CHAPTER 1

INTRODUCTION

Computer graphics is a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Although the term often refers to the study of three-dimensional computer graphics, it also encompasses two-dimensional graphics and image processing. It is divided into two broad classes. It is also called passive graphics. Here the user has no control over the pictures produced on the screen. Interactive graphics provides extensive user-computer interaction. It provides a tool called "motion dynamics" using which the user can move objects. A broad classification of major subfields in computer graphics might be:

1. Geometry: study of different ways to represent and process surfaces.
2. Animation: study of different ways to represent and manipulate motion.
3. Rendering: study of algorithms to reproduce light transport.
4. Imaging: study of image acquisition or image editing.
5. Topology: study of the behavior of spaces and surfaces.

Geometry

The subfield of geometry studies the representation of three-dimensional objects in a discrete digital setting. Because the appearance of an object depends largely on its exterior, boundary representations are most commonly used. Two dimensional surfaces are a good representation for most objects, though they may be non-manifold. Since surfaces are not finite, discrete digital approximations are used.

Animation

The subfield of animation studies descriptions for surfaces (and other phenomena) that move or deform over time. Historically, most work in this field has focused on parametric and data-driven models, but recently physical simulation has become more popular as computers have become more powerful computationally.

Rendering

Rendering generates images from a model. Rendering may simulate light transport to create realistic images or it may create images that have a particular artistic style in non-photorealistic rendering. The two basic operations in realistic rendering are transport (how much light passes from one place to another) and scattering (how surfaces interact with light).

Imaging

Digital imaging or digital image acquisition is the creation of digital images, such as of a physical scene or of the interior structure of an object.

Topology

Topology is the mathematical study of shapes and topological spaces.

1.1 About OpenGL

OpenGL(**O**pen **G**raphics **L**ibrary**)** is an application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering. OpenGL stands for Open Graphics Library. It is a specification of an API for rendering graphics, usually in 3D. OpenGL implementations are libraries that implement the API defined by the specification. OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that is used to specify the objects and operations needed to produce interactive three-dimensional applications.

OpenGL (**Open G**raphics **L**ibrary) is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics the interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, and flight simulation. It is also used in video games , where it competes with Direct3D on Microsoft Windows platforms. OpenGL is managed by the non-profit technology consortium, the Khronos Group**.**

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, user must work through whatever windowing system controls the particular hardware that the user is using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow the user to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation.

OpenGL serves two main purposes:

- To hide the complexities of interfacing with different 3D accelerators, by presenting the programmer with a single, uniform API.
- To hide the differing capabilities of hardware platforms, by requiring that all implementations support the full OpenGL feature set (using software emulation if necessary).

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, the user must work through whatever windowing system controls the particular hardware that the user is using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow the user to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules.

The format for transmitting OpenGL commands (called the *protocol*) from the client to the server is always the same, so OpenGL programs can work across a network even if the client and server are different kinds of computers. If an OpenGL program isn't running across a network, then there's only one computer, and it is both the client and the server.

1.2.1 OpenGL Commands and Primitives

OpenGL draws *primitives*—points, line segments, or polygons—subject to several selectable modes. You can control modes independently of each other; that is, setting one mode doesn't affect whether other modes are set (although many modes may interact to determine what eventually ends up in the frame buffer). Primitives are specified, modes are set, and other OpenGL operations are described by issuing commands in the form of function calls.

Primitives are defined by a group of one or more *vertices*. A vertex defines a point, an endpoint of a line, or a corner of a polygon where two edges meet. Data (consisting of vertex coordinates, colors, normals, texture coordinates, and edge flags) is associated with a vertex, and each vertex and its associated data are processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be *clipped* so that a particular primitive fits within a specified region; in this case, vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent command takes effect. It also means that state-querying commands return data that's consistent with complete execution of all previously issued OpenGL commands.

OpenGL commands use the prefix **gl** and initial capital letters for each word making up the command name (**glClearColor()**, for example). Similarly, OpenGL defined constants begin with GL_, use all capital letters, and use underscores to separate words (like GL_COLOR_BUFFER_BIT). Some seemingly extraneous letters appended to some command names (for example, the **3f** in **glColor3f () and glVertex3f ()**)can also be seen. It's true that the **Color** part of the command name **glColor3f ()** is enough to define the command as one that sets the current color. However, more than one such command has been defined so that the user can use different types of arguments. In particular, the **3** part of the suffix indicates that three arguments are given; another version of the **Color** command takes four arguments. The **f** part of the suffix indicates that the arguments are floating-point numbers. Having different formats allows OpenGL to accept the user's data in his or her own data format.

Some OpenGL commands accept as many as 8 different data types for their arguments. The letters used as suffixes to specify these data types for ISO C implementations of OpenGL are shown in Table 1.1, along with the corresponding OpenGL type definitions. The particular implementation of OpenGL that are used might not follow this scheme exactly; an implementation in C++ or ADA, for example, wouldn't need to implementation in C++ or ADA, for example, wouldn't need to.

Suffix	**Data Type**	**Typical Corresponding C-Language Type**	**OpenGL Type Definition**
B	8-bit integer	Signed char	GLbyte
S	16-bit integer	Short	GLshort
I	32-bit integer	int or long	GLint,GLsizei
F	32-bit floating-point	Float	GLint,GLclampf
D	62-bit floating-point	Double	GLdouble,GLclampd
Ub	8-bit unsigned integer	Unsigned char	GLubyte,GLboolean
Us	16-bit unsigned integer	Unsigned short	GLushort
Ui	32-bit unsigned integer	Unsigned int or long	GLuint,GLenum,GLbitfield

Table 1.1 - Datatypes accepted by OpenGL

1.2.2 OpenGL Rendering Pipeline

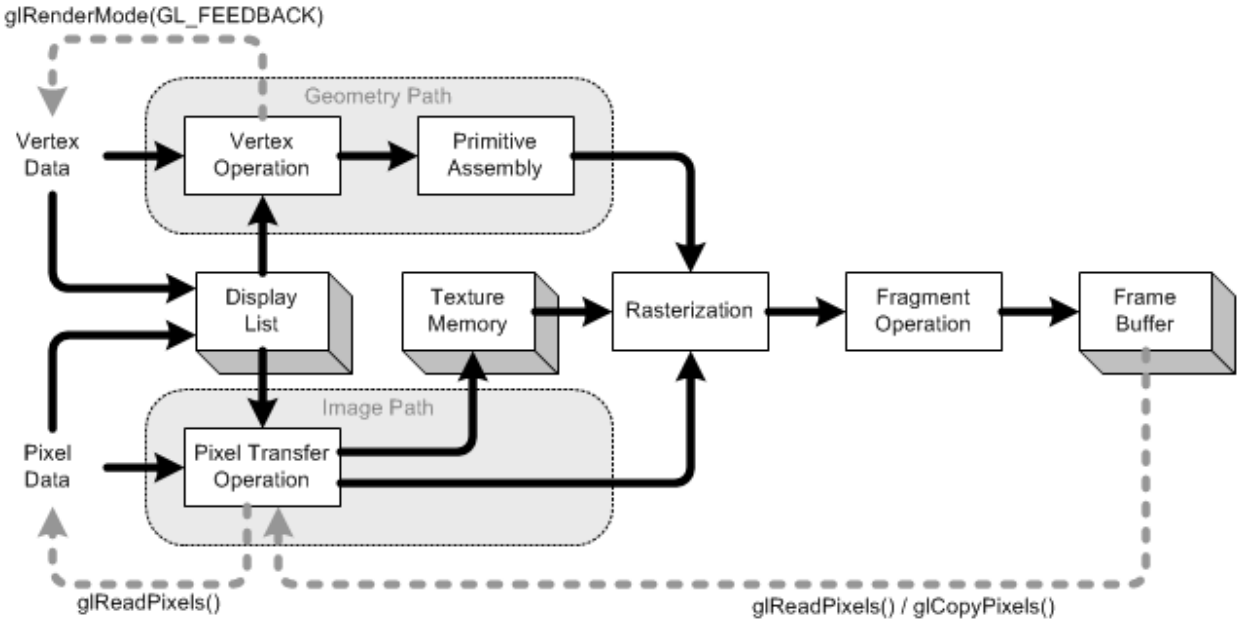

Figure 1.1: Order of Operations

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. This ordering, as shown in the figure below, is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do. The following diagram shows the Henry Ford assembly line approach, which OpenGL takes to processing data. Geometric data (vertices, lines, and polygons) follow the path through the row of boxes that includes evaluators and per-vertex operations, while pixel data are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per fragment operations) before the final pixel data is written into the frame buffer.

Given below are the key stages in the OpenGL rendering pipeline.

Display Lists

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

Evaluators

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

Per-Vertex Operations

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

Primitive Assembly

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines. The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

Pixel Operations

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

Texture Assembly

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

Rasterization

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

1.2.3 OpenGL -GLUT and OpenGL Utility Libraries

There are numerous Windowing system and interface libraries available for OpenGL as well as Scene graphs and High-level libraries build on top of OpenGL

About GLUT

GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs. GLUT routines use the prefix **glut**. It implements a simple windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL Programming.

Other GLUT-like Window System Toolkits

Libraries that are modeled on the functionality of GLUT providing support for things like: windowing and events, user input, menus, full screen rendering, performance timing

About GLX, GLU & DRI

GLX is used on Unix OpenGL implementation to manage interaction with the X Window System and to encode OpenGL onto the X protocol stream for remote rendering. GLU is the OpenGL Utility Library. This is a set of functions to create texture mipmaps from a base image, map coordinates between screen and object space, and draw quadric surfaces and NURBS. DRI is the Direct Rendering Infrastructure for coordinating the Linux kernel, X window system, 3D graphics hardware and an OpenGL-based rendering engine.

GLX, GLU and DRI

GLX Library

GLX 1.3 is used on Unix OpenGL implementation to manage interaction with the X Window System and to encode OpenGL onto the X protocol stream for remote rendering. It supports: pixel buffers for hardware accelerated offscreen rendering; read-only drawables for preprocessing of data in an offscreen window and direct video input; and FBConfigs, a more

powerful and flexible interface for selecting frame buffer configurations underlying an OpenGL rendering window.

GLU Library

The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. GLU routines use the prefix **glu**. This is a set of functions to create texture mipmaps from a base image, map coordinates between screen and object space, and draw quadric surfaces and NURBS. GLU 1.2 is the version of GLU that goes with OpenGL 1.1. GLU 1.3 is available and includes new capabilities corresponding to new OpenGL 1.2 features.

Direct Rendering Infrastructure (DRI)

In simple terms, the DRI enables hardware-accelerated 3D graphics on Linux. More specifically, it is a software architecture for coordinating the Linux kernel, X window system, 3D graphics hardware and an OpenGL-based rendering engine.

Higher Level Libraries built on OpenGL

Leading software developers use OpenGL, with its robust rendering libraries, as the 2D/3D graphics foundation for higher-level APIs. Developers leverage the capabilities of OpenGL to deliver highly differentiated, yet widely supported vertical market solutions.

CHAPTER 2

REQUIREMENTS ANALYSIS

The requirement analysis specifies the requirements of the computer graphic system is needed to develop a graphic project.

2.1 Requirements of the project

A graphics package that attracts the attention of the viewers is to be implemented. The package should provide a platform for user to perform animation.

2.2 Resource Requirements

The requirement analysis phase of the project can be classified into:

- Hardware Requirements
- Software Requirements

2.2.1 Software Requirements

This document will outline the software design and specification of our application. The application is a Windows based C++ implementation with OpenGL. Our main UI will be developed in C/C++, which is a high level language. This application will allow viewing of GDS II files, assignment of Color and Transparency, as well as printing of the rendered object. There will be the ability to save these color/transparency palettes for a given GDS file as well as the foundry used to create the file. These palettes can then be used with future GDS files to save time assigning colors/transparencies to layers. Operator/user interface characteristics from the human factors point of view

- Mouse Interface allows the user to point and click on GDS objects.
- Standard Win Forms keyboard shortcuts to access menus.

- The interface will use OpenGL to display GDS file.
- Any mouse and keyboard that works with Win Forms
- Program will use windows print APIs to interface with printer.

Interface with other software components or products, including other systems, utility software, databases, and operating systems.

OpenGL libraries for required are

- GLUT library
- STDLIB library

The application is very self-contained. Robust error handling will be implemented and code will be object-oriented to allow for easier maintenance and feature additions.

This model runs using Microsoft Visual Studio C++ Express version 10 and runs on Windows XP/07/08.

2.2.2 Hardware Requirements

There are no rigorous restrictions on the machine configuration. The model should be capable of working on all machines and should be capable of being supported by the recent versions of Microsoft Visual Studio.

- Processor: Intel® Pentium 2.6+ GHz
- Hard disk Capacity: 50 GB on a single drive/partition
- RAM: 4 GB
- CPU Speed: 2.6+ GHz
- Keyboard: Standard
- Mouse: Standard

Visual Studio Hardware Requirements

- Some specific hardware requirements are needed for Visual Studio C++ Express 2010 which include the Processor, RAM, Hard disk space, required operating system, Video and Mouse. They can be listen as follows:

Requirement	Professional
Processor	1.6 GHz processor or higher
RAM	1 GB of RAM (1.5 GB if running on a virtual machine)
Available Hard Disk Space	20 GB of available hard disk space
Operating System	Visual Studio 2010 can be installed on the following operating systems: • Windows XP (x86) with Service Pack 3 - all editions except Starter Edition • Windows Vista (x86 & x64) with Service Pack 2 - all editions except Starter Edition • Windows 7 (x86 and x64) • Windows Server 2003 (x86 & x64) with Service Pack 2 - all editions. Users must install MSXML6 if it is not already present. • Windows Server 2003 R2 (x86 and x64) - all editions • Windows Server 2008 (x86 and x64) with Service Pack 2 - all editions • Windows Server 2008 R2 (x64) - all editions
Video	DirectX 9-capable video card that runs at 1024 x 768 or higher display resolution
Mouse	Microsoft mouse or compatible pointing device

- Performance has not been tuned for minimum system configuration. Increasing the RAM above the recommended system configuration will improve performance, specifically when there are running multiple applications, working with large projects, or doing enterprise-level development.

CHAPTER 3

DESIGN PHASE

Design of any software depends on the architecture of the machine on which that software runs, for which the designer needs to know the system architecture. Design process involves design of suitable algorithms, modules, subsystems, interfaces etc.

3.1 Algorithm:

The entire design process can be explained using a simple algorithm. The algorithm gives a detailed description of the design process of 'Air Traffic Control'.
The various steps involved in the design of 'Air Traffic Control' are as shown below:

Step 1: Start

Step 2:Set initial Display Mode.

Step 3:Set initial Window Size to (1000,480).

Step 4: Create Window "Glut Plane".

Step 5: Display function

Set clear to Clear the Color and Depth Buffers.

Call Swap Buffer function.

Step 6: Keyboard function

Assign 'p' to draw pyramid.

Check function.

Assign 'f' to take off.

Check function.

Assign 'a' to add plane.

Check function.

Assign ‘r’ to remove plane.

Check function.

Assign ‘q’ to exit.

Check function.

Step 7: Mouse function

Assign ‘Right button’ to display menu.

Check function.

Assign ‘Left button’ to display menu.

Check function.

Step 8: Init function

Set Clear Color.

Set Viewport function.

Set MatrixMode function.

Set Perspective function.

Step 9: Stop

3.2 Flow Diagram:

Flow diagram is a collective term for a diagram representing a flow or set of dynamic relationships in a system. The term flow diagram is also used as synonym of the flowchart and sometimes as counterpart of the flowchart.

Flow diagrams are used to structure and order a complex system, or to reveal the underlying structure of the elements and their interaction. Flow diagrams are used in analyzing, designing, documenting or managing a process or program in various fields

Flow diagrams used to be a popular means for describing computer algorithms. They are still used for this purpose; modern techniques such as UML activity diagrams can be considered to be extensions of the flow diagram.

However, their popularity decreased when, in the 1970s, interactive computer terminals and third-generation programming languages became the common tools of the trade, since algorithms can be expressed much more concisely and readably as source code in such a language. Often, pseudo-code is used, which uses the common idioms of such languages without strictly adhering to the details of a particular one.

There are many different types of flow diagrams. On the one hand there are different types for different users, such as analysts, designers, engineers, managers, or programmers. On the other hand those flow diagrams can represent different types of objects.

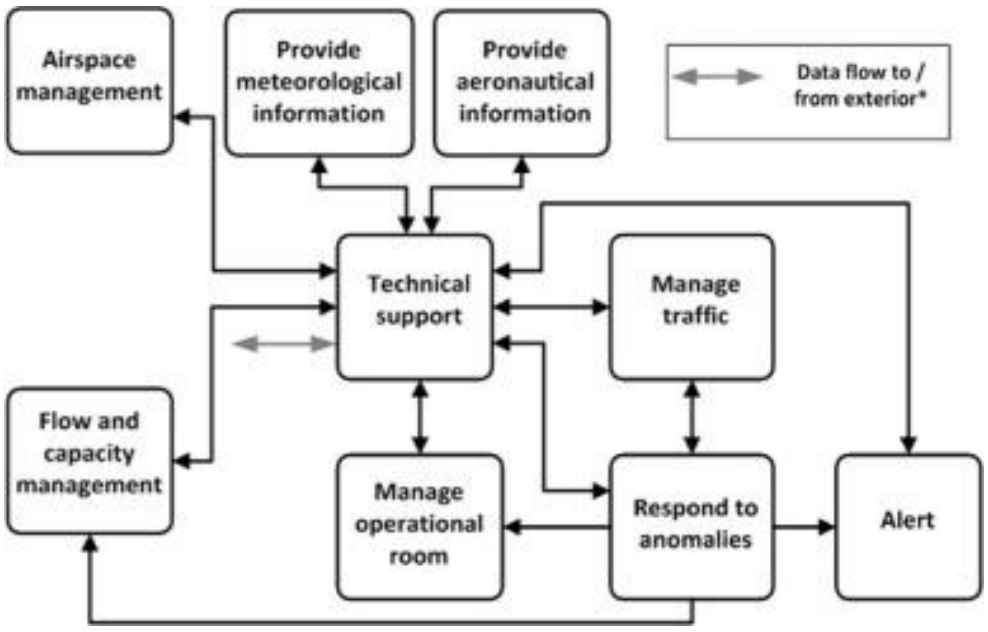

Figure 3.1: Flow Diagram

CHAPTER 4

IMPLEMENTATION

The implementation stage of this model involves the following phases.

- Implementation of OpenGL built in functions.
- User defined function Implementation.

4.1 Implementation of OpenGL Built In Functions

1. glutInit( )

glutInit is used to initialize the GLUT library.
glutInit will initialize the GLUT library and negotiate a session with the window system.

2. glutInitDisplayMode( )

glutInitDisplayMode sets the initial display mode.
The initial display mode is used when creating top-level windows, sub windows, and overlays to determine the OpenGL display mode for the to-be-created window or overlay.

3. glutCreateWindow( )

glutCreateWindow creates a top-level window.
glutCreateWindow creates a top-level window. The name will be provided to the window system as the window's name. The intent is that the window system will label the window with the name.

4. glutDisplayFunc( )

glutDisplayFunc sets the display callback for the current window.glutDisplayFunc sets the display callback for the current window. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called.

5. glutKeyboardFunc( )

glutKeyboardFunc sets the keyboard callback for the current window. 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 x and y callback parameters indicate the mouse location in window relative coordinates when the key was pressed.

6. glutMouseFunc()

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.

7. glutCreateMenu()

glutCreateMenu **creates a new pop-up menu and returns a unique small integer identifier**. The range of allocated identifiers starts at one. The menu identifier range is separate from the window identifier range. Implicitly, the current menu is set to the newly created menu.

8. glutMainLoop( )

glutMainLoop enters the GLUT event processing loop.
glutMainLoop enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

9. glMatrixMode( )

The two most important matrices are the model-view and projection matrix. At any time, the state includes values for both of these matrices, which are initially set to identity matrices.

10. gluOrtho2D( ) It is used to specify the two dimensional orthographic view. It takes four parameters; they specify the leftmost corner and rightmost corner of viewing rectangle.

11.glutIdleFunc( )

glutIdleFunc sets the global idle callback. glutIdleFunc sets the global idle callback to be func so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received. If enabled, the idle callback is continuously called when events are not being received.

12.glutBitmapCharacter( )

glutBitmapCharacter renders a bitmap character using OpenGL. Without using any display lists, glutBitmapCharacter renders the character in the named bitmap font.

13. glLightfv()

glLight — set light source parameters. glLight sets the values of individual light source parameters. *light* names the light and is a symbolic name of the form GL_LIGHT i, where i ranges from 0 to the value of GL_MAX_LIGHTS - 1. *pname* specifies one of ten light source parameters, again by symbolic name. *params* is either a single value or a pointer to an array that contains the new values.

4.2 Implementation of User Defined Functions:

1.drawplane()

The drawplane function is used for plane construction . It is created using the glPushMatrix() functions.

2.strip()

The strip function is used to run away strip. It is created using the glPushMatrix() functions.

3.animate()

The animate function is used for plane transition. It is created using the glPushMatrix() functions.

4.airport()

The airport function is used for airport view. It is created using the glVertex3f() & glPushMatrix() functions.

5.controller()

The controller function is used for drawing pyramid.

6.myinit()

This function is used to maintain the aspect ratio of the window screen. It is called in the main function.

4.3 Source Code:

```
#include <windows.h>
#include<string.h>
#include<stdarg.h>
#include<stdio.h>
#include <glut.h>
static double x[10]={0},x2=0.0,r1=0.0;
static double yaxis[10]={-15,-15,-15,-15,-15,-15,-15,-15,-15,-15};
static double max=0;
static bool takeOff=false;
void
stroke_output(GLfloat x, GLfloat y, char *format,...)
{
        va_list args;
        char buffer[200], *p;
        va_start(args, format);
        vsprintf(buffer, format, args);
```

```
        va_end(args);
        glPushMatrix();
        glTranslatef(-2.5, y, 0);
        glScaled(0.003, 0.005, 0.005);
        for (p = buffer; *p; p++)
   glutStrokeCharacter(GLUT_STROKE_ROMAN, *p);
        glPopMatrix();
}
//runway strip
void strip(float x1)
{
glPushMatrix();
glRotatef(-65,0,1,0);
glColor3f(1,1,1);
glTranslatef(x1,-3.5,7.8);
glScaled(1,0.15,0.1);
glutSolidCube(1);
glPopMatrix();
}
void drawPlane(float y1){
/***********  PLANE CONSTRUCTION  ********************/
        glPushMatrix();
        // Main Body
glPushMatrix();
glScalef(.3,0.3,1.5);
if(y1<=15)
glColor3f(1,1.0,0.5);
if(y1>=15)
glColor3f(1,0.3,0.5);
glutSolidSphere(2.0,50,50);
glPopMatrix();
 glPushMatrix();
glTranslatef(0.0,0.1,-1.8);
glScalef(1.0,1,1.5);
```

```
glColor3f(0,0,1);
glutSolidSphere(0.5,25,25);
glPopMatrix();
//Left Fin
glPushMatrix();
glTranslatef(-1.0,0,0);
glScalef(1.5,0.1,0.5);
glColor3f(0,0,0);
glutSolidSphere(1.0,50,50);
glPopMatrix();
// Right Fin
glPushMatrix();
glTranslatef(1.0,0,0);
glScalef(1.5,0.1,0.5);
glColor3f(0,0,0);
glutSolidSphere(1.0,50,50);
glPopMatrix();
//right Tail fin
glPushMatrix();
glTranslatef(0.8,0,2.4);
glScalef(1.2,0.1,0.5);
glColor3f(0.0,0,0);
glutSolidSphere(0.4,50,50);
glPopMatrix();
//left Tail fin
glPushMatrix();
glTranslatef(-0.8,0,2.4);
glScalef(1.2,0.1,0.5);
glColor3f(0.0,0,0);
glutSolidSphere(0.4,50,50);
glPopMatrix();
//Top tail fin
glPushMatrix();
glTranslatef(0,0.5,2.4);
```

```
glScalef(0.1,1.1,0.5);
glColor3f(0.0,0,0);
glutSolidSphere(0.4,50,50);
glPopMatrix();
// Blades
glPushMatrix();
glRotatef(x2,0.0,0.0,1.0);
glPushMatrix();
glTranslatef(0,0.0,-3.0);
glScalef(1.5,0.2,0.1);
glColor3f(0.0,0,0);
glutSolidSphere(0.3,50,50);
glPopMatrix();
//blades
glPushMatrix();
glRotatef(90,0.0,0.0,1.0);
glTranslatef(0,0.0,-3.0);
glScalef(1.5,0.2,0.1);
glColor3f(0.0,0,0);
glutSolidSphere(0.3,50,50);
glPopMatrix();
glPopMatrix();
/* Blased End */
/*  Wheels  */
//Front
glPushMatrix();
glTranslatef(0.0,-0.8,-1.5);
glRotatef(90,0.0,1,0);
glScaled(0.3,0.3,0.3);
glutSolidTorus(0.18,0.5,25,25);
glColor3f(1,1,1);
glutSolidTorus(0.2,0.1,25,25);
glPopMatrix();
glPushMatrix();
```

```
glTranslatef(0.0,-0.4,-1.5);
glRotatef(20,0.0,1,0);
glScaled(0.05,0.3,0.05);
glutSolidSphere(1.0,25,25);
glPopMatrix();
//Rear
glPushMatrix();
glTranslatef(0.3,-0.8,0.7);
glRotatef(90,0.0,1,0);
glScaled(0.3,0.3,0.3);
glColor3f(0,0,0);
glutSolidTorus(0.18,0.5,25,25);
glColor3f(1,1,1);
glutSolidTorus(0.2,0.1,25,25);
glPopMatrix();
glPushMatrix();
glTranslatef(0.3,-0.4,0.7);
glRotatef(20,0.0,1,0);
glScaled(0.05,0.3,0.05);
glutSolidSphere(1.0,25,25);
glPopMatrix();
//rear 2
glPushMatrix();
glTranslatef(-0.3,-0.8,0.7);
glRotatef(90,0.0,1,0);
glScaled(0.3,0.3,0.3);
glColor3f(0,0,0);
glutSolidTorus(0.18,0.5,25,25);
glColor3f(1,1,1);
glutSolidTorus(0.2,0.1,25,25);
glPopMatrix();
glPushMatrix();
glTranslatef(-0.3,-0.4,0.7);
glRotatef(20,0.0,1,0);
```

```
glScaled(0.05,0.3,0.05);
glutSolidSphere(1.0,25,25);
glPopMatrix();
glPopMatrix();
}
void animate(float y1,float x1){
        // Plane Transition
        glPushMatrix();
        //Move  the Plane towards rotating zone
        if(y1<=-2){
        glTranslatef(5.5+y1,3,0);
        glRotatef(-90,0,1,0);
        }
        // Move the Plane towards 2nd runwat
if(takeOff)
        if(y1>=15){
        glRotatef(140,0,1,0);
                if(y1>=15 && y1<=20)
                glTranslatef(2+15-y1,-3,-3);
                if(y1>=20)
                glTranslatef(2+15-y1,-3-20+y1,-3);
        }
        // keep rotating the plane
        if(y1>=-2 && y1<=2){
        glTranslatef(3.0,3.0,0.0);
        }
        //Start desending the plane
if(y1>=2 && y1<=6.5)
        {
        glTranslatef(3,3-y1+2,0);
        }
        // move towards runway
        if(y1>=6.5 && y1<=8.2)
        {
```

```
        glTranslatef(3-y1+6.5,3-y1+2,0);
        }
        // landing only change the x-axis
                if(y1>=8.2 && y1<=15)
        {
        glTranslatef(3-y1+6.5,3-8.2+2,0);
        }
void airport(){
        //Floor
        glColor3f(0,1,0);
        glBegin(GL_POLYGON);
glVertex3f(-19,-3.5,19);
glVertex3f(-19,-3.5,-19);
glVertex3f(19,-3.5,-19);
glVertex3f(19,-3.5,19);
        glEnd();
glPushMatrix();
// runway landing
glPushMatrix();
glColor3f(1,1,1);
glTranslatef(0,-3.5,-1);
glScaled(17,0.1,1);
glutSolidCube(1);
// Start your Drawing ---Draw pyramid
void controller()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();
        glTranslatef(0.0,0.0,-25.0);
//q      glRotatef(-90,0,1,0);
        animate(yaxis[0],x[0]);
        for(int i=0;i<max;i++){
if(yaxis[i]>=-5){               //wait until previous plane reaches safe location
        animate(yaxis[i+1],x[i+1]);
```

```
        if(yaxis[i+1]>=-2 && yaxis[i+1]<=6.7)
        // Rotate until y-axis of plane is less than 6.7
        x[i+1]+=3.5;
        // Conditions to increase or decrease the speed of plane
        if(yaxis[i+1]<=0)
        yaxis[i+1]+=0.15;
        else if(yaxis[i+1]>=0 && yaxis[i+1]<=6.7)
                yaxis[i+1]+=0.06;
        else if(yaxis[i+1]>=6.7 && yaxis[i+1]<=15)
                yaxis[i+1]+=0.1;
        else if(takeOff && yaxis[i+1]<=30)
                yaxis[i+1]+=0.1;
        }
        }
void display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        glTranslatef(0.0f,0.0f,-13.0f);
        stroke_output(-2.0, 1.7, "p--> Pyramid Clockwise");
                stroke_output(-2.0, 1.0, "P--> Pyramid Anti Clockwise");
stroke_output(-2.0, 0.3, "h--> House Clockwise");
                                stroke_output(-2.0, -0.4, "H--> House Anti-Clockwise");
                                        stroke_output(-2.0, -1.1, "q--> quit");
        GLfloat mat_ambient[]={0.0f,1.0f,2.0f,1.0f};
        GLfloat mat_diffuse[]={0.0f,1.5f,.5f,1.0f};
        GLfloat mat_specular[]={5.0f,1.0f,1.0f,1.0f};
        GLfloat mat_shininess[]={50.0f};
        glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
        glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
        glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
        glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
        GLfloat lightIntensity[]={3.7f,0.7f,0.7f,1.0f};
        GLfloat light_position[]={0.0f,3.0f,2.0f,0.0f};
```

```
        glLightfv(GL_LIGHT0,GL_POSITION,light_position);
        glLightfv(GL_LIGHT0,GL_DIFFUSE,lightIntensity);
        glutIdleFunc(controller);
        glFlush();
        glutSwapBuffers();
}
void menu(int id)
{
        switch(id)
{
        case 1:max+=1;
                break;
        case 2:max-=1;
                break;
        case 3:takeOff=!takeOff;
                break;
        case 4:exit(0);
                break;
        }
        glFlush();
        glutSwapBuffers();
        glutPostRedisplay();
}
int main(int argc, char *argv[])
{
   glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
        glutInitWindowSize(1000,480);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Glut Plane");
        glutDisplayFunc(display);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glShadeModel(GL_SMOOTH);
```

```
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_NORMALIZE);
        glutKeyboardFunc(mykey);
        glutCreateMenu(menu);
   glutAddMenuEntry("Add Plane      'a'",1);
        glutAddMenuEntry("Remove         'r'",2);
        glutAddMenuEntry("Takeoff        'f'",3);
        glutAddMenuEntry("Quit           'q'",4);
        glutAttachMenu(GLUT_LEFT_BUTTON);
        glutAttachMenu(GLUT_RIGHT_BUTTON);
doInit();
   glutMainLoop();
        return 0;
}
```

CHAPTER 5

TESTING AND SNAPSHOTS

5.1 Testing

Testing is the process of executing a program to find the errors. A good test has the high probability of finding a yet undiscovered error. A test is vital to the success of the system. System test makes a logical assumption that if all parts of the system are correct, then goal will be successfully achieved.

5.2 Test Cases

SLNO	Name of the test	Expected output	Result	Remarks
1.	With the press of the key ‘p’	Draw pyramid	Displays pyramid	Pass
2.	With the press of the key ‘f’	Take off	Displays plane taking off	Pass
3.	With the press of the key ‘a’	Add plane	A new plane is added	Pass
4.	With the press of the key ‘r’	Remove plane	A plane is removed	Pass
5.	With the press of the key ‘q’	Exits screen	The window exits	Pass

Table 5.1-Test cases

5.3 SnapShots

In computer file systems, a snapshot is a copy of a set of files and directories as they were at a particular point in the past. The term was coined as an analogy to that in photography.

One approach to safely backing up live data is to temporarily disable write access to data during the backup, either by stopping the accessing applications or by using the locking API provided by the operating system to enforce exclusive read access. This is tolerable for low-availability systems.

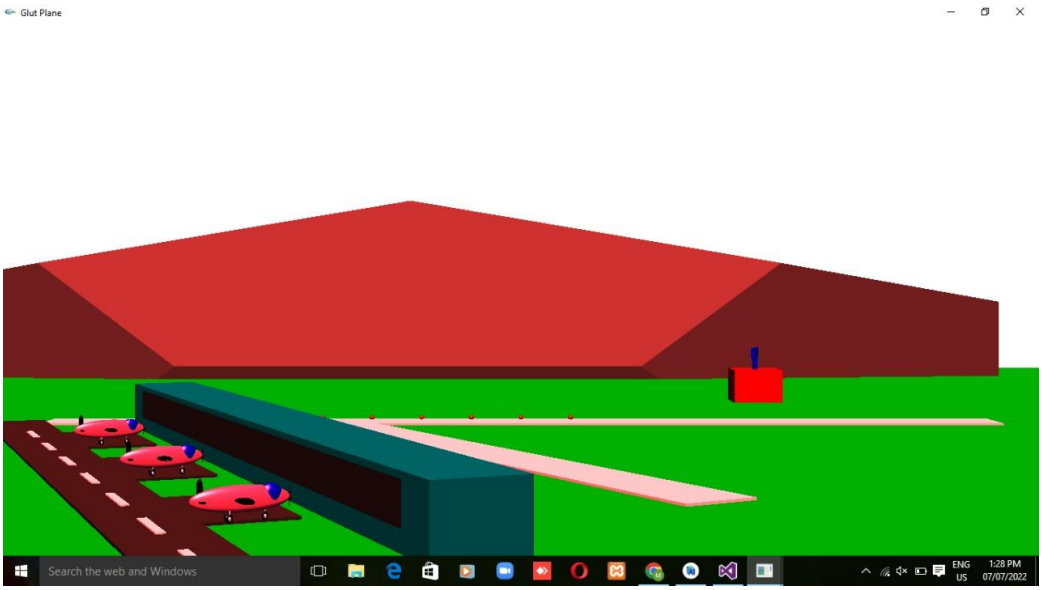

Fig 5.1: Snapshot-Airport view

The above snapshot displays the airport view.

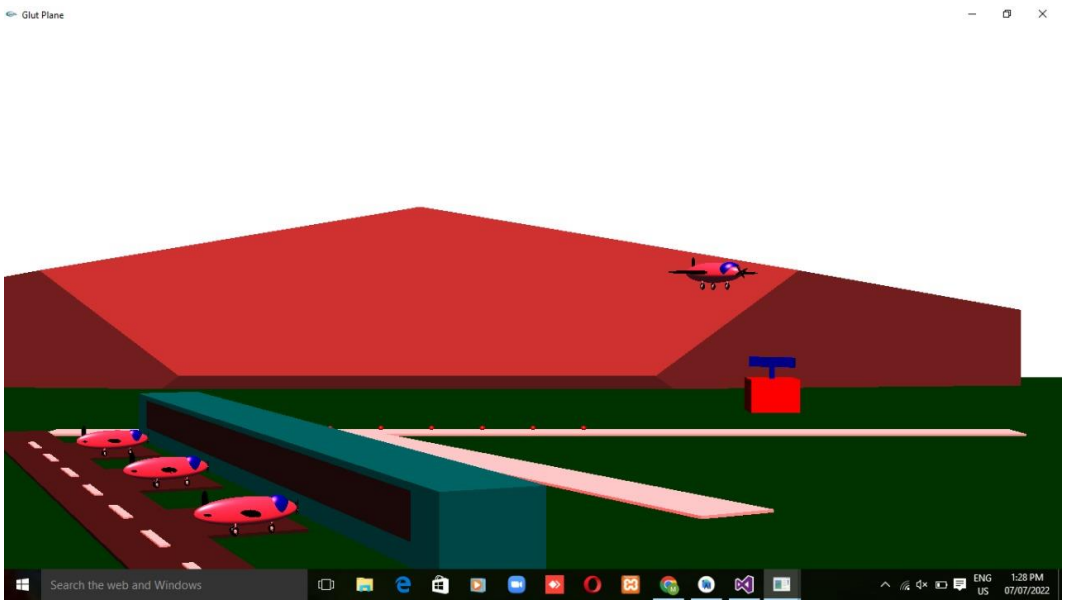

Fig 5.2: Snapshot-Take off

The above snapshot displays the plane taking off.

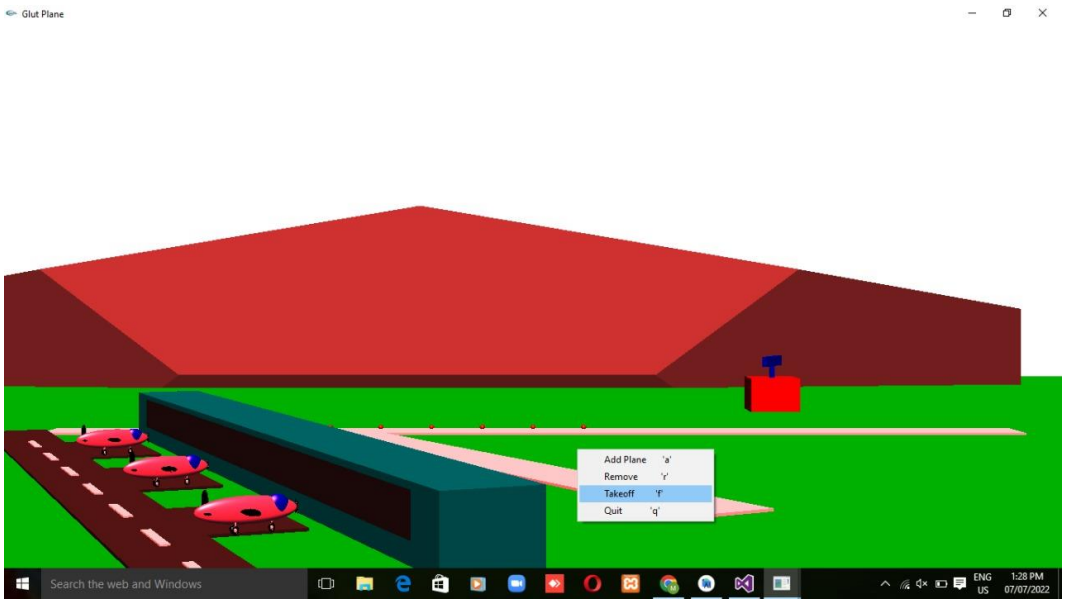

Fig 5.3: Snapshot-Add plane

The above snapshot displays adding plane.

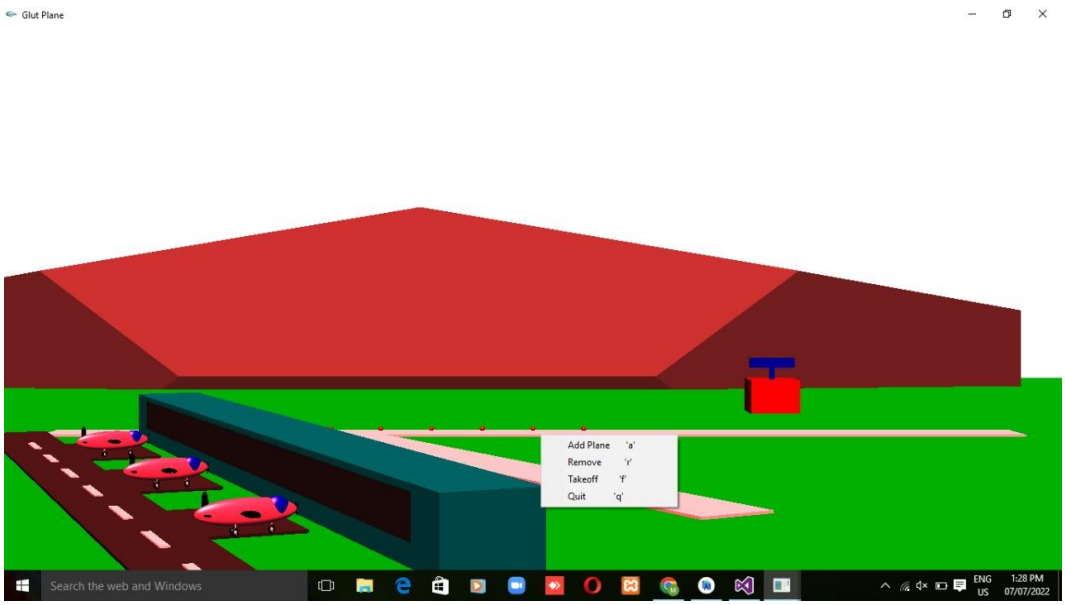

Fig 5.5: Snapshot-Removing plane

The above snapshot displays removing plane.

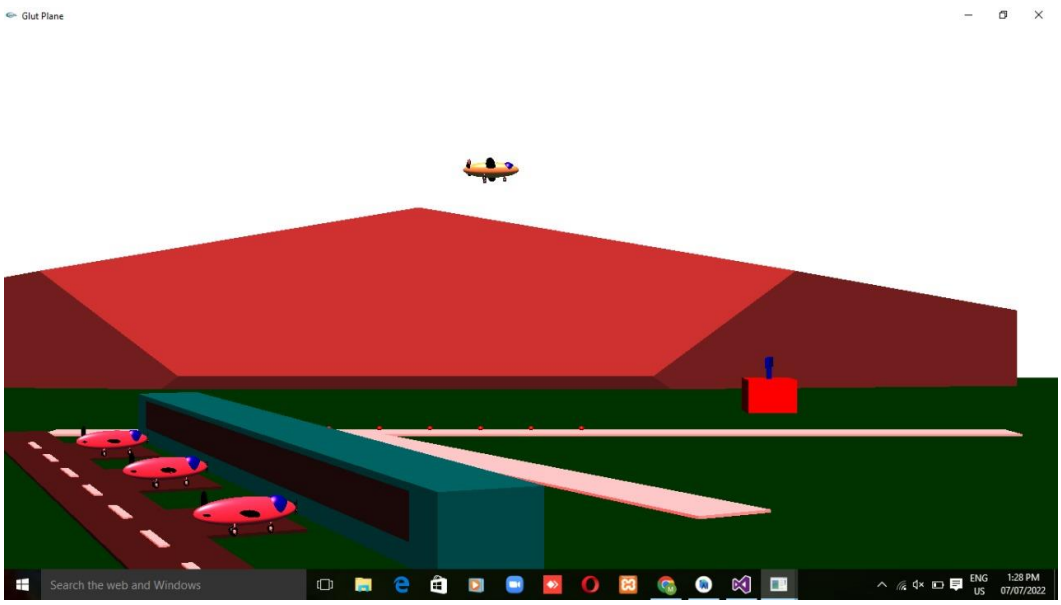

Fig 5.4: Snapshot-Landing plane

The above snapshot displays landing plane.

CHAPTER 6

FUTURE ENHANCEMENT

This project can be improved in many ways. Time constraint and deficient knowledge has been a major factor in limiting the features offered by this display model.

We have put in place a “AIR TRAFFIC CONTROLLER” which is a colorful and simple mini project. And this project includes a lot of options in it. This is the idea to implement the simple objects as we have used in this project we can also develop simple objects to this project and it will look even better. For the future implementations try to create a road and some vehicles moving in the road and etc.

CHAPTER 7

CONCLUSION

A “AIR TRAFFIC CONTROLLER ” which is a colorful and simple mini project. And this project includes a lot of options in it. We have given you the idea to implement the simple objects as we have used in this project you can also develop simple objects to this project and it will look even better. For the future implementations try to create a road and some vehicles moving in the road and etc. This project aims at using glut pre-built model sub-api and callback functions.

Finally we conclude that this program clearly illustrate the use of glut model sub-api. and has been completed successfully and is ready to be demonstrated.

An attempt has been made to develop an OpenGL package which meets necessary requirements of the user successfully. Since it is user friendly, it enables the user to interact efficiently and easily.

The development of the mini project has given us a good exposure to OpenGL by which we have learnt some of the technique which help in development of animated pictures, gaming. Hence it is helpful for us even to take up this field as our career too and develop some other features in OpenGL and provide as a token of contribution to the graphics world.

CHAPTER 8

REFERENCES

We have obtained information from many resources to design and implement our project successively. We have acquired most of the knowledge from related websites. The following are some of the resources :

- Text books :

Interactive computer graphics a top-down approach

-By Edward Angel.

- Computer graphics,principles & practices

  - Foley van dam
  - Feiner hughes

- Web references:

  http://jerome.jouvie.free.fr/OpenGl/Lessons/Lesson3.php

  http://google.com

  http://opengl.org