Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual
CS251
Computer Graphics Fundamental

Preparing the scientific material

A.Prof. Mohamed Salah

T.A. Manar Abbas

T.A. Somaya Ahmed

Weekly Breakdown

Week 1: Introduction to OpenGL & Setup

- **Computer Graphics Definition, applications, and generating.**
- **Overview, advantages & disadvantages, file structure of OpenGL.**
- **Essential OpenGL Functions.**
- **Lab: Install OpenGL, Create window.**

Week 2: OpenGL Primitives & Basic Drawing (Part 1)

- **Difference between primitives.**
- **GL_POINTS (Point Primitive)**
- **GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP (Line Primitives)**
- **Lab: Draw dotted square, square using different line primitives, house outline.**

Week 3: OpenGL Primitives & Basic Drawing (Part 2)

- **GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN**
- **GL_QUADS, GL_QUAD_STRIP, GL_POLYGON**
- **Lab: Draw filled shapes, chessboard.**

Week 4: Orthographic projection & Circle Drawing

- **Orthographic projection (glMatrixMode, glLoadIdentity, gluOrtho2D)**
- **Circle drawing using parametric equation**
- **Lab: Draw circle using GL_POINTS.**

Contents

Lab#	Description
1	Introduction to Computer Graphics and OpenGL
2	OpenGL Primitives part 1: Types, Modes, and How to Draw Them
3	OpenGL Primitives part 2: Types, Modes, and How to Draw Them
4	Orthographic projection and Circle Drawing using GL_POINTS

Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

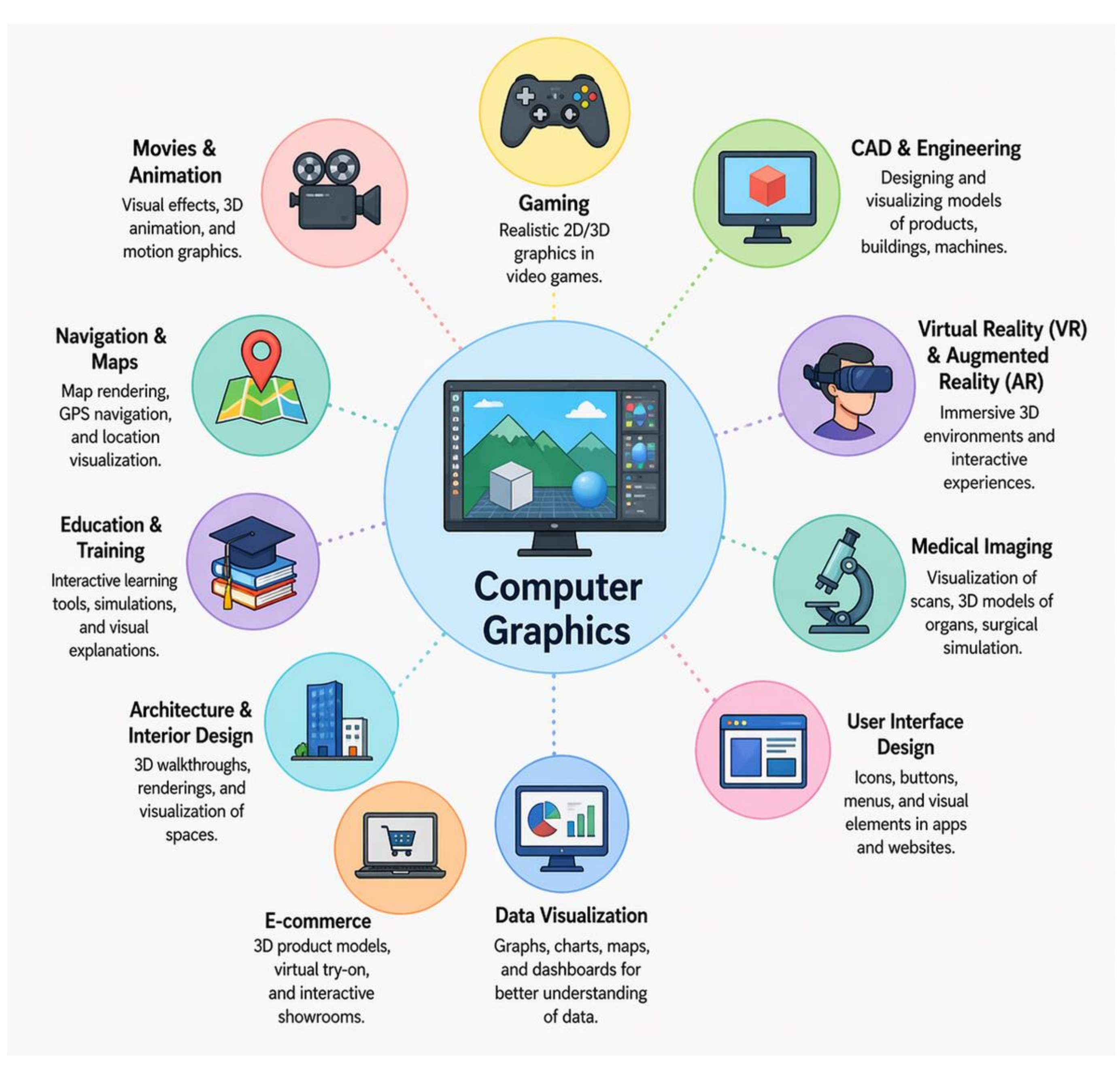
Introduction to Computer Graphics & OpenGL

Lab - 1
Introduction to Computer Graphics

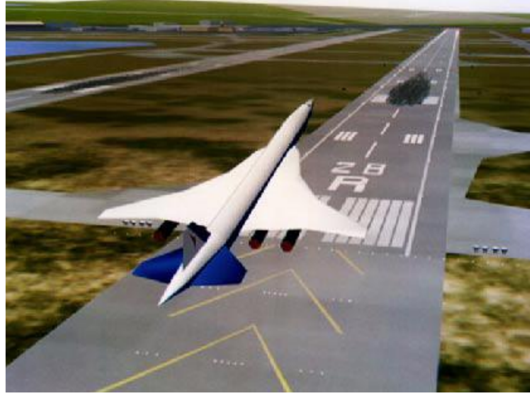
What is Computer Graphics ?

- In simple terms, **computer graphics** is the art and science of creating, manipulating, and displaying visual content (images, animations, 3D models) using computers.
- Think of it as digital drawing and movie-making, but driven by math and code instead of just a paintbrush.
- **The core idea:** You represent every point, line, color, and shadow as numbers and mathematical formulas. The computer then calculates these formulas millions of times per second to show you an image on the screen.
- **Key example:** The 3D character of *Mario* in *Super Mario Odyssey* is a computer graphic. Mario isn't a video of a real person; he's a mathematical model that the game console draws in real-time.

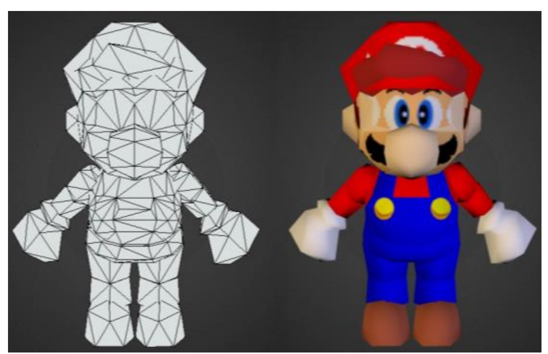
What is Applications of Computer Graphics ?



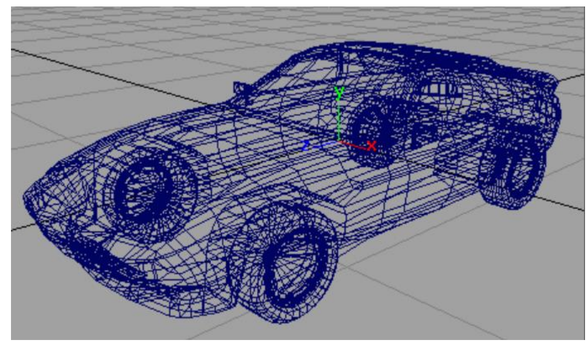
Driving Simulation



Flight Simulation



Entertainments



Computer-Aided Design

Human Skeleton

Interactive kitchen planner

How to generate Graphics ?

There are two main approaches to generating computer graphics. The first uses ready-made software where humans do the drawing directly. The second uses programming libraries where code instructs the computer to draw automatically.

1. **Special-Purpose Programs (Interactive Tools)**:
    - These are applications designed for humans to create graphics manually using a mouse or keyboard. You don't need to write code. Instead, you use menus, buttons, and drag-and-drop actions.
    - **How they work:** The program provides a set of tools (brush, shape tool, color picker). When you click and drag, the program internally calculates which pixels or vectors to create. You are essentially commanding a pre-built graphics engine through a user interface.
    - **Key characteristics:**
        1. User-driven: You decide every line and color.
        2. No programming required: Artists and designers use these.
        3. Output: Images, diagrams, animations saved as files (JPEG, PNG, PDF, etc.).
    - **Example:** Adobe Photoshop, Microsoft PowerPoint, AutoCAD
2. **General Graphics Libraries (Programming-Based):**
    - These are collections of code functions that programmers call to generate graphics automatically. Instead of a human drawing each line, the program runs instructions that tell the computer exactly what to draw, where, and how.
    - **How they work:** You write a program (in C++, Python, Java, etc.). Inside your code, you call functions like drawLine(x1, y1, x2, y2) or drawTriangle(vertices). When you run your program, the computer executes these commands one by one to generate the image.
    - **Key characteristics:**
        1. Code-driven: The program generates graphics dynamically based on logic and data.
        2. Requires programming knowledge: Used by game developers, simulation engineers, and app developers.
        3. Output: Can be real-time (games, UI) or saved to files.
    - **Example:** Windows API, OpenGL, DirectX (Microsoft)

What is OpenGL?

OpenGL (Open Graphics Library) is the computer industry's standard application program interface (API) for defining 2-D and 3-D graphic images. Prior to OpenGL, any company developing a graphical application typically had to rewrite the graphics part of it for each operating system platform.

Advantages of OpenGL

1. Easy to learn (basic level): Simple step-by-step approach; great for beginners and university courses.
2. Cross-platform: Runs on Windows, Linux, macOS, iOS, Android. Write once, run almost anywhere.
3. Easy to understand for teamwork: OpenGL function names are long and descriptive (like glBegin(), glClearColor()), so code reads like English and no separate documentation is needed to know what previous team members did.

Disadvantages of OpenGL

1. Hard to master – Basics are easy, but advanced topics (shaders, buffers, optimization) take months or years to learn.
2. Slower than DirectX on Windows – Higher CPU driver overhead; DirectX is better optimized for Windows.

OpenGL Library File Strucute?

1. **Header Files (include directory)**
    a. Contain function declarations, type definitions, and constants
    b. Tell the compiler what these functions look like.
    c. Your code includes these using #include <GL/gl.h>
    d. Located in the include/ folder of your OpenGL installation
    e. Example: GL.h , GLU.h, GLUT.h

2. **Library Files (lib directory)**
    a. Contain compiled code for linking
    b. Tell the compiler where to find functions in DLLs.
    c. Located in the lib/ folder
    d. Example: freeGlut.lib, GLU32.lib, GLUT.lib

3. **Dynamic Link Libraries (DLLs)**
    a. Loaded at runtime when your program executes
    b. Actually execute the functions.
    c. Example: opengl32.dll , GLU32.dll, GLUT.dll

Functions to create window

1. **glutInit**
   a. Description: Initializes the GLUT library. Must be called before any other GLUT functions.
   b. Usage: glutInit(&argc, argv);
   c. Arguments:
      i. &argc → Pointer to main function's argument count
      ii. argv → Pointer to main function's argument array

2. **glutInitDisplayMode**
   a. Description: Sets the initial display mode (color type, buffering, depth buffer, etc.) for the window.
   b. Usage: glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
   c. Arguments:
      i. GLUT_RGB or GLUT_RGBA → Use RGB color mode
      ii. GLUT_DOUBLE → Use double buffering (smooth animation)
      iii. GLUT_SINGLE → Use single buffering
      iv. Use | (bitwise OR) to combine multiple flags

3. **glutInitWindowSize**
   a. Description: Sets the initial width and height of the window.
   b. Usage: glutInitWindowSize(int width, int height);
   c. Arguments:
      i. width → Desired window width in pixels (example: 950)
      ii. height → Desired window height in pixels (example: 550)

4. **glutInitWindowPosition**
   a. Description: Sets the initial position of the window on the screen.
   b. Usage: glutInitWindowPosition(int x, int y);
   c. Arguments:
      i. x → X coordinate of window's top-left corner from left of screen
      ii. y → Y coordinate of window's top-left corner from top of screen

5. **glutCreateWindow**
   a. Description: Creates the actual window and sets its title bar text.
   b. Usage: glutCreateWindow("Window Title");
   c. Arguments:
      i. "Window Title" → String to display in the window's title bar (example: "Primitives")

6. **glutDisplayFunc**
   a. Description: Registers a callback function that GLUT will call whenever the window needs to be drawn.
   b. Usage: glutDisplayFunc(display);
   c. Arguments:
      i. display → Name of your drawing function (without parentheses)

7. **glutMainLoop**
   a. Description: Sets the display in loop for current window.
   b. Usage: glutMainLoop();
   c. Arguments: None

8. **glClearColor**
   a. Description: Specifies the color to use when clearing the color buffer. This sets the background color that will fill the window.
   b. Usage: glClearColor(red, green, blue, alpha);
   c. Arguments:
      i. red → Red component value (0.0 to 1.0, where 0.0 = none, 1.0 = full red)
      ii. green → Green component value (0.0 to 1.0)
      iii. blue → Blue component value (0.0 to 1.0)
      iv. alpha → Alpha (transparency) component value (0.0 = transparent, 1.0 = opaque)
   d. Note: This function only sets the color. It does NOT clear the screen. Clearing happens with glClear.

9. **glClear**
   a. Description: Clears the specified buffers (color, depth, stencil) using the values previously set by functions like glClearColor.
   b. Usage: glClear(mask);
   c. Arguments:
      i. mask → Bitwise OR combination of buffers to clear (use | to combine multiple)
      ii. GL_COLOR_BUFFER_BIT → Clears the color buffer (fills window with glClearColor value)
   b. Note: Call this at the beginning of your display function before drawing anything.

What Each Buffer Type Requires

1. **For GLUT_SINGLE**
   - Must call: **glFlush()**
   - When to call: After glEnd() or after all drawing commands
   - What it does: Forces any pending drawing commands to be executed immediately

2. For **GLUT_DOUBLE**
   - Must call: **glutSwapBuffers()**
   - When to call: After glEnd() or after all drawing commands
   - What it does: Swaps the back buffer (where you draw) with the front buffer (what you see)

LAB: Setting Up Our Tools

Let's Get Our Hands Dirty!

Tools:

1. **Visual Studio:** The code editor and compiler we will use to write and run OpenGL programs.
2. **GLUT (OpenGL Utility Toolkit):** A pre-built library that helps us create windows and handle keyboard/mouse input (so we don't have to write complex window code from scratch).
3. **OpenGL Library:** The core graphics library that does all the drawing (the engine behind every shape, color, and animation).

installation steps

OpenGL Installation Steps (for Visual Studio)

Download GLUT Library

Step 1: Download GLUT

- Go to: **https://www.opengl.org/resources/libraries/glut/glut_downloads.php**
- Download **glutdlls37beta.zip** (or the latest Windows GLUT package)

Extract and Copy Files

Step 2: Extract the downloaded ZIP file

- You will get the following files:
    1. glut.h (header file)
    2. glut32.lib (library file)
    3. glut32.dll (dynamic link library)
    4. glut.lib
    5. glut.dll

Step 3: Copy 2 files (glut.lib and glut32.lib)

Step 4: Paste these 2 files into Visual Studio's lib folders

- Location depends on your Visual Studio version

Step 5: Copy 2 more files (glut.h and glut.dll)

Step 6: Create a folder named "GL"

- Navigate to Visual Studio's include directory
- Create a new folder called GL

Step 7: Paste these 2 files into the GL folder

Step 8: Copy glut32.dll

Step 9: Paste glut32.dll into System32 folder

- Path: C:\Windows\System32\glut32.dll

Step 10: Paste glut32.dll into SysWOW64 folder (for 64-bit systems)

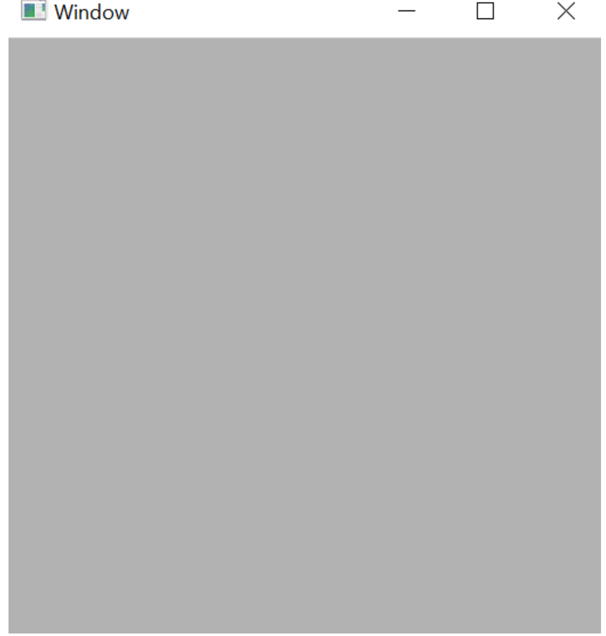
- Path: C:\Windows\SysWOW64\glut32.dll

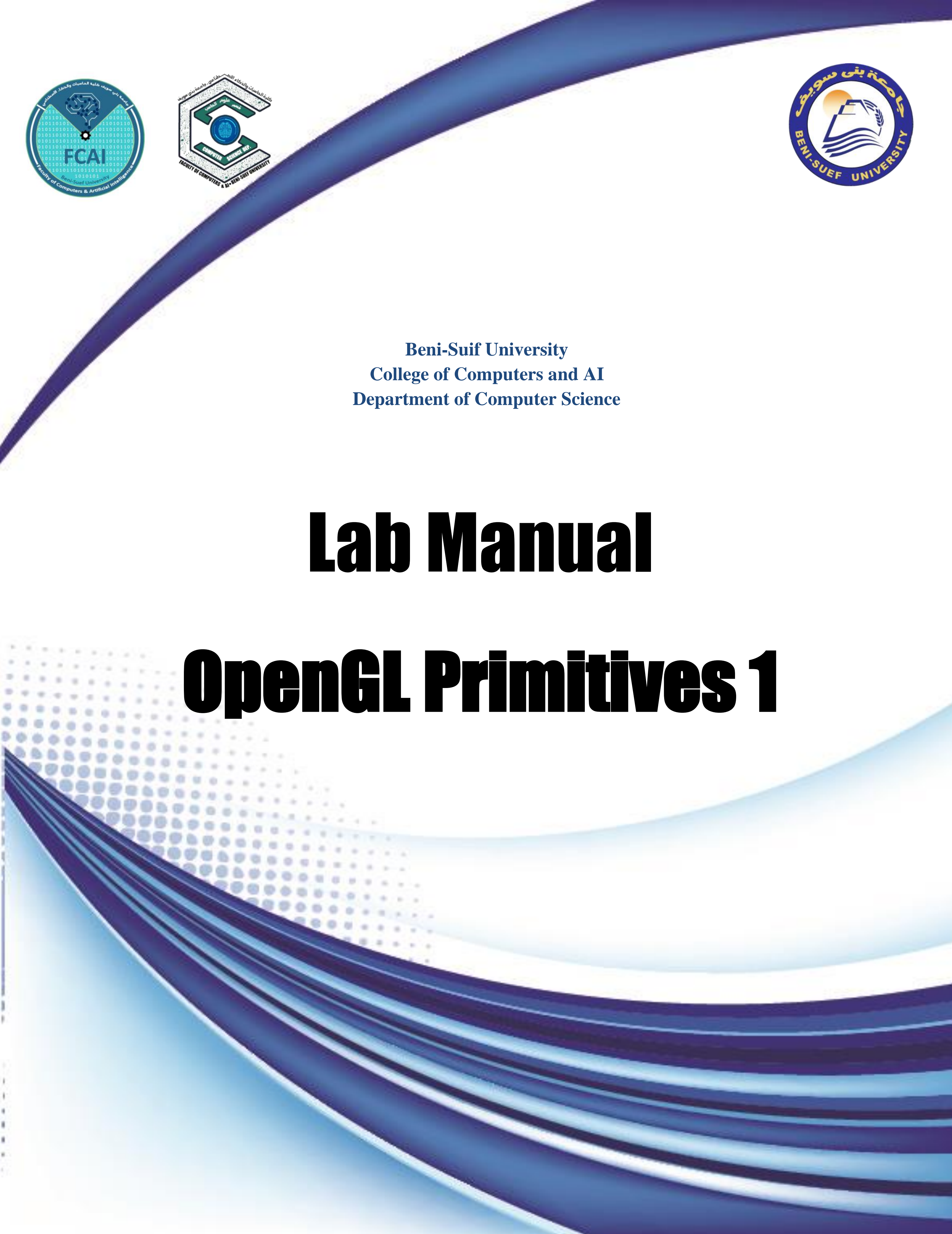
Our Very First Code!

Let's open a new c++ file and write a simple program to make sure everything works.

```
#include<GL\glut.h>       // Include GLUT library (for windows + OpenGL)

void display()
{
    // Set background color to light gray (R=0.7, G=0.7, B=0.7)
    glClearColor(0.7, 0.7, 0.7, 1.0);

    // Clear screen with the background color
    glClear(GL_COLOR_BUFFER_BIT);

    // Swap buffers (show what was drawn)
    glutSwapBuffers();
}

int main(int argc, char** argv)
{
    // Initialize GLUT
    glutInit(&argc, argv);

    // Double buffering + RGB colors
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);

    // Window size: 350x350 pixels
    glutInitWindowSize(350, 350);
    // Window position: at (500, 200) on screen
    glutInitWindowPosition(500, 200);

    // Create window with title "Window"
    glutCreateWindow("Window");

    // Register display function (drawing happens here)
    glutDisplayFunc(display);

    // Start the infinite event loop
    glutMainLoop();
    // Never reached (keeps window open)
    return 0;
}
```

Expected output:





Beni-Suif University
College of Computers and AI
Department of Computer Science

Lab Manual

OpenGL Primitives 1

Lab - 2
OpenGL Primitives 1

1. An Overview?

This lab introduces one of the most fundamental ideas in computer graphics: the concept of **graphics primitives**. By the end of this session, you will understand what makes a primitive the building block of all shapes, how different primitive types work, and the three main primitive families. You will also draw your first shapes in OpenGL.

Learning Objectives

After completing this lab, you should be able to:

- **Define** what OpenGL primitives are and what is modes.
- **Use** the glBegin() and glEnd() structure to start and finish drawing any primitive.
- **Identify** the three families of primitives: Points, Lines, and Polygons.
- **Name and compare** the some primitive modes
  - GL_POINTS
  - GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP
- **Implement** a complete OpenGL program that draws multiple primitives (a house, a star, or a simple scene).
- **Control** primitive appearance using helper functions like glColor3f(), glPointSize(), and glLineWidth().

2. Theory

What are OpenGL Primitives?
OpenGL primitives are the most basic shapes you can draw. Think of them as the alphabet of graphics – every complex 3D model (cars, characters, buildings) is made by connecting many small primitives together.

Simple definition: Primitives are the building blocks of all OpenGL graphics. You cannot draw anything more basic than a point, line, or triangle.

What are Types of Primitives (Modes)?
OpenGL provides three families of primitives: Points, Lines, and Polygons (triangles and quads).

- **Point Primitives:**
  - GL_POINTS.
- **Line Primitives:**
  - GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP
- **Polygon Primitives:**
  - GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
  - GL_QUADS, GL_POLYGON

Let's start to know how to draw this primitives before to know difference for each

How to Draw Primitives (Functions)
The Basic Drawing Pattern
All drawing happens inside the display() function between glBegin() and glEnd().

```
 // Start drawing with specific mode
glBegin(mode);
    glVertex*() // List vertices here
glEnd();          // Finish drawing
```

The Vertex Function
Function: glVertex*()

- Description: Specifies a single point (vertex) in 2D or 3D space
- Syntax variations:
  - glVertex2f(x, y) → 2D with floating point coordinates
  - glVertex3f(x, y, z) → 3D with floating point coordinates
  - glVertex2i(x, y) → 2D with integer coordinates
  - glVertex3d(x, y, z) → 3D with double precision
- Arguments:
  - x → X coordinate (horizontal position)
  - y → Y coordinate (vertical position)
  - z → Z coordinate (depth for 3D)

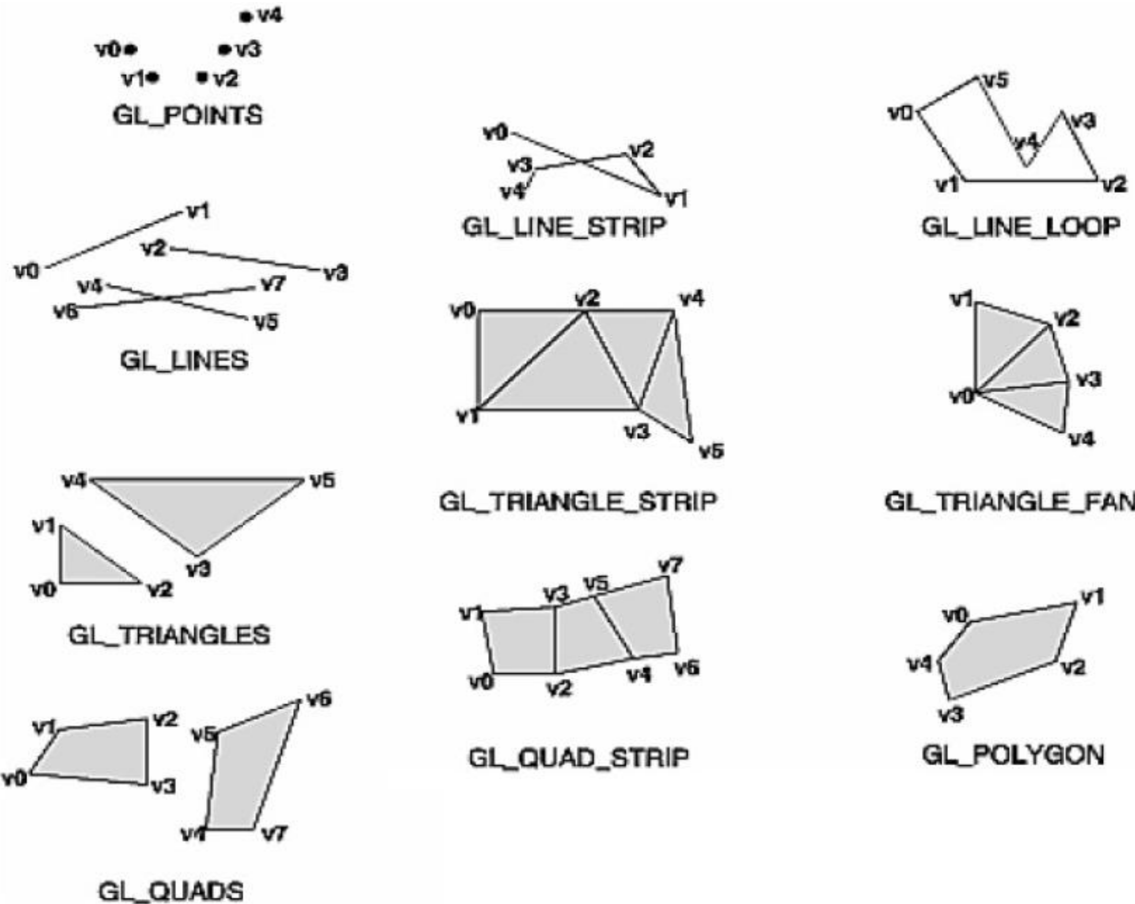


Figure 1 — The Types of modes and number of vertexs for each.

In this Lab lets start with Point Primitive and Line Primitive:

Part 1: Point Primitive

What is GL_POINTS?

- GL_POINTS is the simplest primitive in OpenGL
- It draws each vertex as a single, independent dot on the screen
- Each vertex becomes one pixel (or more if you increase the size)
- No connections exist between points — each point stands alone

How Vertices Are Processed

- Every vertex you specify becomes exactly one point
- If you provide 1 vertex → you get 1 point
- If you provide 5 vertices → you get 5 separate points
- There is no minimum or maximum number of vertices
- Example: glBegin(GL_POINTS); with 10 vertices creates 10 separate dots

How Drawing Works

- The computer takes each vertex coordinate (x, y)
- It finds the corresponding pixel on the screen
- It colors that pixel with the current color
- It moves to the next vertex and repeats
- Nothing connects vertex 1 to vertex 2 — they are separate

Functions to Control Point Appearance

1. glPointSize()
    - Purpose: Controls how large each point appears on screen
    - Syntax: glPointSize(float size);
    - Argument: Size in pixels (1.0 is default, 5.0 is five times bigger)
2. glColor3f()
    - Purpose: Sets the color of all points drawn after this command
    - Syntax: glColor3f(float red, float green, float blue);
3. glEnable(GL_POINT_SMOOTH)
    - Purpose: Makes points appear as smooth circles instead of square blocks
    - Syntax: glEnable(GL_POINT_SMOOTH);
    - Effect: Points become round with anti-aliased edges (no jagged corners)

Example:

```
void display()
{
    glPointSize(10);
    glEnable(GL_POINT_SMOOTH);
    glBegin(GL_POINTS);
    glColor3f(0.0, 0.0, 0.0);
    glVertex3f(-0.5, -0.5, 0);          // Draw a point at bottom-left
    glVertex3f(0.5, -0.5, 0);           // Draw a point at bottom-right
    glVertex3f(-0.5, 0.5, 0);           // Draw a point at top-left
    glVertex3f(0.5, 0.5, 0);            // Draw a point at top-right
    glEnd();
    glutSwapBuffers();
}
```

Result: Four large black dots appear at the four corners of a square shape, none connected

Part 2: Line Primitive

OpenGL provides three different line primitives. Each connects vertices differently.

1. GL_LINES

What is it?

- Draws separate, unconnected line segments
- Each pair of vertices creates one line
- Lines do not connect to each other

How vertices are paired

- Vertices 1 and 2 form line 1
- Vertices 3 and 4 form line 2
- And so on...

Number of vertices required

- Minimum: 2 vertices
- Must be even number: If you give an odd number, the last vertex is ignored
- Example: 2 vertices = 1 line, 4 vertices = 2 lines, 6 vertices = 3 lines

Example:

```
void display()
{
   glBegin(GL_LINES);
   // Line 1 (vertices 1 and 2)
   glVertex2f(-0.8, -0.5);   // Start point of line 1
   glVertex2f(-0.2, 0.5);    // End point of line 1

   // Line 2 (vertices 3 and 4)
   glVertex2f(0.2, -0.5);    // Start point of line 2
   glVertex2f(0.8, 0.5);     // End point of line 2

   // Line 3 (vertices 5 and 6)
   glVertex2f(-0.5, 0.8);    // Start point of line 3
   glVertex2f(0.5, 0.8);     // End point of line 3
   glEnd();
   glutSwapBuffers();
}
```

Result: Three separate lines, none connected

2. GL_LINE_STRIP

What is it?

- Draws a connected chain of lines
- Each new vertex connects to the previous vertex
- The strip is open (does not connect back to the start)

How vertices are connected

- Vertex 1 to vertex 2 forms line 1
- Vertex 2 to vertex 3 forms line 2
- Vertex 3 to vertex 4 forms line 3
- And so on...

Number of vertices required

- Minimum: 2 vertices
- No restriction on count: Any number works (2, 3, 4, 5, etc.)
- Number of lines drawn: (Number of vertices) – 1
- Examples:
    a. 2 vertices → 2 – 1 = 1 line (just a single segment)
    b. 3 vertices → 3 – 1 = 2 lines connected at vertex 2

Example:

```
void display()
{
   glBegin(GL_LINE_STRIP);
      glVertex2f(-0.8, -0.5);   // Vertex 1 (start)
      glVertex2f(-0.4, 0.3);    // Vertex 2 (connects to vertex 1)
      glVertex2f(0.0, -0.2);    // Vertex 3 (connects to vertex 2)
      glVertex2f(0.4, 0.4);     // Vertex 4 (connects to vertex 3)
      glVertex2f(0.8, -0.4);    // Vertex 5 (connects to vertex 4)
   glEnd();
   glutSwapBuffers();
}
```

Result: One continuous line (Like M Character)

3. GL_LINE_LOOP

What is it?

- Draws a connected chain of lines that closes back to the start
- Same as GL_LINE_STRIP, plus an extra line connecting last vertex back to first

How vertices are connected

- Vertex 1 to vertex 2 forms line 1
- Vertex 2 to vertex 3 forms line 2
- Vertex 3 to vertex 4 forms line 3
- And so on...
- Last vertex back to vertex 1 forms the final line

Number of vertices required

- Minimum: 2 vertices
- No restriction on count: Any number works (2, 3, 4, 5, etc.)
- Number of lines drawn: Same number as vertices
- Examples:
    c. 2 vertices → 2 lines (back and forth — a line drawn twice)
    d. 3 vertices → 3 lines (a triangle outline)
    e. 4 vertices → 4 lines (a rectangle or square outline)

Example:

```
void display()
{
   glBegin(GL_LINE_LOOP);
      glVertex2f(-0.5, 0.5);    // Vertex 1 (top-left)
      glVertex2f(0.5, 0.5);     // Vertex 2 (top-right)
      glVertex2f(0.5, -0.5);    // Vertex 3 (bottom-right)
      glVertex2f(-0.5, -0.5);   // Vertex 4 (bottom-left)
   glEnd();
   glutSwapBuffers();
}
```

Result: A square outline

Difference Between GL_LINE_STRIP and GL_LINE_LOOP

Both GL_LINE_STRIP and GL_LINE_LOOP draw connected lines. The key difference is that GL_LINE_LOOP closes the shape by connecting the last vertex back to the first, while GL_LINE_STRIP leaves the shape open.

Features	GL_LINE_STRIP	GL_LINE_LOOP
Connection style	Connects each vertex to the next one	Connects each vertex to the next one
Last to first connection	Does NOT connect last vertex back to first	Automatically connects last vertex back to first
Shape type	Produces an open shape	Produces a closed shape
Number of lines	vertices − 1	vertices
Visual appearance	Open chain (like a path)	Closed polygon (like a frame)
Best used for	Graphs, paths, open curves	Outlines, borders, closed polygons

Functions to Control Line Appearance

1. glLineWidth()
   a. Purpose: Controls how thick each line appears on screen
   b. Syntax: glLineWidth(float width);
   c. Argument: Width in pixels (1.0 is default)
2. glColor3f()
   a. Purpose: Sets the color of all lines drawn after this command
   b. Syntax: glColor3f(float red, float green, float blue);

3. Lab

Lab Goal — Exercise 1

Draw a dotted square using GL_POINTS at the four corners. Then in Exercise 2, draw a square using different line primitives. Compare how each primitive connects vertices.

Exercise 1: Dotted square using GL_POINTS

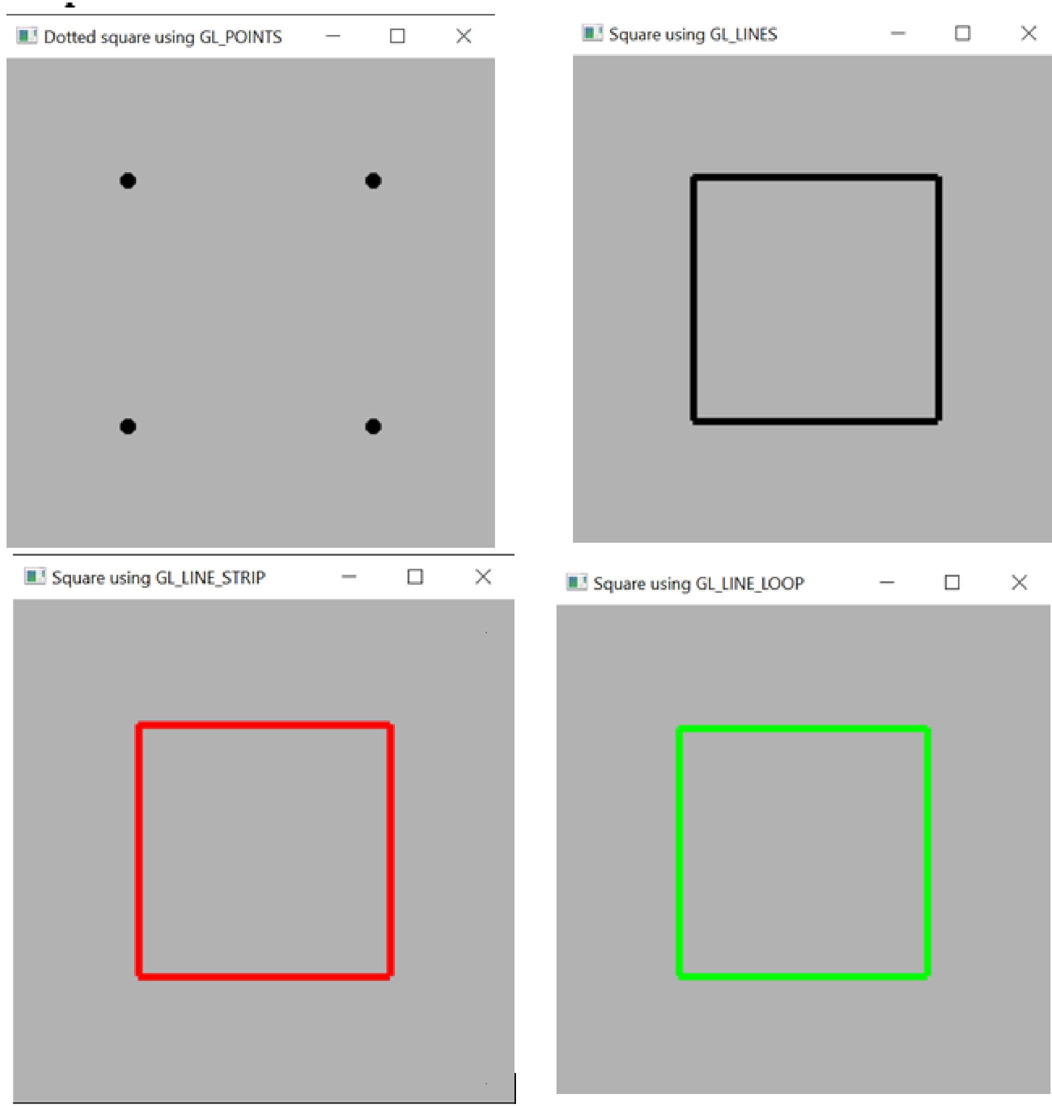
```
#include<GL/glut.h>

void display()
{
    glClearColor(0.7,0.7,0.7,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glPointSize(10);
    glEnable(GL_POINT_SMOOTH);
    glBegin(GL_POINTS);
    glColor3f(0.0, 0.0, 0.0);
    glVertex3f(-0.5, -0.5, 0);    // Draw a point at bottom-left
    glVertex3f(0.5, -0.5, 0);     // Draw a point at bottom-right
    glVertex3f(-0.5, 0.5, 0);     // Draw a point at top-left
    glVertex3f(0.5, 0.5, 0);      // Draw a point at top-right
    glEnd();
    glutSwapBuffers();
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(350, 350);
    glutInitWindowPosition(500, 200);
    glutCreateWindow("Dotted square using GL_POINTS");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Exercise 2: Draw a square using different line primitives

```
#include<GL/glut.h>
void squareLines()
{
   glClearColor(0.7,0.7,0.7,1.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glLineWidth(5);
   glColor3f(0.0, 0.0, 0.0);
   glBegin(GL_LINES);
      //bottom line
      glVertex3f(-0.5, -0.5, 0);
      glVertex3f(0.5, -0.5, 0);
      //right line
      glVertex3f(0.5, -0.5, 0);
      glVertex3f(0.5, 0.5, 0);
      //top line
      glVertex3f(0.5, 0.5, 0);
      glVertex3f(-0.5, 0.5, 0);
      //left line
      glVertex3f(-0.5, 0.5, 0);
      glVertex3f(-0.5, -0.5, 0);
   glEnd();
   glutSwapBuffers();
}

void squareLineStrip()
{
   glClearColor(0.7,0.7,0.7,1.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glLineWidth(5);
   glColor3f(1.0, 0.0, 0.0);
   glBegin(GL_LINE_STRIP);
      //bottom line
      glVertex3f(-0.5, -0.5, 0);
      glVertex3f(0.5, -0.5, 0);
      //right line
      glVertex3f(0.5, 0.5, 0);
```

```
        //top line
        glVertex3f(-0.5, 0.5, 0);
        //left line
        glVertex3f(-0.5, -0.5, 0);
    glEnd();
    glutSwapBuffers();
}
void squareLineLoop()
{
    glClearColor(0.7,0.7,0.7,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glLineWidth(5);
    glColor3f(0.0, 1.0, 0.0);
    glBegin(GL_LINE_LOOP);
        //bottom line
        glVertex3f(-0.5, -0.5, 0);
        glVertex3f(0.5, -0.5, 0);
        //right line
        glVertex3f(0.5, 0.5, 0);
        //top line
        glVertex3f(-0.5, 0.5, 0);
        //left line connect to first line automatically
    glEnd();
    glutSwapBuffers();
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(350, 350);
    glutInitWindowPosition(500, 200);
    glutCreateWindow("Square using GL_LINE_LOOP");
    glutDisplayFunc(squareLineLoop);
    glutMainLoop();
    return 0;
}
```

Output:



🔍 Student Task

Draw a simple house outline using different line primitives. You must use each of the three line primitives (GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP) at least once .in your drawing.

4. Summary

Here is everything we covered this week:

✦ What is a Primitive?

The most basic shape OpenGL can draw (point, line, triangle). Building blocks of all computer graphics.

✦ Three Primitive Families

Points, Lines, Polygons

✦ Point primitive

Draws each vertex as a single, independent dot. No connections between points. Each vertex = one point.

✦ Line Primitive

✦ GL_LINES

Draws separate, unconnected line segments. Needs even number of vertices.

✦ GL_LINE_STRIP

Draws a connected chain of lines. Each new vertex connects to the previous vertex. Number of lines = vertices − 1.

✦ GL_LINE_LOOP

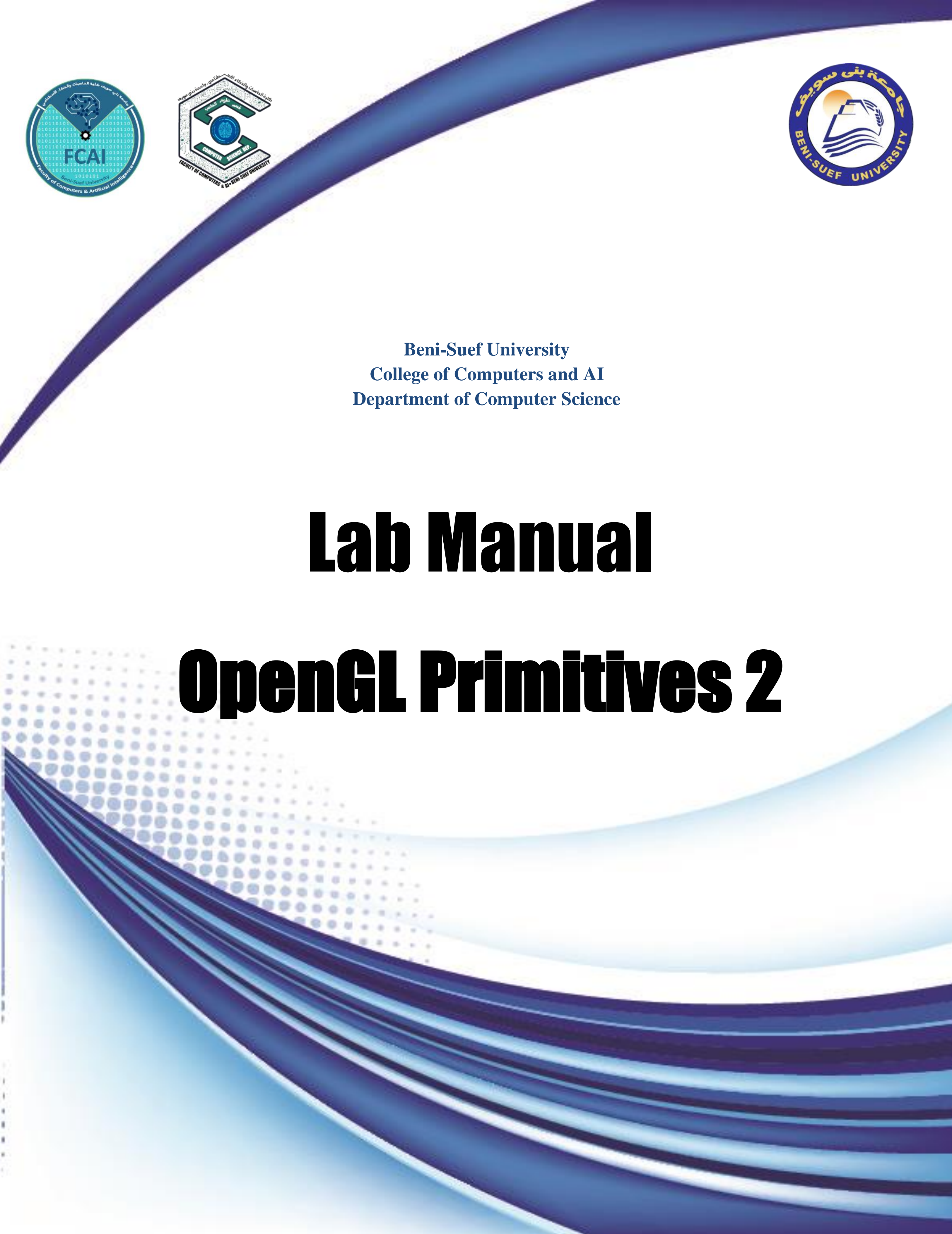
Draws a connected chain that closes back to the start. Number of lines = vertices.

✦ Functions to Control Appearance

glPointSize(size) → Sets point thickness in pixels
glColor3f(r,g,b) → Sets shape color
glEnable(GL_POINT_SMOOTH) → Makes points round and smooth
glLineWidth(width) → Sets line thickness in pixels

✦ Key Lab Takeaway

Understanding how vertices connect is the foundation of computer graphics. Choose LINE_LOOP for closed shapes (square, triangle), LINE_STRIP for open paths (graph), and LINES for separate segments (window cross bars, parallel lines).



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

OpenGL Primitives 2

Lab - 3
OpenGL Primitives 2

1. An overview

This lab introduces **triangles and polygon** – the foundation of all filled shapes. By the end of this session, you will understand how triangles and quads form every complex shape. You will also draw your first filled shapes in OpenGL.

Learning Objectives

After completing this lab, you should be able to:

- **Define** what polygon primitives are .
- **Use** GL_TRIANGLES, GL_TRIANGLE_STRIP, and GL_TRIANGLE_FAN to draw filled triangles.
- **Use** GL_QUADS and GL_POLYGON to draw four-sided and multi-sided shapes.
- **Identify** the difference between separate triangles, triangle strips, and triangle fans.
- **Calculate** the number of triangles drawn from a given number of vertices for each mode.
- **Implement** a complete OpenGL program that draws filled shapes (a rectangle, a circle, a hexagon).
- **Control** primitive appearance using glColor3f() for filled shapes.

2. Theory

What are Polygon Primitives?
Polygon primitives are filled shapes – they have area, not just outlines. While points and lines are useful, polygons are what make 2D and 3D graphics look solid. Every model you see in video games and movies is made of thousands of connected polygons.

Simple definition: Polygons are the building blocks of solid graphics. You cannot create a solid-looking object without them.

What are Types of Polygon Primitives (Modes)?
OpenGL provides five polygon primitives:

- **Triangle Primitives:**
  - GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
- **Quad Primitives:**
  - GL_QUADS
- **General Primitives:**
  - GL_POLYGON

How to Draw Polygons (The Basic Pattern)
Same as points and lines – drawing happens between glBegin() and glEnd():

```
// Start drawing with polygon mode
    glBegin(mode);
        glVertex2f(x, y);  // List vertices in order
        glVertex2f(x, y);
        glVertex2f(x, y);
        // ... more vertices
    glEnd();            // Finish drawing (shape becomes filled)
```

Part 1: Triangles Primitive

1. GL_TRIANGLES

What is it?

- Draws separate, independent triangles
- Every 3 vertices creates one triangle
- Triangles do NOT share edges or vertices

How vertices are grouped

- Vertices 1, 2, 3 → Triangle 1
- Vertices 4, 5, 6 → Triangle 2
- Vertices 7, 8, 9 → Triangle 3
- And so on...

Number of vertices required

- Minimum: 3 vertices
- Must be multiple of 3: 3, 6, 9, 12, etc.
- If not multiple of 3: Extra vertices are ignored
- Formula: Number of triangles = vertices ÷ 3

Example:

```
void display()
{
   glBegin(GL_TRIANGLES);
      // Triangle 1 (separate)
      glVertex2f(-0.7, -0.5);
      glVertex2f(-0.2, 0.5);
      glVertex2f(0.0, -0.5);

      // Triangle 2 (separate - not connected to Triangle 1)
      glVertex2f(0.3, -0.5);
      glVertex2f(0.5, 0.5);
      glVertex2f(0.9, -0.5);
   glEnd();
   glutSwapBuffers();
}
```

Result: Two separate triangles, not connected.

2. GL_TRIANGLE_STRIP

What is it?

- Draws a connected strip of triangles
- Each new vertex forms a new triangle with the previous 2 vertices
- Highly efficient – uses fewer vertices to draw many triangles

How vertices are Connected

Triangle	Uses Vertices
Triangle 1	1, 2, 3
Triangle 2	2, 3, 4
Triangle 3	3, 4, 5
And so on...	

Number of vertices required

- Minimum: 3 vertices
- Any number works: 3, 4, 5, 6, etc.
- Formula: Number of triangles = vertices − 2

Example:

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(0.2, 0.2, 0.2, 1.0);

    glColor3f(0.0, 1.0, 0.0);  // Green

    glBegin(GL_TRIANGLE_STRIP);
        glVertex2f(-0.8, -0.5);   // Vertex 1
        glVertex2f(-0.5, 0.5);    // Vertex 2
        glVertex2f(0.0, -0.5);    // Vertex 3 → Triangle 1 (1,2,3)
        glVertex2f(0.3, 0.5);     // Vertex 4 → Triangle 2 (2,3,4)
        glVertex2f(0.8, -0.5);    // Vertex 5 → Triangle 3 (3,4,5)
    glEnd();

    glutSwapBuffers();
}
```

Result: One connected strip of 3 green triangles (total 5 vertices).

3. GL_TRIANGLE_FAN

What is it?

- Draws triangles that all share a common first vertex
- Forms a fan or pie-slice shape
- Perfect for circles, cones, and rounded shapes

How vertices are Connected

Triangle	Uses Vertices
Triangle 1	1, 2, 3
Triangle 2	1, 3, 4
Triangle 3	1, 4, 5
And so on...	

Note: Vertex 1 is the center – shared by ALL triangles.

Number of vertices required

- Minimum: 3 vertices
- Any number works: 3, 4, 5, 6, etc.
- Formula: Number of triangles = vertices − 2

Example:

```
void display()
{
   glBegin(GL_TRIANGLE_FAN);
      // Center vertex (shared by all triangles)
      glVertex2f(0.0, 0.0);

      // Outer vertices (forming a circle)
      glVertex2f(0.0, 0.6);
      glVertex2f(0.4, 0.4);
      glVertex2f(0.6, 0.0);
      glVertex2f(0.4, -0.4);
      glVertex2f(0.0, -0.6);
      glVertex2f(-0.4, -0.4);
      glVertex2f(-0.6, 0.0);
      glVertex2f(-0.4, 0.4);
      glVertex2f(0.0, 0.6);  // Close the circle
   glEnd();

   glutSwapBuffers();
}
```

Result: A filled white circle made of 9 triangles sharing the center.

Difference Between GL_TRIANGLES, GL_TRIANGLE_STRIP, and GL_TRIANGLE_FAN

Features	GL_TRIANGLES	GL_TRIANGLE_STRIP	GL_TRIANGLE_FAN
Connection style	Separate triangles	Connected strip	Connected fan
Shared vertices	None	Each triangle shares 2 vertices	All triangles share first vertex
Shape type	Independent triangles	Long strip	Pie/circle shape
Vertices needed for N triangles	N × 3	N + 2	N + 2
Best used for	Separate objects	Terrain, ribbons	Circles, cones, sectors
Example with 4 vertices	1 triangle (ignores vertex 4)	2 triangles	2 triangles

Part 2: Quad and Polygon Primitives

4. GL_QUADS

What is it?

- Draws separate, independent four-sided polygons
- Every 4 vertices creates one quadrilateral (rectangle or square)
- Quads are efficient for drawing flat rectangular surfaces

How vertices are Grouped

- Vertices 1, 2, 3, 4 → Quad 1
- Vertices 5, 6, 7, 8 → Quad 2
- And so on...

Number of vertices required

- Minimum: 4 vertices
- Must be multiple of 4: 4, 8, 12, 16, etc.
- If not multiple of 4: Extra vertices are ignored
- Formula: Number of quads = vertices ÷ 4

Example:

```
void display()
{
    glColor3f(1.0, 1.0, 0.0);  // Yellow
    glBegin(GL_QUADS);
        // Quad 1 (square)
        glVertex2f(-0.7, 0.3);
        glVertex2f(-0.1, 0.3);
        glVertex2f(-0.1, -0.3);
        glVertex2f(-0.7, -0.3);

        // Quad 2 (separate rectangle)
        glVertex2f(0.2, 0.5);
        glVertex2f(0.8, 0.5);
        glVertex2f(0.9, -0.2);
        glVertex2f(0.1, -0.2);
    glEnd();

    glutSwapBuffers();
}
```

Result: Two separate filled yellow rectangles.

5. GL_QUAD_STRIP

What is it?

- Draws a connected strip of quads (rectangles)
- Each new pair of vertices forms a new quad connected to the previous one
- Highly efficient for drawing long chains of connected rectangles

How vertices are Connected

- Quad 1 uses vertices 1, 2, 3, 4
- Quad 2 uses vertices 3, 4, 5, 6
- And so on...

Order of Vertices for Each Quad

For a strip, vertices should be ordered in a zigzag pattern:

First quad: bottom-left, top-left, bottom-right, top-right

Next quad: shares bottom-right and top-right, then adds next bottom-left, top-left

Number of vertices required

- Minimum: 4 vertices
- Must be even number: 4, 6, 8, 10, 12, etc.
- Number of quads = (vertices ÷ 2) – 1
- Formula: Number of quads = (vertices ÷ 2) − 1

Example:

```
void display()
{
    glColor3f(0.0, 1.0, 1.0);  // Cyan
    glBegin(GL_QUAD_STRIP);
        // Quad 1 (left rectangle)
        glVertex2f(-0.8, -0.3);   // Vertex 1: bottom-left of quad 1
        glVertex2f(-0.8, 0.3);    // Vertex 2: top-left of quad 1
        glVertex2f(-0.2, -0.3);   // Vertex 3: bottom-right of quad 1
        glVertex2f(-0.2, 0.3);    // Vertex 4: top-right of quad 1
        // Quad 2 (middle rectangle - shares vertices 3 and 4)
        glVertex2f(0.4, -0.3);    // Vertex 5: bottom-right of quad 2
        glVertex2f(0.4, 0.3);     // Vertex 6: top-right of quad 2
        // Quad 3 (right rectangle - shares vertices 5 and 6)
        glVertex2f(0.9, -0.3);    // Vertex 7: bottom-right of quad 3
        glVertex2f(0.9, 0.3);     // Vertex 8: top-right of quad 3
    glEnd();
    glutSwapBuffers();
}
```

Result: Three connected cyan 2D rectangles forming a strip.

Difference Between GL_QUADS and GL_QUAD_STRIP

Features	GL_QUADS	GL_QUAD_STRIP
Connection style	Separate, independent quads	Connected strip (sharing edges)
Vertex grouping	4 vertices per quad	New quad shares last 2 vertices
Vertices needed for N quads	N × 4	(N × 2) + 2
Sharing between quads	No sharing	Each quad shares edge with next
Best used for	Separate rectangles (windows, doors)	Connected rectangles (path, fence, platform)
Example with 2 quads	8 vertices	6 vertices

6. GL_POLYGON

What is it?

- Draws a single filled polygon with any number of sides
- Unlike other primitives, all vertices form ONE shape
- Useful for irregular or multi-sided shapes

How vertices are Connected

- All vertices are connected in order
- Last vertex automatically connects back to first
- Creates one closed, filled polygon

Number of vertices required

- Minimum: 3 vertices
- Any number works: 3, 4, 5, 6, 7, etc.

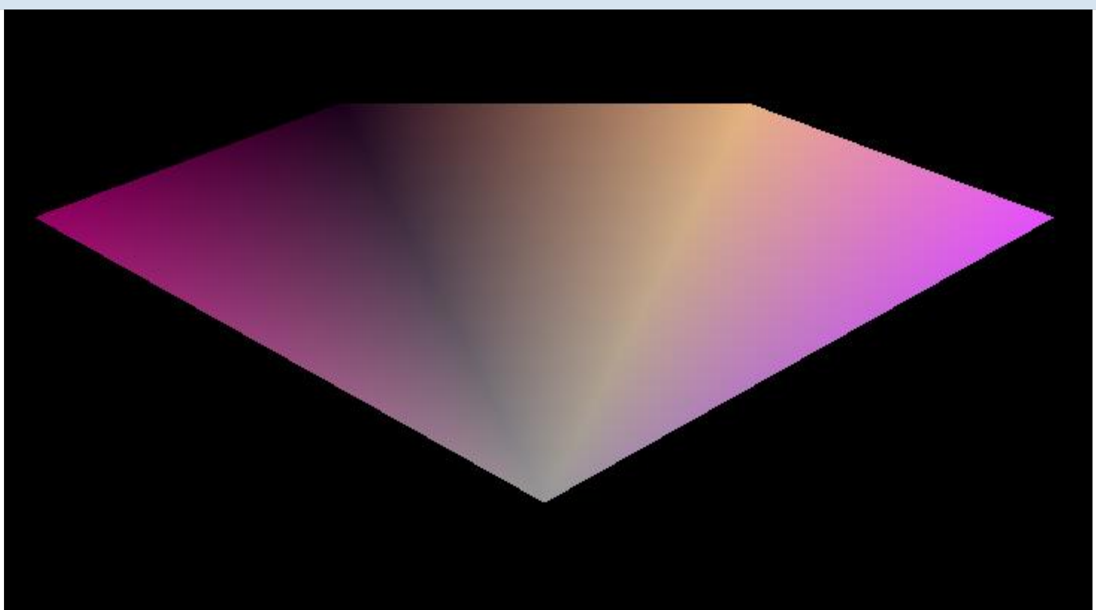
Example:

```
void display()
{
    glColor3f(1.0, 0.5, 0.0);  // Orange

    glBegin(GL_POLYGON);
        // Hexagon vertices (counter-clockwise)
        glVertex2f(0.0, 0.5);
        glVertex2f(0.43, 0.25);
        glVertex2f(0.43, -0.25);
        glVertex2f(0.0, -0.5);
        glVertex2f(-0.43, -0.25);
        glVertex2f(-0.43, 0.25);
    glEnd();

    glutSwapBuffers();
}
```

Result: One filled orange hexagon.

3. Lab

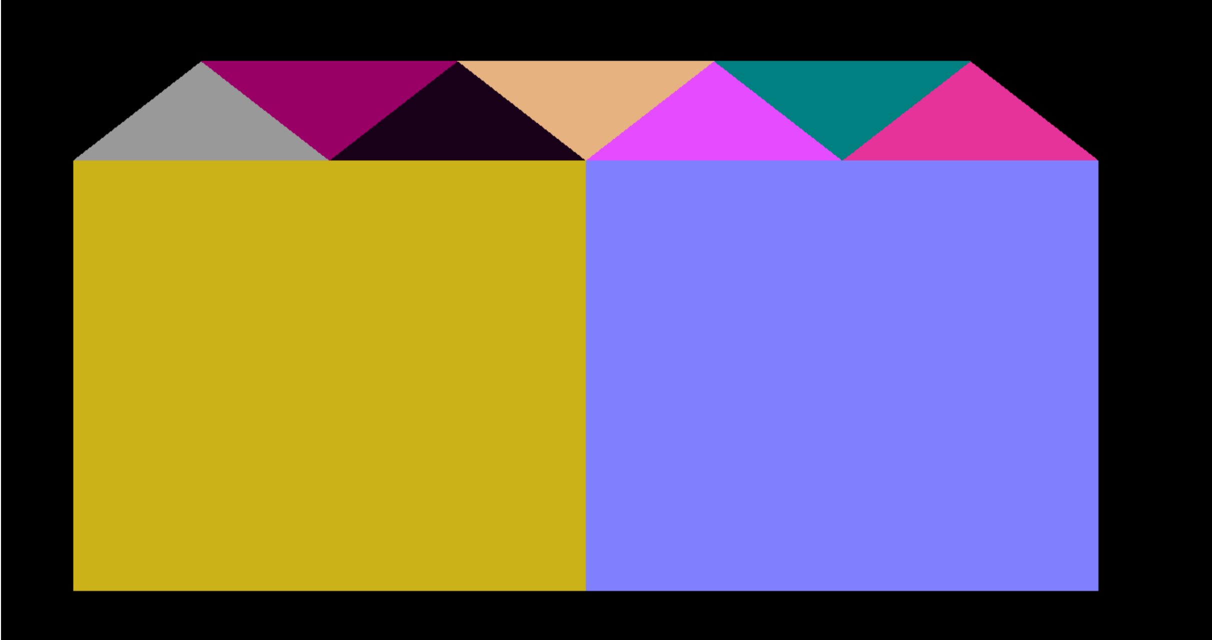
Lab Goal — Exercise 1
Draw the below shape



```
#include<GL\glut.h>
void display()
{
   glBegin(GL_TRIANGLE_FAN);
      glColor3f(0.6, 0.6, 0.6);
      glVertex2f(0,0);//0
      glColor3f(0.6, 0.0, 0.4);
      glVertex2f(-.5, .5);//1
      glColor3f(0.1, 0.0, 0.1);
      glVertex2f(-0.2, .7);//2
      glColor3f(.9, 0.7, 0.5);
      glVertex2f(0.2, .7);//3
      glColor3f(0.9, 0.3, 1.0);
      glVertex2f(.5, 0.5);//4
   glEnd();
   glutSwapBuffers();
}
int main(int argc, char** argv)
```

```
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize(600, 350);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("Primitives");
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Exercise 2: Draw this shape



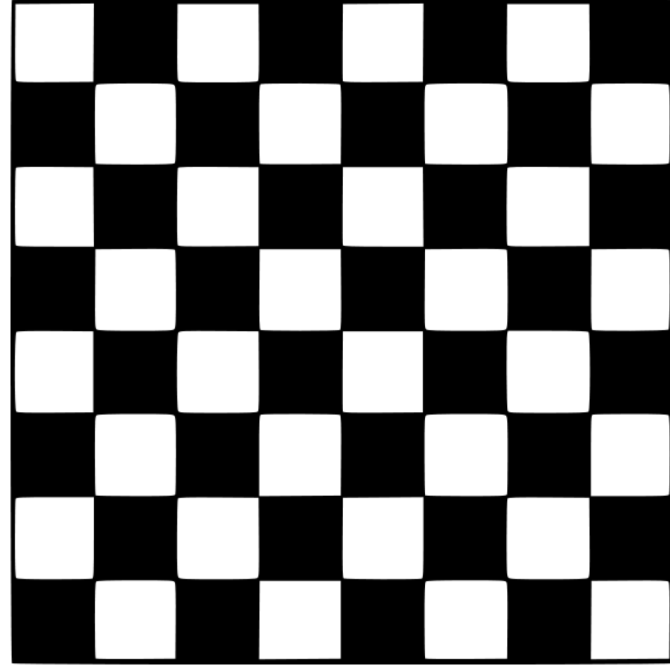
```
#include<GL\glut.h>
void display()
{
   glBegin(GL_QUADS);
      glColor3f(0.8, 0.7, 0.1);
      glVertex2f(-.8, -.8);
      glVertex2f(-.8,.5);
      glVertex2f(0, 0.5);
      glVertex2f(0, -.8);
```

```
    glColor3f(0.5, 0.5, 1.0);
    glVertex2f(0, 0.5);
    glVertex2f(0, -.8);
    glVertex2f(.8, -.8);
    glVertex2f(.8, .5);
glEnd();

glBegin(GL_TRIANGLES);
    glColor3f(0.6, 0.6, 0.6);
    glVertex2f(-0.8, .5);
    glVertex2f(-0.6, .8);
    glVertex2f(-0.4, .5);
    glColor3f(0.6, 0.0, 0.4);
    glVertex2f(-0.6, .8);
    glVertex2f(-0.4, .5);
    glVertex2f(-0.2, .8);
    glColor3f(0.1, 0.0, 0.1);
    glVertex2f(-0.4, .5);
    glVertex2f(-0.2, .8);
    glVertex2f(0.0, .5);
    glColor3f(.9, 0.7, 0.5);
    glVertex2f(-0.2, .8);
    glVertex2f(0.0, .5);
    glVertex2f(0.2, .8);
    glColor3f(0.9, 0.3, 1.0);
    glVertex2f(0, 0.5);
    glVertex2f(0.2, .8);
    glVertex2f(.4, 0.5);
    glColor3f(0.0, 0.5, 0.5);
    glVertex2f(0.2, .8);
    glVertex2f(0.4, .5);
    glVertex2f(0.6, .8);
    glColor3f(0.9, 0.2, 0.6);
    glVertex2f(0.4, .5);
    glVertex2f(0.6, .8);
    glVertex2f(0.8, .5);
glEnd();
glutSwapBuffers();
```

```
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(600, 550);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Arguments");
    glutDisplayFunc(display);
    glutMainLoop();
return 0;
}
```

🔍 Student Task

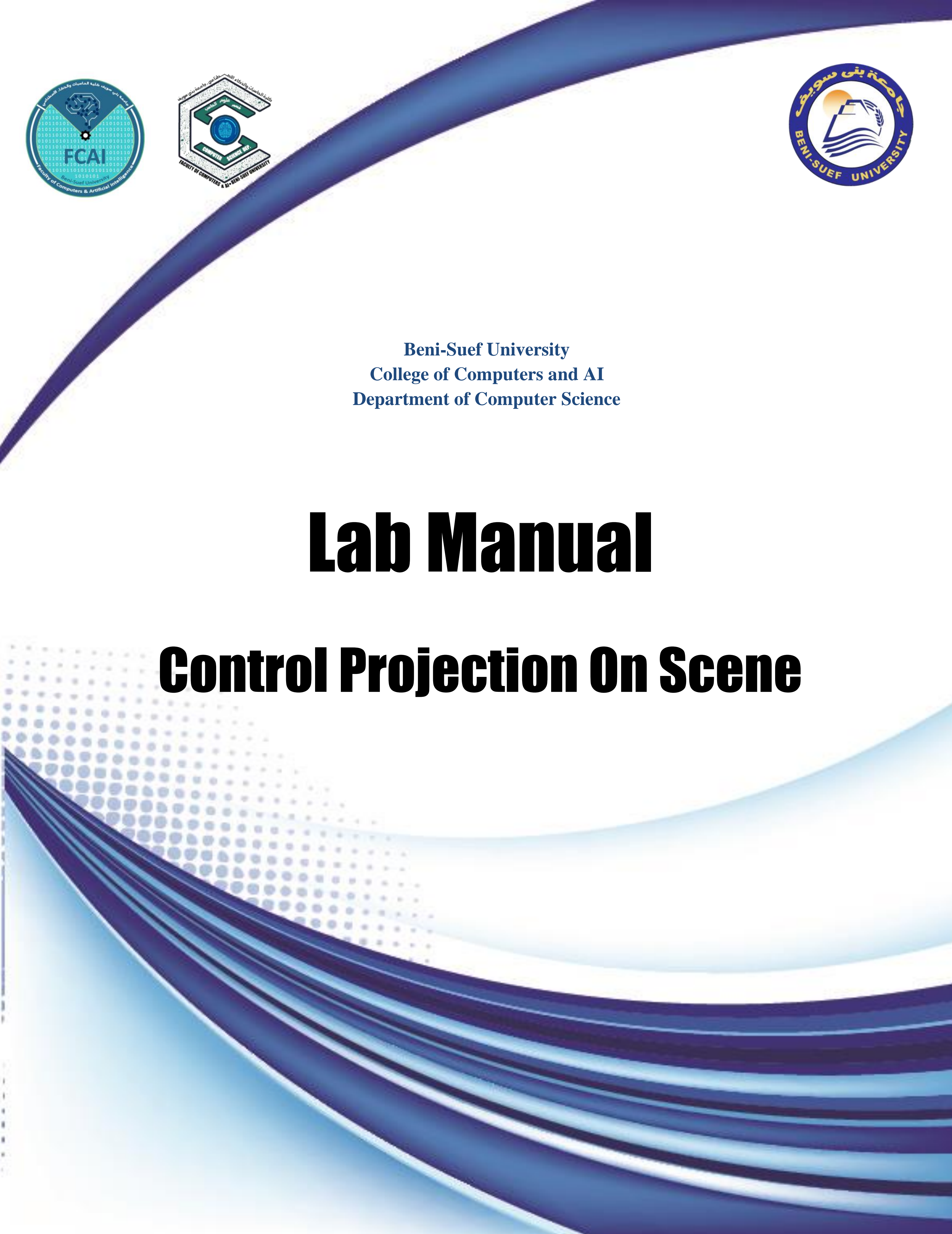
Draw ChessBoard shape using OpenGl Primitive functions.



4. Summary

Here is everything we covered this week:

✦ What is a Polygon Primitive? **A filled shape with area – the foundation of solid graphics. Every complex model is made of thousands of connected polygons.**
✦ GL_TRIANGLES **Draws separate, independent triangles. Every 3 vertices = 1 triangle. Vertices must be multiple of 3.** **✦ GL_TRIANGLE_STRIP** **Draws connected strip of triangles. Each new vertex forms triangle with previous 2 vertices. Number of triangles = vertices − 2.** **✦ GL_TRIANGLE_FAN** **Draws triangles sharing a common center vertex. Number of triangles = vertices − 2**
✦ GL_QUADS **Draws separate four-sided polygons. Every 4 vertices = 1 quad. Vertices must be multiple of 4**
✦ GL_POLYGON **Draws ONE filled polygon with any number of sides. Vertices must be convex and counter-clockwise.**



Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab Manual

Control Projection On Scene

Lab - 4
Control Projection On Scene

1. An overview

This lab introduces one of the most fundamental concepts in 2D graphics: setting up the coordinate system using orthographic projection. By the end of this session, you will understand how to define your own coordinate system, why it matters, and how to map your 2D shapes to the screen. You will also draw a circle using GL_POINTS.

Learning Objectives

After completing this lab, you should be able to:

- **Define** what orthographic projection is and why we need it in 2D graphics.
- **Use** gluOrtho2D() to set custom coordinate boundaries for your scene.
- **Use** glMatrixMode() to switch to the projection matrix.
- **Use** glLoadIdentity() to reset the matrix before setting projection.
- **Understand** the correct order of projection setup commands.
- **Implement** a complete OpenGL program that draws a circle using GL_POINTS with custom coordinates.

2. Theory

Why Do We Need Projection Control?

The Problem
By default, OpenGL uses a coordinate system where:

- X ranges from -1.0 to 1.0 (left to right)
- Y ranges from -1.0 to 1.0 (bottom to top)
- Origin (0,0) is at the center of the screen

This is fine for simple shapes, but what if you want to:

- Use pixel coordinates (e.g., 0 to 800, 0 to 600)?
- Use world coordinates (e.g., -300 to 300)?
- Draw shapes that need precise positioning?

Solution: We need to tell OpenGL how to map our desired coordinates to the screen.

What is Orthographic Projection?

Orthographic projection is a way to map 3D coordinates to 2D screen coordinates without perspective. In 2D graphics, it simply means:

"Define a rectangle in your world, and map it exactly to the window rectangle."

Simple definition: You tell OpenGL: "My world goes from X_left to X_right and from Y_bottom to Y_top. Please map that to the entire window."

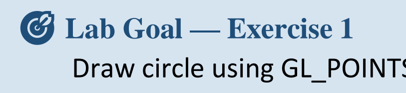
Functions for Projection Control

1. **glMatrixMode()**
    a. Purpose: Tells OpenGL which matrix you want to modify (projection, modelview, texture, etc.)
    b. Syntax: glMatrixMode(GLenum mode);
    c. Arguments:
        i. GL_PROJECTION → For setting up camera/view/projection
        ii. GL_MODELVIEW → For positioning objects.
        iii. GL_TEXTURE → For texture mapping
    d. When to use: Always before setting up projection (gluOrtho2D)

2. **glLoadIdentity()**
   a. Purpose: Resets the current matrix to the identity matrix (clears any previous transformations)
   b. Syntax: glLoadIdentity();
   c. Arguments: None
   d. Why important: Without this, previous transformations would combine with your new projection, causing unexpected results

3. **gluOrtho2D()**
   a. Purpose: Sets up a 2D orthographic projection with specified boundaries
   b. Syntax: gluOrtho2D(double left, double right, double bottom, double top);
   c. Arguments:
      i. left → X coordinate of the left edge of the window
      ii. right → X coordinate of the right edge of the window
      iii. bottom → Y coordinate of the bottom edge of the window
      iv. top → Y coordinate of the top edge of the window
   d. Effect: Maps the rectangle (left, bottom) to (right, top) to the entire window

3. Lab



Lab Goal — Exercise 1

Draw circle using GL_POINTS

How the Circle Drawing Works

The circle is drawn using the parametric equation:

- x = radius × cos(angle)
- y = radius × sin(angle)
- Angle goes from 0 to 2π (360 degrees)

Each point on the circle is one vertex using GL_POINTS.

```
#include<GL/glut.h>
#include<math.h>

void display()
{
   glClearColor(0.4, 1, 0.8, 1.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glPointSize(5);

   gluOrtho2D(-300,300,-300,300);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   float x, y, i;
   glBegin(GL_POINTS);
   glColor3f(0.0, 0.0, 0.0);
   for (i = 0; i <= 2 * 3.14; i += 0.1)
   {
   x = 200 * cos(i);
   y = 200 * sin(i);
   glVertex2d(x, y);
   }
   glEnd();
   glFlush();
```

```
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(500, 200);
    glutCreateWindow("Draw Circle");
    glutDisplayFunc(display);

    glutMainLoop();
    return 0;
}
```

🔍 **Student Task 1**

Run the circle code. Then answer:

What happens if you change gluOrtho2D(-300, 300, -300, 300) to gluOrtho2D(-200, 200, -200, 200)?

What happens if you change the radius from 200 to 350?

What happens if you remove glLoadIdentity()?

🔍 **Student Task 2**

Modify the code to draw:

A half-circle (angle from 0 to π only)

Two circles (one inside the other) using different colors

Beni-Suef University

College of Computers and AI

Department of Computer Science

Lab Manual

Computer Graphics Fundamentals

Preparing the scientific material

Prof Mohamed salah

T.A. Manar Abbas

T.A. Somia Ahmed

Contents

Section #	Description
Section 5	Draw Circle Using GL_POINTS — OpenGL DDA Algorithm

Beni-Suef University
College of Computers and AI
Department of Computer Science

Section - 6

Computer Graphics Fundamentals

1. INTRODUCTION

Computer graphics is the discipline concerned with generating and manipulating visual content using computers. At its core, every image displayed on a screen is composed of discrete picture elements called pixels, and the algorithms that determine which pixels to illuminate are fundamental to the field. This section covers two important topics: drawing a circle using GL_POINTS in OpenGL, and the Digital Differential Analyzer (DDA) line-drawing algorithm.

2. OPENGL 2D COORDINATE SYSTEM

Before rendering any 2D shape in OpenGL, the coordinate system must be configured correctly. OpenGL provides two key functions for this purpose:

gluOrtho2D

gluOrtho2D(left, right, bottom, top) is an OpenGL utility function that sets up a 2D orthographic projection. It defines the viewing area (clipping area) for rendering 2D shapes with no perspective distortion. Objects outside this clipping area will be discarded and cannot be seen on screen.

Signature: void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)

glMatrixMode & glLoadIdentity

glMatrixMode(GL_PROJECTION) sets the current matrix mode to the projection matrix, which controls how objects are projected onto the screen. glLoadIdentity() then resets this matrix to the identity matrix, removing any previous transformations before the new projection is applied.

The OpenGL 2D coordinate system places the origin (0,0) at the center of the window, with x increasing to the right and y increasing upward — opposite to typical screen coordinates where (0,0) is at the top-left corner.

Fig. 1 — OpenGL 2D Coordinates vs. Viewport Coordinates

3. DRAW CIRCLE USING GL_POINTS

A circle can be drawn in OpenGL by plotting individual points along the circle's circumference. The parametric equations of a circle of radius r centered at the origin are:

$$x = r \cdot \cos(\theta) \quad \text{and} \quad y = r \cdot \sin(\theta)$$

By iterating the angle θ from 0 to 2π in small increments and computing the corresponding (x, y) coordinates, a series of points is plotted that visually approximates a circle. The smaller the increment, the smoother the circle appears.

Implementation

```
void display() {
  glClearColor(0.7, 0.7, 0.7, 0.1);
  glClear(GL_COLOR_BUFFER_BIT);
  glPointSize(5);
  gluOrtho2D(-300, 300, -300, 300);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();

  float x, y, i;
  glBegin(GL_POINTS);
  glColor3f(0.0, 0.0, 0.0);
  for (i = 0; i <= 2 * 3.14; i += 0.1) {
    x = 200 * cos(i);
    y = 200 * sin(i);
    glVertex2f(x, y);
  }
  glEnd();
  glFlush();
}
```

Fig. 2 — Output: Circle drawn using GL_POINTS with radius 200

Note: In the code above, gluOrtho2D and glMatrixMode must be called before plotting the points to ensure the coordinate system is set up correctly. The point size is set to 5 pixels for visibility.

---

4. DDA (DIGITAL DIFFERENTIAL ANALYZER) ALGORITHM

DDA is the simplest line-drawing algorithm in computer graphics. It works on an incremental method: it plots pixels step by step from the starting point to the ending point by incrementing the source coordinates in each step, then rounding to the nearest integer to identify the screen pixel to illuminate.

5. DDA ALGORITHM — STEP BY STEP

Given: starting point $(x0, y0)$ and ending point $(x1, y1)$, the algorithm proceeds as follows:

- Calculate $\Delta x = x1 - x0$ and $\Delta y = y1 - y0$.
- Determine the number of steps: if $|\Delta x| > |\Delta y|$ then steps = $|\Delta x|$, else steps = $|\Delta y|$.
- Calculate increments: x_inc = Δx / steps and y_inc = Δy / steps.
- Set initial position: x = x0, y = y0. Plot the first pixel at (round(x), round(y)).
- For each step from 1 to steps: x += x_inc; y += y_inc; Plot pixel at (round(x), round(y)).

6. WORKED EXAMPLE

Draw a line from (1, 2) to (7, 5).

Step 1: $\Delta x = 7 - 1 = 6$, $\Delta y = 5 - 2 = 3$

Step 2: $|\Delta x| > |\Delta y| \rightarrow$ steps = 6

Step 3: x_inc = 6/6 = 1, y_inc = 3/6 = 0.5

t	X	Y	R(x)	R(y)
0	1	2	1	2
1	2	2.5	2	3
2	3	3	3	3
3	4	3.5	4	4
4	5	4	5	4
5	6	4.5	6	5
6	7	5	7	5

Table 1 — DDA iteration table for line from (1,2) to (7,5)

7. EXPERIMENT

Using the DDA algorithm, manually compute the pixel positions for a line drawn from (2, 1) to (6, 3). Fill in the table below following the same procedure shown in Section 6.

t	X	Y	R(x)	R(y)

Table 2 — Student experiment: DDA for line from (2,1) to (6,3)

8. DDA IMPLEMENTATION IN OPENGL

The following C++ program implements the DDA algorithm inside an OpenGL window. The user enters the start and end coordinates at runtime, and the program draws the line using GL_POINTS.

```
#include<GL/freeglut.h>
#include<math.h>
#include <cstdlib>
#include<iostream>
using namespace std;

double x_start, x_end, y_start, y_end;

void display() {
    glClearColor(0.7, 0.7, 0.7, 0);
    glClear(GL_COLOR_BUFFER_BIT);
```

```
    glPointSize(5);
    gluOrtho2D(-250, 250, -250, 250);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    double delta_x = x_end - x_start;
    double delta_y = y_end - y_start;
    double num_steps = delta_x;

    if (abs(delta_x) < abs(delta_y)) {
        num_steps = abs(delta_y);
    }

    double x = x_start;
    double y = y_start;
    double x_inc = delta_x / num_steps;
    double y_inc = delta_y / num_steps;

    glBegin(GL_POINTS);
    glVertex2f(x, y);
    for (int i = 0; i <= num_steps; i++) {
        x += x_inc;
        y += y_inc;
        glVertex2f(round(x), round(y));
    }
    glEnd();
    glFlush();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    cout << "Enter first point:" << endl;
    cin >> x_start >> y_start;
    cout << "Enter End point:" << endl;
    cin >> x_end >> y_end;
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(500, 200);
    glutCreateWindow("DDA Algorithm");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Fig. 3 — DDA line drawing program in C++ / OpenGL (freeglut)

9. KEY CONCEPTS SUMMARY

The following table summarises the core OpenGL functions used in this section:

Function	Purpose

gluOrtho2D(l, r, b, t)	Sets the 2D orthographic projection / clipping area
glMatrixMode(GL_PROJECTION)	Switches current matrix mode to the projection matrix
glLoadIdentity()	Resets the current matrix to identity (removes transforms)
glBegin(GL_POINTS)	Starts defining a set of individual points to render
glVertex2f(x, y)	Specifies a 2D vertex (point) in OpenGL coordinates
glPointSize(n)	Sets the diameter of rendered points in pixels
glColor3f(r, g, b)	Sets the current drawing color using RGB floats (0.0–1.0)
glFlush()	Forces execution of all pending OpenGL commands

Table 3 — OpenGL function reference for Section 6

Lab Exercises

1. Modify the circle program to draw a circle of radius 150 centered at the origin. What changes are needed?
2. Manually apply DDA to draw the line from (0, 0) to (5, 3). Show the full iteration table.
3. In the DDA implementation, what happens if you remove the round() call? Explain the visual effect.
4. Modify the DDA program to draw two lines: one from (−200, −200) to (200, 0) and another from (0, 0) to (200, 200).

Lab Manual

Computer Graphics Lab

Preparing the scientific material

Prof .Mohamed Salah

T.A. Manar Abbas

T.A Somia Ahmed

Beni-Suef University

College of Computers and AI

Department of Computer Science

Contents

Lab #	Description
Section 6	Midpoint Circle Drawing Algorithm

Lab Manual

Computer Graphics Lab

Beni-Suef University
College of Computers and AI
Department of Computer Science

Lab
Midpoint Circle Drawing Algorithm

1. INTRODUCTION

In computer graphics, drawing geometric shapes on a raster display involves mapping continuous mathematical curves onto a finite grid of pixels. The circle is one of the most fundamental geometric primitives. Rather than relying on expensive floating-point arithmetic, computer graphics engineers developed integer-based incremental algorithms that approximate circular arcs efficiently. The Midpoint Circle Drawing Algorithm is one such algorithm — analogous to Bresenham's line algorithm — and is widely used due to its speed and simplicity.

2. DEFINITION OF A CIRCLE

A circle is defined as the set of all points that are equidistant from a given center point (Xc, Yc). The common distance from the center is called the radius r.

Two standard mathematical representations are used:

2.1 Polar Coordinates

In polar form, any point on the circle is expressed as:

$$x = Xc + r \cdot \cos(\theta)$$

$$y = Yc + r \cdot \sin(\theta)$$

By stepping θ from 0° to 360°, one can trace the full circle. However, this approach requires trigonometric computations (cos, sin) at each step, making it computationally expensive for real-time graphics.

2.2 Cartesian Coordinates

Using the Pythagorean theorem, the Cartesian equation of a circle centered at (Xc, Yc) is:

$$(x - Xc)^2 + (y - Yc)^2 = r^2$$

Solving for y:

$$y = Yc \pm \sqrt{r^2 - (Xc - x)^2}$$

While correct, this approach involves a square-root operation at every pixel step — computationally inefficient for hardware rasterization.

3. DRAWBACKS OF TRADITIONAL APPROACHES

Both traditional approaches suffer from performance limitations:

- The Cartesian equation requires multiplication and square-root calculations at every pixel position.
- The parametric polar equations require trigonometric computations (sin, cos) at each step.
- Floating-point arithmetic introduces rounding errors and is inherently slower than integer arithmetic.
- Neither approach is suitable for real-time or hardware-accelerated rendering without optimization.

More efficient circle algorithms are based on incremental calculation of a decision parameter, eliminating the need for square roots and trigonometric functions entirely.

4. MIDPOINT CIRCLE DRAWING ALGORITHM

The Midpoint Circle Algorithm exploits the 8-way symmetry of a circle: if a point (x, y) lies on the circle, then the seven other symmetric points (−x, y), (x, −y), (−x, −y), (y, x), (−y, x), (y, −x), (−y, −x) all lie on the circle as well. This means we only need to calculate pixels for the first octant (0° to 45°) and reflect them into all eight octants.

The algorithm works as follows:

- Calculate pixel positions for a circle centered at the origin (0, 0).
- For each calculated position (x, y), translate it to screen coordinates by adding (Xc, Yc).
- Use the circle function $f(x, y) = x^2 + y^2 - r^2$ to determine whether a midpoint is inside or outside the circle.

The circle function evaluates as:

f(x, y) Value	Point Location
$f(x, y) < 0$	Inside the circle
$f(x, y) = 0$	On the circle boundary
$f(x, y) > 0$	Outside the circle

At each step, after plotting pixel (xk, yk), the algorithm must choose the next pixel from two candidates: (xk+1, yk) or (xk+1, yk−1). The decision is made by evaluating the circle function at the midpoint between these two candidates.

5. DECISION PARAMETER DERIVATION

The initial decision parameter is computed from the midpoint of the first candidate pixels, starting at (0, r):

$$p_0 = 1 - r$$

At each subsequent step k, the decision parameter is updated incrementally:

Condition	Next Pixel Chosen	Update Formula
pk < 0	(xk+1, yk)	pk+1 = pk + 2xk+1 + 1
pk ≥ 0	(xk+1, yk−1)	pk+1 = pk + 2xk+1 + 1 − 2yk+1

These update formulas use only integer addition and subtraction — no multiplication, division, or square roots — making the algorithm extremely fast and suitable for hardware implementation.

6. WORKED EXAMPLE — RADIUS = 5, CENTER = (0, 0)

Given: r = 5, center = (0, 0)

Initial values: $p_0 = 1 - r = 1 - 5 = -4$ → first point: (0, 5)

6.1 Decision Parameter Table

k	pk	xk+1	yk+1	2xk+1	2yk+1
0	−4	1	5	2	10
1	−1	2	5	4	10
2	4	3	4	6	8
3	3	4	3	8	6

6.2 Eight-Way Symmetry Point Table

Using 8-way symmetry, the full set of plotted pixel coordinates for r = 5 is:

+x,+y	−x,+y	+x,−y	−x,−y	+y,+x	−y,+x	+y,−x	−y,−x
(0,5)	(0,5)	(0,−5)	(0,−5)	(5,0)	(−5,0)	(5,0)	(−5,0)
(1,5)	(−1,5)	(1,−5)	(−1,−5)	(5,1)	(−5,1)	(5,−1)	(−5,−1)
(2,5)	(−2,5)	(2,−5)	(−2,−5)	(5,2)	(−5,2)	(5,−2)	(−5,−2)
(3,4)	(−3,4)	(3,−4)	(−3,−4)	(4,3)	(−4,3)	(4,−3)	(−4,−3)
(4,3)	(−4,3)	(4,−3)	(−4,−3)	(3,4)	(−3,4)	(3,−4)	(−3,−4)

7. ALGORITHM STEPS (PSEUDOCODE)

The complete Midpoint Circle Drawing Algorithm can be stated as follows:

```
Input:  radius r, center (Xc, Yc)
Output: set of pixel coordinates approximating the circle

Step 1:  Set x = 0, y = r
Step 2:  Calculate initial decision parameter: p = 1 - r
Step 3:  Plot the eight symmetric points of (x, y) around (Xc, Yc)
Step 4:  While x < y:
    a.  Increment x:  x = x + 1
    b.  If p < 0:
            p = p + 2x + 1
        Else:
            y = y - 1
            p = p + 2(x - y) + 1
    c.  Plot eight symmetric points of (x, y) around (Xc, Yc)
Step 5:  Stop when x ≥ y (first octant complete)
```

8. OPENGL IMPLEMENTATION (C++)

The following C++ program implements the Midpoint Circle Algorithm using OpenGL and FreeGLUT. The user enters the center coordinates and radius at runtime, and the program renders the circle on screen.

8.1 Point-Plotting Function

The circleplotpoints() function exploits 8-way symmetry by plotting all eight symmetric pixels for each calculated (x, y) pair:

```
void circleplotpoints(int xcenter, int ycenter, int x, int y) {
    glVertex2i(xcenter + x, ycenter + y);
    glVertex2i(xcenter - x, ycenter + y);
    glVertex2i(xcenter + x, ycenter - y);
    glVertex2i(xcenter - x, ycenter - y);
    glVertex2i(xcenter + y, ycenter + x);
    glVertex2i(xcenter - y, ycenter + x);
    glVertex2i(xcenter + y, ycenter - x);
    glVertex2i(xcenter - y, ycenter - x);
}
```

8.2 Main Circle Drawing Function

```
#include <GL/freeglut.h>
#include <iostream>
using namespace std;

int xcenter, ycenter, radius;

void circlemidpoint() {
    int x = 0;
    int y = radius;
    int p = 1 - radius;

    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glPointSize(5.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-300, 300, -300, 300);

    glBegin(GL_POINTS);
    circleplotpoints(xcenter, ycenter, x, y);

    while (x < y) {
        x++;
        if (p < 0) {
            p += 2 * x + 1;
        } else {
            y--;
            p += 2 * (x - y) + 1;
        }
        circleplotpoints(xcenter, ycenter, x, y);
    }
```

```
    glEnd();
    glFlush();
}

int main(int argc, char** argv) {
    cout << "Enter center of the circle (x y): ";
    cin >> xcenter >> ycenter;
    cout << "Enter radius: ";
    cin >> radius;

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(0, 0);
    glutInitWindowSize(600, 600);
    glutCreateWindow("Mid Point Circle Algorithm");
    glutDisplayFunc(circlemidpoint);
    glutMainLoop();
    return 0;
}
```

8.3 Compilation Instructions

To compile and run on Linux with FreeGLUT installed:

```
g++ -o circle circle.cpp -lGL -lGLU -lfreeglut
./circle
```

9. ADVANTAGES OF THE MIDPOINT CIRCLE ALGORITHM

- Uses only integer arithmetic (addition and subtraction) — no floating-point operations.
- Exploits 8-way symmetry, reducing the computation to just one octant (0° to 45°).
- Highly efficient: only a few additions are needed per pixel.
- Produces a visually symmetric, smooth-looking circle on raster displays.
- Easy to implement in hardware for GPU rasterization pipelines.

Lab Exercise. PRACTICE PROBLEMS

1. Using the Midpoint Circle Algorithm, manually calculate the pixel coordinates for a circle with radius = 8 centered at (0, 0). Fill in a decision parameter table similar to the example in Section 6.
2. Modify the provided OpenGL code to draw a circle centered at (100, 50) with radius 80. What change must be made to the main() function?
3. Explain why the algorithm only needs to compute pixels for the first octant (x from 0 to y). How does 8-way symmetry reduce computational cost?
4. Compare the Cartesian, Polar, and Midpoint approaches for drawing circles. For each, state the types of arithmetic required and identify the most computationally efficient approach.

Lab Manual

Computer Graphics Fundamentals

Section 7: Transformations

Prepared by:

T.A. Manar Abbas

T.A.Somia Ahmed

Beni-Suef University

College of Computers and AI

Department of Computer Science

FCAI
Faculty of Computers & Artificial Intelligence
Beni-Suef University
COMPUTER SCIENCE DEP.
FACULTY OF COMPUTERS & AI • BENI-SUEF UNIVERSITY
جامعة بنى سويف
BENI-SUEF UNIVERSITY

We also explore how transformations can be composed (combined), animated using timer functions, and isolated using matrix stack operations (glPushMatrix / glPopMatrix).

2. TRANSLATION

2.1 Definition

Translation moves an object by a specified displacement in the x and y (and z in 3D) directions. Every point (x, y) of the object is mapped to a new point (x', y') by adding the translation amounts Tx and Ty.

2.2 Equations

```
x' = x + Tx
y' = y + Ty
```

2.3 Translation Matrix (2D)

In homogeneous coordinates the translation is expressed as a 3×3 matrix multiplication:

2D Translation Matrix:

1	0	Tx
0	1	Ty
0	0	1

In 3D, the matrix becomes 4×4 with an added Tz term for the z-axis. The function glTranslated() internally multiplies the current OpenGL matrix by this translation matrix.

2.4 OpenGL Functions

```
glTranslatef( GLfloat  x, GLfloat  y, GLfloat  z );
glTranslated( GLdouble x, GLdouble y, GLdouble z );
```

The function translates an object x units along the x-axis, y units along the y-axis, and z units along the z-axis.

2.5 Example — Worked Problem

Move the line with endpoints (5, 5) and (20, 5) seven units in the x-direction and five units in the y-direction.

Solution:

```
Point A:  x' = 5 + 7 = 12,   y' = 5 + 5 = 10   →  A' = (12, 10)
```

```
Point B:  x' = 20 + 7 = 27,  y' = 5 + 5 = 10   →  B' = (27, 10)
```

2.6 Lab Exercise — Translation Code

The following program draws a triangle centred at the origin and translates it 4 units along the x-axis using glTranslatef:

```
#include<GL/freeglut.h>
#include<iostream>
using namespace std;

void display() {
    glClearColor(0, 0, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-10, 10, -10, 10);
    glTranslatef(4, 0, 0);    // Translate 4 units along x-axis
    glBegin(GL_TRIANGLES);
        glVertex2f(0, 2);
        glVertex2f(-2, -2);
        glVertex2f(2, -2);
    glEnd();
    glFlush();
}

int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Translation");
    glutDisplayFunc(display);
    glutMainLoop();
}
```

3. SCALING

3.1 Definition

Scaling changes the size of an object by multiplying each coordinate by a scaling factor. A factor greater than 1 enlarges the object, a factor between 0 and 1 shrinks it, and a negative factor additionally reflects the object.

3.2 Equations

```
x' = x * Sx
y' = y * Sy
```

3.3 Scaling Matrix (2D)

In homogeneous coordinates the scaling transformation is:

2D Scaling Matrix:

Sx	0	0
0	Sy	0
0	0	1

In 3D the diagonal extends to include Sz. The function glScaled() multiplies the current OpenGL matrix by this scaling matrix.

3.4 OpenGL Functions

```
glScalef( GLfloat  x, GLfloat  y, GLfloat  z );
glScaled( GLdouble x, GLdouble y, GLdouble z );
```

glScalef(u, v, w) maps each point (x, y, z) of the object to the new point (u·x, v·y, w·z).

3.5 Example — Worked Problem

Scale the rectangle with vertices (12, 4), (20, 4), (20, 8), (12, 8) using Sx = 2 and Sy = 2.

Solution:

```
(12,4) → (24, 8)
(20,4) → (40, 8)
(20,8) → (40, 16)
(12,8) → (24, 16)
```

3.6 Lab Exercise — Scaling Code

The following program draws a triangle and scales it by a factor of 4 in both x and y using glScaled:

```
#include<GL/freeglut.h>
#include<iostream>
using namespace std;

void display() {
    glClearColor(0, 0, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-10, 10, -10, 10);
    glScaled(4, 4, 1);     // Scale 4x in x and y
    glBegin(GL_TRIANGLES);
        glVertex2f(0, 2);
        glVertex2f(-2, -2);
        glVertex2f(2, -2);
    glEnd();
    glFlush();
}

int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
```

```
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Scaling");
    glutDisplayFunc(display);
    glutMainLoop();
}
```

4. ROTATION

4.1 Definition

Rotation moves every point of an object through a specified angle about a pivot (usually the origin). Positive angles produce counter-clockwise rotation in standard mathematical convention.

4.2 Equations

$$x' = x \cdot \cos(\theta) - y \cdot \sin(\theta)$$
$$y' = x \cdot \sin(\theta) + y \cdot \cos(\theta)$$

4.3 Rotation Matrix (2D)

In homogeneous coordinates:

2D Rotation Matrix:

cos θ	**-sin θ**	**0**
sin θ	**cos θ**	**0**
0	**0**	**1**

The function glRotated() multiplies the current OpenGL matrix by a rotation matrix constructed from the specified angle and axis vector.

4.4 OpenGL Functions

```
glRotatef( GLfloat  angle, GLfloat  x, GLfloat  y, GLfloat  z );
glRotated( GLdouble angle, GLdouble x, GLdouble y, GLdouble z );
```

glRotatef(A, p, q, r) rotates the object A degrees counter-clockwise around the axis defined by the vector from the origin (0,0,0) to the point (p, q, r). For 2D rotation about the z-axis, use glRotatef(angle, 0, 0, 1).

4.5 Example — Worked Problem

Rotate the line with endpoints (1, 6) and (5, 1) anticlockwise by 90 degrees.

Solution (using θ = 90°, cos 90° = 0, sin 90° = 1):

```
Point A(1,6):  x' = 1·0 − 6·1 = −6,   y' = 1·1 + 6·0 = 1   →
A' = (−6, 1)
Point B(5,1):  x' = 5·0 − 1·1 = −1,   y' = 5·1 + 1·0 = 5   →
B' = (−1, 5)
```

4.6 Lab Exercise — Rotation Code

The following program draws a triangle and rotates it 45 degrees counter-clockwise around the z-axis using glRotated:

```
#include<GL/freeglut.h>
#include<iostream>
using namespace std;

void display() {
    glClearColor(0, 0, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-10, 10, -10, 10);
    glRotated(45, 0, 0, 1);    // Rotate 45° around z-axis
    glBegin(GL_TRIANGLES);
        glVertex2f(0, 2);
        glVertex2f(-2, -2);
        glVertex2f(2, -2);
    glEnd();
    glFlush();
}

int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Rotation");
    glutDisplayFunc(display);
    glutMainLoop();
}
```

5. MATRIX STACK — glPushMatrix() & glPopMatrix()

When applying multiple transformations in a scene, it is often necessary to apply a transformation to one object without affecting others. OpenGL provides a matrix stack for this purpose.

- glPushMatrix() — saves (pushes) a copy of the current transformation matrix onto the stack.
- glPopMatrix() — restores (pops) the previously saved matrix from the stack, undoing any transformations applied since the last push.

This mechanism allows transformations to be isolated: apply glPushMatrix() before transforming an object, draw the object, then call glPopMatrix() to restore the original coordinate state for subsequent objects.

6. ANIMATION

6.1 glutTimerFunc()

OpenGL animation is achieved by repeatedly updating object state and redrawing the scene. The function glutTimerFunc registers a callback that fires after a specified delay:

```
glutTimerFunc( unsigned int msecs, void (*func)(int value), int value );
```

The callback function is invoked msecs milliseconds after glutTimerFunc is called. To create a continuous animation loop, the timer callback re-registers itself before returning.

6.2 glutPostRedisplay()

This function marks the current window as requiring a redraw. On the next iteration of the GLUT main loop it triggers a call to the registered display() function, causing the scene to be redrawn with updated values. It is typically called from inside the timer callback.

6.3 Lab Exercise — Animated Translation

The following program animates a red triangle bouncing horizontally across the screen:

```
#include <GL/freeglut.h>
float tx = -8;
float step = 0.1;

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(1, 1, 1, 1);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-10, 10, -10, 10);
    glPushMatrix();
    glTranslatef(tx, 0, 0);
    glBegin(GL_TRIANGLES);
        glColor3f(1, 0, 0);
        glVertex2f(0, 2);
        glVertex2f(-2, -2);
        glVertex2f(2, -2);
    glEnd();
    glPopMatrix();
    glFlush();
}

void timer(int) {
    tx += step;
    if (tx > 8 || tx < -8) step = -step;
    glutPostRedisplay();
    glutTimerFunc(60, timer, 0);
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutCreateWindow("Animated Triangle");
    glutDisplayFunc(display);
    glutTimerFunc(0, timer, 0);
```

```
    glutMainLoop();
    return 0;
}
```

7. COMPOSING TRANSFORMATIONS

Transformations can be combined (composed) by calling multiple OpenGL transformation functions in sequence. An important rule to remember is:

> **Transformations are applied to the object in reverse order through the code — the last transformation written is the first applied.**

Example 1: Translate then rotate (rotation is applied first to the object):

```
glTranslatef(0.0, 0.0, -15.0);
glTranslatef(10.0, 0.0,   0.0);
glRotatef(45.0,  0.0, 0.0,  1.0);
```

Example 2: Translate, scale, then rotate:

```
glTranslatef(0.0, 0.0, -15.0);
glScalef(1.0, 3.0, 1.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
```

7.1 Lab Exercise — Composite Animation

The following program combines translation and rotation in a single animation — the triangle moves horizontally while spinning continuously:

```
#include <GL/freeglut.h>
float angle = 0;
float tx = -8;

void display() {
    glClearColor(1, 1, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-10, 10, -10, 10);
    glPushMatrix();
    glTranslatef(tx, 0, 0);
    glRotatef(angle, 0, 0, 1);
    glColor3f(0, 0.6, 1.0);
    glBegin(GL_TRIANGLES);
        glVertex2f(0, 2);
        glVertex2f(-2, -2);
        glVertex2f(2, -2);
    glEnd();
    glPopMatrix();
    glFlush();
}

void timer(int) {
    tx += 0.05;
    if (tx > 8) tx = -8;
```

```
    angle += 20;
    if (angle >= 360) angle = 0;
    glutPostRedisplay();
    glutTimerFunc(60, timer, 0);
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutCreateWindow("Transformations Combo");
    glutDisplayFunc(display);
    glutTimerFunc(0, timer, 0);
    glutMainLoop();
    return 0;
}
```

8. SUMMARY

This section covered the three fundamental geometric transformations in computer graphics and their OpenGL implementations:

Summary of OpenGL Transformation Functions

Transformation	OpenGL Function	Effect
Translation	glTranslatef(x, y, z)	Moves object by (x, y, z)
Scaling	glScalef(x, y, z)	Resizes object by factors (x, y, z)
Rotation	glRotatef(a, x, y, z)	Rotates object a° around axis (x,y,z)
Push Matrix	glPushMatrix()	Saves current transformation state
Pop Matrix	glPopMatrix()	Restores last saved transformation state
Timer	glutTimerFunc(ms, fn, v)	Schedules animation callback after ms ms
Redisplay	glutPostRedisplay()	Marks window for redraw on next loop

Understanding and combining these transformations is essential for building any non-trivial graphics application, from simple shape manipulation to complex animated 3D scenes.

— End of Section 8 —

Lab Manual

Computer Graphics Fundamentals

Section 8: Transformations

Prepared by:

T.A. Manar Abbas
T.A.Somia Ahmed

Beni-Suef University
College of Computers and AI
Department of Computer Science

FCAI
Beni-Suef University
Faculty of Computers & Artificial Intelligence
COMPUTER SCIENCE DEP.
FACULTY OF COMPUTERS & AI • BENI-SUEF UNIVERSITY
جامعة بنى سويف
BENI-SUEF UNIVERSITY

We also explore how transformations can be composed (combined), animated using timer functions, and isolated using matrix stack operations (glPushMatrix / glPopMatrix).

8. REFLECTION AND SHEARING

This section introduces two additional geometric transformations: Reflection and Shearing. Both are affine transformations and can be expressed as matrix multiplications in homogeneous coordinates.

8.1 Definition (Reflection)

Reflection produces a mirror image of an object across a chosen axis. Every point is mapped symmetrically to the opposite side of the axis. Reflection across the x-axis negates the y-coordinates, reflection across the y-axis negates the x-coordinates, and reflection across both axes negates both.

8.2 Equations (Reflection)

Reflection across the x-axis:

```
x' = x
y' = -y
```

Reflection across the y-axis:

```
x' = -x
y' = y
```

8.3 Reflection Matrix (2D)

In homogeneous coordinates, reflection across the x-axis is represented by:

2D Reflection Matrix (across x-axis):

1	0	0
0	-1	0
0	0	1

Reflection across the y-axis:

2D Reflection Matrix (across y-axis):

-1	0	0
0	1	0
0	0	1

8.4 OpenGL Functions (Reflection)

OpenGL has no dedicated reflection function. Reflection is achieved by using glScalef() with a negative factor on the target axis:

```
glScalef( 1, -1, 1 );   // Reflect across x-axis (negate y)
glScalef(-1,  1, 1 );   // Reflect across y-axis (negate x)
glScalef(-1, -1, 1 );   // Reflect across both axes
```

8.5 Example — Worked Problem (Reflection)

Reflect the triangle with vertices A(1, 4), B(3, 1), and C(5, 4) across the x-axis.

Solution (x' = x, y' = -y):

```
A(1,  4)  →  A'( 1, -4)
B(3,  1)  →  B'( 3, -1)
C(5,  4)  →  C'( 5, -4)
```

8.6 Lab Exercise — Reflection Code

The following program draws a triangle and reflects it across the x-axis using glScalef with a negative y factor:

```
#include<GL/freeglut.h>
#include<iostream>
using namespace std;

void display() {
    glClearColor(0, 0, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-10, 10, -10, 10);

    glScalef(1, -1, 1);   // Reflect across x-axis

    glBegin(GL_TRIANGLES);
        glVertex2f(0, 2);
        glVertex2f(-2, -2);
        glVertex2f(2, -2);
    glEnd();
    glFlush();
}

int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Reflection");
    glutDisplayFunc(display);
    glutMainLoop();
}
```

8.7 Definition (Shearing)

Shearing (skewing) displaces each point in proportion to its distance from a reference axis. An x-shear shifts points horizontally by an amount proportional to their y-coordinate; a y-shear shifts them vertically in proportion to their x-coordinate. The result is a slanted version of the original shape.

8.8 Equations (Shearing)

X-direction shear (parameter Shx):

```
x' = x + Shx * y
y' = y
```

Y-direction shear (parameter Shy):

```
x' = x
y' = y + Shy * x
```

8.9 Shearing Matrix (2D)

X-shear matrix in homogeneous coordinates:

2D X-Shear Matrix:

1	Shx	0
0	1	0
0	0	1

Y-shear matrix:

2D Y-Shear Matrix:

1	0	0
Shy	1	0
0	0	1

> **OpenGL has no built-in shear function. Shearing is applied by loading a custom 4x4 matrix using glLoadMatrixf() on the GL_MODELVIEW matrix mode.**

8.10 Example — Worked Problem (Shearing)

Apply an x-shear with Shx = 2 to the triangle A(0, 0), B(1, 0), C(1, 1).

Solution (x' = x + 2y, y' = y):

```
A(0, 0)  →  x' = 0 + 2*0 = 0,   y' = 0   →  A'(0, 0)
B(1, 0)  →  x' = 1 + 2*0 = 1,   y' = 0   →  B'(1, 0)
C(1, 1)  →  x' = 1 + 2*1 = 3,   y' = 1   →  C'(3, 1)
```

8.11 Lab Exercise — Shearing Code

The following program applies an x-shear of Shx = 2 to a triangle using a custom matrix loaded via glLoadMatrixf:

```
#include<GL/freeglut.h>
#include<iostream>
using namespace std;
```

```
void display() {
    glClearColor(0, 0, 1, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-10, 10, -10, 10);

    // X-shear matrix: Shx = 2
    // OpenGL stores matrices column-major
    float shearMatrix[16] = {
        1.0f, 0.0f, 0.0f, 0.0f,   // col 0
        2.0f, 1.0f, 0.0f, 0.0f,   // col 1: Shx in row 0
        0.0f, 0.0f, 1.0f, 0.0f,   // col 2
        0.0f, 0.0f, 0.0f, 1.0f    // col 3
    };
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(shearMatrix);

    glBegin(GL_TRIANGLES);
        glVertex2f(0, 2);
        glVertex2f(-2, -2);
        glVertex2f(2, -2);
    glEnd();
    glFlush();
}

int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(600, 600);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Shearing");
    glutDisplayFunc(display);
    glutMainLoop();
}
```

— End of Section 8 —

Lab Manual

Computer Graphics Fundamentals

Section 9: Transformations

Prepared by:

T.A. Manar Abbas

T.A. Somia Ahmed

Beni-Suef University

College of Computers and AI

Department of Computer Science

FCAI
Faculty of Computers & Artificial Intelligence
COMPUTER SCIENCE DEP.
جامعة بنى سويف
BENI-SUEF UNIVERSITY

9. SOLVED PROBLEMS

The following problems provide step-by-step solutions covering all five transformation types: Translation, Scaling, Rotation, Reflection, and Shearing.

9.1 Problem 1 — Translation

Translate the rectangle with vertices A(2, 3), B(6, 3), C(6, 7), D(2, 7) by Tx = -3 and Ty = 4.

Solution (x' = x + Tx, y' = y + Ty):

```
A(2, 3)  →  x' = 2 + (-3) = -1,   y' = 3 + 4 = 7    →  A'(-1, 7)
B(6, 3)  →  x' = 6 + (-3) =  3,   y' = 3 + 4 = 7    →  B'( 3, 7)
C(6, 7)  →  x' = 6 + (-3) =  3,   y' = 7 + 4 = 11   →  C'( 3, 11)
D(2, 7)  →  x' = 2 + (-3) = -1,   y' = 7 + 4 = 11   →  D'(-1, 11)
```

Note: A negative Tx moves the object left; a positive Ty moves it upward.

9.2 Problem 2 — Scaling

Scale the triangle A(2, 2), B(4, 2), C(3, 5) with Sx = 3 and Sy = 0.5.

Solution (x' = x * Sx, y' = y * Sy):

```
A(2, 2)  →  x' = 2 * 3 = 6,   y' = 2 * 0.5 = 1.0   →  A'(6, 1.0)
B(4, 2)  →  x' = 4 * 3 = 12,  y' = 2 * 0.5 = 1.0   →  B'(12, 1.0)
C(3, 5)  →  x' = 3 * 3 = 9,   y' = 5 * 0.5 = 2.5   →  C'(9, 2.5)
```

Note: Sx > 1 enlarges the object horizontally; Sy = 0.5 compresses it to half its height.

9.3 Problem 3 — Rotation

Rotate the line endpoints P(3, 0) and Q(0, 4) anticlockwise by 90 degrees about the origin.

Solution ($\theta = 90°$, $\cos 90° = 0$, $\sin 90° = 1$):

Equations: $x' = x \cdot \cos\theta - y \cdot \sin\theta$, $y' = x \cdot \sin\theta + y \cdot \cos\theta$

```
P(3, 0)  →  x' = 3*0 - 0*1 = 0,    y' = 3*1 + 0*0 = 3    → P'(0,  3)
Q(0, 4)  →  x' = 0*0 - 4*1 = -4,   y' = 0*1 + 4*0 = 0    → Q'(-4, 0)
```

> **Note: A 90° CCW rotation maps (x, y) → (-y, x). Verify: P(3,0) → P'(0,3) ✓**

9.4 Problem 4 — Reflection

Reflect the quadrilateral A(1, 2), B(4, 2), C(4, 5), D(1, 5) across the y-axis.

Solution (x' = -x, y' = y):

```
A(1, 2)  →  A'(-1, 2)
B(4, 2)  →  B'(-4, 2)
C(4, 5)  →  C'(-4, 5)
D(1, 5)  →  D'(-1, 5)
```

> **Note: Reflection across the y-axis is equivalent to glScalef(-1, 1, 1) in OpenGL.**

9.5 Problem 5 — Shearing

Apply a y-direction shear with Shy = 1.5 to the triangle A(0, 0), B(2, 0), C(2, 3).

Solution (x' = x, y' = y + Shy * x = y + 1.5x):

```
A(0, 0)  →  x' = 0,   y' = 0 + 1.5*0 = 0.0   →  A'(0,   0.0)
B(2, 0)  →  x' = 2,   y' = 0 + 1.5*2 = 3.0   →  B'(2,   3.0)
C(2, 3)  →  x' = 2,   y' = 3 + 1.5*2 = 6.0   →  C'(2,   6.0)
```

> **Note: A y-shear does not change x-coordinates. The shape is slanted upward as x increases.**

9.6 Problem 6 — Composite Transformation

Apply the following sequence to point P(2, 3): first scale by Sx=2, Sy=2, then translate by Tx=1, Ty=-1.

Step 1 — Scaling (x' = x*Sx, y' = y*Sy):

```
P(2, 3)  →  x' = 2*2 = 4,   y' = 3*2 = 6   →  P1(4, 6)
```

Step 2 — Translation (x'' = x' + Tx, y'' = y' + Ty):

```
P1(4, 6)  →  x'' = 4+1 = 5,   y'' = 6+(-1) = 5   →  P'(5, 5)
```

> **Important: In OpenGL, transformations are written in reverse order in code. To scale then translate, write glTranslatef first, then glScalef — the last written is first applied.**

8. SUMMARY

This section covered the three fundamental geometric transformations in computer graphics and their OpenGL implementations:

Summary of OpenGL Transformation Functions

Transformation	OpenGL Function	Effect
Translation	glTranslatef(x, y, z)	Moves object by (x, y, z)
Scaling	glScalef(x, y, z)	Resizes object by factors (x, y, z)
Rotation	glRotatef(a, x, y, z)	Rotates object a° around axis (x,y,z)
Push Matrix	glPushMatrix()	Saves current transformation state
Pop Matrix	glPopMatrix()	Restores last saved transformation state
Timer	glutTimerFunc(ms, fn, v)	Schedules animation callback after ms ms
Redisplay	glutPostRedisplay()	Marks window for redraw on next loop

Understanding and combining these transformations is essential for building any non-trivial graphics application, from simple shape manipulation to complex animated 3D scenes.

— End of Section 8 —