

CHAPTER 1

INTRODUCTION

Client-server architecture provides a structured framework for dividing the functionalities of a system into two major components: the client and the server.

In this architecture, the client is a software application or device that interacts with the user and initiates requests for services or resources. It can be a web browser, a mobile app, or any other software that enables user interaction. The client is responsible for presenting information to the user, gathering user input, and transmitting requests to the server.

In our project, we have implemented this architecture through computer graphics with some useful functions like OpenGL. Through this visual illustration, one can understand how the client and server communication works. Here handshaking process, uploading and downloading operations are shown.

This project makes it simple to everyone to understand the way how the communication between client and server happens. The communication use TCP/IP - UDP protocols and channels which is visualised in a form of pipelines that connect from one client to the server.

1.1 OBJECTIVE OF THE PROJECT

- 1) The main objective of the project is to display the communication between client and server using controls.
- 2) To implement the concepts of Computer Graphics we have learnt.
- 3) To implement movements of the planets.
- 4) To incorporate colouring effects.

CHAPTER 2

LITERATURE REVIEW

2.1 HISTORY OF CLIENT SERVER ARCHITECTURE

Client-server architecture is a model that has been widely used in the field of computer networking and distributed computing. It defines a relationship between two entities: a client, which requests services or resources, and a server, which provides those services or resources. The architecture has evolved over several decades, so let's explore its history.

1. EARLY DAYS:

The concept of client-server architecture emerged in the late 1960s and early 1970s when mainframe computers were prevalent. During this time, the dominant model was a centralized system where a single mainframe served multiple dumb terminals. The mainframe acted as a server, processing requests from the terminals (clients) and sending the results back.

2. 1980S - RISE OF LOCAL AREA NETWORKS (LANs):

With the advent of LANs in the 1980s, client-server architecture gained more prominence. Local networks enabled the connection of personal computers (PCs), which could act as both clients and servers. This decentralization of computing power allowed for greater flexibility and improved resource utilization.

3. 1990S - CLIENT-SERVER COMPUTING BOOM:

The 1990s witnessed a significant expansion of client-server architecture. This was fueled by the growth of the internet, advancements in networking technologies, and the increasing affordability of personal computers. The World Wide Web, introduced in the early 1990s, popularized the use of client-server architecture for web applications. The client, typically a web browser, made HTTP requests to web servers and received HTML documents in response.

4. DISTRIBUTED COMPUTING PARADIGMS:

As technology advanced, client-server architecture extended beyond the traditional model. Distributed computing paradigms such as peer-to-peer (P2P) networks, grid computing, and cloud computing emerged. These models utilized the principles of client-server architecture but distributed the workload across multiple servers or nodes, allowing for improved scalability and fault tolerance.

5. MODERN ERA - WEB SERVICES AND APIS:

In the 2000s and beyond, client-server architecture continued to evolve with the rise of web services and application programming interfaces (APIs). Web services allowed applications to expose their functionalities as services that could be accessed remotely over the internet. Clients could consume these services by making requests to the corresponding servers using standardized protocols such as SOAP (Simple Object Access Protocol) and later REST (Representational State Transfer).

6. CLIENT-SERVER IN MOBILE AND IOT:

The proliferation of mobile devices and the Internet of Things (IoT) brought client-server architecture to new frontiers. Mobile apps and IoT devices often rely on client-server interactions to access cloud-based services, process data, and perform computations on remote servers.

Overall, client-server architecture has played a crucial role in enabling distributed computing, scalability, and the seamless integration of various technologies. Its evolution continues to shape the way applications and systems are designed, developed, and deployed in the modern computing landscape.

2.2 HISTORY OF COMPUTER GRAPHICS

Computer graphics deals with generating images with the aid of computers. Today, computer graphics is a core technology in digital photography, film, video games, cell phone and computer display, and many specialized applications.

The first cathode ray tube, the Braun tube, was invented in 1897 – it in turn would permit the oscilloscope and the military control panel – the more direct precursors of the field, as they provided the first two-dimensional electronic displays that responded to programmatic or user input. New kinds of displays were needed to process the wealth of information resulting from such projects, leading to the development of computer graphics as a discipline.

In 1996, Krishnamurthy and Lavoy invented normal mapping – an improvement on Jim Blinn's bump mapping. By the end of the decade, computers adopted common frameworks for graphics processing such as DirectX and OpenGL. Since then, computer graphics have only become more detailed and realistic, due to more powerful graphics hardware and 3D modelling software.

2.3 HISTORY OF OPENGL

By the early 1990s, Silicon Graphics (SGI) was a leader in 3D graphics for workstations. Their IRIS GL API became the industry standard, used more widely than the open standards-based PHIGS. This was because IRIS GL was considered easier to use, and because it supported immediate mode rendering. By contrast, PHIGS was considered difficult to use and outdated in functionality.

SGI's competitors (including Sun Microsystems, Hewlett-Packard and IBM) were also able to bring to market 3D hardware supported by extensions made to the PHIGS standard, which pressured SGI to open source a version of IrisGL as a public standard called OpenGL.

However, SGI had many customers for whom the change from IrisGL to OpenGL would demand significant investment. Moreover, IrisGL had API functions that were irrelevant to 3D graphics. For example, it included a windowing, keyboard and mouse API, in part because it was developed before the X Window System and Sun's NeWS. And, IrisGL libraries were unsuitable for opening due to licensing and patent issues. These factors required SGI to continue to support the advanced and proprietary Iris Inventor and Iris Performer programming APIs while market support for OpenGL matured.

CHAPTER 3

REQUIREMENT SPECIFICATION

3.1 SYSTEM REQUIREMENT

The basic requirements for the development of this mini project are as follows:

3.1.1 HARDWARE CONSTRAINTS

- Processor : Pentium PC
- RAM : 512MB
- Hard Disk : 20GB(approx)
- Display : VGA Color Monitor

3.1.2 SOFTWARE CONSTRAINTS

- Software: OpenGL 4.3 or above
- Operating System : Windows 98SE/2000/XP/Vista/UBUNTU
- Compiler : Eclipse/Microsoft Visual studio 2005
- Programming languages: C/C++

3.2 DEVELOPMENT ENVIRONMENT

1. SOFTWARE – OPENGL:

OpenGL (Open Graphics Library) is a cross-language, cross-platform 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.

Silicon Graphics, Inc. (SGI) began developing OpenGL in 1991 and released it on June 30, 1992; applications use it extensively in the fields of computer-aided design (CAD), virtual reality, scientific visualization, information visualization, flight simulation, and video games. Since 2006, OpenGL has been managed by the non-profit technology consortium Khronos Group.

The OpenGL specification describes an abstract API for drawing 2D and 3D graphics. Although it is possible for the API to be implemented entirely in software, it is designed to be implemented mostly or entirely in hardware. The API is defined as a set of functions which may be called by the client program, alongside a set of named integer constants.

	
Original author(s)	Silicon Graphics
Developer(s)	Khronos Group
Initial release	June 30, 1992
Stable release	4.6 / July 31, 2017
Written in	C
Type	3D graphics API
License	<p>Open source license for use of the S.I. This is a Free Software License B closely modeled on BSD, X, and Mozilla licenses.</p> <p>Trademark license for new licensees who want to use the OpenGL trademark and logo and claim conformance.</p>
Website	opengl.org

Fig 3.1 OpenGL

2. DEVELOPMENT TOOL – MICROSOFT VISUAL STUDIO:

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop computer programs, as well as websites, web apps, web services, mobile apps and GUI applications. Visual Studio uses Microsoft software development platforms such as Windows API, Windows Forms, Windows Presentation Foundation, Windows Store and Microsoft Silverlight. It can produce both native code and managed code.


Microsoft Visual Studio	
	
Developer(s)	Microsoft
Stable release	2019 version 16.9.4 (April 13, 2021)
Preview release	2019 version 16.10.0 (April 22, 2021)
Operating system	Windows 7 SP1 and later, Windows Server 2012 R2 and later
Available in	13 languages
Type	Integrated development environment
License	Freemium
Website	visualstudio.microsoft.com

Fig 3.2 Microsoft Visual Studio

CHAPTER 4

SYSTEM DESIGN

4.1 FLOW CHART OF THE PROJECT

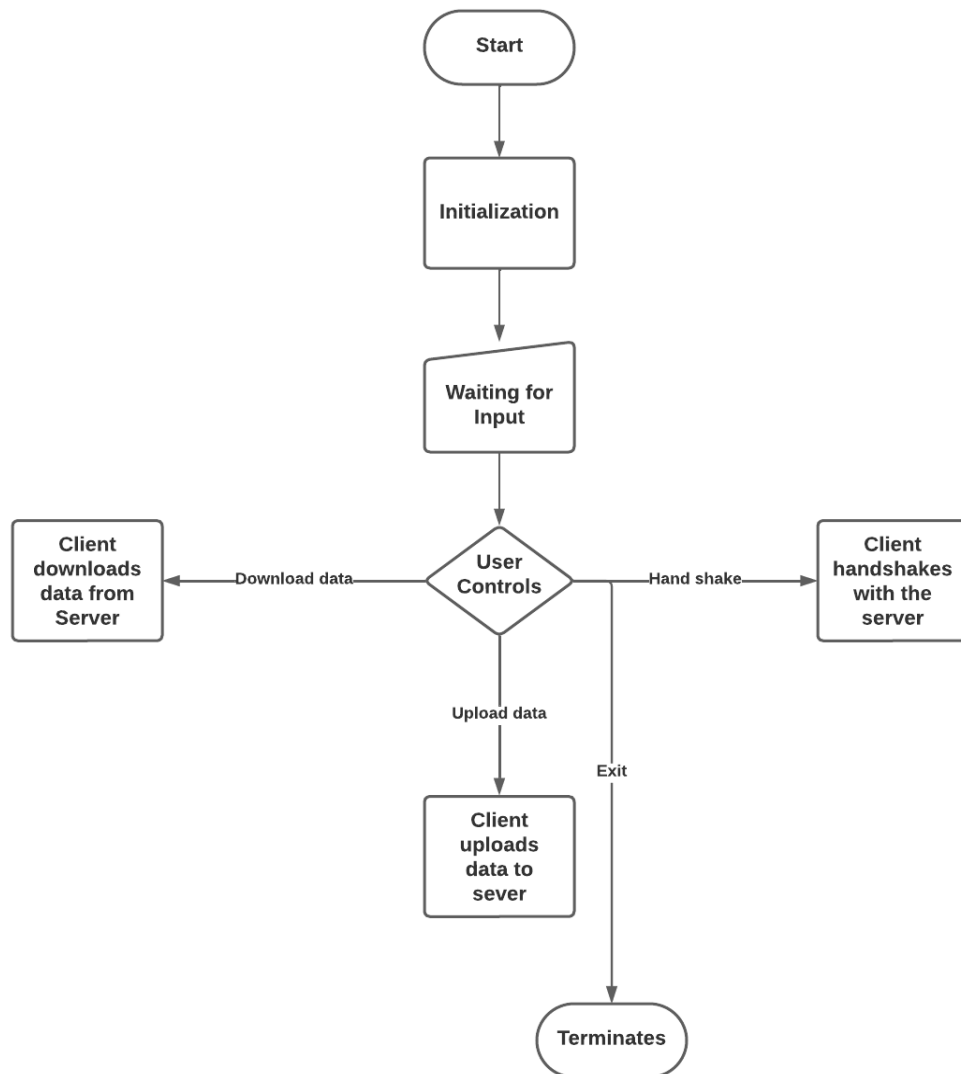


Fig 4.1 - Flow Chart of the Project

The above flow chart describes how our project works and the flow of the process. The user can use the available options to illustrate each process. The user controls are defined in the following section.

4.2 USER CONTROLS

Using the mouse control and the options available, we are able to start the communication between client and server.

On right click , 5 options will be listed

- Client 1 - Server(TCP)
- Client 2 - Server(UDP)
- Client 3 – Server
- Client 4 – Server

On clicking first option , we get three more options

- Hand shake
- Upload data
- Download data

We get similar options on clicking the rest of the options. But handshake option is available for the first option as the project is designed that way.

By using these options communication can be visualised on the output screen which are shown in the sample screenshots of the project section

CHAPTER 5

SYSTEM IMPLEMENTATION

5.1 OPENGL LIBRARIES

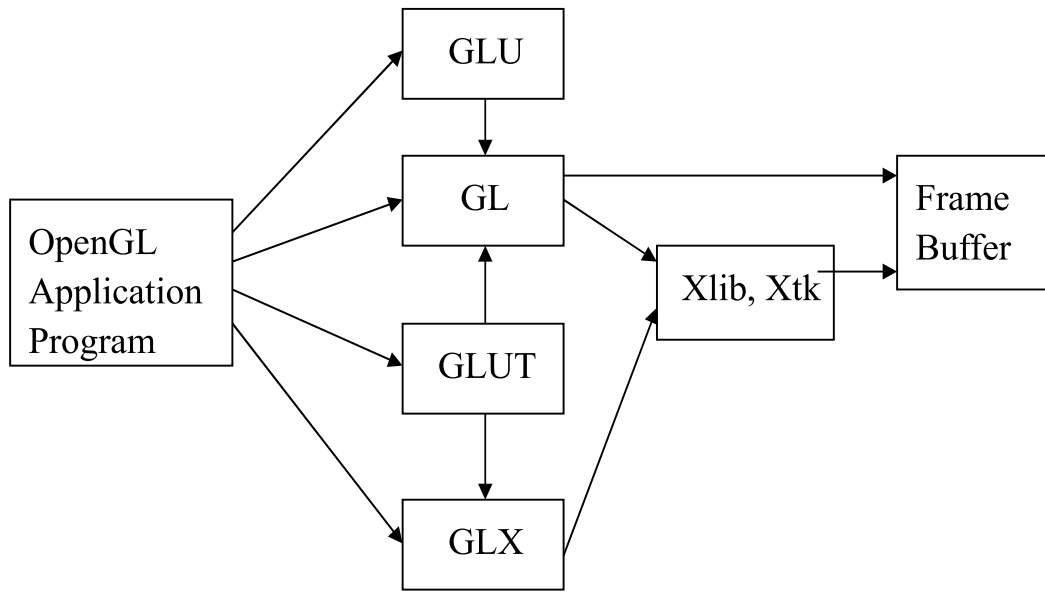


Fig 5.1 OpenGL Libraries

Fig 5.1 shows the organization of the libraries for an X Window System environment. For this window system, GLUT will use GLX and the X libraries. The application program, however, can use only GLUT functions and thus can be recompiled with the GLUT library for other window systems.

1. GLU LIBRARY:

The OpenGL Utility Library (GLU) is a computer graphics library for OpenGL. It consists of a number of functions that use the base OpenGL library to provide higher-level drawing routines from the more primitive routines that OpenGL provides. It is usually distributed with the base OpenGL package.

Among these features are mapping between screen- and world-coordinates, generation of texture mipmaps, drawing of quadric surfaces, NURBS, tessellation of polygonal primitives, interpretation of OpenGL error codes, an extended range of transformation routines for setting up

viewing volumes and simple positioning of the camera, generally in more human-friendly terms than the routines presented by OpenGL. All GLU functions start with the 'glu' prefix.

2. GL LIBRARY:

A graphics library is a program library designed to aid in rendering computer graphics to a monitor. This typically involves providing optimized versions of functions that handle common rendering tasks. This can be done purely in software and running on the CPU, common in embedded systems, or being hardware accelerated by a GPU, more common in PCs. By employing these functions, a program can assemble an image to be output to a monitor. Graphics libraries are mainly used in video games and simulations.

3. GLUT LIBRARY:

The OpenGL Utility Toolkit (GLUT) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres and the Utah teapot. All GLUT functions start with the 'glut' prefix. (for example, "glutPostRedisplay" marks the current window as needing to be redrawn).

FreeGLUT is an open-source alternative to the OpenGL Utility Toolkit (GLUT) library. GLUT (and hence FreeGLUT) allows the user to create and manage windows containing FreeGLUT is intended to be a full replacement for GLUT, and has only a few differences.

4. GLX LIBRARY:

GLX (initialism for "OpenGL Extension to the X Window System") is an extension to the X Window System core protocol providing an interface between OpenGL and the X Window System as well as extensions to OpenGL itself. It enables programs wishing to use OpenGL to do so within a window provided by the X Window System. GLX distinguishes two "states": indirect state and direct state.

5. FRAME BUFFER:

A Framebuffer is a collection of buffers that can be used as the destination for rendering. OpenGL has two kinds of framebuffers: the Default Framebuffer, which is provided by the OpenGL Context; and user-created framebuffers called Framebuffer Objects (FBOs). The buffers for default framebuffers are part of the context and usually represent a window or display device. The buffers for FBOs reference images from either Textures or Render buffers; they are never directly visible.

Default framebuffers cannot change their buffer attachments, but a particular default framebuffer may not have images associated with certain buffers. For example, the `GL_BACK_RIGHT` buffer will only have an image if the default framebuffer is double-buffered and uses stereoscopic 3D.

5.2 OPENGL PRIMITIVES

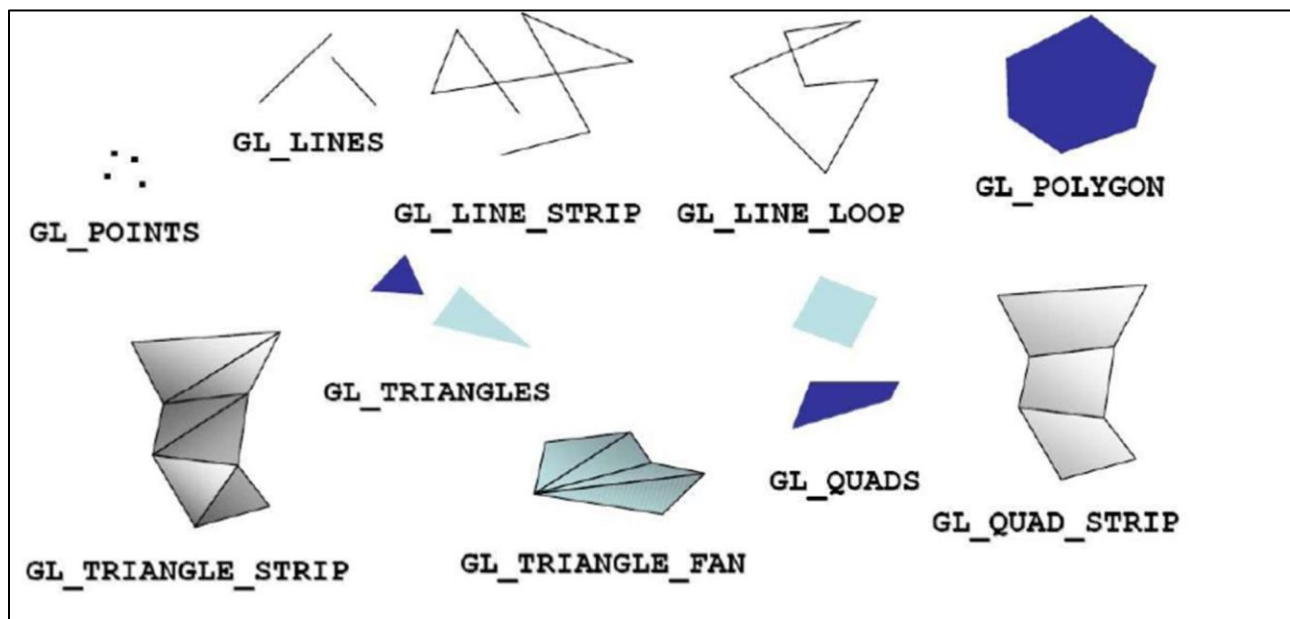


Fig 5.2 OpenGL Primitives

The term Primitive in OpenGL is used to refer to two similar but separate concepts. The first meaning of "Primitive" refers to the interpretation scheme used by OpenGL to determine what a stream of vertices represents when being rendered ex: "`GL_POINTS`". Such sequences of vertices can be arbitrarily long.

1. POINT PRIMITIVE:

There is only one kind of point primitive:

- `GL_POINTS`: This will cause OpenGL to interpret each individual vertex in the stream as a point. Points that have a Texture mapped onto them are often called "point sprites".

2. LINE PRIMITIVE:

There are 3 kinds of line primitives, based on different interpretations of a vertex stream.

- `GL_LINES`: Vertices 0 and 1 are considered a line. Vertices 2 and 3 are considered a line.
- `GL_LINE_STRIP`: The adjacent vertices are considered lines. Thus, if you pass n vertices, you will get $n-1$ lines. If the user only specifies 1 vertex, the drawing command is ignored.
- `GL_LINE_LOOP`: As line strips, except that the first and last vertices are also used as a line. Thus, you get n lines for n input vertices. If the user only specifies 1 vertex, the drawing command is ignored. The line between the first and last vertices happens after all of the previous lines in the sequence.

3. TRIANGLE PRIMITIVES:

A triangle is a primitive formed by 3 vertices. It is the 2D shape with the smallest number of vertices, so renderers are typically designed to render them. Since it is created from only 3 vertices, it is also guaranteed to be planar. There are 3 kinds of triangle primitives, based again on different interpretations of the vertex stream:

- `GL_TRIANGLES`: Vertices 0, 1, and 2 form a triangle, Vertices 3, 4, and 5 form a triangle, and so on.
- `GL_TRIANGLE_STRIP`: Every group of 3 adjacent vertices forms a triangle. The face direction of the strip is determined by the winding of the first triangle. Each successive triangle will have its effective face order reversed, so the system compensates for that by testing it in the opposite way. A vertex stream of n length will generate $n-2$ triangles.

4. QUADS:

A quad is a 4-vertex quadrilateral primitive. The four vertices are expected to be coplanar; failure to do so can lead to undefined results. A quad is typically rasterized as a pair of triangles. This is not defined by the GL specification, but it is allowed by it. This can lead to some artifacts due to how vertex/geometry shader outputs are interpolated over the 2 generated triangles.

- `GL_QUADS`: Vertices 0-3 form a quad, vertices 4-7 form another, and so on. The vertex stream must be a number of vertices divisible by 4 to work
- `GL_QUAD_STRIP`: Similar to triangle strips, a quad strip uses adjacent edges to form the next quad. In the case of quads, the third and fourth vertices of one quad are used as the edge of the next quad.

5.3 HEADER FILES

The headers that are used are as follows:

- `#include<GL/glut.h>`: To include glut library files.
- `#include<stdio.h>`: To include standard input and output files.
- `#include<math.h>`: To include various mathematical functions.

5.4 FUNCTIONS

- `void glScalef (TYPE sx, TYPE sy, TYPE sz)` alters the current matrix by a scaling of (sx, sy, sz). TYPE here is `GLfloat`. Here in the above considered example we use scaling to minimize the length of the curve at each iteration. For this curve we use the scale factor to be 3 units because we substitute a line by 4 lines in each iteration.
- `void glRotatef(TYPE angle, TYPE dx, TYPE dy, TYPE dz)` alters the current matrix by a rotation of angle degrees about the axis(dx, dy, dz). TYPE heris `GLfloat`. For a Koch curve we rotate by 60° about the z-axis.

- `void glTranslatef(TYPE x, TYPE y, TYPE z)` alters the current matrix by a displacement of (x, y, z). TYPE here is GLfloat. We need to translate to display the new position of the line from the old position and also to go out to the beginning of the next side while drawing.
- `void glLoadIdentity()` sets the current transformation matrix to an identity matrix.
- `void glPushMatrix()` pushes to the matrix stack corresponding to the current matrix mode.
- `void glPopMatrix()` pops from the matrix stack corresponding to the current matrix mode.
- `void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)` defines a two-dimensional viewing rectangle in the plane $z=0$.
- `void glutMouseFunc(myMouse)` refers to the mouse callback function. Here mouse interface is given to increase a level of recursion by clicking mouse button and also to decrease a level of recursion by doing the same holding the shift on the keyboard.
- `void glutKeyboardFunc(myKey)` refers to the keyboard callback function. Here keyboard interface is given to quit, the user can quit by pressing 'q' and to see next example of the implementation, the user should press 'n'.
- `void glutInit(int *argc, char**argv)` Initializes GLUT< the arguments from main are passed in and can be by the application.
- `void glutCreateWindow(char *title)` Creates a window on the display. The string title can be used to label the window. The return value provides a reference to the window that can be used when there are multiple windows.
- `void glutInitDisplayMode(unsigned int mode)` Requests a display with the properties in mode. The value of mode is determined by the logical OR of options including the color model (GLUT_RGB<GLUT_INDEX) and buffering (GLUT_SINGLE<GLUT_DOUBLE).
- `void glutInitWindowSize(int width,int heights)` Specifies the initial height and width of the window in pixels.
- `void glutInitWindowPosition(int x,int y)` Specifies the initial position of the top-left corner of the window in pixels.
- `void glutMainLoop()` Cause the program to enter an event –processing loop.it should be the statement in main.

- `void glutPostRedisplay()` Requests that the display callback be executed after the current callback returns.
- `void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble atx, GLdouble aty, GLdouble atz, GLdouble upx, GLdouble upy, GLdouble upz)` o Postmultiplies the current matrix determined by the viewer at the eye point looking at the point with specified up direction.
- `void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far)` o Defines a perspective viewing volume using the y direction field of view fovy measured in degree, the aspect ratio of the front clipping plane, and the near and far distance.

5.5 SOURCE CODE

```

#include<GL/glut.h>
#include<stdio.h>
#include<math.h>
const float DEG2RAD = 3.14159 / 180;
int flag = 0, count = 0, count1 = 0, dt = 0,
dd = 0,
dx = 0, da = 0, db = 0, dat1 = 0, dat2 = 0;
float pos1 = 110.0, pos2 = 300.0;
char* ptr1, * ptr2, * ptr3, * ptr4, * ptr5, *
ptr6,
* ptr7, * ptr8, * ptr9;
int len1, len2, len3, len4, len5, len6, len7;
void one();
float angle, p1 = 107.0, p2 = 300.0, p3 =
175.0,
p4 = 200.0, p5 = 175.0, p6 = 200.0, p7 =
100.0, p8 = 115.0, p9 = 185.0, p10 = 99.0,
p11 = 210.0, p12 = 280.0;
float p13 = 300.0, p14 = 262.0, p15 =
300.0, p16 = 114.0, p17 = 216.0, p18 =
99.0;
void download();
void response();
void downloading();
void upload();
void draw(float radius)
{
float delta_theta = 0.001;
glBegin(GL_POLYGON);
GL_LINE_LOOP
glEnable(GL_POINT_SMOOTH);
glHint(GL_POINT_SMOOTH_HINT,
GL_NICEST);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
for (angle = 0; angle < 2 * 3.1415; angle +=
delta_theta)
glVertex3f(radius * cos(angle), radius*
sin(angle), 0);
glEnd();
}
void Init()
{
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT
GL_DEPTH_BUFFER_BIT);
glMatrixMode(GL_PROJECTION);
gluOrtho2D(0.0, 400.0, 0.0, 400.0);
glMatrixMode(GL_MODELVIEW);
glOrtho(-1.0, 1.0, -1.0, 1.0, -10.0, 10.0);
}
void line()
{
glColor3f(0.22, 0.22, 0.22);
glBegin(GL_POLYGON);
glVertex2i(30, 60);
glVertex2i(80, 60);
glVertex2i(80, 115);
glVertex2i(30, 115);
glEnd();glColor3f(0.0, 0.0, 0.0);
glBegin(GL_POLYGON);
glVertex2f(32.0, 64.0);
glVertex2f(78.0, 64.0);
glVertex2f(78.0, 112.0);
glVertex2f(32.0, 112.0);
glEnd();
glColor3f(0.22, 0.22, 0.22);
glBegin(GL_POLYGON);
glVertex2f(30.0, 44.0);
glVertex2f(80.1, 44.0);
glVertex2f(80.1, 60.0);
glVertex2f(30.0, 60.0);
glEnd();
glColor3f(0.22, 0.22, 0.22);
glBegin(GL_POLYGON);
glVertex2f(51.0, 34.0);
glVertex2f(51.0, 44.0);
glVertex2f(59.0, 44.0);
glVertex2f(59.0, 34.0);
glVertex2f(64.0, 34.0);
glVertex2f(64.0, 32.0);
glVertex2f(46.0, 32.0);
glVertex2f(46.0, 34.0);
glVertex2f(51.0, 34.0);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_LINES);

```

```
glVertex2f(29.9, 44);
glVertex2f(29.99, 115.2);
glVertex2f(80.1, 44);
glVertex2f(80.1, 115.2);
glVertex2f(29.9, 115.2);
glVertex2f(80.1, 115.29);
glEnd();
glColor3f(0.22, 0.22, 0.22);
glBegin(GL_POLYGON);
glVertex2f(90.0, 32.0);
glVertex2f(110.0, 32.0);
glVertex2f(110.0, 115.0);
glVertex2f(90.0, 115.0);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_POLYGON);
glVertex2f(92.0, 103.0);
glVertex2f(108.0, 103.0);
glVertex2f(108.0, 112.0);
glVertex2f(92.0, 112.0);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_POLYGON);
glVertex2f(95.0, 89.0);
glVertex2f(103.0, 89.0);
glVertex2f(103.0, 98.0);
glVertex2f(95.0, 98.0);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_POLYGON);
glVertex2f(105.0, 95.0);
glVertex2f(108.0, 95.0);
glVertex2f(108.0, 98.0);
glVertex2f(105.0, 98.0);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_POLYGON);
glVertex2f(105.0, 89.0);
glVertex2f(108.0, 89.0);
glVertex2f(108.0, 92.0);
glVertex2f(105.0, 92.0);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glPointSize(3.5);
glBegin(GL_POINTS);
glVertex2f(93.0, 96.0);

glVertex2f(93.0, 91.0);
glEnd();
glColor3f(1.0, 1.0, 1.0);
glPointSize(0.5);
glBegin(GL_POINTS);
glVertex2f(106.0, 96.5);
glVertex2f(107.0, 96.5);
glEnd();
glEnd();
glColor3f(2.0, 0.2, 0.2);
glBegin(GL_LINES);
glVertex2f(92.0, 80.0);
glVertex2f(108.0, 80.0);
glEnd();
glColor3f(2.0, 0.2, 0.2);
glBegin(GL_LINES);
glVertex2f(92.0, 75.0);
glVertex2f(108.0, 75.0);
glEnd();
glColor3f(2.0, 0.2, 0.2);
glBegin(GL_LINES);
glVertex2f(92.0, 70.0);
glVertex2f(108.0, 70.0);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_POLYGON);
glVertex2f(97.5, 65.0);
glVertex2f(102.5, 65.0);
glVertex2f(102.5, 41.0);
glVertex2f(97.5, 41.0);
glEnd();
glColor3f(1.0, 0.0, 0.0);
glPointSize(5.0);
glBegin(GL_POINTS);
glVertex2f(100.0, 58.0);
glEnd();
glColor3f(0.0, 0.0, 1.0);
glPointSize(5.0);
glBegin(GL_POINTS);
glVertex2f(100.0, 48.0);
glEnd();
glColor3f(0.22, 0.22, 0.22);
glBegin(GL_POLYGON);
glVertex2f(175.0, 100.0);
glVertex2f(225.0, 100.0);
glVertex2f(225.0, 280.0);
```

```
glVertex2f(175.0, 280.0);
glEnd();
glColor3f(0.0, 0.0, 0.0);
glBegin(GL_POLYGON);
glVertex2f(177.2, 104.0);
glVertex2f(222.8, 104.0);
glVertex2f(222.8, 276.0);
glVertex2f(177.2, 276.0);
glEnd();
glColor3f(2.55, 2.55, 0.2);
glBegin(GL_POLYGON);
glVertex2f(177.2, 139.0);
glVertex2f(222.8, 139.0);
glVertex2f(222.8, 141.0);
glVertex2f(177.2, 141.0);
glEnd();
glColor3f(2.55, 2.55, 0.2);
glBegin(GL_POLYGON);
glVertex2f(207.0, 233.0);
glVertex2f(222.8, 233.0);
glVertex2f(222.8, 240.0);
glVertex2f(207.0, 240.0);
glEnd();
glColor3f(2.55, 2.55, 0.2);
glBegin(GL_POLYGON);
glVertex2f(207.0, 220.0);
glVertex2f(222.8, 220.0);
glVertex2f(222.8, 227.0);
glVertex2f(207.0, 227.0);
glEnd();
```

```
glColor3f(2.55, 2.55, 0.2);
glBegin(GL_POLYGON);
glVertex2f(207.0, 207.0);
glVertex2f(222.8, 207.0);
glVertex2f(222.8, 214.0);
glVertex2f(207.0, 214.0);
glEnd();
glColor3f(0.0, 0.0, 1.0);
glPointSize(5.0);
glBegin(GL_POINTS);
glVertex2f(186.0, 260.0);
glEnd();
glColor3f(1.0, 1.0, 0.0);
glPointSize(5.0);
glBegin(GL_POINTS);
```

```
glVertex2f(186.0, 252.0);
glEnd();
}
void upload4()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr8 = " D A T A - 2 U P L O A D E D ";
len7 = strlen(ptr8);
glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(138.0, 156.0, 0.0);
for (int i = 0; i < len7; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr8[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void uploading4().
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr8 = " U P L O A D I N G D A T A - 2
...";
len7 = strlen(ptr8);
glRasterPos3f(138.0, 156.0, 0.0);
for (int i = 0; i < len7; i++)
glutBitmapCharacter(GLUT_BITMAP_H
E
LVETICA_12, ptr8[i]);

glPopMatrix();
glutPostRedisplay();
glFlush();
}
void download4()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);

len6 = strlen(ptr6);
glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(120.0, 53.0, 0.0);
for (int i = 0; i < len6; i++)
glutBitmapCharacter(GLUT_BITMAP_H
E
```

```

LVETICA_12, ptr6[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void downloading4()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr6 = "D O W N L O A D I N G   D A T A
“
len6 = strlen(ptr6);
//glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(120.0, 53.0, 0.0);
for (int i = 0; i < len6; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr6[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void download3(s.
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr6 = "D A T A - 2   D O W N L O A D “
len6 = strlen(ptr6);
//glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(120.0, 318.0, 0.0);
for (int i = 0; i < len6; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr6[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void downloading3()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr6 = "D O W N L O A D I N G   D A T A
“
len6 = strlen(ptr6);
//glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(120.0, 318.0, 0.0);
for (int i = 0; i < len6; i++)

```

```

glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr6[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void upload3()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr8 = " D A T A - 1   U P L O A D     E D
";
len7 = strlen(ptr8);
glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(140.0, 213.0, 0.0);
for (int i = 0; i < len7; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr8[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void uploading3()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr6 = "U P L O A D I N G   D A T A - 1
...";
len6 = strlen(ptr6);
//glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(140.0, 213.0, 0.0);
for (int i = 0; i < len6; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr6[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void fourfourU()
{
if (p15 == 300.0 && p16 >= 114.0 && p16
< 145.0)
{
p16 = p16 + 0.08;
glutPostRedisplay();
}
}

```

```
else
{
if (p15 <= 300.0 && p15 > 225.7)
{
p15 = p15 - 0.08;
glutPostRedisplay();
}
}
if (p15 > 227.5)
{
glPushMatrix();
glColor3f(0.6, 0.6, 0.6);
uploading4();
glPopMatrix();
}
else
{
glPushMatrix();
glColor3f(0.0, 0.0, 0.0);
uploading4();
glPopMatrix();
upload4();
}
}
void fourfourD()
{
if (p17 == 216.0 && p18 <= 99.0 && p18
> 70.0)
{
p18 = p18 - 0.08;
glutPostRedisplay();
}
}
else
{
if (p17 >= 216.0 && p17 < 289.0)
{
p17 = p17 + 0.08;
glutPostRedisplay();
}
}
if (p17 < 287.0)
{
glPushMatrix();
glColor3f(0.6, 0.6, 0.6);
downloading4();
glPopMatrix();
}
}
else
{
glPushMatrix();
glColor3f(0.0, 0.0, 0.0);
downloading4();
glPopMatrix();
}
}
void threethreeU().
{
if (p13 == 300.0 && p14 <= 262.0 && p14
> 225.0)
{
p14 = p14 - 0.08;

glutPostRedisplay();
}
else
{
if (p13 <= 300.0 && p13 > 226.0)
{
p13 = p13 - 0.08;
glutPostRedisplay();
}
}
if (p13 > 229.0)
{
glPushMatrix();
glColor3f(0.6, 0.6, 0.6);
uploading3();
glPopMatrix();
}
else
{
glPushMatrix();
glColor3f(0.0, 0.0, 0.0);
uploading3();
glPopMatrix();
upload3();
}
}
void threethreeD
{
```

```
if (p11 == 210.0 && p12 < 310.0)
{
    p12 = p12 + 0.08;

    glutPostRedisplay();
}
else
{
    if (p11 >= 210.0 && p11 < 289.2)
    {
        p11 = p11 + 0.08;
        glutPostRedisplay();
    }
}
if (p11 < 286.0)
{
    glPushMatrix();
    glColor3f(0.6, 0.6, 0.6);
    downloading3();
    glPopMatrix();
}
else
{
    glPushMatrix();
    glColor3f(0.0, 0.0, 0.0);
    downloading3();
    glPopMatrix();
    glPushMatrix();
    glColor3f(0.6, 0.6, 0.6);
    download3();
    glPopMatrix();
}
}

void download2()
{
    glPushMatrix();
    glTranslatef(100.0, 0.0, 0.0);
    ptr6 = "D O W N L O A D E D ";
    len6 = strlen(ptr6);
    //glColor3f(0.60, 0.60, 0.60);
    glRasterPos3f(30.0, 55.0, 0.0);
    for (int i = 0; i < len6; i++)
        glutBitmapCharacter(GLUT_BITMAP_H
            ELVETICA_12, ptr6[i]);
    glPopMatrix();
    glutPostRedisplay();

    glFlush();
}

void downloading2()
{
    glPushMatrix();
    glTranslatef(100.0, 0.0, 0.0);
    ptr6 = "D O W N L O A D I N G ...";
    len6 = strlen(ptr6);
    glRasterPos3f(30.0, 55.0, 0.0);
    for (int i = 0; i < len6; i++)
        glutBitmapCharacter(GLUT_BITMAP_H
            ELVETICA_12, ptr6[i]);
    glPopMatrix();
    glutPostRedisplay();
    glFlush();
}

void twotwoD()
{
    if (p9 == 185.0 && p10 <= 99.0 && p10 >
        68.0)
    {
        p10 = p10 - 0.08;
        glutPostRedisplay();
    }
    else
    {
        if (p9 <= 185.0 && p9 > 111.2)
        {
            p9 = p9 - 0.08;
            glutPostRedisplay();
        }
    }
    if (p9 > 112.0)
    {
        glPushMatrix();
        glColor3f(0.6, 0.6, 0.6);
        downloading2();
        glPopMatrix();
    }
    else
    {
        glPushMatrix();
        glColor3f(0.0, 0.0, 0.0);
        downloading2();
        glPopMatrix();
        glPushMatrix();
    }
}
```

```
glColor3f(0.6, 0.6, 0.6);
download2();
glPopMatrix();
glEnd();}
void upload2()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr6 = "U P L O A D E D ";
len6 = strlen(ptr6);
glRasterPos3f(27.0, 155.0, 0.0);
for (int i = 0; i < len6; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr6[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void uploading2()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr6 = "U P L O A D I N G ...";
len6 = strlen(ptr6);
//glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(27.0, 155.0, 0.0);
for (int i = 0; i < len6; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr6[i]);
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void twotwoU()
{
if (p7 == 100.0 && p8 >= 115.0 && p8 <
145.0)
{
p8 = p8 + 0.08;
glutPostRedisplay();
}
else
{
if (p8 >= 145.0 && p7 < 174.0)
{
p7 = p7 + 0.08;
glutPostRedisplay();
}
}
}
void oneoneD()
{
if (flag == 1)
{
if (p5 <= 175.0 && p6 == 200.0 && p5 >
100.0)
{
p5 = p5 - 0.08;
glutPostRedisplay();
}
else
{
if (p5 <= 100.0 && p6 < 250.0)
{
p6 = p6 + 0.08;
glutPostRedisplay();
}
}
glPushMatrix();
glColor3f(0.0, 0.0, 0.0);
response();
glPopMatrix();
if (p6 < 248.0)
```

```

{
    glPushMatrix();
    glColor3f(0.6, 0.6, 0.6);
    downloading();
    glPopMatrix();
}
else
{
    glPushMatrix();
    glColor3f(0.0, 0.0, 0.0);
    downloading();
    glPopMatrix();
    glPushMatrix();
    glColor3f(0.6, 0.6, 0.6);
    download();
    glPopMatrix();
    glEnd();
}
}
}
void downloading()
{
    glPushMatrix();
    glTranslatef(100.0, 0.0, 0.0);
    ptr5 = "D O W N L O A D I N G ... ";
    len5 = strlen(ptr5);
    glRasterPos3f(25.0, 190.0, 0.0);
    for (int i = 0; i < len5; i++)
        glutBitmapCharacter(GLUT_BITMAP_H
            ELVETICA_12, ptr5[i]);
    glPopMatrix();
    glutPostRedisplay();
    glFlush();
}
void download()
{
    glPushMatrix();
    glTranslatef(100.0, 0.0, 0.0);
    ptr6 = "D O W N L O A D E D ";
    len6 = strlen(ptr6);
    glRasterPos3f(25.0, 190.0, 0.0);
    for (int i = 0; i < len6; i++)
        glutPostRedisplay();
    glFlush();
}

void request()
{
    glPushMatrix();
    glTranslatef(100.0, 0.0, 0.0);
    ptr3 = "H A N D S H A K E   P R O C E S
        S";
    len1 = strlen(ptr3);
    glRasterPos3f(25.0, 315.0, 0.0);
    for (int i = 0; i < len1; i++)
        glutBitmapCharacter(GLUT_BITMAP_H
            ELVETICA_12, ptr3[i]);
    glPopMatrix();
    glutPostRedisplay();
    glFlush();
}
void response()
{
    glPushMatrix();
    glTranslatef(100.0, 0.0, 0.0);
    ptr4 = "H A N D S H A K E   P R O C E S
        S ";
    len2 = strlen(ptr4);
    glRasterPos3f(15.0, 190.0, 0.0);
    for (int i = 0; i < len2; i++)
        glutBitmapCharacter(GLUT_BITMAP_H
            ELVETICA_12, ptr4[i]);
    glPopMatrix();
    glutPostRedisplay();
    glFlush();
}
void handshake1V()
{
    if (p3 <= 175.0 && p4 == 200.0 && p3 >
        100.0)
    {
        p3 = p3 - 0.08;
        glPushMatrix();
        glPointSize(2.0);
        glColor3f(1.0, 1.0, 1.0);
        glBegin(GL_POINTS);
        glVertex2f(p3, p4);
        glEnd();
        glPopMatrix();
        glutPostRedisplay();
    }
    else

```



```
{
if (p3 <= 100.0 && p4 < 252.0)
{
p4 = p4 + 0.08;
glPushMatrix();
glColor3f(1.0, 1.0, 1.0);
glPointSize(2.0);
glBegin(GL_POINTS);
glVertex2f(p3, p4);
glEnd();
glPopMatrix();
glutPostRedisplay();
}
}
glPushMatrix();
glColor3f(0.6, 0.6, 0.6);
response();
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void handshake1H(){
if (p1 >= 107.0 && p2 == 300.0 && p1 <
200.0)
{
p1 = p1 + 0.08;
glPushMatrix();
glPointSize(2.0);
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_POINTS);
glVertex2f(p1, p2);
glEnd();
glPopMatrix();
glutPostRedisplay();
}
else
{
if (p1 >= 200.0 && p2 > 278.0)
{
p2 = p2 - 0.08;
glPushMatrix();
glColor3f(1.0, 1.0, 1.0);
glPointSize(2.0);
glBegin(GL_POINTS);
glVertex2f(p1, p2);
glEnd();

glPopMatrix();
glutPostRedisplay();
}
}
}
void handshake1V();
flag = 1;
glFlush();
}
void upload()
{
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
ptr1 = "U P L O A D ";
ptr2 = "E D ";
len1 = strlen(ptr1);
len2 = strlen(ptr2);
glColor3f(0.60, 0.60, 0.60);
glRasterPos3f(40.0, 315.0, 0.0);
for (int i = 0; i < len1; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr1[i]);
for (int i = 0; i < len2; i++) {
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_12, ptr2[i]);
}
glPopMatrix();
glutPostRedisplay();
glFlush();
}
void uploading()
void two()
{
if (flag == 1) {
glPushMatrix();
glPointSize(6.0);
glColor3f(2.0, 0.20, 0.20);
glBegin(GL_POINTS);
glVertex2f(p5, p6);
glEnd();
}
```

```
glPopMatrix();
glFlush();
}
}
void three()
{
glPushMatrix();
//glPointSize(4.0);
glPointSize(6.0);
glColor3f(0.20, 0.20, 2.0);
glBegin(GL_POINTS);
glVertex2f(p7, p8);
glEnd();
glPopMatrix();
glFlush();
}
void four()
{
glPushMatrix();
//glPointSize(4.0);
glPointSize(6.0);
glColor3f(2.0, 0.20, 0.20);
glBegin(GL_POINTS);
glVertex2f(p9, p10);
glEnd();
glPopMatrix();
glFlush();
}
void five()
{
glPushMatrix();
glPointSize(6.0);
glColor3f(0.20, 0.20, 2.0);
glBegin(GL_POINTS);
glVertex2f(p13, p14);
glEnd();
glPopMatrix();
glFlush();
}
void six()
{
glPushMatrix();
glPointSize(6.0);
glColor3f(2.0, 0.20, 0.20);
glBegin(GL_POINTS);
glVertex2f(p11, p12);

glEnd();
glPopMatrix();
glFlush();
}
void seven()
{
glPushMatrix();
//glPointSize(4.0);
glPointSize(6.0);
glColor3f(0.2, 0.20, 2.0);
glBegin(GL_POINTS);
glVertex2f(p15, p16);
glEnd();
glPopMatrix();
glFlush();
}
void eight()
{
glPushMatrix();
//glPointSize(4.0);
glPointSize(6.0);
glColor3f(2.0, 0.20, 0.20);
glBegin(GL_POINTS);
glVertex2f(p17, p18);
glEnd();
glPopMatrix();
glFlush();
}
void display()
{
glPushMatrix();
line();
glPopMatrix();

glPushMatrix();
glTranslatef(0.0, 220.0, 0.0);
line();
glPopMatrix();
glPushMatrix();
glTranslatef(260.0, 220.0, 0.0);
line();
glPopMatrix();
glPushMatrix();
glTranslatef(260.0, 0.0, 0.0);
line();
glPopMatrix();
}
```

```

glPushMatrix();
server();
glPopMatrix();
glPushMatrix();
glColor3f(0.22, 0.22, 0.22);
glTranslatef(200.0, 140.0, 0.0);
draw(7.0);
glPopMatrix();
glPushMatrix();
glColor3f(0.0, 0.0, 0.0);
glTranslatef(200.0, 140.0, 0.0);
draw(5.0);
glPopMatrix();
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
char* ptr5 = "Client 2";
int len5 = strlen(ptr5);
glColor3f(0.3, 0.3, 0.3);
glRasterPos3f(-53, 20, 0.0);
for (int i = 0; i < len5; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_18, ptr5[i]);
glPopMatrix();
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
char* ptr1 = "Client 1";
int len1 = strlen(ptr1);
glColor3f(0.3, 0.3, 0.3);
glRasterPos3f(-53, 240, 0.0);
for (int i = 0; i < len1; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_18, ptr1[i]);
glPopMatrix();
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
char* ptr3 = "Client 3";
int len3 = strlen(ptr3);
glColor3f(0.3, 0.3, 0.3);
glRasterPos3f(207, 240, 0.0);
for (int i = 0; i < len3; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_18, ptr3[i]);
glPopMatrix();
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
char* ptr4 = "Client 4";

int len4 = strlen(ptr4);
glColor3f(0.3, 0.3, 0.3);
glRasterPos3f(207, 19, 0.0);
for (int i = 0; i < len4; i++)
glutBitmapCharacter(GLUT_BITMAP_H
ELVETICA_18, ptr4[i]);
glPopMatrix();
glPushMatrix();
glTranslatef(100.0, 0.0, 0.0);
char* ptr7 = "SERVER";
int len7 = strlen(ptr4);
glColor3f(0.3, 0.3, 0.3);
Init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
GLint id1 = glutCreateMenu(CS1);
glutAddMenuEntry(" Handshake ", 1);
glutAddMenuEntry(" Upload Data", 2);
glutAddMenuEntry(" Download Data", 3);
GLint id2 = glutCreateMenu(CS2);
glutAddMenuEntry(" Upload Data", 1);
glutAddMenuEntry(" Download Data ",
2);
GLint id3 = glutCreateMenu(CS3);
glutAddMenuEntry(" Upload Data 1", 1);
glutAddMenuEntry(" Download Data 2",
2);
GLint id4 = glutCreateMenu(CS4);
glutAddMenuEntry(" Upload Data 2", 1);
glutAddMenuEntry(" Download Data 1",
2);
glutCreateMenu(cs);
glutAddSubMenu("Client 1 - Server (TCP)
", id1);
glutAddSubMenu("Client 2 - server (UDP)
", id2);
glutAddSubMenu("Client 3 - Server ",
id3);
glutAddSubMenu("Client 4 - server ",
id4);
glutAddMenuEntry("Exit ", 5);
glutAttachMenu(GLUT_LEFT_BUTTON
);
glutAttachMenu(GLUT_RIGHT)
glutMainLoop();
}

```

CHAPTER 6

RESULTS

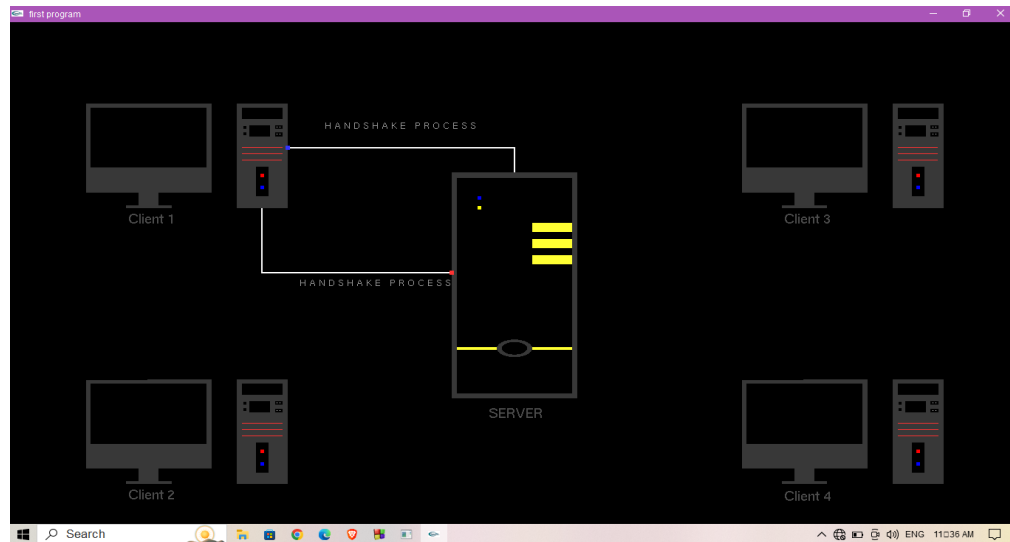


Fig 6.1 – Hand shaking process: The client 1 hand shakes with the server for further communication.

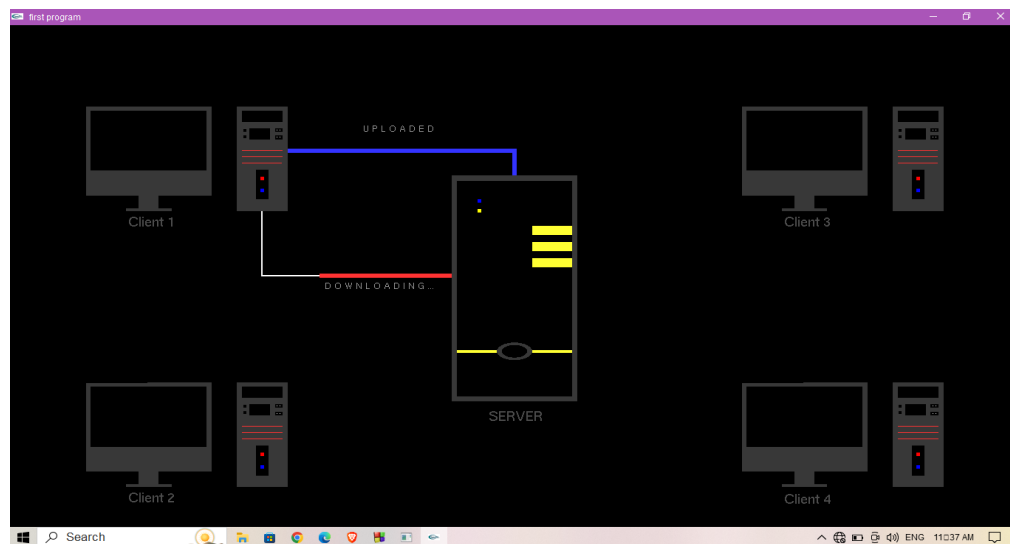


Fig 6.2 - Uploading and Downloading from server: Client 1 can upload the data to server after hand shake and it can also download the data from the server.

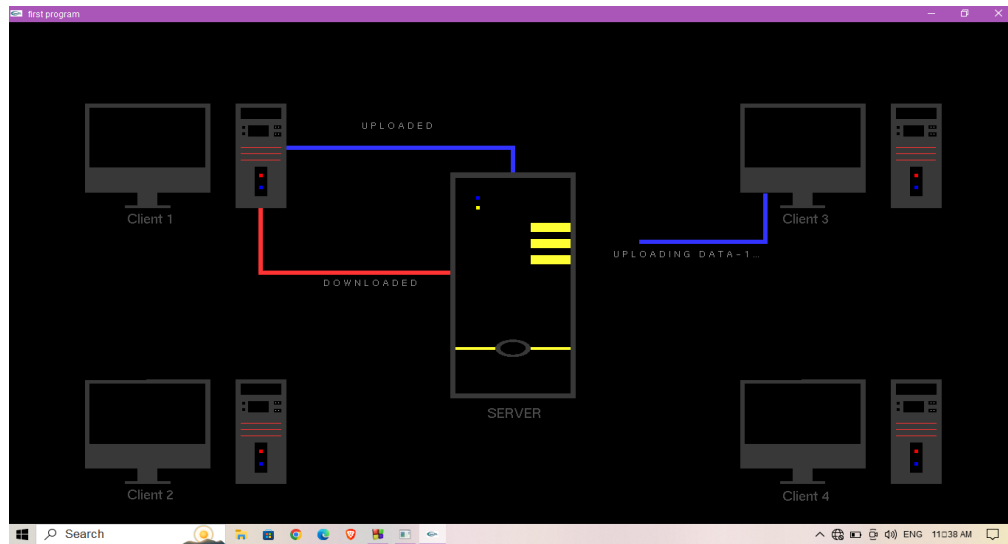


Fig 6.3 - Client 3 sending request to server: As he above figure illustrates the clients perform the same operations i.e., uploading and downloading the data.

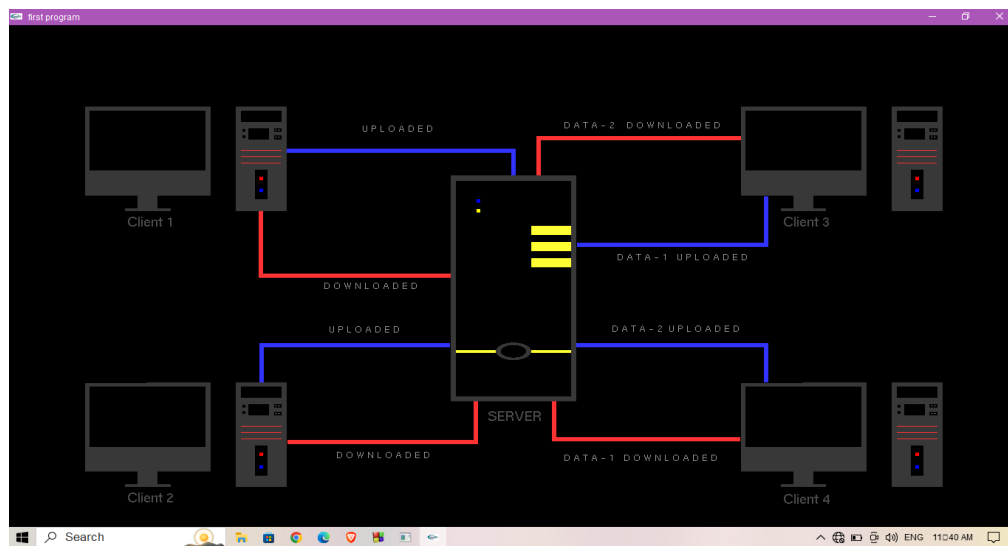


Fig 6.4 - Final Output after all the processes: The above figure shows how the clients communicate with the server with the use of computer graphics.

CHAPTER 7

CONCLUSION

The code we have implemented for our project is working well to the best of our knowledge. In this project the clients and server communicate as per the user's command. In conclusion, the client-server architecture is a well-liked and incredibly effective method for developing networked systems. The foundation of this architecture is the idea that the system should be split into two main parts: the client, which starts requests and engages with users, and the server, which fulfils requests for services and processes them. This project will serve as a visual treat for every networking student that illustrates handshaking process, uploading and downloading from the server in an innovative way.

This project is both informative and entertaining. This project provided an opportunity to learn the various concepts of the subject in detail and provided us a platform to express our creativity and imagination come true.

BIBLIOGRAPHY

1. "Client-Server Architecture for Distributed Rendering of OpenGL-based Graphics" by A. Santinelli et al. This paper presents a client-server architecture for distributed rendering of OpenGL-based graphics.
2. https://en.wikipedia.org/wiki/Computer_graphics - History of Computer Graphics
3. <https://en.wikipedia.org/wiki/OpenGL> - History of OpenGL
4. <https://en.wikipedia.org/wiki/OpenGL> - About OpenGL
5. https://en.wikipedia.org/wiki/OpenGL_Utility_Library - About GLU
6. https://en.wikipedia.org/wiki/OpenGL_Utility_Toolkit - About GLUT
7. <https://www.khronos.org/opengl/wiki/Primitive> - About OpenGL Primitives