase of development of the project by knowing the drawbacks of the existing computer graphics and proposing a new system .To determine the objectives and scope of the project.

CHAPTER -2

LITERATURE SURVEY

The Rocket Shooter is a game that includes the player and the intruder rockets. It consists bullets whose post hit eliminates the intruder rockets .The player will be given points for each hit .The player will be deducted some points for each collision.

2.1 MAIN FEATURES OF THE PROJECT

- The game application is user friendly.
- It serves a gaming and modeling aid.
- Enhanced by the inclusion of user interaction functions.

2.2. TECHNICAL OVERVIEW

Most of our applications will be designed to access OpenGL directly through functions in three libraries. Firstly functions in the main GL library begin with the letters ***gl*** and are stored in the library usually referred to as ***GL*** **.**The second is the OpenGL utility library ***(GLU)*** and the Third is the OpenGL utility toolkit (***GLUT)***, which provides the minimum functionality that should be expected in any modern windowing system.

In GLUT, our application structures use event handling to use callback functions. (This method is similar to using the Xt Toolkit, also known as the X Intrinsics, with a widget set.) For example, first you open a window and register callback routines for specific events. Then, you create a main loop without an exit. In that loop, if an event occurs, its registered callback functions are executed. Upon completion of the callback functions, flow of control is returned to the main loop.

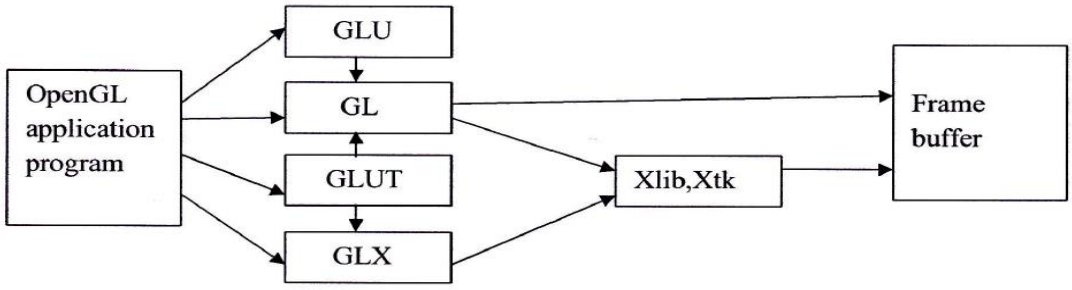


Fig 1.2 :Library organization.

Geometric data (vertices, lines, and polygons) follows the path through the row of boxes that include evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) is treated differently for part of the process. Both types of data undergo the rasterization and per-fragment operations before the final pixel data is written into the frame buffer.

All the data, whether geometry or pixels, can be saved in a display list or processed immediately. When a display list is executed, the data is sent from the display list just as if it were sent by the application.

All geometric primitives are eventually described by vertices. If evaluators are used, that data is converted to vertices and treated as vertices from then on. Vertex data may also be stored in and used from specialized vertex arrays. Per-vertex calculations are performed on each vertex, followed by rasterization to fragments. For pixel data, pixel operations are performed, and the results are either stored in the texture memory, used for polygon stippling, or rasterized to fragments. Finally, the fragments are subjected to a series of per-fragment operations, after which the final pixel values are drawn into the frame buffer.

2.4 SUMMARY :

In this chapter we were briefed about the main features of the project. The technical overview specified the architecture used, the project capabilities and the different modules used.

CHAPTER 3

REQUIREMENT SPECIFICATION

A software requirement definition is an abstract description of the services which the system should provide, and the constraints under which the system must operate. It should only specify the external behavior of the system. The requirements are specified as below:

3.1 FUNCTIONAL REQUIREMENTS:

In software engineering, a functional requirement defines a function of a software system or its component. A function is described as a set of inputs, the behavior, and outputs (see also software). Functional requirements may be calculations, technical details, data manipulation and processing and other specific functionality that define what a system is supposed to accomplish. Behavioral requirements describing all the cases where the system uses the functional requirements are captured in use cases.

The various methods used in this project are as follows:-

Init - The module sets the initial OpenGL variables.

Display- The module draws the output on the screen and the functions in it

Keyboard Movement- The module specifies the user keyboard interaction.

Special Function- It sets the special keyboard callback for the *current window*.

3.2 NON-FUNCTIONAL REQUIREMENTS:

These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Nonfunctional requirements often apply to the system as a whole. The Non-Functional Requirements are as follows:-

3.2.1 Dependability

The dependability of a computer system is a property of the system that equates to its trustworthiness. Trustworthiness essentially means the degree of user confidence that the system will operate as they expect and that the system will not 'fail' in normal use.

3.2.2 Availability

It is the ability of the system to deliver services when requested. There is no error in the program while executing the program.

3.2.3 Reliability

The ability of the system to deliver services as specified. The program is compatible with all types of operating system without any failure.

3.2.4 Safety

It is the ability of the system to operate without catastrophic failure. This program is user friendly and it will never effects the system

3.2.5 Security

It is the ability of the system to protect itself against accidental or deliberate intrusion.

3.3 DETAILS OF THE SOFTWARE

Here, the coding of the project is done in Code Blocks 17.12 which is a commercial integrated development environment (IDE) with OpenGL (Open Graphics Library) which is a standard specification to produce 2D and 3D computer graphics.
The OpenGL Utility Toolkit called GLUT which is a library of utilities for OpenGL programs is also used.

3.3.1 Code Blocks 17.12

Code::Blocks is a *free C, C++ and Fortran IDE* built to meet the most demanding needs of its users. It is designed to be very extensible and fully configurable. Built around a plugin framework, Code::Blocks can be *extended with plugins*. Any kind of functionality can be added by installing/coding a plugin. For instance, compiling and debugging functionality is already provided by plugins!

3.3.2 OpenGL and GLUT

OpenGL (Open Graphics Library) is a standard specification defining a cross language, cross-platform API for writing applications that produce 2D and 3D computer graphics, describing a set of functions and the precise behaviors that they must perform. From this specification, hardware vendors create implementations - libraries of functions created to match the functions stated in the OpenGL specification, making use of hardware acceleration where possible. Hardware vendors have to meet specific tests to be able to qualify their implementation as an OpenGL implementation.
GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing Programs in OpenGL. It implements a simple windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL programming. GLUT provides a portable API so you can write a

single OpenGL program that works across all PC and workstation OS platforms.

3.4 SOFTWARE REQUIREMENTS

- OpenGL(Graphics Library Utility Toolkit)
- Code::Blocks 17.12
- Operating System- Windows XP,8,8.1,10

3.5 HARDWARE REQUIREMENTS:

- Hard Disk - 32 GB and Above
- Memory - 256 MB and Above
- Keyboard
- Visual Display Unit

CHAPTER 4

DESIGN

Graphics systems used general - purpose computers with the standard von Neumann architecture. Such computers are characterized by a single processing unit that processes a single instruction at a time. Information had to be sent to the display at a rate high enough to avoid flicker on the display. In the early days of computer graphics, computers were so slow that refreshing even simple images, containing a few hundred line segments, would burden an expensive computer.

The following diagram shows the Henry ford assembly line approach which OpenGL takes to process data.

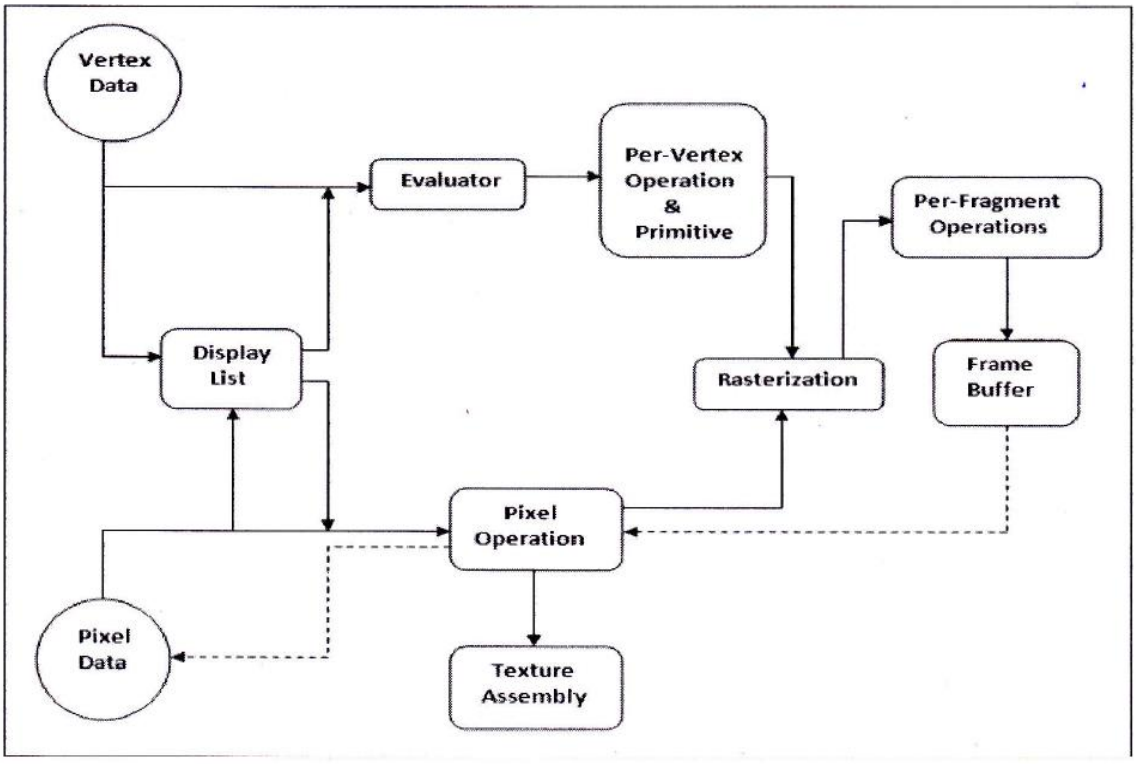


Fig 4.1:Henry Ford Assembly Line Approach

4.1 Display Processors:

Built special-purpose graphics systems were concerned primarily with relieving the general purpose computer from the task of refreshing the display continuously. These display processes had conventional architectures but included instructions to display primitives on the CRT. The main advantage of the display processor was that the instructions to generate the image could be assembled once in the host and sent to the display processor, where they were stored in the display processor's own memory as a display list, or display file. The display processor would then repetitively execute the program in the display list, at a rate sufficient to avoid flickering independently of the host, thus freeing the host for other tasks. This architecture has become closely associated with the client-server architectures.

4.2 Pipeline Architectures:

The major advances in graphics architectures closely parallel the advances in work stations. In both cases, the ability to create special-purpose VLSI chops was the key enabling technology development. In addition, the availability of inexpensive solid state memory led to the universality of raster displays. For computer-graphics applications, the most important use of customs VLSI circuits has been in creating pipeline architectures.

4.3 The graphics pipeline:

3 major steps in the imaging process:

- **Vertex processing**
- **Clipping and primitive assembly**
- **Fragment processing**

4.3.1 Vertex Processing:

In the first block of our pipeline, each vertex is processed independently. The two major functions of the block are to carry out coordinate transformations and to compute a color for each vertex. The assignment of vertex colors can be as simple as the program specifying a color or as complex as the computation of a color from a physically realistic lighting model that incorporates the surface properties of the object and the characteristic light sources in the scent.

4.3.2 Clipping and primitive assembly:

The second fundamental block in the implementation of the standard graphics pipeline is for clipping and primitive assembly. We must do clipping because of the limitation that no imaging system can see the whole world at once. The human retina has a limited size corresponding to an approximately 9O-degree field of view. Cameras have a film of limited size, and we can adjust their fields of view by selecting different lenses.

4.3.3 Rasterization.

The primitives that emerge from the clipper are still represented in terms of their vertices and must be further processed to generate pixels in the frame buffer. For example, if three vertices specify a triangle filled with a solid color, the raster must determine which pixels in the frame buffer are inside the polygon. The output of rasterization is a set fragment for each primitive. A fragment can be thought of as a potential pixel that carries with it information, including its color and location, that is used to update the corresponding pixel in the frame buffer. Fragment can also carry along depth information that allows later stages to determine if a particular fragment lies behind other previously rasterized fragments for a given pixel.

4.3.4 Fragment Processing:

The final block in our pipeline takes in the fragments generated by the rasterizer and updates the pixels in the frame buffer. If the application generated three-dimensional data, some fragments may not be visible because the surfaces that they define are behind other surfaces. The color of the pixel that corresponds to a fragment can also be read from buffer and blended with the fragment's color to create translucent effects.

4.4 CONTROLS

The controls provided in the project are given below:

KEYBOARD CONTROLS

- 'left arrow'- move left.
- 'right arrow'-move right.
- 'space bar'-fire.
- 'r'-restart

CHAPTER 5

IMPLEMENTATION

- **glloadldentity**

  Syntax: glloadldentity();

  Purpose: It loads the identity matrix.

- **glMatrixMode**

  Syntax: a) glMatrixMode(GL_PROJECTION);

  b) glMatrixMode(GL_MODEL_VIEW);

Purpose: It helps in selecting the matrix(model view or projection matrix) to which the operations apply by setting the matrix mode a variable that is set to one type of matrix and also part of the state.

- **glOrtho**

Syntax: glorthd Gldouble left, Gldouble right, Gldouble bottom, Gldlouble top, Gldouble near, Gldouble far);
Purpose: It gives an orthographic projection with a right parallelepiped viewing volume. All the parameters me distances measured from the camera.

- **glBegin**

Syntax: glBegin(glEnum mode);
Purpose: Initiates a new primitive of type mode and starts the collection of vertices. Values of mode include GL_POINTS, GL_LINES and GL_POLYGON.

- **glEnd**

Syntax: glEnd0;
Purpose: Terminates a list of vertices.

- **glutlnit**

Syntax: glutlnit(int *argc, char ** argv);
Purpose: Initializes GLUT. The arguments from main are passed In and can be used by the application.

- **glVertex**

Syntax:a)glVertex[23a][sifd(TYPE xcoordinate, TYPE ycoordinate, .);
b) glVertex[B4][sifd]v(TYPE *coordinate);

Purpose: Specifies the position of a vertex inZ,3,or 4 dimensions. The coordinates can be specifie as short s, int i, float f or double d. If the v is presant, the argument is a pointer to an array containing the coordinates.

- **glutCreateWindow**

Syntax: glutCreateWindow(char * title);

Purpose: It creates a window on the display. The string title can be used to label the window.

- **glutlnitDisplayMode**

Syntax: glutlnitDisplayMode(unsigned int mode);

Purpose: It requests a display with the properties in mode. The value of mode is determined by the logical OR of options including the color model(GLUT_RGB, GLUT_INDEX) and buffering(GLUT_SINGLE, GLUT_DOUBLE).

- **glutMainloop**

Syntax: glutMainlLoop0;

Purpose: It causes the program to enter an event processing loop. It should be the last statement in main.

- **glutKeyboardFunc**

Syntax: glutKepoardFunc(void *(char key, int width, int height)

Purpose: Registers the keyboard callback function f. The callback function returns the ASCtr code of the key pressed and the position of the mouse.

- **glutSpecialFunc**

Syntax: glutSpecialFunc(void(*f)(unsigrred char key,int x, int y))

Purpose: Registers the mottion keys such as (->,<-)callback function The position of the mouse is retumed by the callback when the mouse is moved at least one of thesekeys pressed.

- **glEnable**

Syntax :glEnable(Glenum feature)

Purpose: Enables an OpenGL feature. Features that can be enabled include GL DEPTHJEST, GL-LIGHTING, GI LIGHTi, GL_LINE_SMOOTH, GL_BLEND, GL_NORMALVE etc.

- **glutMotionFunc**

Syntax: void glutMotionFunc(void(*func)(int x,int y));

Purpose: The motion callback for a window is called when the mouse moves within the window while one or more mouse buttons are used.

- **glColor**

Syntax: glColor3[b i f d ub us ui](TYPE r, TYPE g, TYPE b);

Purpose:Sets the present RGB colors. Valid types are byte(b), int(i), float(f), double(d), unsigrred byte(ub), unsigrred short(us), and unsigned int(ui).

- **glRasterPos**

Syntax: glRasterPos[234]sifd(TYPE xcoord, TYPE ycoord, - . ..);
Purpose: Specifi es a raster position.

- **glNewList**

Syntax: void glNewList(GLuint list,GLenum mode);
Purpose: Specifies the display list name, the compilation mode to be used.

- **GlutSwapBuffers**

Syntax : glut SwapBuffers0 ;
Purpose: Swaps the front and back buffers.

- **glFlush**

Syntax: glFlush0;
Purpose: Forces any buffered OpenGL commands to execute.

- **glutPostRedisplay**

Syntax : glutPostRedisplayQ
Purpose: Requests that the display callback be executed after the current callback returns.

- **gluOrtho2D**

Syntax: gluOrtho2D(Gldouble left, Glxlouble right, Gldouble bottom, Glclouble top);

Purpose: Defines a two dimensional viewing rectangle in the plane Z=0

.

- **glutKeyboardFunc**

Syntax: glutKeyboardFunc(void *f(char key, int width, int height);

Purpose: Registers the keyboard callback function f. The callback function returns the ASCII code of the key pressed and the position of the, mouse.

5.1: Source Code:

```
#include<GL/glut.h>
#include<stdlib.h>
#include<iostream.h>
#include<math.h>
#include<string.h>
using namespace std;
#define n 600
int counter=0;
int gamestate=0;
int Health=50;
int c=0;
string convertInt(int number)
{
stringstream ss;

ss << number;
return ss.str();
}
string convertHealth(int number)
{
stringstream ss;//create a stringstream
```

```
Health=Health-1;
ss << Health;//add number to the stream
return ss.str();//return a string with the contents of the stream
}


int NOB;              //Number of Bullets
int NOF=4;            //Number of enemies per frame
int enemyX[481];      //Generates Random X positions for X

class enemy           //enemy class
{
public:
double x;         //Position initial position of x coordinate of enemy spaceship
double y;         // initial position of y  coordinate of enemy spaceship
int alive;        // this variable will whether the enemy is alive not not 0 means alive 1
means dead
double x2;        //this the same x2 coordinate as we have declared earlier but this one
will be retrieved for collision detection
double y2;        //same as the above x2 t
double w2;        //width of the object
double h2;        //height of the object
enemy()           // constructor for enemy
{
alive=1;      // we will make the alive  variable 1


}
void getcollsioninformation(){  // this function send information about current position

x2=x-10;
w2=40;
y2=y+40;
h2=50;

}

void init()
{
x=enemyX[rand()%481];
y=500;
alive=1;
}

void draw()           //this draws the enemy
{
glColor3f(0.9,0.91,0.98);
glBegin(GL_QUADS);
```

```
glVertex2f(x,y);

glVertex2f(x+20,y);       //BODY
glVertex2f(x+20,y+40);
glVertex2f(x,y+40);

glEnd();
glBegin(GL_QUADS);
glVertex2f(x-10,y+15);  //wing 1
glVertex2f(x,y+15);
glVertex2f(x,y+15-5);
glVertex2f(x-10,y+15-5);
glEnd();
glBegin(GL_QUADS);
glVertex2f(x+30-10,y+15);
glVertex2f(x+40-10,y+15);  // wing2
glVertex2f(x+40-10,y+10);
glVertex2f(x+30-10,y+10);
glEnd();

glColor3f(0,0,1);


glBegin(GL_POLYGON);
glVertex2f(x,y);

glVertex2f(x+10,y-10);  // warhead
//glVertex2f(x+10,y-10);
glVertex2f(x+20,y);

glEnd();


}
void move(float offset) //this function will be descend the enemy according a give speed
which is offset in this case
{
y=y-offset;
}

};
```

class star //start object we have made start with glpoints

```
{
public:

double x;               //x and y coordinates of start
double y;
void move()             // since stars will be falling y has to be decreased
{
y--;
}
void show()     // draws the start
{
//glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glPointSize(1);
glColor3f(1,1,1);
glBegin(GL_POINTS);
glVertex2f(x,y);
glEnd();
//glFlush();
}


};

class myship            //class for spaceship
{
public:
double x;       //x and y coordinates
double y;
int shoot;
int alive;    // checks whether alive of not
double x1;
double y1;
double w1;
double h1;
myship()        //constructor
{
x=250;       //initial position
y=40;
shoot=0;
alive=1;      //alive or not
}
void information_for_collision(){   // info about collision


x1=x-10;
y1=0;
w1=50;
h1=60;
```

```
}

void move_left(int offset)     //moves the object left
{
x=x-offset;
}
void move_right(int offset)    //right
{
x=x+offset;
}


void displayShip()   //draws the ship
{

glColor3f(1,1,1);
glBegin(GL_QUADS);
glVertex2f(x,0);
glVertex2f(x+30,0);
glVertex2f(x+30,0+40);  //body
glVertex2f(x,0+40);
glEnd();

glBegin(GL_QUADS);
glVertex2f(x-10,5);  // the left wing
glVertex2f(x,5);
glVertex2f(x,20);
glVertex2f(x-10,20);
glEnd();

glBegin(GL_QUADS);
glVertex2f(x+30,5);
glVertex2f(x+40,5); // the right wing
glVertex2f(x+40,20);
glVertex2f(x+30,20);
glEnd();


glColor3f(1,0,0);
glBegin(GL_TRIANGLES);
glVertex2f(x,0+40);
glVertex2f(x+30,0+40); // warhead
glVertex2f(x+15,0+60);

glEnd();

}
```

```
void Constructor() //resets the ship object
{
x=250;
y=40;
shoot=0;
alive=1;
}
};

class bullet
{
public:
double x;
double y;
int firing;
double x3;
double y3;
double w3;
double h3;

bullet()
{
firing=0;
}
void getPosition(myship ship)   // takes the position of ship
{
x=ship.x+15;
y=ship.y+35;
}
void fire()
{
firing=1;
}
void draw()            //draws the bullet
{
glColor3f(1,0,0);
glLineWidth(3);
glBegin(GL_LINES);
glVertex2f(x,y);
glVertex2f(x,y+10);
glEnd();
}
void move(int offset)   //ascends the bullet
{
y=y+offset;
}
void reinit()        //initialize
{
```

```
firing=0;
}
};
bullet b[n];       //instance of bullet object

star s[n];        //start
void showstars()  //renders the start;
{
int i;

for(i=0;i<n;i++)
{
if(s[i].y >= 0)
{
s[i].show();    //render each object
s[i].move();    //moves starts
}
else
{
s[i].y=500;     //initial y position 500
s[i].x=rand()%500; //initial x position
}
}

}

myship ship;

void FireBulletsIfShot()
{
if(ship.shoot)              //when ship.ship=1
{
b[NOB-1].fire();       //sets firing of bullet into 1
b[NOB-1].getPosition(ship); //collects the current x position of ship
ship.shoot=0;         //sets the shoot variable of ship into 0
}
}
void drawship()       //renders the ship object
{
if(ship.alive) //as long as the ship is alive
{
ship.displayShip(); //it will render the ship object
}

FireBulletsIfShot(); //if ship.shot is 1 it sets the bullet ready to shot

}
```

```
//////////////////////////////////////////////Move Object with mouse////////////////////////////////////////

void move(int x, int y)  //takes the current position of mouse and sets the ship according
to that
{
ship.x=x;
glutPostRedisplay();
}
int bulletspeed=26;  //this value will be added to the y of bullet
void drawbullet()    //renders bullet
{
int i;

for(i=0;i<NOB;i++)
{
if(b[i].firing)
{
b[i].draw(); //renders
b[i].move(bulletspeed); //move
}
if(b[i].y > 500)
{
b[i].reinit();  //resets the bullet object when it goes beyond the screen

}
}
if(NOB>30)  // number of bullets can never go beyond 30
{
NOB=0;
}
}
enemy e[4];         //enemy object
int enemyspeed=5;   // speed at which enemy will fall
void drawenemy()
{

int i;
for(i=0;i<NOF;i++)
{
if(e[i].alive)  //as long as the enemy is alive  it will be rendered
{
e[i].draw();            //render the each enemy
e[i].move(enemyspeed);  //enemies will fall at this speed
if(e[i].y -10 < 0)  //when the y coordinate of y will be 0 new enemies will be initialized
```

```
{
e[i].init();  //initialize
}
}
if(e[i].alive==0) //if the current enemy is dead it will also be initialized
{

e[i].init();
}
}
}

//////////////////////Collision detection of enemy with space ship////////////////////////////////
void collisionship(){

for(int i=0;i<NOF;i++){

e[i].getcollsioninformation();
ship.information_for_collision();

if((ship.x1<(e[i].x2+e[i].w2)) &&(e[i].x2<ship.x1+ship.w1)
&&((ship.y1+ship.h1==e[i].y2-e[i].h2))){//|| ((ship.x1+ship.w1)>e[i].x2) &&
(ship.y1+ship.h1>e[i].y2+e[i].h2))){

Health=Health-5;
}

}

}
int sco=0;
//////////////////////////////////////Collision detection for bullet and enemy//////////////////////
void BulletsVsEnemyCollisionTest()
{
int i;
int j;
for(i=0;i<NOB;i++)
{

for(j=0;j<NOF;j++){
e[j].getcollsioninformation();
if(e[j].x2<=b[i].x && b[i].x <= (e[j].x2+e[j].w2) && e[j].alive)
{

e[j].alive=0;
b[i].firing=0;
b[i].x=0;
b[i].y=0;
```

```
sco +=1;

}
}
}

}


//////////////////displays text////////////////////////////////
void rendertext(float x,float y, string strings){

glColor3d(1,0,0);

glRasterPos2d(x,y);
glDisable(GL_TEXTURE);
glDisable(GL_TEXTURE_2D);
int l=strings.length();
for( int i=0;i<=l;i++){
glutBitmapCharacter(GLUT_BITMAP_9_BY_15,(int)strings[i]);


}
glEnable(GL_TEXTURE);
glEnable(GL_TEXTURE_2D);

}
void overdisplay()              //displays text when game is over
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
string text1="GAME OVER";
string text2="You Scored ";
string text3="Press r to restart";

string temp=convertInt(sco);

glColor3f(1,0,0);
glRasterPos2f(180,250);
int i;
int l=text1.length();
for(i=0;i<=l;i++){
glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24,(int)text1[i]);

}
glRasterPos2f(180,150);
l=text2.length();
for(i=0;i<=l;i++){
glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24,(int)text2[i]);
```

```
}
l=temp.length();
for(i=0;i<=l;i++){
glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24,(int)temp[i]);
}
glRasterPos2f(180,130);
l=text3.length();
for(i=0;i<=l;i++){
glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24,(int)text3[i]);
}
glFlush();
glutSwapBuffers();
}

void display()
{
start:
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
showstars();
drawship();
drawenemy();
drawbullet();
ship.information_for_collision();

BulletsVsEnemyCollisionTest();
collisionship();
char score[10]={0};
string sf="Score :"+ convertInt(sco);
string ss="Health :"+ convertInt(Health);
rendertext(10,480,ss);
rendertext(10,460,sf);

if(Health<=0){
ship.alive=0;

gamestate=1;


}
glFlush();
system("sleep 0.00001");
glutSwapBuffers();
glutPostRedisplay();
}

//////////////////////////////////////////Reset  All the function////////////////////////////////////////////////

void Reinitialization()
```

```
{

ship.Constructor();
Health=100;
sco=0;
int i;
NOF=4;
for(i=0;i<NOF;i++)
{
e[i].init();
}


return;
}
//////////////////////////////////////////Keyboard//////////////////////////////////////////
void keyboard(unsigned char key, int x, int y)
{
switch(key)
{
//case 32:
case 32:          if(ship.alive)
{ship.shoot=1;
NOB++;}
break;
case 'D':
case 'd':
break;
case 'A':
case 'a':
break;
case 'R':
case 'r': if(ship.alive==0)
{
Reinitialization();
gamestate=0;
}
break;

}
glutPostRedisplay();
}
void keyboard(int button, int x, int y)
{
if (button == GLUT_KEY_RIGHT)
{
ship.x=ship.x+10;
```

```
}


else if (button == GLUT_KEY_LEFT)
{
ship.x=ship.x-10;
}
//update display
glutPostRedisplay();
}
////////////////////////////////////////////////////Main Display Function//////////////////////////////////////
void loop()
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
switch(gamestate)
{
case 0: display();
break;
case 1: overdisplay();
break;
}
}
void myinit()
{
int i;
int inc=10;
for(i=0;i<n;i++)
{
s[i].x=rand()%500;
s[i].y=rand()%500;
}


for(i=0;i<481;i++)
{
enemyX[i]=inc;
inc++;
}

glClearColor(0.0, 0.0, 0.0, 0.0);
glPointSize(4.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 500.0, 0.0, 500.0);
}

int main(int argc, char** argv)
{
```

```
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(500,500);
glutInitWindowPosition(500,100);
glutCreateWindow("Space Shooter");
glutDisplayFunc(loop);
glutPassiveMotionFunc(move);//void glutPassiveMotionFunc(void (*func)(int x, int
y));
glutKeyboardFunc(keyboard);
glutSpecialFunc(keyboard);//void glutSpecialFunc(void (*func)(int key, int x, int y));

myinit();
glutMainLoop();
}
```

CHAPTER 6

TESTING

Once source code has been generated, software must be tested to uncover as many errors as possible before delivery to the customer. Software testing is a critical element of software quality assurance and represents the ultimate review of specification design and code generation.

6.1 TEST PLANS

In this test plan all major activities are Unit test, Integration test, Validation test, System test.

6.1.1 Unit Testing

Unit testing focuses verification effort on the unit of software design or module. using the unit test plans, prepared in the design phase of the system development as a guide, important control paths are tested to uncover efforts within the boundary of the modules. The inter phase of each of the module were tested to ensure proper flow of the information into and out of the modules under the consideration.

Table 6.1.1 Test of source code

S1# Test Case:	**1**
Name of Test:	**Test of Source code**
Item being tested:	**Source code**
Expected Output:	**Rocket Shooter**
Actual Output:	**Rocket Shooter**
Remarks:	**Pass**

6.1.2 Integration testing

Data can be lost across an interface: a module and sub-functions, when combined may not produce the desired Can present problems. have an adverse effect on another's major function; global data structures can present problems

Table 6.1.2 Source code compilation

S1# Test Case:	**2**
Name of Test:	**Compilation of Source Code**
Item being tested:	**Source code**
Expected Output:	**Executed Command Prompt**
Actual Output:	**Executed Command Prompt**
Remarks:	**Pass**

6.1.3 Validation testing

The whole of validation testing is to expose latent defects in a software system before the system is delivered. This contrasts with validation testing which is intended to demonstrate that a system meets its specification. Validation testing requires the system to perform correctly using given acceptance test cases.

Table 6.1.3 Keyboard function

S1# Test Case:	**3**
Name of Test:	**Keyboard Function**
Item being tested:	**Space Bar**
Expected Output:	**Corresponding action to be executed**
Actual Output:	**Corresponding action to be executed**
Remarks:	**Pass**

6.1.4 System Testing

Creation of application .Expected output rocketshooter.exe with 0 error, 0 warning. Actual output pendulum.exe with 0 error,0 warning.

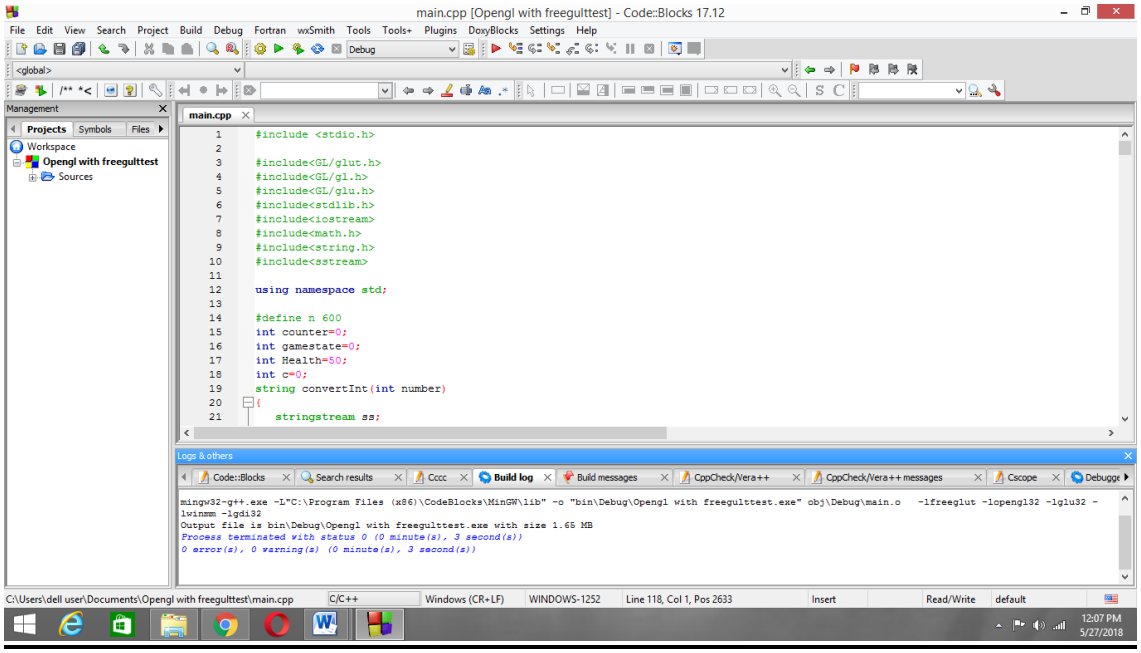
Table 6.1.4 Creation of application

S1# Test Case:	**4**
Name of Test:	**Creation of Application**
Item being tested:	**Source Code**
Expected Output:	**rocketshooter with 0error and 0 warning**
Actual Output:	**rocketshooter with 0error and 0 warning**
Remarks:	**Pass**

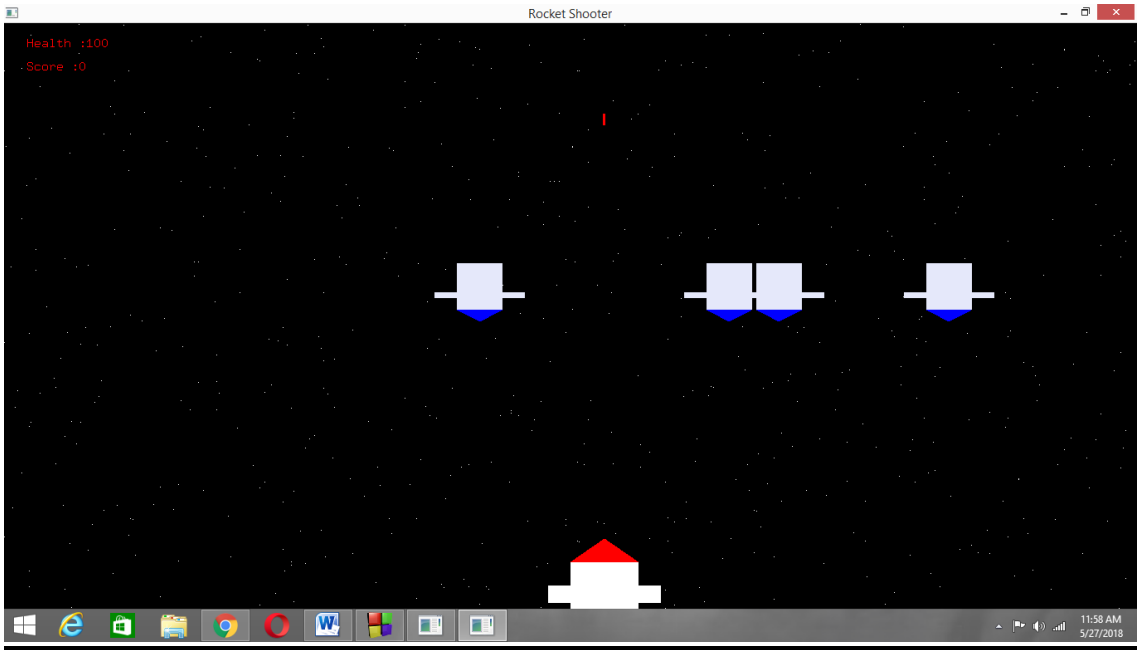
CHAPTER 7

SCREEN SHOTS

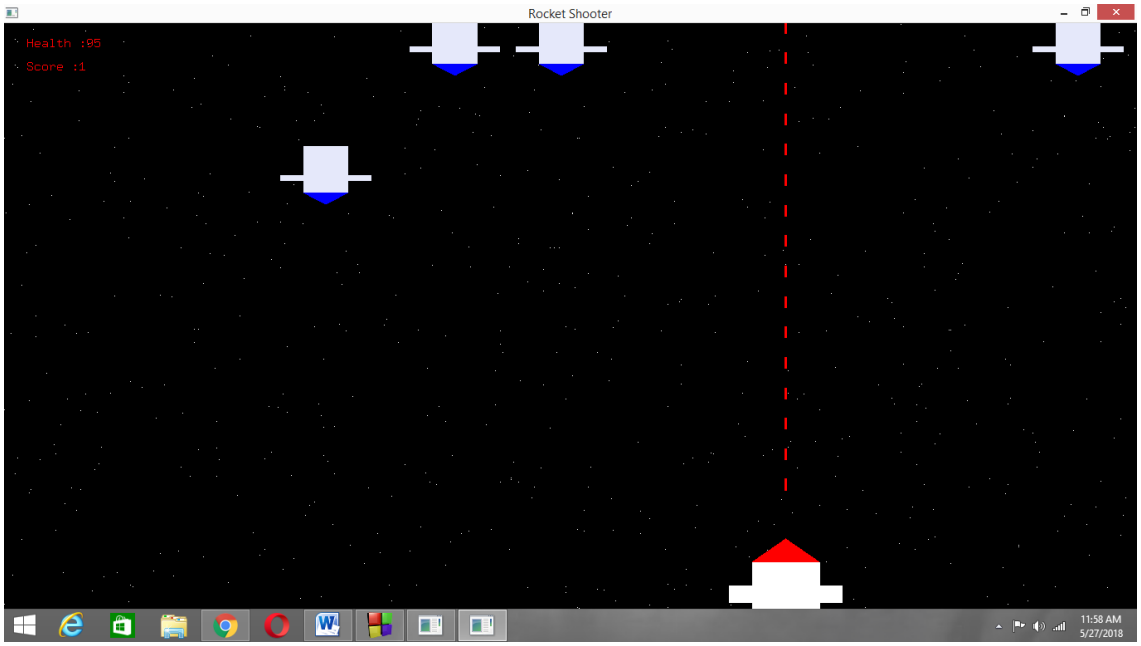
7.1.1 Application Build



7.1.2 Game in Progress



7.1.3 Rocket Firing



7.1.4 Game Over

CHAPTER 8

CONCLUSION

In this project, the design and implementation of 2-dimensional graphics Interfacing program has been attempted. We have seen the functions performed by this application and code for the same. In OpenGL, C/C++ provide enormous flexibility in the design and the use of C/C++ graphics programs.

The presence of many in-built OpenGL functions and libraries to take care of many of the functionalities reduces the burden of coding and makes the implementation simpler.

The project started with the designing phase in which we figured the requirements needed the system design, Data flow diagram etc. Then comes the details of the implementation phase where in we have included various functionalities. And now after the testing phase, the project comes to an end.

.

BIBLIOGRAPHY

- During the course of this project reference to the following books and materials were made:

[1] Edward Angel, *"Interactive Computer Graphics"*, Pearson Publication, 5th edition

[2] F.S. Hill Jr,"Computer Graphics Using OpenGL",Pearson Publication,2nd edition

[3] James D. Foley,"Computer Graphics",Addision Wesley,1997

- Internet was extensively browsed for various materials related to this project

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

[2] http://www.cs.usr.edu

[3] http://math.ucsd.edu/~sbuss/MathCG/OpenGLsoft