CHAPTER 1

INTRODUCTION

Computer graphics is concerned with all aspects of producing images using a computer. The field began humbly 50 years ago, with the display of few lines on a cathode ray tube (CRT); now, we can create images by computer that are indistinguishable from photographs of real objects.

In this chapter we discuss about the application of computer graphics, overview of the graphic system, graphics architectures. In this project we discuss about OpenGL- open graphics library API- which is used to develop the application program which is the matter in question.

APPLICATIONS OF COMPUTER GRAPHICS

The applications of computer graphics are many and varied. However we can divide them into four major areas:

- Display of Information
- Design
- Simulation and Animation
- User interfaces

Display of Information

Classical graphics techniques arose as a medium to convey information among people. Although spoken and written languages serve this purpose, images are easier to communicate with and hence the computer graphics plays an important role.

Information visualization is becoming increasingly important in the field of security, medicine weather forecasting and mapping. Medical imaging like CT, MRI, ultrasound scan are produced by medical imaging systems but are displayed by computer graphics system.

Design

Professions such as engineering and architecture are concerned with design which is an iterative process. The power of the paradigm of humans interacting with the computer displays was recognized by Ivan Sutherland.

Simulation and Animation

Graphics system, now are capable of generating sophisticated images in real time. Hence engineers and researchers use them as simulators. The most important use has been in the training of pilots.

User interfaces

User interaction with computers has become dominated by a visual paradigm that includes windows, icons, menus and pointing devices. Applications such as office suites, web browsers are accustomed to this style.

1.1 PROBLEM STATEMENT

To design and implement a helicopter game using OpenGL.

1.2 OBJECTIVES

In this project we are going to design a Helicopter game using OpenGL. This project displays a helicopter in flight with its fan and rotor stabilizer moving.

The basic idea of the game is to dodge the rectangles and the game ends when the helicopter touches one of the rectangles.

The score is calculated based on the distance covered by the helicopter before the crash.

1.3 SCOPE

It can be used for gaming and entertainment purpose.It can also be extended to the development of android application.

CHAPTER 2

LITERATURE SURVEY

People use the term “computer graphics” to mean different things in different context. Computer graphics are pictures that are generated by a computer. Everywhere you look today, you can find examples, especially in magazines and on television. Some images look so natural you can’t distinguish them from photographs of a real scene. Others have an artificial look, intended to achieve some visual effects.

There are several ways in which the graphics generated by the program can be delivered.

- Frame- by- frame: A single frame can be drawn while the user waits.
- Frame-by-frame under control of the user: A sequence of frames can be drawn, as in a corporate power point presentation; the user presses to move on to the next slide, but otherwise has no way of interacting with the slides
- Animation: A sequence of frames proceeds at a particular rate while the user watches with delight.
- Interactive program: An interactive graphics presentation is watched, where the user controls the flow from one frame to another using an input device such as a mouse or keyboard, in a manner that was unpredictable at the time the program was written. This can delight the eye.

A GRAPHICS SYSTEM

A computer graphics system is a computer system which has five major elements:

1. Input devices
2. Processor
3. Memory
4. Frame buffer
5. Output devices

This model is general enough to include workstations and personal computers, interactive game systems, and sophisticated image-generating systems. Although all the

components, with the exception of the frame buffer, are present in standard computer, it is the way each element is specialized for computer graphics that characteristics this diagram as a portrait of graphics system.

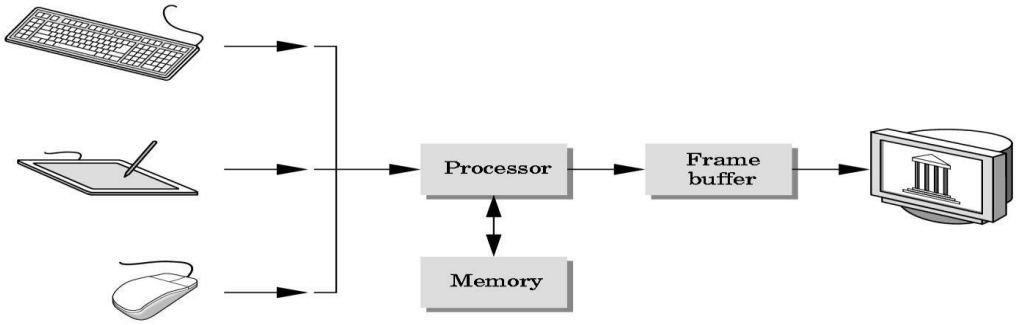

Fig 1.1: A Graphics System

GRAPHICS ARCHITECTURES

On one side of the API is the application program. On the other side is some combination of hardware and software that implements the functionality of the API. There are various approaches to developing architectures to support graphics API as discussed below.

Display Processors

Display processors had a conventional architecture that relived the general-purpose computer from the task of refreshing the screen. These display processors included instructions to display primitives on the display like CRT. The main advantage of display processor was that the instructions to generate the image could be assembled once in the host and sent to display processor where they were stored in display processor's own memory as display list. Then the display processor would execute these programs in display list repetitively, at a rate sufficient to avoid flicker.

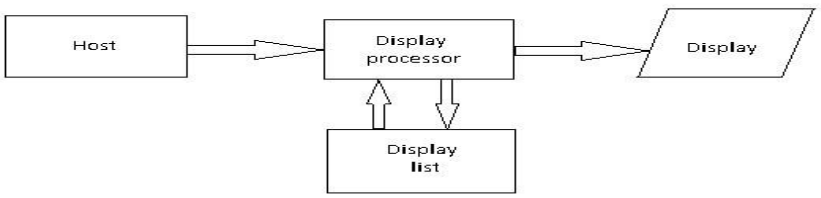

Fig 1.2: Display processor Architecture

Pipeline Architecture

Pipeline architecture is the process of giving output of one process as the input to other. This architecture is prominently used in modern days with the advancement of VLSI. In graphics system; we start with a set of objects. Each object comprises a set of graphical primitives. Each primitive comprises a set of vertices. We can think of the collection of primitive types and vertices as defining the geometry of the scene. Thus the graphics pipeline has four major steps for processing the image:

1. Vertex processing
2. Clipping and primitive assembly
3. Rasterization
4. Fragment processing

Vertex Processing

Vertex processing is the first step to be performed in the pipeline architecture. Vertex Processing mainly involves the transformation of objects, transformation of the coordinate system and projection transforming. It also involves color computation for each vertex.

Clipping and Primitive Assembly

Clipping and Primitive Assembly is the second step in the pipeline architecture. The output of the vertex processor is given as the input to this stage. This step is mainly involved in the clipping of the primitive; Clipping is necessary since the camera film has a limited size and hence cannot image the whole world at once. This stage computes the clipping volume and considers only those vertices that falls within this volume for image formation. Those vertices that are outside this volume do not appear in the image and are said to be clipped.

Rasterization

Rasterization is the third step of pipeline architecture. This step is necessary because the primitives that emerge out of clipper are still represented in terms of vertices. Rasterizer would further process there vertices to generate pixels in the frame buffer. The Rasterizer determines which pixels in the frame buffer are inside the polygon and which pixels fall outside.

Fragment Processing

Fragment Processing is the final step of the pipeline architecture. In the output generated by the rasterizer, some portions may not be actually visible since it may be present behind another primitive (object).

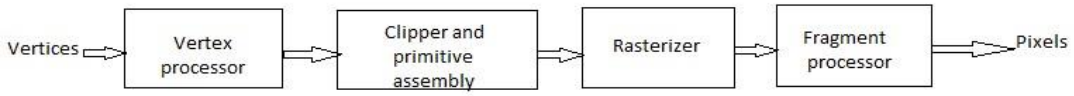

Fig 2.1: Graphics Pipeline Architecture

2.1 HISTORY

OpenGL was developed by **'Silicon Graphics Inc'**(SGI) on 1992 and is popular in the gaming industry where it competes with the Direct3D in the Microsoft Windows platform. OpenGL is broadly used in CAD (Computer Aided Design), virtual reality, scientific visualization, information visualization, flight simulation and video games development.

OpenGL is a standard specification that defines an API that is multi-language and multi-platform and that enables the codification of applications that output computerized graphics in 2D and 3D.

The interface consists in more than 250 different functions, which can be used to draw complex tridimensional scenes with simple primitives. It consists of many functions that help to create a real world object and a particular existence for an object can be given.

2.2 CHARACTERISTICS

- OpenGL is a better documented API.
- OpenGL is also a cleaner API and much easier to learn and program.
- OpenGL has the best demonstrated 3D performance for any API.
- Microsoft's Direct3D group is already planning a major API change called Direct Primitive that will leave any existing investment in learning Direct3D immediate mode largely obsolete.

2.3 COMPUTER GRAPHICS LIBRARY ORGANISATION

OpenGL stands for Open Source Graphics Library. Graphics Library is a collection of APIs (Application Programming Interfaces).

Graphics Library functions are divided in three libraries. They are as follows-

i. GL Library (OpenGL in Windows)
ii. GLU (OpenGL Utility Library)
iii. GLUT (OpenGL Utility Toolkit)

Functions in main GL library name function names that begin with the letter ‘gl’.

- GLU library uses only GL functions but contains code for creating objects and simplify viewing.
- To interface with the window system and to get input from external devices GLUT library is used, which is a combination of three libraries GLX for X windows, ‘wgl’ for Windows and ‘agl’ for Macintosh.
- These libraries are included in the application program using preprocessor directives. E.g.: #include<GL/glut.h>
- The following figure shows the library organization in OpenGL.

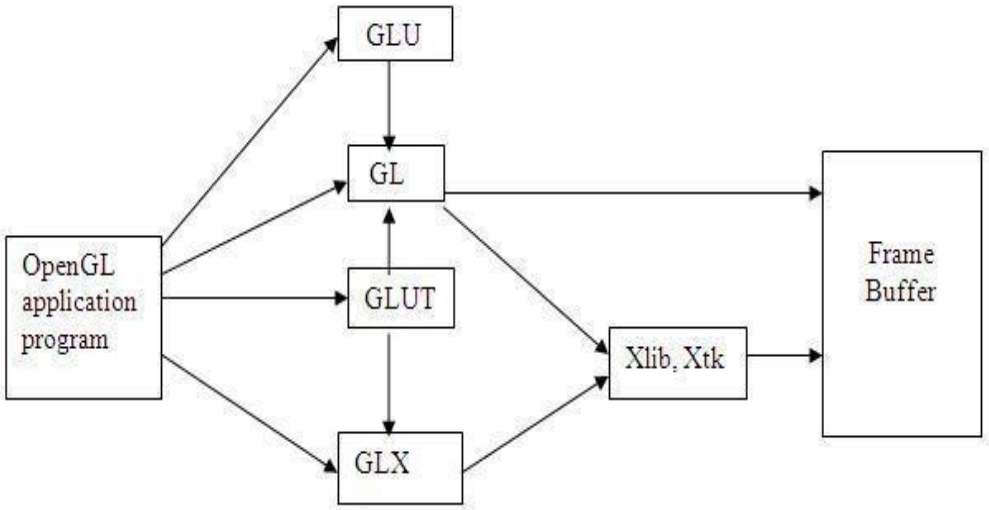

Fig 2.2 Library Organization

2.4 GRAPHICS SYSTEM AND FUNCTIONS

- Graphics system and functions can be considered as a black box, a term used to denote a system whose properties are only described by its inputs and output without knowing the internal working.
- Inputs to graphics system are functions calls from application program, measurements from input devices such as mouse and keyboard.
- Outputs are primarily the graphics sent to output devices.

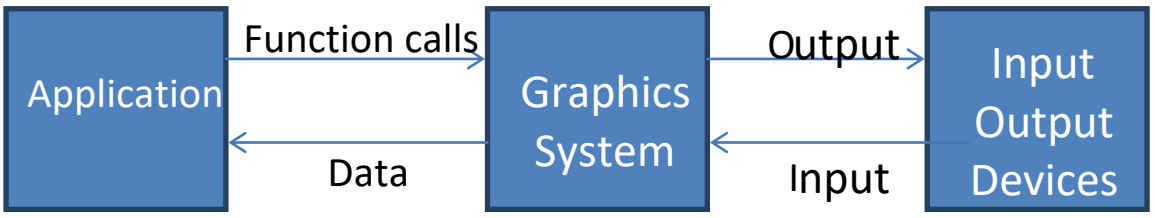

Fig 2.3 Graphics System as a Black Box

API's are described through functions in its library. These functions are divided into seven major groups.

1) Primitive Functions: Primitive functions define the low level objects or atomic entities that a system can display, the primitives include line segments, polygons, pixels, points, text and various types of curves and surfaces.
2) Attribute Functions: Attribute Functions allow us to perform operations ranging from choosing the color to display a line segment, to packing a pattern to fill inside any solid figure.
3) Viewing Functions: Viewing functions allow us to specify various views.
4) Transformation Functions: Transformation functions allow us to carry out transformation of objects such as rotation, translation and scaling.
5) Input Functions: Input functions allow us to deal with the diverse forms of input that characterize modern graphics system. It deals with devices such as keyboard, mouse and data tablets.
6) Control Functions: Control Functions enable us to communicate with the window system, to initialize the programs, and to deal with any errors that occur during the execution of the program.
7) Query Functions: Query Functions provides information about the API.

CHAPTER 3

SYSTEM REQUIREMENTS

Requirements analysis is critical for project development. Requirements must be documented, actionable, measurable, testable and defined to a level of detail sufficient for system design. Requirements can be architectural, structural, behavioural, functional, and non-functional. A software requirements specification (SRS) is a comprehensive description of the intended purpose and the environment for software under development.

3.1 Hardware Requirement

- Minimum of 2GB of main memory
- Minimum of 3GB of storage
- Keyboard
- Mouse
- Display Unit
- Dual-Core or AMD with minimum of 1.5GHz speed

3.2 Software Requirement

- Windows – XP/7/8
- Microsoft Visual Studio C/C++ 7.0 and above versions
- OpenGL Files
- DirectX 8.0 and above versions

Header Files

- glut.h

Object File Libraries

- glut32.lib

DLL files

- glut32.dll

CHAPTER 4

DESIGN AND IMPEMENTATION

4.1 HEADER FILES USED

- **STDIO.H :** Input and Output operations can also be performed in C++.Using the **C** standard input and output Library.
- **STDLIB.H :** The stdlib.h header includes many memory management functions such as malloc, calloc, free which are used to allocate or de-allocate bytes of memory.
- **GLUT.H :** The glut.h is a readily available library called the OpenGL Utility Toolkit which provides a simple interface between the System.

4.2 OPENGL API'S USED

4.2.1 Primitive functions

OpenGL supports two classes of primitives: Geometric primitives and Raster primitives.

Geometric primitives are specified in the problem domain and include points, line segments, polygons, curves and surfaces. The geometric primitives exist in two- and three-dimensional space and hence they can be manipulated by operations such as rotation and translation. Raster primitives are used to convert geometric primitives into pixels. Raster primitives, such as array of pixels, lack geometric properties and cannot be manipulated in the same way as geometric primitives.The basic OpenGL primitives are specified by sets of vertices. Thus the programmer can define the objects in the following form: glBegin(GLenum mode);

glVertex*(. . .);

.

glVertex*(. . .); glEnd();

The value mode defines how OpenGL assembles the vertices to define geometric objects. Other code and OpenGL function calls occur between glBegin and glEnd.

glBegin & glEnd

glBegin, glEnd - delimit the vertices of a primitive or a group of primitives

C Specification void glBegin(GLenum *mode*)

Parameters

Mode Specifies the primitive or primitives that will be created from vertices presented between glBegin and the subsequent glEnd. Ten symbolic constants are accepted: GL_POINTS,GL_LINES,GL_LINE_STRIP,GL_LINE_LOOP,GL_TRIANGLES,GL_TRIANGLE_STRIP,GL_TRIANGLE_FAN,GL_QUADS,GL_QUAD_STRIP,and GL_POLYGON.

C Specification

void glEnd( )

Description

glBegin and glEnd delimit the vertices that define a primitive or a group of like primitives.

GL_POINTS Treats each vertex as a single point. Vertex *n* defines point *n*. *N* points are drawn.

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 *1*. *N* lines are drawn.

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

glVertex

glVertex2f, glVertex2i, glVertex3f, glVertex3i - specify a vertex

C Specification void glVertex2f(GLfloat *x*, GLfloat *y*) void

glVertex2i(GLint *x*, GLint *y*) void glVertex3f(GLfloat *x*,

GLfloat *y*, GLfloat *z*) void glVertex3i(GLint *x*, GLint *y*, GLint

z) **Parameters**

x, *y*, *z* Specify *x*, *y*, *z* coordinates of a vertex. Not all parameters are present in all forms of the command. **Description**

glVertex commands are used within glBegin/glEnd pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called.

4.2.2 Attribute functions

An attribute is any property that determines how a geometric primitive is to be rendered.

glColor

glColor3f- set the current color. **C Specification** void

glColor3f(GLfloat *red*, GLfloat *green*, GLfloat *blue*)

Parameters

Red, *green*, *blue* Specify new red, green, and blue values for the current color. **Description**

glColor3 variants specify new red, green, and blue values explicitly, and set the current alpha value to 1.0 implicitly.Current color values are stored in floating-point format, with unspecified mantissa and exponent sizes. Unsigned integer color components, when specified, are linearly mapped to floating-point values such that the largest representable value maps to 1.0 (full intensity), and zero maps to 0.0 (zero intensity). Signed integer color components, when specified, are linearly mapped to floating-point values such that t h e m o s t positive represent able value maps to 1.0, and the most negative represent able value maps to -1.0.Floating-point values are mapped directly.

Neither floating-point nor signed integer values are clamped to the range [0,1] before updating the current color. However, color components are clamped to this range before they are interpolated or written into a color buffer.

glClearColor

glClearColor - specify clear values for the color buffers. C Specification void

glClearColor(GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*)

Parameters

red, *green*, *blue*, *alpha* Specify the red, green, blue, and alpha values used when the color buffers are cleared. The default values are all zero. Description

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

glClear - clear buffers to preset values.

C Specification void

glClear(GLbitfield *mask*)

Parameters

Mask Bitwise OR of masks that indicate the buffers to be cleared. The masks are

GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT.

Description

glClear sets the bitplane area of the window to values previously selected by glClearColor, glClearIndex, glClearDepth. Multiple color buffers can be cleared simultaneously by selecting more than one buffer at a time using glDrawBuffer. The pixel ownership test, the scissor test, dithering, and the buffer write masks affect the operation of glClear. The scissor box bounds the cleared region. Alpha function, blend function, logical operation, stenciling, texture mapping, and z-buffering are ignored by glClear. glClear takes a single argument that is the bitwise OR of several values indicating which buffer is to be cleared.

The values are as follows:

GL_COLOR_BUFFER_BIT Indicates the buffers currently enabled for color writing.
GL_DEPTH_BUFFER_BIT Indicates the depth buffer.

4.2.3 Viewing functions

glOrtho

glOrtho - multiply the current matrix by an orthographic matrix. gluOrtho2D- special case of glOrtho for two- dimensional viewing. **C Specification**

void glOrtho(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*)

void gluOrtho2D(GLdouble *left*, GLdouble *right*, GLdouble

bottom, GLdouble *top*) **Parameters**

left, *right* Specify the coordinates for the left and right vertical clipping planes. *bottom*,

top Specify the coordinates for the bottom and top horizontal clipping planes.

near, *far* Specify the distances to the nearer and farther depth clipping planes. These distances are negative if the plane is to be behind the viewer.

Description glOrtho describes a perspective matrix that produces a

parallel projection.

(*left*, *bottom*, *-near*) and (*right*, *top*, *-near*) specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively, assuming that the eye is located at (0, 0, 0).

-*far* specifies the location of the far clipping plane. Both *near* and *far* can be either positive or negative.

glMatrixMode

glMatrixMode - specify which matrix is the current matrix.

C Specification void glMatrixMode(GLenum *mode*)

Parameters

Mode Specifies which matrix stack is the target for subsequent matrix operations. Three values are accepted: GL_MODELVIEW, GL_PROJECTION, and GL_TEXTURE. The default value is GL_MODELVIEW. **Description** glMatrixMode sets the current matrix mode. *mode* can assume one of three values:

GL_MODELVIEW Applies subsequent matrix operations to the modelview matrix stack.
GL_PROJECTION Applies subsequent matrix operations to the projection matrix stack.

glLoadIdentity

glLoadIdentity - replace the current matrix with the identity matrix.

C Specification void glLoadIdentity(void) **Description**

glLoadIdentity replaces the current matrix with the identity matrix. It is semantically equivalent to calling glLoadMatrix with the identity matrix but in some cases it is more efficient.

4.2.4 Transformation functions

glTranslate

glTranslated, glTranslatef – multiply the current matrix by a translation matrix.

C Specification void glTranslated(GLdouble *x*, GLdouble *y*, GLdouble *z*)

void glTranslatef(GLfloat *x*, GLfloat *y*, GLfloat *z*) **Parameters**

x, *y*, *z* Specify the *x*, *y* and *z* coordinates of a translation vector. **Description**

glTranslate moves the coordinate system origin to the point specified by (*x*,*y*,*z*). The translation vector is used to compute a 4x4 translation matrix:

1	0	0	x
0	1	0	y
0	0	1	z
0	0	0	1

The current matrix (see glMatrixMode) is multiplied by this translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with M *T.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after glTranslate are translated. Use glPushMatrix and glPopMatrix to save and restore the untranslated coordinate system.

glRotate

glRotated, glRotatef - multiply the current matrix by a rotation matrix **C**

Specification void glRotated(GLdouble *angle*, GLdouble *x*, GLdouble

y, GLdouble *z*) void glRotatef(GLfloat *angle*, GLfloat *x*, GLfloat *y*,

GLfloat *z*) **Parameters**

angle Specifies the angle of rotation, in degrees.

x, *y*, *z* Specify the *x*, *y*, and *z* coordinates of a vector, respectively. **Description**

glRotate computes a matrix that performs a counter clockwise rotation of *angle* degrees about the vector from the origin through the point (*x*, *y*, *z*).

glPushMatrix & glPopMatrix

glPushMatrix, glPopMatrix - push and pop the current matrix stack.

C Specification void glPushMatrix(void) **C Specification** void

glPopMatrix(void)

Description

There is a stack of matrices for each of the matrix modes. In GL_MODELVIEW mode, the stack depth is at least 32. In the other two modes, GL_PROJECTION and GL_TEXTURE, the depth is at least 2.The current matrix in any mode is the matrix on the top of the stack for that mode. glPushMatrix pushes the current matrix stack down by one, duplicating the current matrix. That is, after a glPushMatrix call, the matrix on the top of the stack is identical to the one below it. glPopMatrix pops the current matrix stack, replacing the current matrix with the one below it on the stack. Initially, each of the stacks contains one matrix, an identity matrix. It is an error to push a full matrix stack, or to pop a matrix stack that contains only a single matrix. In either case, the error flag is set.

glRasterPos

C specification glRasterPos2d(GLfloat x,GLfloat y)

glRasterPos2f(GLdouble x,GLdouble y)

glRasterPos3d(GLfloat x,GLfloat y,GLfloat z)

glRasterPos3f(GLdouble x,GLdouble y,GLdouble

z) **Parameters**

x,y,z Specify the x, y, z, and w object coordinates (if present) for the raster position.

Description

The GL maintains a 3D position in window coordinates. This position, called the raster position, is used to position pixel and bitmap write operations. It is maintained with subpixel accuracyThe current raster position consists of three window coordinates (x, y, z), a clip coordinate value (w), an eye coordinate distance, a valid bit, and associated color data and texture coordinates. The w coordinate is a clip coordinate, because w is not projected to window coordinates. glRasterPos4 specifies object coordinates x, y, z, and w explicitly. glRasterPos3 specifies object coordinate x, y, and zexplicitly, while w is implicitly set to 1. glRasterPos2 uses the argument values for x and y while implicitly setting z and w to 0 and 1. The current raster position also includes some associated color data and texture coordinates. If lighting is enabled, then GL_CURRENT_RASTER_COLOR

(inRGBAmode) or GL_CURRENT_RASTER_INDEX (in color index mode) is set to the color produced by the lighting calculation (see glLight, glLightModel, and glShadeModel). If lighting is disabled, current color (in RGBA mode, state variable GL_CURRENT_COLOR) or color index (in color index mode, state variable GL_CURRENT_INDEX) is used to update the current raster color. GL_CURRENT_RASTER_SECONDARY_COLOR (in RGBA mode) is likewise updated.

4.3 USER DEFINED FUNCTIONS

We are using the GLUT toolkit in this program and hence the main function consist of calls to GLUT functions to set up our window and display properties. The main has a display callback function and for interaction it has mouse and keyboard callback functions. The init() is used to set up user options through OpenGL functions on GL and GLU libraries.The main function for this program is as follows:

Main Function:

void main(int argc,char **argv)

It contains calls to various functions to perform different operations.Before a programmer can open a window, he/she must specify its characteristics: Should it be single-buffered or double-buffered? Should it store colors as RGBA values or as color indices? Where should it appear on your display? To specify the answers to these questions, call **glutInit**(), **glutInitDisplayMode**(), glutInitWindowSize(), and **glutInitWindowPosition**() before you call **glutCreateWindow**() to open the window.

void **glutInit**(int argc, char **argv);

glutInit() should be called before any other GLUT routine, because it initializes the GLUT library. **glutInit()** will also process command line options, but the specific options are window system dependent. For the X Window System, -iconic, -geometry, and -display are examples of command line options, processed by **glutInit()**. (The parameters to the **glutInit()** should be the same as those to **main()**.) void **glutInitDisplayMode**(unsigned int mode);

Specifies a display mode (such as RGBA or color-index, or single- or double-buffered) for windows created when **glutCreateWindow()** is called. You can also specify that the

window have an associated depth, stencil, and/or accumulation buffer. The mask argument is a bitwise ORed combination of GLUT_RGBA or GLUT_INDEX, GLUT_SINGLE or GLUT_DOUBLE, and any of the buffer-enabling flags: GLUT_DEPTH, GLUT_STENCIL, or GLUT_ACCUM. For example, for a double-buffered, RGBA-mode window with a depth and stencil buffer, use GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL. The default value is GLUT_RGBA | GLUT_SINGLE (an RGBA, single-buffered window).

void **glutInitWindowSize**(int width, int height); void

glutInitWindowPosition(int x, int y);

Requests windows created by **glutCreateWindow()** to have an initial size and position. The arguments (x, y) indicate the location of a corner of the window, relative to the entire display. The width and height indicate the window's size (in pixels). The initial window size and position are hints and may be overridden by other requests.int **glutCreateWindow**(char *name);

Opens a window with previously set characteristics (display mode, width, height, and so on). The string name may appear in the title bar if your window system does that sort of thing. The window is not initially displayed until **glutMainLoop()** is entered, so do not render into the window until then. The value returned is a unique integer identifier for the window. This identifier can be used for controlling and rendering to multiple windows (each with an OpenGL rendering context) from the same application.

Display Function:

void display()

This function is used to call various functions that draw different objects to be displayed at the location depending upon where the function call to the particular function is made and messages to be displayed. It also has functions to set the color and also has function that forces the content of the frame buffer onto the display (glFlush( )).

Msg Function:

void msg(char *string)

string is the variable which collects the string that is to be displayed which is sent as a parameter in the function call.It uses the GLUT_BITMAP_TIMES_ROMAN_24 font to display the message.

Rectangle Function:

void rectangle(int b)

void rectangle1(int d)

void rectangle2(int d)

These functions are responsible for drawing rectangles at various positions. The values b and d are responsible for deciding the exact position of the rectangles.

Reset function: void

reset()

Used to reset the values of all the variables to default values.

Limit function:

void limit()

This function is used to calculate the score.

Helicopter Function:

void helicopter(int x, int y) x and y represents the x and

y co ordinates respectively.

This function is used to draw the helicopter at the specified position. This function contains code to draw various parts of the helicopter such as black line, head, big wing, small wing, foot, foot line.

Crashed Helicopter Function: void crashed_helicopter(int

x, int y) x and y represents the co ordinates of the crashed

helicopter.

This function is used to draw various parts of the crashed helicopter on the game over screen once the helicopter has touched the rectangle and the game is over.

Init Function:

void init(void)

This function is used to set up the window.

And finally for interaction- to play the game according to a player input- we have mouse and keyboard events.

Up and Down Bar Functions:

void down_bar() void

up_bar()

Used to display the Up and down green color bars present on the screen.

Mouse function:

void mouse(int btn,int state,int x,int y)

glutMouseFunc sets the mouse callback for the *current window*. When a user presses and releases mouse buttons in the window, each press and each release generates a mouse callback.

The button parameter is one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. For systems with only two mouse buttons, it may not be possible to generate GLUT_MIDDLE_BUTTON callback. For systems with a single mouse button, it may be possible to generate only a GLUT_LEFT_BUTTON callback. The state parameter is either GLUT_UP or GLUT_DOWN indicating whether the callback was due to a release or press respectively. The x and y callback parameters indicate the window relative coordinates when the mouse button state changed. If a GLUT_DOWN callback for a specific button is triggered, the program can assume a GLUT_UPcallback for the same button will be generated (assuming the window still has a mouse callback registered) when the mouse button is released even if the mouse has moved outside the window.

If the left mouse button is pressed the helicopter moves upwards and if the right mouse button is pressed the helicopter moves downwards.

Keyboard function:

void mykeyboard(unsigned char key, int x, int y)

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. The key callback parameter is the generated ASCII character. The state of modifier keys such as Shift cannot be determined directly; their only effect will be on the returned ASCII data. The x and y callback parameters indicate the mouse location in window relative coordinates when the key was pressed.

When a new window is created, no keyboard callback is initially registered, and ASCII key strokes in the window are ignored. Passing NULL to glutKeyboardFunc disables the generation of keyboard callbacks.

In this the input interaction happens through keyboard. Pressing Enter starts the game and pressing the key 8 causes the helicopter to move upwards and pressing the key 2 causes the helicopter to move downwards. Once the game is over we can restart the game by pressing the key c or exit the game by pressing the Esc key.

DATA FLOW DIAGRAM

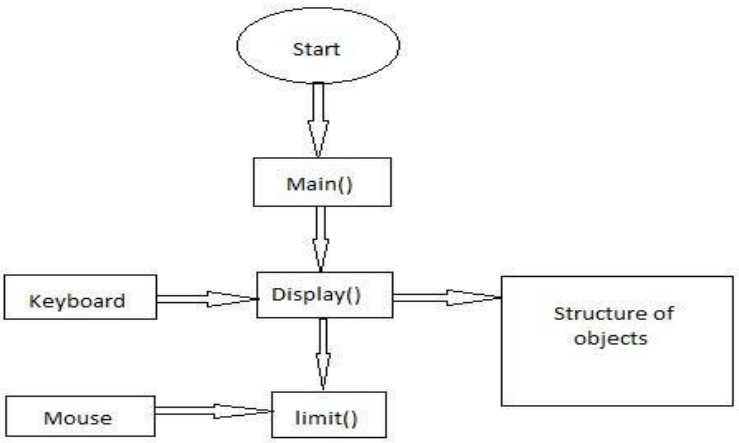

Fig 4.1:Data Flow Diagram

4.4 ALGORITHM

Step 1: Initialize the position of the Helicopter.

Step2: Initialize the position of the rectangle blocks- rectangle, rectangle1 and rectangle2.

Step 3: Set the limits of the screen for the game.

Step 4: Display welcome screen.

If player presses ‘enter’ then goto game screen.

Step 5: Move the rectangular blocks along x- axis towards the left side of the screen. Move the Helicopter along y- axis(up or down) according to the player’s input through a mouse or a keyboard.

Step 6: if crashed or cross the limits then goto game over screen.

Select to quit or restart the game.

4.5 CODE SNIPPETS

```
#include <stdio.h>
#include<stdlib.h>    #include<process.h>
#include<math.h>
#include<GL/glut.h> void
helicopter(int x, int y); void
crashed_helicopter(int x, int y); void
rectangle(int d); void rectangle1(int
d); void rectangle2(int d); void
rectangle3(); void down_bar(); void
up_bar(); float score = 0; int start =
0, end = 0, q = 0;
float n = 60, x = 200, y = 500, d = 0, a = 0, count = 1, b = 0, c = 0, t = 0, xcor = 0, xdor =
0, xbor = 0, yy = 0;
GLfloat theta = 0.0, theta1 = 0.0, theta2 = 0.0, theta3 = 0.0;



void msg(char *string)
{
        while (*string)
                glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, *string++);

}

void reset() {

n = 60, x = 200, y = 500, d = 0, a = 0, count = 1, b = 0, c = 0, t = 0, xcor = 0, xdor = 0, xbor
= 0, yy = 0;
        theta = 0.0, theta1 = 0.0, theta2 = 0.0, theta3 = 0.0;
score = 0;
        start = 1, end = 0, q = 0;
}

void display()
{
        glClear(GL_COLOR_BUFFER_BIT);

        if (!start)
        {
         glColor3f(1.0, 1.0, 1.0);   helicopter(x, y);
        glClearColor(0, 0, 0, 0); glColor3f(.3, .5, .8);
        glRasterPos2f(200, 400);
```

```
        msg("PRESS LEFT MOUSE BUTTON OR 8 TO GO UP");  glRasterPos2f(200,
        300);
                msg("PRESS RIGHT MOUSE BUTTON OR 2 TO GO DOWN");
                glRasterPos2f(200, 200);
                msg("PRESS ENTER TO BEGIN");

                glutIdleFunc(NULL);
        }
        else if (end)
        {
 glClearColor(0, 0, 0, 0);  glColor3f(.3, .5, .8);
 glRasterPos2f(400, 500);  msg("GAME OVER");
                glRasterPos2f(700, 250);
msg("press esc to quit");
glRasterPos2f(700, 200);                msg("press c
to restart");           glRasterPos2f(100, 250);
        msg("Project By");
glRasterPos2f(150, 200);
msg("DEEPAK R and DIVYA K");
glRasterPos2f(80, 150);
                msg("(SJB INSTITUTE OF TECHNOLOGY)");
        }

        else
        {
                down_bar();
                up_bar();

                glColor3f(1.0, 1.0, 1.0);
                helicopter(x, y);

                glColor3f(0.0, 1.0, 0.0);
                if (t>10 && t<20)
                rectangle(b);
if (t>20 && t<30)
rectangle1(c);                  if
(t<10)          rectangle2(d);

                glColor3f(0.0, 1.0, 0.0);

                glutSwapBuffers();
        }
        glFlush();

}

void limit()
```

```
{
        char tmp_str[40];

 if (!end)score += 0.005;    glColor3f(1, 0, 0);
glRasterPos2f(400, 770);    sprintf_s(tmp_str, "
score = %.0f", score);  msg(tmp_str);
        if (y>120 && y<860)
        {
                if (a == 1 && count == 0)
                {
                        count = 1;
                        y += 40;
                }
                if (a == 0 && count == 0)
                {
                        y -= 40;
                        count = 1;
                }
                if (t>10 && t<20)
                {
                        if (b<1000)
                                b    +=    1;
                else
                                b = 0;
                }
                if (t>20 && t<30)
                {
                        if (c<1000)
                        c += 1;
                else
        c = 0;
                }

                if (t<10)
                {
                        if (d<1000)
                                d +=
        1;              else
                        d = 0;
                }
                if (t>30)
                        t = 0;
                t += 0.01;
                if (xcor == 340 && y>465)
                {
                        yy = y;
                        y = 1000;
```

```
 }
                if (xdor == 340 && y>450)
                {
                        yy = y;
                        y = 1000;
                }
                if (xbor == 340 && y>125 && y<600)
                {
                        yy  =  y;
                y = 1000;
                }
                if (theta<100)
                theta += 50.0;
        else theta = 0;
  if  (theta1<100)      theta1  +=  50.0;
else theta1 = 0;
  if  (theta2<100)      theta2  +=  50.0;
else theta2 = 0;
         if  (theta3<100)     theta3  +=
        50.0;  else theta3 = 0;

                glutPostRedisplay();
        }
        if          (y          ==          1000)
                crashed_helicopter(x, yy);

        else if (y >= 860 || y<120)
                crashed_helicopter(x, y);
        //glFlush();

}

void helicopter(int x, int y)

{
        GLfloat i, cosine, sine;
        GLint r = 35, k = 55, m = 20;

        glColor3f(1., 1., 1.);
        glBegin(GL_POLYGON);
glVertex2f(x, y + 5);   glVertex2f(x
+ 75, y + 20);  glVertex2f(x + 75, y -
20);     glVertex2f(x, y - 5);
glEnd();

        //head
        glColor3f(1.0, 1.0, 1.0);
```

```
        glBegin(GL_POLYGON);
        for (i = 0; i<360; i++)
        {
                cosine = (x + 100) + (r*cos(i));
        sine = y + (r*sin(i));
                glVertex2f(cosine, sine);
        }
        glEnd();

 //black line (seperator)
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_LINES);
        glVertex2f(x + 75, y - 20);
        glVertex2f(x + 75, y + 20);
glEnd();

        glColor3f(.3, .5, .8);
        glBegin(GL_POLYGON);
        glVertex2f(x + 85, y - 20);
        glVertex2f(x + 85, y + 5);
        glVertex2f(x + 97, y + 5);
        glVertex2f(x + 97, y - 20);
        glEnd();
        glBegin(GL_POLYGON);
        glVertex2f(x + 105, y - 20);
        glVertex2f(x + 105, y);
        glVertex2f(x + 115, y);
        glVertex2f(x + 115, y - 20);
        glEnd();

        //wing big
        glColor3f(.3, .5, .8);
        glPushMatrix();
 glTranslatef(x + 90, y + 30, 0);  glRotatef(-
60, 0.0, 0.0, 1.0);  glRotatef(theta, 0.0, 0.0,
1.0);  glTranslatef(-(x + 90), -(y + 30), 0);
        glBegin(GL_POLYGON);
glVertex2i(x + 90, y + 30);
glVertex2i(x + 80, y + 100);
glVertex2i(x + 100, y + 100);
glEnd();        glPopMatrix();

        glPushMatrix();
        glTranslatef(x + 90, y + 30, 0);
glRotatef(-60, 0.0, 0.0, 1.0);   glRotatef(theta1, 0.0,
0.0, 1.0);        glTranslatef(-(x + 90), -(y + 30), 0);
```

```
        glBegin(GL_POLYGON);
glVertex2i(x + 90, y + 30);
glVertex2i(x + 80, y - 20);
glVertex2i(x + 100, y - 20);
        glEnd();
        glPopMatrix();

        glPushMatrix();
 glTranslatef(x + 90, y + 30, 0);
glRotatef(-60, 0.0, 0.0, 1.0);
glRotatef(theta2, 0.0, 0.0, 1.0);
        glTranslatef(-(x + 90), -(y + 30), 0);
        glBegin(GL_POLYGON);
glVertex2i(x + 90, y + 30);
glVertex2i(x + 30, y - 10);
        glVertex2i(x + 30, y + 10);
        glEnd(); glPopMatrix();

        glPushMatrix(); glTranslatef(x
        + 90, y + 30, 0); glRotatef(-60,
        0.0, 0.0, 1.0); glRotatef(theta3,
        0.0, 0.0, 1.0);
        glTranslatef(-(x + 90), -(y + 30), 0);
        glBegin(GL_POLYGON);
        glVertex2i(x + 160, y + 50); glVertex2i(x
        + 90, y + 30);
        glVertex2i(x + 160, y + 70);
        glEnd();
        glPopMatrix();

        //wing                small
glPushMatrix();
 glTranslatef(x, y + 10, 0);  glRotatef(-
60, 0.0, 0.0, 1.0);  glRotatef(theta, 0.0,
0.0, 1.0);  glTranslatef(-(x), -(y + 10),
0);
        glBegin(GL_POLYGON);
        glVertex2i(x,       y       +       10);
glVertex2i(x, y + 30);
        glVertex2i(x - 10, y + 30);
        glEnd();
glPopMatrix();
        glPushMatrix();
 glTranslatef(x, y + 10, 0);  glRotatef(-
60,  0.0,  0.0,  1.0);   glRotatef(theta1,
0.0, 0.0, 1.0);  glTranslatef(-(x), -(y +
10), 0);
```

```
        glBegin(GL_POLYGON);
        glVertex2i(x, y + 10);
glVertex2i(x, y - 10);
glVertex2i(x - 10, y - 10);
        glEnd();
        glPopMatrix();

        glPushMatrix();
 glTranslatef(x, y + 10, 0);  glRotatef(-
60,  0.0,  0.0,  1.0);   glRotatef(theta2,
0.0, 0.0, 1.0);  glTranslatef(-(x), -(y +
10), 0);
        glBegin(GL_POLYGON);
        glVertex2i(x,      y      +      10);
glVertex2i(x + 20, y + 10);
        glVertex2i(x    +    20,    y);
glEnd();
        glPopMatrix();

        glBegin(GL_POINTS); for
        (i = 0; i<360; i++)
        {
         cosine = (x)+(m*cos(i));  sine = (y
        +       10)       +       (m*sin(i));
        glVertex2f(cosine, sine);

        } glEnd();

        glColor3f(1.0, 1.0, 0.0);
        //foot
glBegin(GL_LINE_LOOP);
        glVertex2i(x + 80, y - 45);
glVertex2i(x + 80, y - 40);
glVertex2i(x + 120, y - 40);
glVertex2i(x + 125, y - 45);
        glEnd();

        //foot                          line
glBegin(GL_LINES);
        glVertex2i(x + 100, y - 35);
        glVertex2i(x + 90, y - 40);
        glEnd();

        glBegin(GL_LINES);
        glVertex2i(x    +    100,    y    -    35);
glVertex2i(x + 110, y - 40);
        glEnd();
```

```
}


void crashed_helicopter(int x, int y)
{
        GLfloat i, cosine, sine;
        GLint r = 35, k = 50, m = 10;

 glColor3f(1.0,      0.0,      0.0);
glBegin(GL_LINES);
glVertex2i(x, y + 10);
        glVertex2i(x   +   75,   y   +   25);
glEnd();

        //down big line
glBegin(GL_LINES);
glVertex2i(x, y - 10);
        glVertex2i(x + 75, y - 25); glEnd();

        //centre line
        glBegin(GL_LINES);
        glVertex2i(x, y - 10);
        glVertex2i(x - 3, y + 10);
        glEnd();

        //small line connected to small
        wing glBegin(GL_LINES);
        glVertex2i(x, y + 10);
        glVertex2i(x - 3, y + 20);
        glEnd();

        //small      line      connected      to      small      wing
glBegin(GL_LINES);
        glVertex2i(x    -    3,    y    +    10);
glVertex2i(x - 3, y + 20);
        glEnd();

        //head
glBegin(GL_POINTS);
        for (i = 0; i<360; i++)
        {
                cosine = (x + 100) + (r*cos(i));
        sine = y + (r*sin(i));
glVertex2f(cosine, sine);
        }
        glEnd();
```

```
        //wing                          big
glBegin(GL_POINTS);
        for (i = 0; i<360; i++)
        {
                cosine = (x + 90) + (k*cos(i));
        sine = (y + 20) + (k*sin(i));
                glVertex2f(cosine, sine);
        }
        glEnd();

 //wing                  small
glBegin(GL_POINTS);  for (i
= 0; i<360; i++)
        {
                cosine = (x - 5) + (m*cos(i));
        sine = (y + 30) + (m*sin(i));
                glVertex2f(cosine, sine);
        }
        glEnd();



        /*foot*/
        glBegin(GL_LINE_LOOP);
        glVertex2i(x + 80, y - 45);
glVertex2i(x + 80, y - 40);
glVertex2i(x + 120, y - 40);
glVertex2i(x + 125, y - 45);
        glEnd();

        //foot                          line
glBegin(GL_LINES);
        glVertex2i(x + 100, y - 35);
glVertex2i(x + 90, y - 40);
        glEnd();

        glBegin(GL_LINES);
            glVertex2i(x + 100, y - 35);
            glVertex2i(x + 110, y - 40);
        glEnd();

        //big wing rotating line
glBegin(GL_LINES);
        glVertex2i(x + 90, y + 20);
glVertex2i(x + 60, y + 60);
glEnd();
```

```
        glBegin(GL_LINES);
        glVertex2i(x + 90, y + 20);
glVertex2i(x + 48, y + 48);
glEnd();

        glBegin(GL_LINES);
        glVertex2i(x + 90, y + 20);
        glVertex2i(x + 75, y - 25);
glEnd();

        glBegin(GL_LINES);
        glVertex2i(x + 90, y + 20);
        glVertex2i(x + 85, y - 28);
        glEnd();

        glBegin(GL_LINES);
        glVertex2i(x + 90, y + 20);
glVertex2i(x + 135, y + 45);
glEnd();

        glBegin(GL_LINES);
        glVertex2i(x + 90, y + 20);
        glVertex2i(x + 125, y + 53);
        glEnd();
        //small wing rotatin wings
        glBegin(GL_LINES);
 glVertex2i(x - 5, y + 30);  glVertex2i(x - 10,
y + 38);
        glEnd();

        glBegin(GL_LINES);
        glVertex2i(x - 5, y + 30);
        glVertex2i(x - 10, y + 20);
        glEnd();

        glBegin(GL_LINES);
        glVertex2i(x - 5, y + 30);
        glVertex2i(x + 5, y + 30);
glEnd();

 //rounded circle  glColor3f(1.0,
1.0, 1.0);  glPointSize(3);
        glBegin(GL_POINTS);
        for (i = 0; i<90; i++)
        {
```

```
                cosine = (x + 60) + (100 * cos(i));
sine = (y + 20) + (100 * sin(i));
glVertex2f(cosine, sine);
        }
        glEnd();
up_bar();
down_bar();    end = 1;
        glFlush();
}


void rectangle(int b)
{
        glColor3f(0.0, 1.0, 0.0);
        glBegin(GL_POLYGON);
 glVertex2i(950    -    b,    800);
glVertex2i(950    -    b,    550);
glVertex2i(1000    -    b,    550);
glVertex2i(1000    -    b,    800);
glEnd();
        xcor = 950 - b;
}
void rectangle1(int d)
{
        glColor3f(0.0, 1.0, 0.0);
        glBegin(GL_POLYGON);
        glVertex2i(950 - d, 800);
glVertex2i(950 - d, 500);
glVertex2i(1000 - d, 500);
glVertex2i(1000 - d, 800);
glEnd();
        xdor = 950 - d;
}

void rectangle2(int d)
{
        glColor3f(0.0, 1.0, 0.0);
        glBegin(GL_POLYGON);
        glVertex2i(950 - d, 200);
glVertex2i(950 - d, 550);
glVertex2i(1000 - d, 550);
glVertex2i(1000 - d, 200);
glEnd();
        xbor = 950 - d;


}
```

```
void down_bar()
{
        glColor3f(0, 1, 0);
glBegin(GL_POLYGON);
glVertex2i(0, 0);       glVertex2i(0,
100);   glVertex2i(1000, 100);
glVertex2i(1000, 0);
        glEnd();
}


void up_bar()
{
        glColor3f(0, 1, 0);
glBegin(GL_POLYGON);
glVertex2i(0, 900);     glVertex2i(0,
1000);          glVertex2i(1000,
1000);          glVertex2i(1000,
900);   glEnd();
}


void mouse(int btn, int state, int x, int y)
{
        if (btn == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        {
                a = 1;
                count = 0;
        }
        if (btn == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        {
                a = 0;
                count = 0;
        }
}



void keyboard(unsigned char key, int x, int y)
{
        switch (key)
        {
 case 13:   start = 1;

                glutIdleFunc(limit);
```

```
                break;
case 27:
                exit(0);
        case 'c':end = 0;
                reset();
                display();

                break;
        }
        if (key == '8')
        {
 a = 1;   count = 0;
        }
        if (key == '2')
        {
                a = 0;
                count = 0;
        }
}


void init(void)
{
        glClearColor(0.0, 0.0, 0.0, 0.0);
glPointSize(1.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
        gluOrtho2D(0.0, 1000.0, 0.0, 1000.0);
}

int main()
{
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(1300, 800);
glutCreateWindow("THE HELICOPTER");
glutDisplayFunc(display);      glutIdleFunc(limit);
glutMouseFunc(mouse);
glutKeyboardFunc(keyboard);
        init();
glutMainLoop();         return
0;
```

REFERENCES

[1]. Interactive Computer Graphics: A Top- down Approach Using OpenGL, Fifth Edition by Edward Angel, Pearson education, 2009.

[2]. Computer Graphics with OpenGL, Third Edition, by Hearn & Baker, Pearson education.

[3]. http://www.cs.uccs.edu/~ssemwal/glman.html

[4]. http://www.opengl.czweb.org/ewtoc.html

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

[6]. http://www.en.wikipedia.org/wiki/OpenGL

[7]. https://www.opengl.org/documentation/specs/glut/spec3/node49.html

CHAPTER 6

CONCLUSION

6.1 Conclusion of the Project

We have successfully implemented a simple game in this project using OpenGL. OpenGl supports enormous flexibility in the design and the use of OpenGl graphics programs. The presence of many built in classes methods take care of much functionality and reduce the job of coding as well as makes the implementation simpler.

This game shows the use of computer graphics throughout an application especially when it comes to interaction of computers with humans. In this program, we saw how alphabetical characters and stored data like scores are rendered on screen. We also saw how the game or animated characters- like the helicopter in this game- are created, and how objects can be moved from one co-ordinate position to another representing the movement of the object.

6.2 Future Enhancements

Every game has one property in common, that is they can be never ending. The games can blossom like trees spanning new levels and new avatars. Similarly the enhancements that can be made for this game are:

- We can create a new background that makes the game look more colourful.
- We can change the obstacles from rectangular blocks to more realistic objects like buildings or other flying objects.
- We can include more levels and increase the difficulty level as the player advances.
- We can include the feature of Multi-player to this game.