SCHOOL OF COMPUTER SCIENCE

UNIVERSITY OF PETROLEUM AND ENERGY STUDIES
DEHRADUN, UTTARAKHAND

COMPUTER GRAPHICS
LABORATORY FILE
(2024-2025)

For
Vth Semester

Submitted To:
Mr. Dinesh
Assistant Professor
[Vth Semester]
School of Computer Science

Submitted By:
Akshat Negi
500106533(SAP ID)
R2142220414(Roll No.)
B.Tech. CSF (Batch-1)

LAB EXPERIMENT – 1

Introduction to OpenGL:

(Lab Environment Setup)

a) What is OpenGL?

OpenGL (Open Graphics Library) is a powerful and versatile application programming interface (API) used for rendering 2D and 3D vector graphics. Essentially, it's a set of functions that allow software to communicate with a computer's graphics hardware (GPU) to create stunning visuals.

b) What is GLU/GLUT?

GLU: OpenGL Utility Library

(OpenGL Utility Library) is a higher-level library built on top of OpenGL. It provides a set of functions that simplify common tasks in 3D graphics programming. These functions offer a more convenient interface for performing operations like:

- **Projection and viewing transformations:** Defining the camera's position and orientation.
- **Quadric surfaces:** Creating shapes like spheres, cylinders, and cones.
- **NURBS:** Handling non-uniform rational B-splines for complex curves and surfaces.
- **Tessellation:** Breaking down complex shapes into simpler polygons.
- **Error handling:** Managing OpenGL errors.

While GLU was useful in the past, it's gradually being phased out as modern OpenGL provides more direct ways to accomplish these tasks.

GLUT: OpenGL Utility Toolkit

(OpenGL Utility Toolkit) is a cross-platform windowing library designed to simplify creating OpenGL applications. It provides basic functions for:

- **Window creation and management:** Opening, closing, and resizing windows.
- **Input handling:** Managing keyboard and mouse events.

- **OpenGL context management:** Creating and destroying OpenGL rendering contexts.
- **Idle callback:** Executing code when the application is idle.

GLUT is primarily used for educational purposes and small-scale projects. For more complex applications, modern windowing libraries like GLFW or Qt are often preferred.

c) What is OpenGL Architecture?

The OpenGL architecture refers to the design and structure of the OpenGL system, which is a software interface to graphics hardware. This architecture outlines how OpenGL interacts with the underlying hardware, software, and the various components that make up the OpenGL system.

Key Components of OpenGL Architecture:

OpenGL Client and Server Model:

a) **Client:** The application that makes OpenGL API calls is considered the "client."

b) **Server:** The "server" is typically the graphics hardware or the driver software that executes the OpenGL commands. In a distributed system, the server could be a remote machine with the necessary hardware.

c) OpenGL operates in a client-server model, where the client sends drawing commands to the server, which then processes these commands and renders the graphics.

OpenGL Pipeline: The OpenGL rendering pipeline is a sequence of steps that the system follows to transform 3D models into a 2D image on the screen.
Vertex Processing: Vertices (points in 3D space) are processed, including transformations (e.g., scaling, rotation) and lighting calculations.
Primitive Assembly: Vertices are assembled into geometric primitives, like points, lines, or triangles.
Rasterization: The primitives are converted into fragments, which are potential pixels on the screen.
Fragment Processing: Each fragment is processed to determine its final colour and other attributes, taking into account textures, shading, and lighting.
Framebuffer Operations: The processed fragments are written to the framebuffer, where they become pixels that will be displayed on the screen.

OpenGL Context:

d) The OpenGL context is an environment in which OpenGL functions operate. It includes all the state information needed to perform rendering operations, such as textures, shaders, and buffer objects.

e) Each window or rendering surface has its own OpenGL context, and multiple contexts can share resources.

State Machine:

f) OpenGL operates as a state machine, meaning it maintains various states (e.g., current color, current texture) that persist until explicitly changed by the application.

g) The state machine allows for efficient rendering, as OpenGL doesn't need to repeatedly set the same parameters unless they change.

Extensions:

h) OpenGL is designed to be extensible. Hardware vendors can introduce new features through extensions, allowing developers to access advanced capabilities beyond the core OpenGL specification.

i) Extensions provide a way to experiment with new features before they become part of the official OpenGL standard.

GLU and GLUT:

As mentioned earlier, GLU and GLUT are utility libraries that work on top of the core OpenGL architecture to provide additional functionality and simplify certain tasks.

These libraries help manage higher-level operations (GLU) and handle windowing and input (GLUT).

Shaders and Programmable Pipeline:

In modern OpenGL, the fixed-function pipeline has largely been replaced by the programmable pipeline, where shaders (small programs written in GLSL, the OpenGL Shading Language) control various stages of the rendering process.

- **Vertex Shader:** Handles the transformation and lighting of individual vertices.
- **Fragment Shader:** Determines the color and other attributes of individual fragments.

d) Setting up the environment.

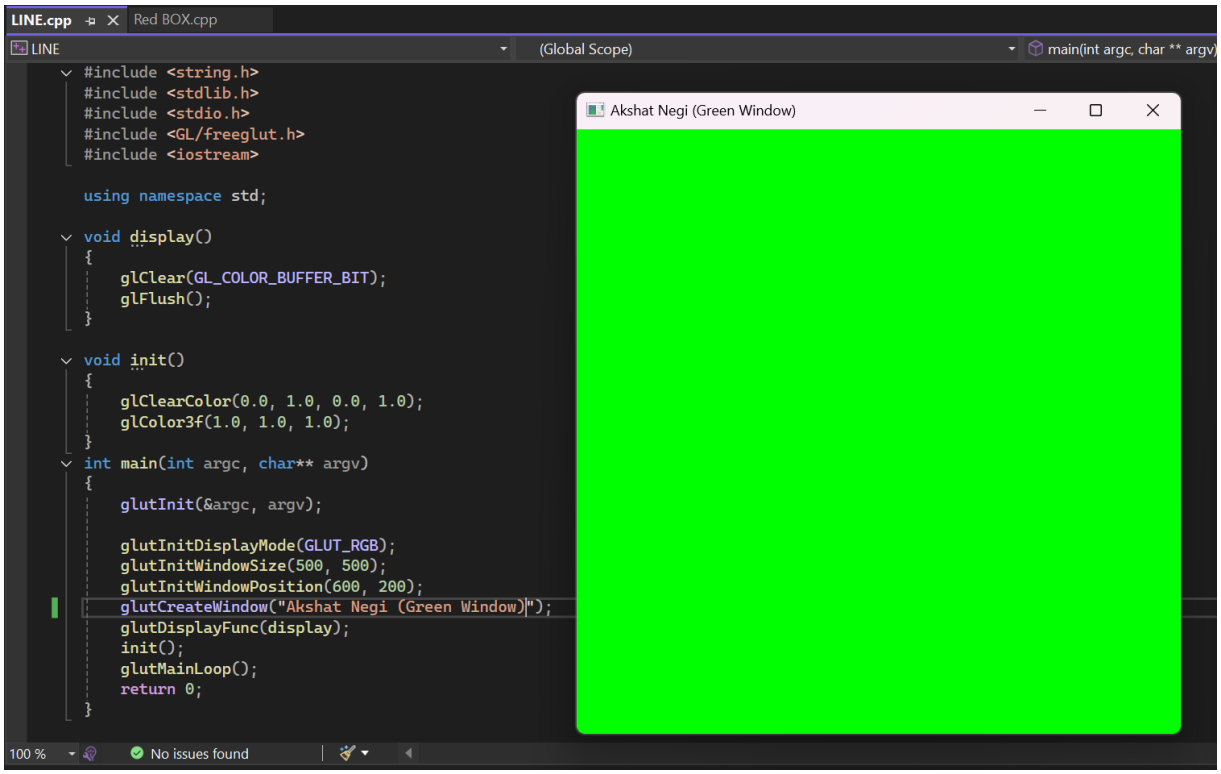
e) First OpenGL Program: This initializes a window of green color.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <GL/freeglut.h>
#include <iostream>
using namespace std;
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

void init()
{
    glClearColor(0.0, 1.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(600, 200);
    glutCreateWindow("Akshat Negi (Green Window)");
    glutDisplayFunc(display);
    init();
    glutMainLoop();
    return 0;

}
```



f) Draw a Hut.

```
#include <GL/freeglut.h>
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.6f, 0.4f, 0.2f); // Brown color
    glBegin(GL_POLYGON);
    glVertex2f(-0.5f, -0.5f);
    glVertex2f(0.5f, -0.5f);
    glVertex2f(0.5f, 0.0f);
    glVertex2f(-0.5f, 0.0f);
    glEnd();
    glColor3f(0.8f, 0.2f, 0.0f); // Red color
    glBegin(GL_TRIANGLES);
    glVertex2f(-0.6f, 0.0f);
    glVertex2f(0.6f, 0.0f);
    glVertex2f(0.0f, 0.5f);
    glEnd();
    glColor3f(0.3f, 0.2f, 0.1f); // Dark brown color
    glBegin(GL_POLYGON);
    glVertex2f(-0.1f, -0.5f);
    glVertex2f(0.1f, -0.5f);
    glVertex2f(0.1f, -0.2f);
    glVertex2f(-0.1f, -0.2f);
    glEnd();
    glColor3f(0.0f, 0.6f, 1.0f); // Blue color
    glBegin(GL_POLYGON);
    glVertex2f(-0.4f, -0.2f);
    glVertex2f(-0.2f, -0.2f);
    glVertex2f(-0.2f, 0.0f);
    glVertex2f(-0.4f, 0.0f);
    glEnd();
    glFlush();
}
void init() {
    glClearColor(0.5f, 0.8f, 1.0f, 1.0f); // Light blue background
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Simple Hut");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Experiment 2: Drawing a line [Usage of Open GL]

a. Draw a line using equation of line Y=m*X+C.

b. Draw a line using DDA algorithm for slope m<1 and m>1.

```
#include <GL/freeglut.h>
#include <iostream>
#include <cmath>
using namespace std; // Function to plot points
void plot(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}  // DDA Line Drawing Algorithm
void DDA(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);  // Maximum steps
    float xIncrement = dx / (float)steps;
    float yIncrement = dy / (float)steps;
    float x = x1;
    float y = y1; // Draw the line by plotting points
    for (int i = 0; i <= steps; i++) {
        plot(round(x), round(y));
        x += xIncrement;
        y += yIncrement;
    }
}// Function to get input from the user and call DDA
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    int x1, y1, x2, y2;
    cout << "Enter the coordinates of the first point (x1, y1): ";
    cin >> x1 >> y1;
    cout << "Enter the coordinates of the second point (x2, y2): ";
    cin >> x2 >> y2;
    DDA(x1, y1, x2, y2);
} // Initialize the OpenGL Graphics
void init() {
    glClearColor(1.0, 1.0, 1.0, 0.0);  // Background color
    glColor3f(0.0, 0.0, 0.0);          // Drawing color
    glPointSize(2.0);                  // Point size
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 500.0, 0.0, 500.0);  // Define the drawing area
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);       // Window size
    glutInitWindowPosition(100, 100);   // Window position
    glutCreateWindow("DDA Line Drawing Algorithm");
    init();
    glutDisplayFunc(display);           // Register display function
    glutMainLoop();                     // Enter the event-processing loop
    return 0;
}
```

c. Draw a line using Bresenhan algorithm for slope m<1 and m>1.

```
#include <GL/freeglut.h>
#include <stdio.h> // Function to set pixel at (x, y)
void setPixel(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
} // Bresenham's algorithm for slope |m| < 1 (dy < dx)
void bresenhamLineLow(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int D = 2 * dy - dx;
    int y = y1;
    for (int x = x1; x <= x2; x++) {
        setPixel(x, y);
        if (D > 0) {
            y += (y2 > y1) ? 1 : -1;  // Increase/decrease y depending on the
slope direction
            D = D + (2 * (dy - dx));
        }
        else {
            D = D + 2 * dy;
        }
    }
}// Bresenham's algorithm for slope |m| > 1 (dy > dx)
void bresenhamLineHigh(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1;
    int dy = y2 - y1;
    int D = 2 * dx - dy;
    int x = x1;
    for (int y = y1; y <= y2; y++) {
        setPixel(x, y);
        if (D > 0) {
            x += (x2 > x1) ? 1 : -1;  // Increase/decrease x depending on the
slope direction
            D = D + (2 * (dx - dy));
        }
        else {
            D = D + 2 * dx;
        }
    }
} // Main function that checks the slope and calls the appropriate function
void drawLine(int x1, int y1, int x2, int y2) {
    if (abs(y2 - y1) < abs(x2 - x1)) {
        if (x1 > x2) {
            bresenhamLineLow(x2, y2, x1, y1);  // Line from (x2, y2) to (x1, y1)
        }
        else {
            bresenhamLineLow(x1, y1, x2, y2);  // Line from (x1, y1) to (x2, y2)
        }
    }
    else {
        if (y1 > y2) {
            bresenhamLineHigh(x2, y2, x1, y1); // Line from (x2, y2) to (x1, y1)
        }
        else {
            bresenhamLineHigh(x1, y1, x2, y2); // Line from (x1, y1) to (x2, y2)
        }
    }
} // User input handling and initialization
```

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    int x1, y1, x2, y2;
    printf("Enter coordinates of the first point (x1, y1): ");
    scanf_s("%d %d", &x1, &y1);
    printf("Enter coordinates of the second point (x2, y2): ");
    scanf_s("%d %d", &x2, &y2);
    drawLine(x1, y1, x2, y2);
}
void init() {
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glColor3f(0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 500, 0, 500);  // Set the orthographic projection
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Bresenham's Line Algorithm");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

 # Take the input from user for all the three scenarios i.e. value of (x1, y1) and (x2, y2).