



Daffodil
International
University

Lab Task

Course Code: CSE 422

Course Title: Computer Graphics Lab

Assignment Title: Draw a rocket using Knowledge of OpenGL and some basic built in function.

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 13th October, 2025

Title

Draw a rocket using fundamental knowledge of OpenGL and some basic built in functions.

Introduction

In this lab task, I used OpenGL with GLUT to draw a simple rocket shape using basic geometric primitives. The program demonstrates how to create 2D graphics by combining triangles and polygons with different colors. Through this task, we learned how to initialize an OpenGL window, set background and object colors, and use functions like glBegin(), glVertex3f(), and glFlush() to display objects on the screen. This experiment helps in understanding the fundamentals of computer graphics and how shapes are formed using coordinates in OpenGL.

Contents

In this lab task :

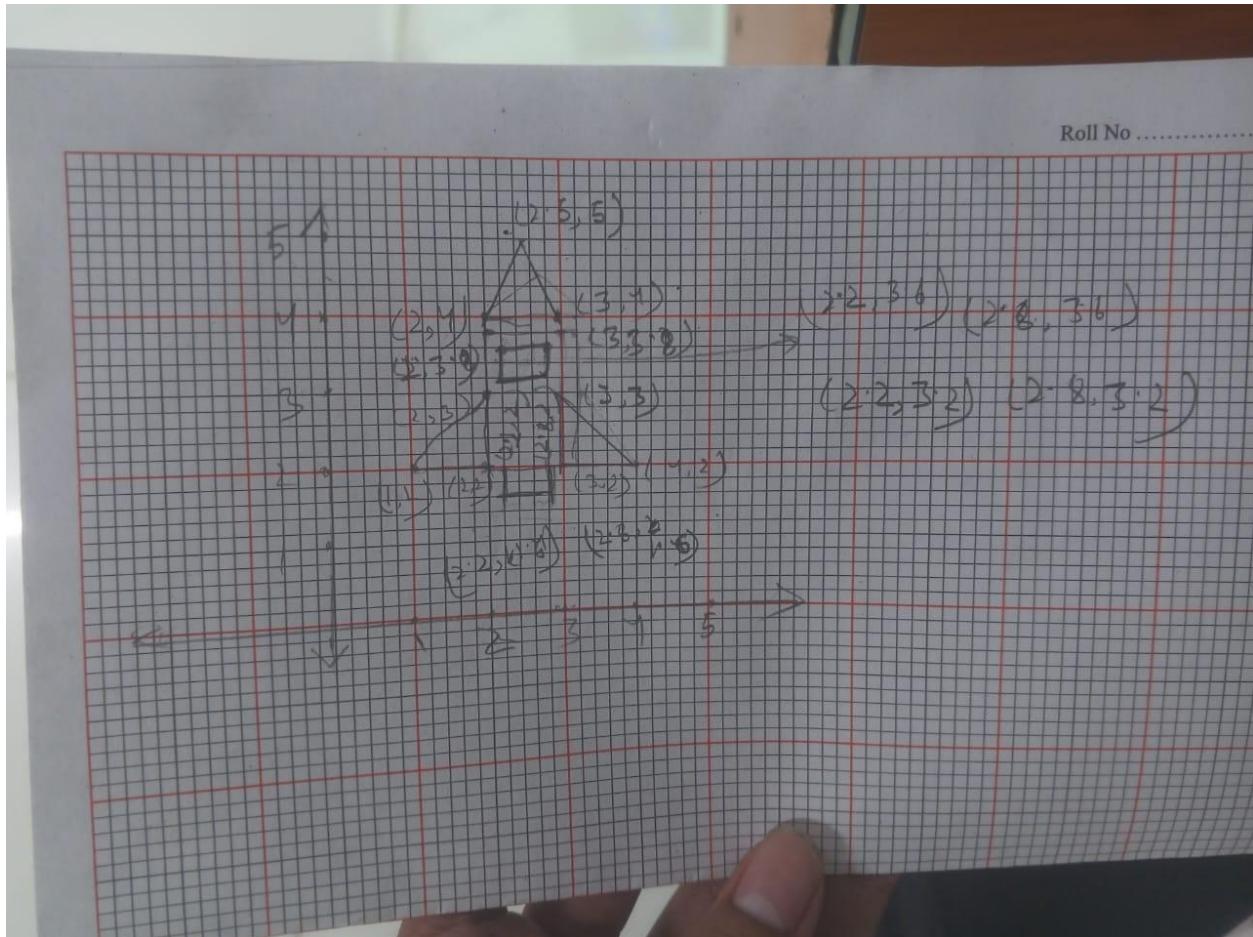
1. Functions used

- ‘glClear()’ – clears the screen.
- ‘glColor3f()’ – sets color.
- ‘glBegin()’/ ‘glEnd()’ – start and end shape drawing.
- ‘glVertex3f()’ – defines shape corners.
- ‘glFlush()’ – displays the drawing.

2. Shapes used:

- Triangles – rocket top and two wings.
- Polygons (rectangles)– rocket body, window, and blast.

Graph



Code

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

void display(void)
{
    /* clear all pixels */
    glClear(GL_COLOR_BUFFER_BIT);
    /* draw white polygon (rectangle) with corners at
     * (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
```

```
*/  
//upper triangle  
glColor3f(0, 0, 1.0);  
glBegin(GL_TRIANGLES);  
 glVertex3f(.25, .5, 0);  
 glVertex3f(.2, .4, 0);  
 glVertex3f(.3, .4, 0);  
//glVertex3f(0.25, 0.75, 0.0);  
glEnd();
```

```
// main body
```

```
glBegin(GL_POLYGON);  
 glVertex3f(.2, .39, 0);  
 glVertex3f(.3, .39, 0);  
 glVertex3f(.3, .2, 0);  
 glVertex3f(.2, .2, 0);  
glEnd();
```

```
// wing 1
```

```
glColor3f(0.0, 1.0, 0.0);  
  
glBegin(GL_TRIANGLES);  
 glVertex3f(.2, .3, 0);  
 glVertex3f(.2, .2, 0);  
 glVertex3f(.1, .2, 0);  
//glVertex3f(0.25, 0.75, 0.0);
```

```
glEnd();

// wing 2

glBegin(GL_TRIANGLES);

glVertex3f (.3, .3, 0);
glVertex3f (.4, .2, 0);
glVertex3f (.3, .2, 0);

//glVertex3f (0.25, 0.75, 0.0);

glEnd();

// blast

glColor3f(1.0, 0.4, 0.7);

glBegin(GL_POLYGON);

glVertex3f (.22, .2, 0);
glVertex3f (.28, .2, 0);
glVertex3f (.28, .16, 0);
glVertex3f (.22, .16, 0);

glEnd();

//window

glColor3f(0.0, 1.0, 0.0);

glBegin(GL_POLYGON);

glVertex3f (.22, .36, 0);
glVertex3f (.28, .36, 0);
glVertex3f (.28, .32, 0);
glVertex3f (.22, .32, 0);

glEnd();
```

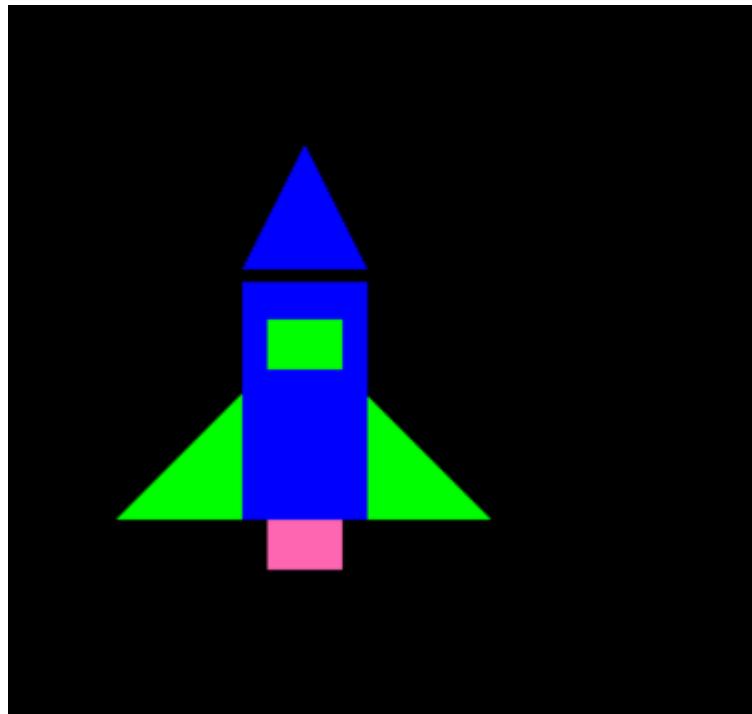
```
/* don't wait!
 * start processing buffered OpenGL routines
 */
glFlush ();
}

void init (void)
{
/* select clearing (background) color */
glClearColor (0.0, 0.0, 0.0, 0.0);
/* initialize viewing values */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

/*
 * Declare initial window size, position, and display mode
 * (single buffer and RGBA). Open window with "hello"
 * in its title bar. Call initialization routines.
 * Register callback function to display graphics.
 * Enter main loop and process events.
*/
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
```

```
glutCreateWindow ("hello");
init ();
glutDisplayFunc(display);
glutMainLoop();
return 0; /* ISO C requires main to return int.*/
}
```

Output



Discussion

In this lab task, I successfully created a rocket using basic OpenGL functions. The rocket was drawn by combining simple shapes such as triangles for the nose and wings, and polygons for the body, window, and burst. Each part was given a different color using `glColor3f()` to make the figure more visually clear. The program used `glBegin()` and `glEnd()` to define shapes, while `glVertex3f()` specified their coordinates. Overall, this lab demonstrated how OpenGL can be used to design graphical objects using coordinate geometry & color control.



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 02

Title: Drawing a robot using different OpenGL functions.

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 20th October, 2025

Title

Drawing a robot using different OpenGL functions.

Introduction

In this lab task, I used OpenGL with GLUT to draw a Robot shape using basic geometric primitives. The program demonstrates how to create 2D graphics by combining triangles and polygons with different colors. Through this task, we learned how to initialize an OpenGL window, set background and object colors, and use functions like glBegin(), glVertex3f(), and glFlush() to display objects on the screen. This experiment helps in understanding the fundamentals of computer graphics and how shapes are formed using coordinates in OpenGL.

Contents

In this lab task :

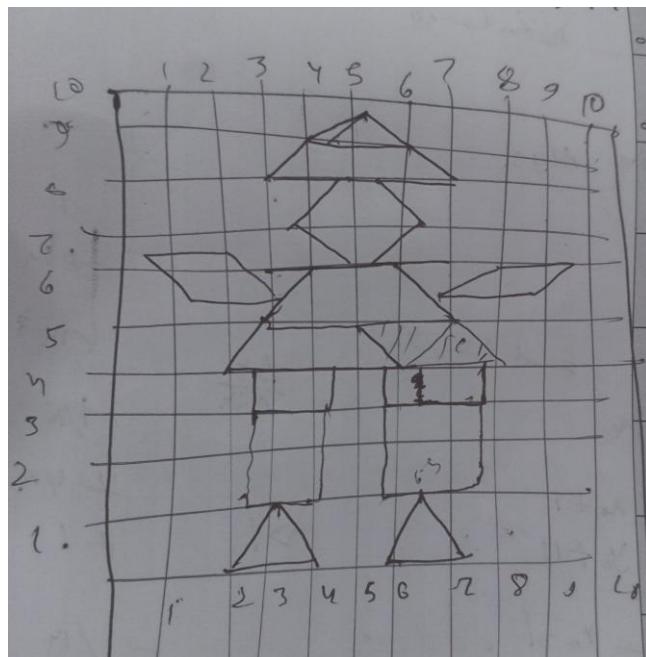
1. Functions used

- ‘glClear()’ – clears the screen.
- ‘ glColor3f()’ – sets color.
- ‘glBegin()’/ ‘glEnd()’ – start and end shape drawing.
- ‘glVertex3f()’ – defines shape corners.
- ‘glFlush()’ – displays the drawing.

2. Shapes used:

- Triangles – rocket top and two wings.
- Polygons (rectangles) – rocket body, window, and blast.

Graph



Code

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

void display(void)
{
/* clear all pixels */
glClear (GL_COLOR_BUFFER_BIT);

/* draw white polygon (rectangle) with corners at
* (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)

*/
//left wing
glColor3f (0, 1.0, 1.0);
glBegin(GL_POLYGON);
 glVertex3f (.65, .55, 0);
 glVertex3f (.75, .65, 0);
 glVertex3f (.9, .65, 0);
 glVertex3f (.8, .55, 0);
glEnd();

//right wing
glColor3f (0, 1.0, 1.0);
glBegin(GL_POLYGON);
 glVertex3f (.15, .55, 0);
 glVertex3f (.05, .65, 0);
 glVertex3f (.20, .65, 0);
 glVertex3f (.3, .55, 0);
```

```
glEnd();

//hat 1
glColor3f(0, 0, 1.0);
glBegin(GL_TRIANGLES);
glVertex3f(.35, .9, 0);
glVertex3f(.475, 1.0, 0);
glVertex3f(.6, .9, 0);
glEnd();

//hat 2
glColor3f(0, 1.0, 1.0);
glBegin(GL_POLYGON);
glVertex3f(.25, .8, 0);
glVertex3f(.35, .9, 0);
glVertex3f(.6, .9, 0);
glVertex3f(.7, .8, 0);
glEnd();

//head
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_POLYGON);
glVertex3f(.38, .6, 0);
glVertex3f(.28, .7, 0);
glVertex3f(.38, .8, 0);
glVertex3f(.58, .8, 0);
```

```
glVertex3f (.68, .7, 0);
glVertex3f (.58, .6, 0);
glEnd();

// belly
glColor3f (0, 1.0, 1.0);
glBegin(GL_POLYGON);
glVertex3f (.25, .5, 0);
glVertex3f (.35, .6, 0);
glVertex3f (.60, .6, 0);
glVertex3f (.7, .5, 0);
glEnd();

// left thigh
glColor3f (0, 0, 1.0);
glBegin(GL_POLYGON);
glVertex3f (.15, .4, 0);
glVertex3f (.25, .5, 0);
glVertex3f (.5, .5, 0);
glVertex3f (.6, .4, 0);
glEnd();

//right thigh 1
glColor3f(0.0, 1.0, 0.0);
glBegin(GL_TRIANGLES);
glVertex3f (.5, .5, 0);
glVertex3f (.7, .5, 0);
glVertex3f (.6, .4, 0);
glEnd();
```

```
glColor3f(0.0, 1.0, 3.0);
glBegin(GL_TRIANGLES);
glVertex3f (.6, .4, 0);
glVertex3f (.7, .5, 0);
glVertex3f (.8, .4, 0);
glEnd();
```

```
// left leg
glColor3f(1.0, 0.4, 0.7);
glBegin(GL_POLYGON);
glVertex3f (.2, .1, 0);
glVertex3f (.2, .2, 0);
glVertex3f (.4, .2, 0);
glVertex3f (.4, .1, 0);
glEnd();
```

```
glColor3f(1.0, 1.4, 0.7);
glBegin(GL_POLYGON);
glVertex3f (.2, .2, 0);
glVertex3f (.2, .3, 0);
glVertex3f (.4, .3, 0);
glVertex3f (.4, .2, 0);
glEnd();
```

```
glColor3f(1.0, 0.4, 1.7);
glBegin(GL_POLYGON);
```

```
glVertex3f (.2, .3, 0);
glVertex3f (.2, .4, 0);
glVertex3f (.4, .4, 0);
glVertex3f (.4, .3, 0);
glEnd();
//right leg
```

```
glColor3f(1.0, 0.4, 0.7);
glBegin(GL_POLYGON);
glVertex3f (.55, .1, 0);
glVertex3f (.55, .2, 0);
glVertex3f (.75, .2, 0);
glVertex3f (.75, .1, 0);
glEnd();
```

```
glColor3f(1.0, 1.4, 0.7);
glBegin(GL_POLYGON);
glVertex3f (.55, .2, 0);
glVertex3f (.55, .3, 0);
glVertex3f (.75, .3, 0);
glVertex3f (.75, .2, 0);
glEnd();
```

```
glColor3f(1.0, 0.4, 1.7);
glBegin(GL_POLYGON);
glVertex3f (.55, .3, 0);
glVertex3f (.55, .4, 0);
```

```
glVertex3f (.75, .4, 0);
glVertex3f (.75, .3, 0);
glEnd();
//left foot

glColor3f(0.0, 1.0, 0.0);

glBegin(GL_TRIANGLES);
glVertex3f (.2, 0, 0);
glVertex3f (.3, .1, 0);
glVertex3f (.4, .0, 0);
glEnd();
// right foot

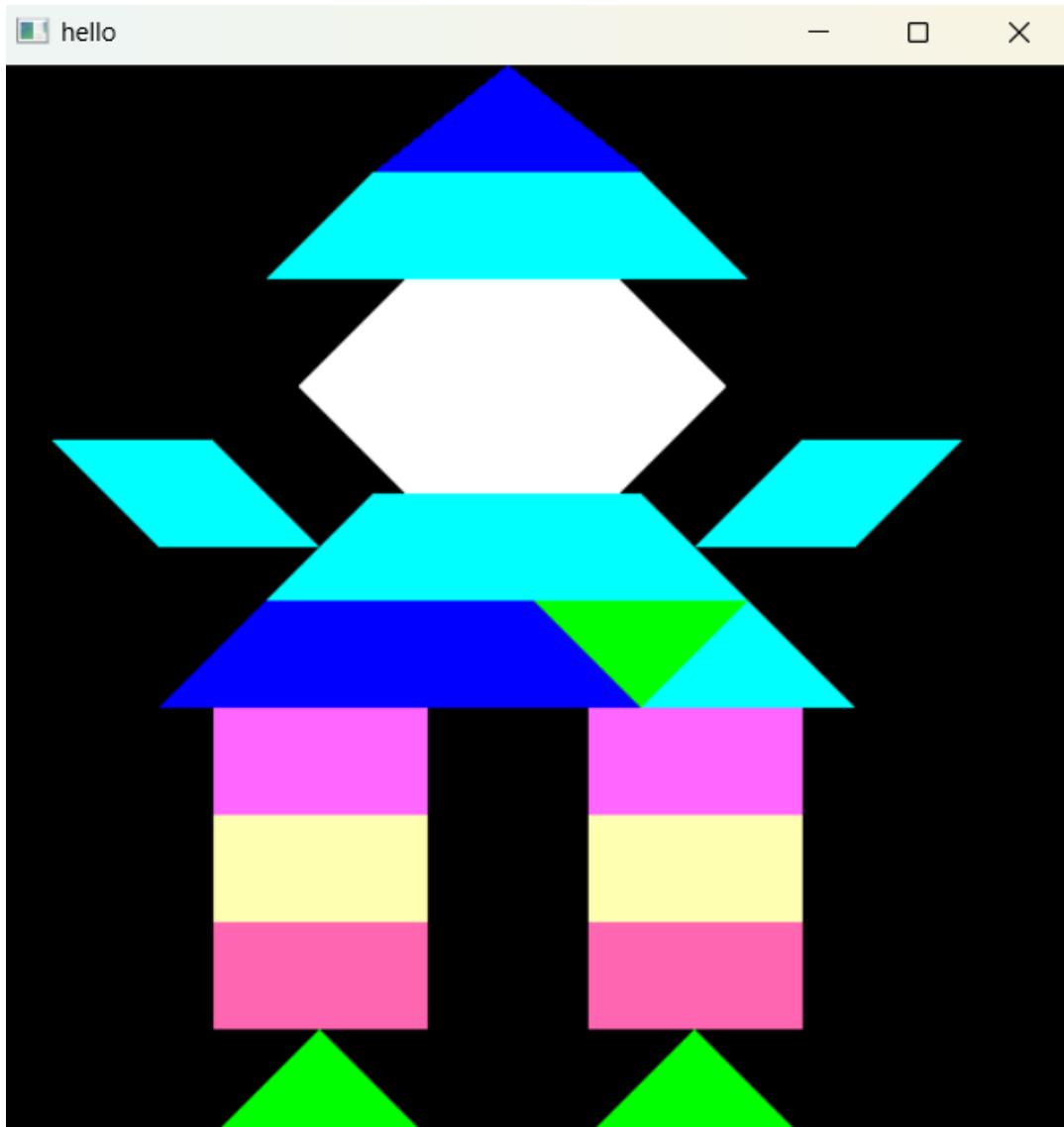
glColor3f(0.0, 1.0, 0.0);
glBegin(GL_TRIANGLES);
glVertex3f (.55, 0, 0);
glVertex3f (.65, .1, 0);
glVertex3f (.75, 0, 0);
glEnd();

/* don't wait!
 * start processing buffered OpenGL routines
 */
glFlush ();
}

void init (void)
```

```
{  
/* select clearing (background) color */  
glClearColor (0.0, 0.0, 0.0, 0.0);  
/* initialize viewing values */  
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);  
}  
/*  
* Declare initial window size, position, and display mode  
* (single buffer and RGBA). Open window with "hello"  
* in its title bar. Call initialization routines.  
* Register callback function to display graphics.  
* Enter main loop and process events.  
*/  
  
int main(int argc, char** argv)  
{  
    glutInit(&argc, argv);  
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);  
    glutInitWindowSize (500, 500);  
    glutInitWindowPosition (100, 100);  
    glutCreateWindow ("hello");  
    init ();  
    glutDisplayFunc(display);  
    glutMainLoop();  
    return 0; /* ISO C requires main to return int. */  
}
```

Output



Discussion

In this lab task, I successfully created a robot using basic OpenGL functions. The rocket was drawn by combining simple shapes such as triangles for the nose and wings, and polygons for the body, window, and burst. Each part was given a different color using `glColor3f()` to make the figure more visually clear. The program used `glBegin()` and `glEnd()` to define shapes, while `glVertex3f()` specified their coordinates. Overall, this lab demonstrated how OpenGL can be used to design graphical objects using coordinating geometry & color control.



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 03

Title: Implementation of DDA line drawing algorithm.

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 28th November, 2025

Title

Implementation of DDA line drawing algorithm.

Introduction

In this lab task, I implemented the DDA (Digital Differential Analyzer) Line Drawing Algorithm using OpenGL and GLUT. The objective of the experiment was to draw a straight line between two user-defined points by incrementally calculating intermediate positions. This program demonstrates how computer graphics systems generate lines using floating-point arithmetic and coordinate-based plotting. Through this task, we learned how to initialize an OpenGL window, take input from the user, apply the DDA algorithm, and use OpenGL functions to visualize the computed line. This experiment strengthens the understanding of line rendering concepts and how digital devices convert mathematical line equations into pixels on the screen.

Contents

In this lab task :

1. Functions used
 - glClear() – clears the screen
 - glColor3f() – sets the drawing color
 - glBegin() / glEnd() – starts and ends drawing operations
 - glVertex2i() – plots individual points of the line
 - glFlush() – displays the rendered output
 - drawLineDDA() – implements the DDA algorithm to draw a line by calculating intermediate pixel coordinates.
2. Input Taken:
 - Starting point (x_0, y_0)
 - Ending point (x_1, y_1)

Code

```
#include <GL/gl.h>
#include <GL/glut.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int x0_user, y0_user, x1_user, y1_user;
```

```
void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    gluOrtho2D(0, 50, 0, 50);
    glPointSize(5.0);
}
```

```
void drawLineDDA(int x0, int y0, int x1, int y1)
{
    float dx = x1 - x0;
    float dy = y1 - y0;
    float steps = fabsf(dx) > fabsf(dy) ? fabsf(dx) : fabsf(dy);
    float xInc = dx / steps;
    float yInc = dy / steps;
    float x = x0;
    float y = y0;
```

```
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_POINTS);
    for (int k = 0; k <= steps; k++)
    {
        glVertex2i((int)roundf(x), (int)roundf(y));
        x += xInc;
        y += yInc;
    }
    glEnd();
```

```
}
```

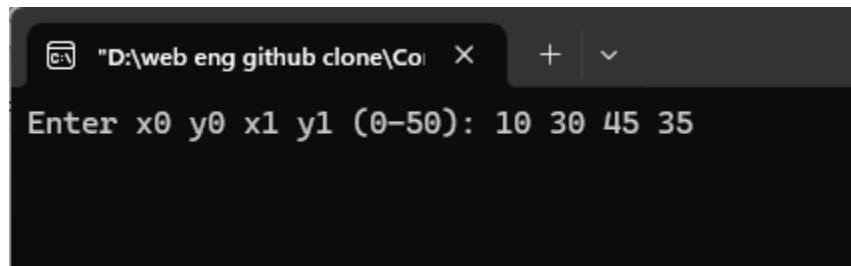
```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawLineDDA(x0_user, y0_user, x1_user, y1_user);
    glFlush();
}
```

```
int main(int argc, char** argv)
{
    printf("Enter x0 y0 x1 y1 (0-50): ");
    scanf("%d %d %d %d", &x0_user, &y0_user, &x1_user, &y1_user);

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(300, 300);
    glutCreateWindow("DDA Line Drawing");

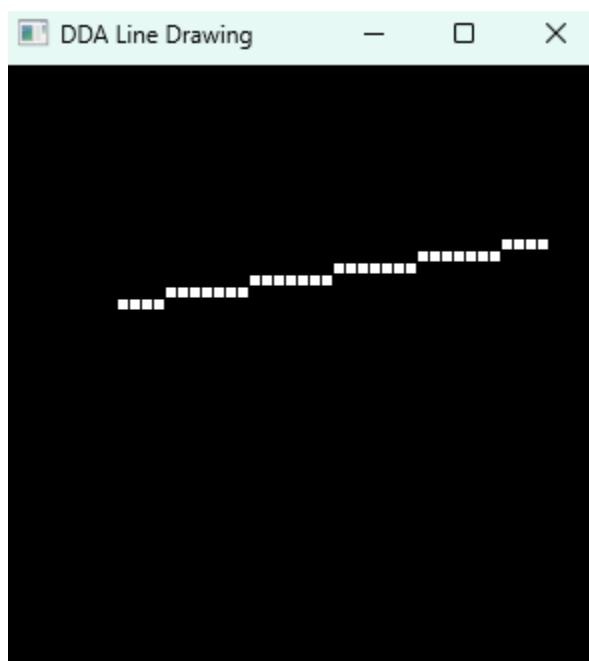
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Input



```
"D:\web eng\github\clone\Co" + v
Enter x0 y0 x1 y1 (0-50): 10 30 45 35
```

Output



Discussion

In this lab task, I successfully implemented the DDA Line Drawing Algorithm using OpenGL. The program reads two endpoints from the user and calculates intermediate points along the line using incremental floating-point steps in both x and y directions. Each calculated point is plotted using `glVertex2i()`, and `glColor3f()` is used to set a clear visual color for the line. The implementation demonstrates how DDA performs step-by-step pixel plotting based on the slope of the line without relying on complex arithmetic. Overall, this task enhanced my understanding of how lines are generated in computer graphics and how OpenGL can be used to visualize algorithmic output efficiently.



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 04

Title: Implementation of Bresenham's Line Drawing algorithm.

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 28th November, 2025

Title

4. Implementation of Bresenham's Line Drawing algorithm.

Introduction

In this lab task, I implemented the Bresenham Line Drawing Algorithm using OpenGL and GLUT. The purpose of this experiment was to draw a line between two user-defined points using only integer arithmetic, making the process faster and more efficient compared to algorithms that rely on floating-point calculations. Bresenham's algorithm determines the next pixel position based on a decision parameter, allowing the line to be rendered with high accuracy and minimal computational cost. Through this task, I learned how to initialize an OpenGL environment, take input from the user, execute the Bresenham algorithm, and display the resulting line on the screen. This lab enhanced my understanding of raster graphics and how precise lines can be generated using incremental decision-based logic.

Contents

In this lab task :

1. Functions used
 - ‘glClear() – clears the display
 - glColor3f() – sets the drawing color
 - glBegin() / glEnd() – starts and ends point plotting
 - glVertex2i() – plots each selected pixel on the line
 - glFlush() – outputs the rendered graphics
 - drawLineBresenham() – applies Bresenham's decision formula to determine the next pixel for the line.
2. Inputs taken:
 - Starting point (x_0, y_0)
 - Ending point (x_1, y_1)

Code

```
#include <GL/gl.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

int x0_user, y0_user, x1_user, y1_user;
```

```
void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    gluOrtho2D(0, 50, 0, 50);
    glPointSize(5.0);

}

void drawLineBresenham(int x0, int y0, int x1, int y1)
{
    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);
    int sx = x0 < x1 ? 1 : -1;
    int sy = y0 < y1 ? 1 : -1;
    int err = dx - dy;

    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_POINTS);
    while (1)
    {
        glVertex2i(x0, y0);
        if (x0 == x1 && y0 == y1) break;
        int e2 = 2 * err;
        if (e2 > -dy)
        {
            err -= dy;
            x0 += sx;
        }
    }
}
```

```

    if (e2 < dx)
    {
        err += dx;
        y0 += sy;
    }
}

glEnd();

}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawLineBresenham(x0_user, y0_user, x1_user, y1_user);
    glFlush();
}

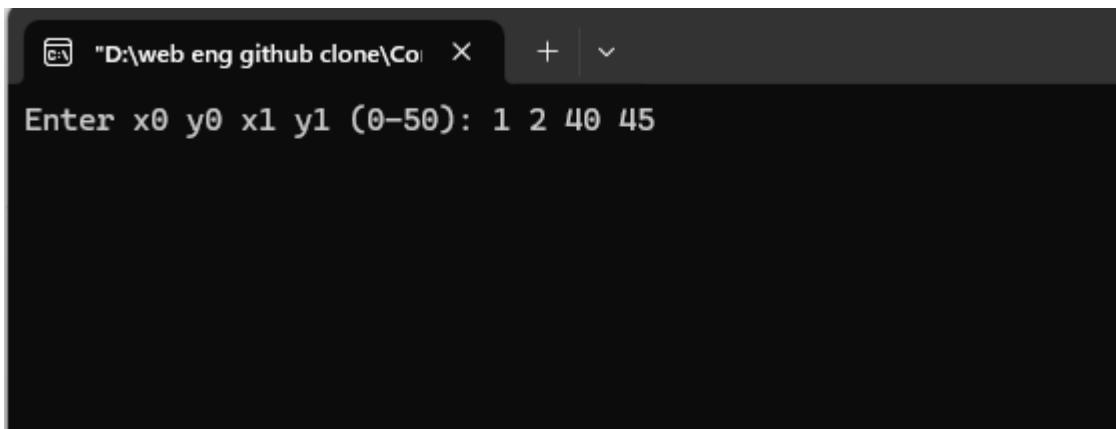
int main(int argc, char** argv)
{
    printf("Enter x0 y0 x1 y1 (0-50): ");
    scanf("%d %d %d %d", &x0_user, &y0_user, &x1_user, &y1_user);

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(300, 300);
    glutCreateWindow("Bresenham Line Drawing");
}

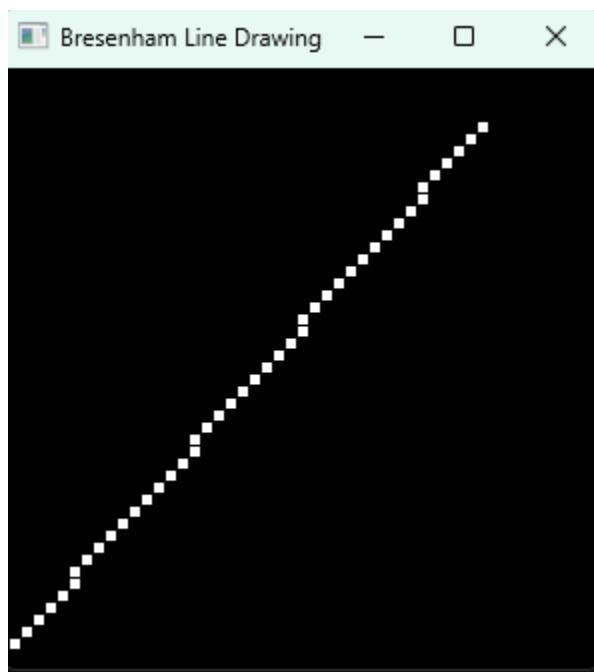
```

```
init();  
glutDisplayFunc(display);  
glutMainLoop();  
return 0;  
}
```

Input



Output



Discussion

In this lab task, I successfully implemented the Bresenham Line Drawing Algorithm using OpenGL. Unlike the DDA algorithm, Bresenham uses integer calculations and a decision parameter to determine whether the next plotted pixel should move horizontally or diagonally. This makes the algorithm faster and avoids floating-point rounding errors. Each selected pixel along the line was plotted using `glVertex2i()`, while `glColor3f()` was used to give the line a clear visual appearance. The program demonstrated how Bresenham's approach provides accurate and efficient rasterized line drawing on a pixel grid. Through this lab, I gained a deeper understanding of how optimized algorithms can enhance rendering performance while maintaining line precision in computer graphics.



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 05

Title: Implementation of Mid-Point Circle Drawing algorithm
and Implement it and draw half moon using it.

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 14th December, 2025

Title

Implementation of Mid-Point Circle Drawing algorithm and Implement it and draw half moon using it.

Introduction

In this project, an OpenGL-based graphical program has been developed to draw a Moon shape. The concept is implemented using two overlapping circles, where one circle is drawn in warm white and another circle matching the night sky background is drawn slightly shifted to create the crescent shape. The circle is generated using trigonometric calculations inside OpenGL's GL_TRIANGLE_FAN primitive, which is similar in spirit to the Mid-Point Circle Drawing approach for approximating circular shapes. GLUT is used for window management, rendering, and handling display events.

Contents

The following functions and shapes have been used in this project:

1. init()
 - Sets the background color of the window to simulate a night sky.
 - glClearColor(0.05f, 0.05f, 0.2f, 1) is used to make the background dark blue.
2. draw()
 - This function performs all rendering tasks.
 - The first circle is drawn in warm white (1.0f, 1.0f, 0.9f), representing the main moon body.
 - A second circle matching the background color is drawn slightly offset, creating a shadow effect to form the crescent moon shape.
 - glutSwapBuffers() is used for double buffering.
3. reshape()
 - Adjust the viewport and projection when the window is resized.
 - Uses gluOrtho2D() to define a fixed 2D coordinate system from -300 to 300 on both axes.
4. circle()
 - Draws a circle using GL_TRIANGLE_FAN with 100 segments.
 - Takes radius (rx, ry) and center (cx, cy) as parameters
 - Uses trigonometric functions cos() and sin() to calculate each vertex of the circle.

Code

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

```
#include <cmath>
```

```
void init();
```

```
void draw();

void reshape(int w, int h);

void circle(GLfloat rx, GLfloat ry, GLfloat cx, GLfloat cy);

int main(int argc, char **argv)

{

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);

    glutInitWindowPosition(300, 100);

    glutInitWindowSize(600, 600);

    glutCreateWindow("Draw Moon");




    glutDisplayFunc(draw);

    glutReshapeFunc(reshape);

    init();



    glutMainLoop();



}

void init()

{

    glClearColor(0.05f, 0.05f, 0.2f, 1);



}

void draw()

{

    glClear(GL_COLOR_BUFFER_BIT);

    glLoadIdentity();



    // Draw white circle (main moon body)
```

```

glColor3f(1.0f, 1.0f, 0.9f);
circle(120, 120, -20, 50);

// Draw dark circle (shadow on moon to create crescent)
glColor3f(0.05f, 0.05f, 0.2f);
circle(120, 120, 50, 50);

glutSwapBuffers();
}

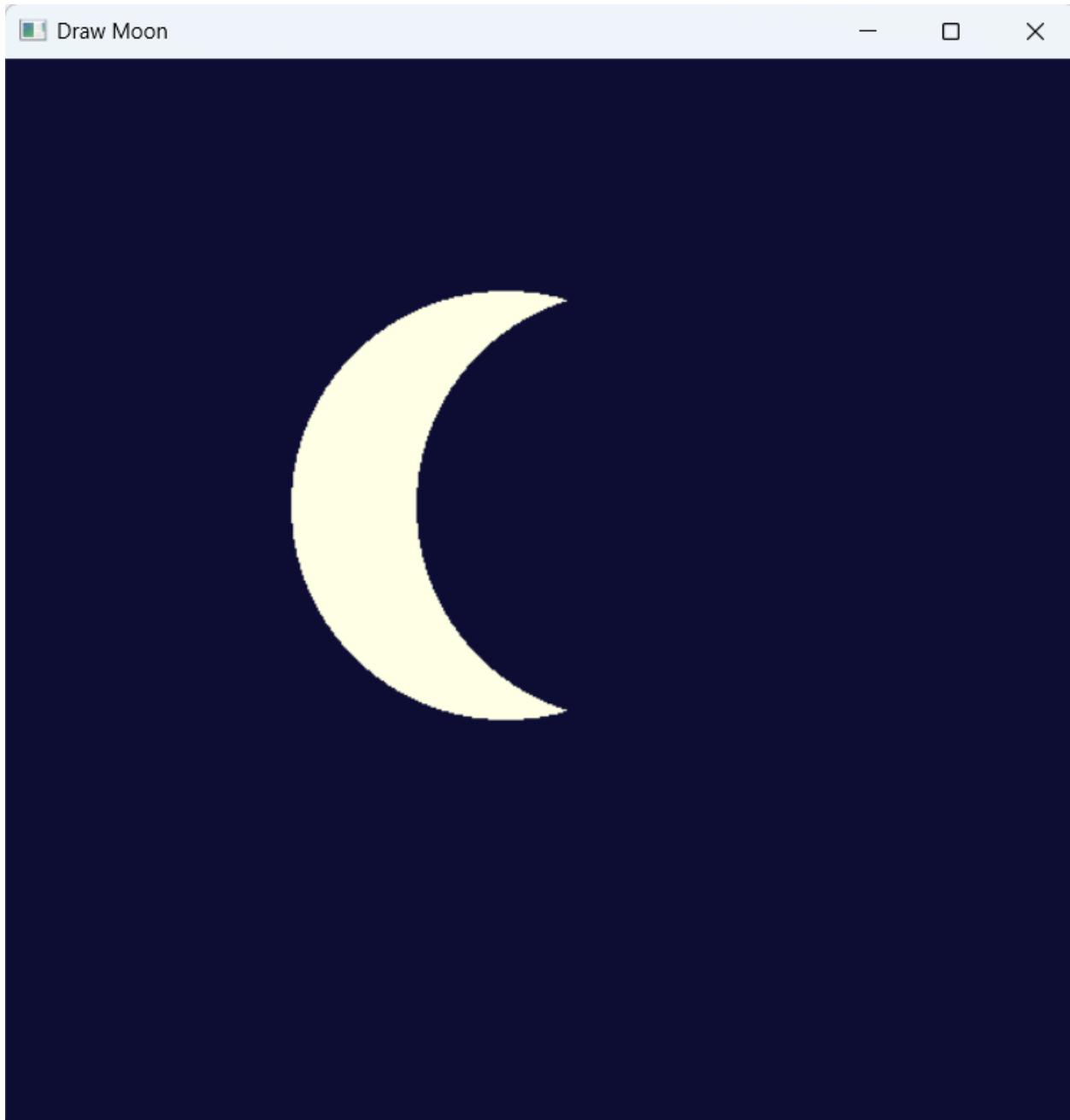
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei)w, (GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-300, 300, -300, 300);
    glMatrixMode(GL_MODELVIEW);
}

void circle(GLfloat rx, GLfloat ry, GLfloat cx, GLfloat cy)
{
    glBegin(GL_TRIANGLE_FAN);
    glVertex2f(cx, cy);
    for (int i = 0; i <= 100; i++)
    {
        float angle = 2 * M_PI * i / 100;
        float x = rx * cosf(angle);
        float y = ry * sinf(angle);
        glVertex2f((x + cx), (y + cy));
    }
}

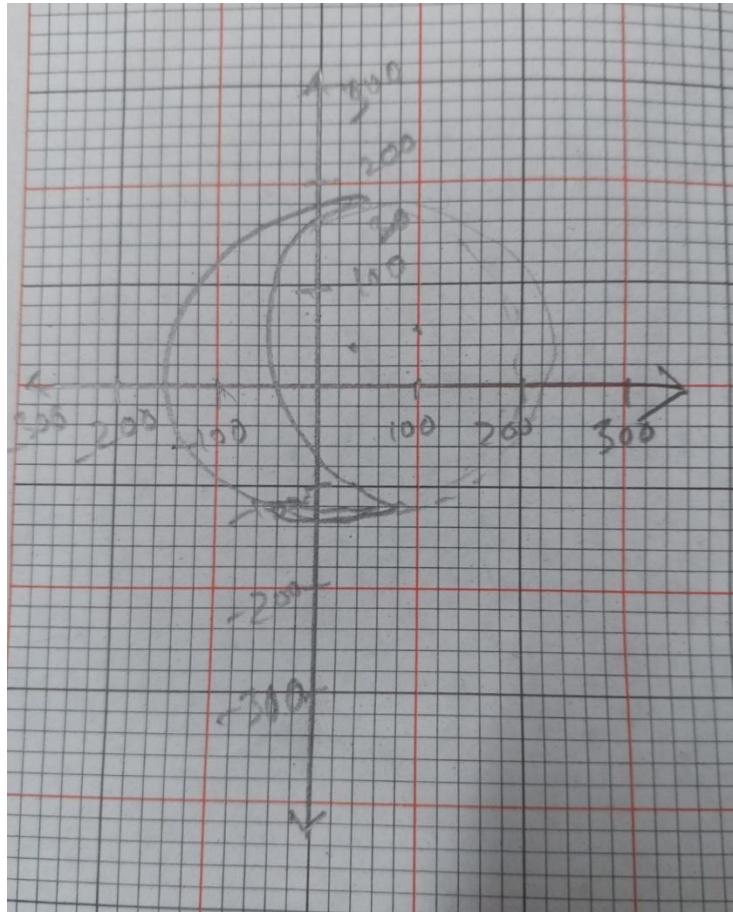
```

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

Output



Graph



Discussion

The main idea of this project is to create a crescent moon by overlapping two circles. First, a full warm white circle is drawn to represent the moon body. Then a circle matching the dark night sky background is drawn slightly offset, creating a shadow effect that hides part of the white circle and producing the realistic crescent moon shape.

Although the traditional Mid-Point Circle Drawing algorithm plots pixels step-by-step, in OpenGL we use trigonometric calculations and primitives to approximate the circle smoothly. The background color simulates a night sky, enhancing the visual appearance of the moon. This project demonstrates how simple 2D shapes can be combined to form more complex shapes. Furthermore, GLUT is used to manage window creation, display callbacks, and reshape events, making the program interactive and scalable.



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 06

Title: Drawing different object using Circle.

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 24th November, 2025

Title

Drawing different object using Circle.

Introduction

In this lab task, I used OpenGL with GLUT to draw a Scenery with a tree, sun, cloud and traffic light shape using basic geometric primitives. The program demonstrates how to create 2D graphics by combining Circles and polygons with different colors. Through this task, we learned how to initialize an OpenGL window, set background and object colors, and use functions like glBegin(), glVertex3f(), and glFlush() to display objects on the screen. This experiment helps in understanding the fundamentals of computer graphics and how shapes are formed using coordinates in OpenGL.

Contents

In this lab task :

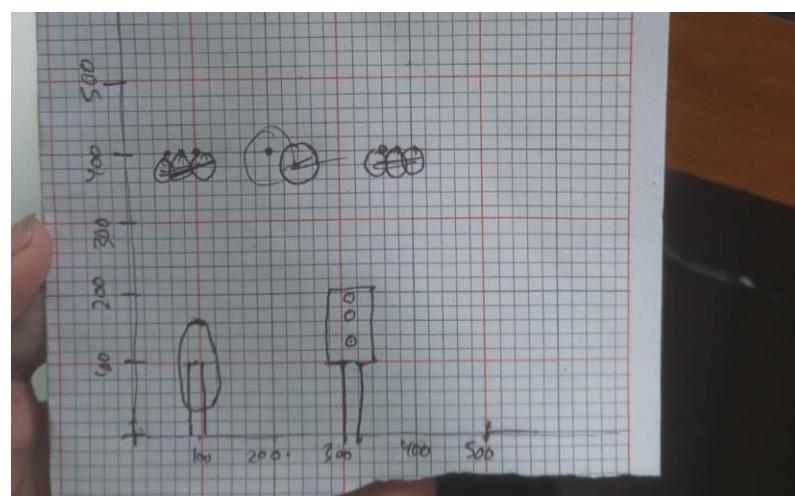
1. Functions used

- ‘glClear()’ – clears the screen.
- ‘ glColor3f()’ – sets color.
- ‘glBegin()’/ ‘glEnd()’ – start and end shape drawing.
- ‘glVertex3f()’ – defines shape corners.
- ‘glFlush()’ – displays the drawing.
- ‘drawCircle()’ – draws circle.

2. Shapes used:

- Circle – Sun, clouds, tree leafs, cloud.
-
- Polygons (rectangles)– tree body, traffic light stand & body.

Graph



Code

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

void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0); // Black background as per original values
    gluOrtho2D(0, 500, 0, 500);      // Set coordinate system
}

void drawCircle(int h, int k, int rx, int ry)
{
    glColor3f(1.0, 1.0, 0.0); // Yellow color (1.0, 1.0, 1.0 is White)
    glBegin(GL_POLYGON);
    for (int i = 0; i <= 360; i++) {
        // Corrected the line by replacing non-breaking spaces
        // The angle needs to be in radians for sin/cos
        float angle = 3.14159 * i / 180.0;
        glVertex2f(h + rx * cos(angle), k + ry * sin(angle));
    }
    glEnd();
}

void drawCircle1(int h, int k, int rx, int ry)
{
    glColor3f(1.0, 0.0, 0.0); // Yellow color (1.0, 1.0, 1.0 is White)
```

```

glBegin(GL_POLYGON);
for (int i = 0; i <= 360; i++) {
    // Corrected the line by replacing non-breaking spaces
    // The angle needs to be in radians for sin/cos
    float angle = 3.14159 * i / 180.0;
    glVertex2f(h + rx * cos(angle), k + ry * sin(angle));
}
glEnd();

}

void drawCircle3(int h, int k, int rx, int ry)
{
    glColor3f(0.0, 1.0, 0.0); // Yellow color (1.0, 1.0, 1.0 is White)
    glBegin(GL_POLYGON);
    for (int i = 0; i <= 360; i++) {
        // Corrected the line by replacing non-breaking spaces
        // The angle needs to be in radians for sin/cos
        float angle = 3.14159 * i / 180.0;
        glVertex2f(h + rx * cos(angle), k + ry * sin(angle));
    }
    glEnd();
}

void drawCirclew(int h, int k, int rx, int ry)
{
    glColor3f(1.0, 1.0, 1.0); // Yellow color (1.0, 1.0, 1.0 is White)
    glBegin(GL_POLYGON);

```

```
for (int i = 0; i <= 360; i++) {  
    // Corrected the line by replacing non-breaking spaces  
    // The angle needs to be in radians for sin/cos  
    float angle = 3.14159 * i / 180.0;  
    glVertex2f(h + rx * cos(angle), k + ry * sin(angle));  
}  
glEnd();  
}
```

```
// Function to draw a triangle  
void drawTriangle(void)  
{  
    glColor3f(1.0, 1.0, 0.0); // Yellow color  
    glBegin(GL_TRIANGLES);  
    glVertex2i(150, 150); // Bottom-left  
    glVertex2i(350, 150); // Bottom-right  
    glVertex2i(250, 300); // Top peak  
    glEnd();  
}
```

```
// Function to display both the circle and triangle  
void display(void)  
{  
    glClear(GL_COLOR_BUFFER_BIT);  
    // Clear screen (sets background to black)
```

```
drawCircle(250, 350, 50, 50); // Draw the circle at (250, 350)

//drawTriangle();           // Draw the triangle at predefined coordinates

//r cloud

drawCirclew(350, 350, 15, 15);
drawCirclew(365, 350, 15, 15);
drawCirclew(380, 350, 15, 15);

//l cloud

drawCirclew(120, 350, 15, 15);
drawCirclew(130, 350, 15, 15);
drawCirclew(140, 350, 15, 15);
drawCirclew(125, 340, 15, 15);
drawCirclew(135, 340, 15, 15);

// tree

drawCircle3(90, 100, 40,70);
glColor3f(1.0, 0.4, 1.7);
glBegin(GL_POLYGON);
 glVertex3f (80, 0, 0);
 glVertex3f (80, 100, 0);
 glVertex3f (100, 100, 0);
 glVertex3f (100, 0, 0);
 glEnd();

//trafic light

//lightbox
```

```
glColor3f(1.0, 1.4, 1.7);
glBegin(GL_POLYGON);
    glVertex3f (280, 100, 0);
    glVertex3f (280, 200, 0);
    glVertex3f (340, 200, 0);
    glVertex3f (340, 100, 0);
    glEnd();
//stand
glColor3f(1.0, 0.4, 1.7);
glBegin(GL_POLYGON);
    glVertex3f (300, 0, 0);
    glVertex3f (300, 100, 0);
    glVertex3f (320, 100, 0);
    glVertex3f (320, 0, 0);
    glEnd();
drawCircle3(310, 125, 10,10);
drawCircle(310, 150, 10,10);
drawCircle1(310, 175, 10,10);

glFlush();
}

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

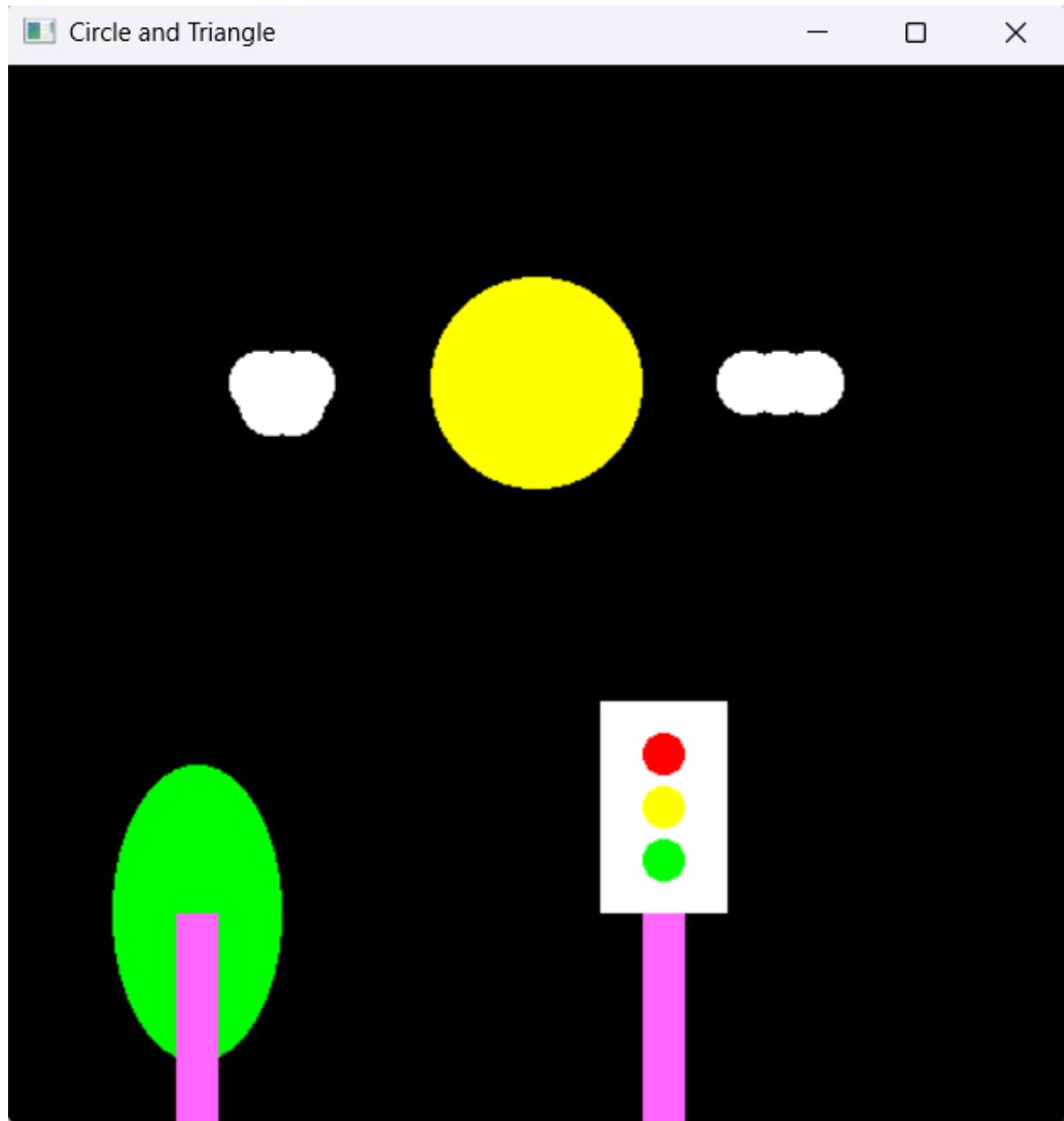
```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowPosition(100, 100);
glutInitWindowSize(500, 500);
glutCreateWindow("Circle and Triangle");

init();
glutDisplayFunc(display);
glutMainLoop();

return 0; // Added return 0 for good practice
```

```
}
```

Output



Discussion

In this lab task, I successfully created a robot using basic OpenGL functions. The Scenery was drawn by combining simple shapes such as circles and rectangles for the traffic lights, cloud, sun, tree leaves and polygons for the tree body, traffic light body. Each part was given a different color using `glColor3f()` to make the figure more visually clear. The program used `glBegin()` and `glEnd()` to define shapes, while `glVertex3f()` specified their coordinates. Overall, this lab demonstrated how OpenGL can be used to design graphical objects using coordinating geometry & color control.



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 07

Title: 2D Implementation (Translation).

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 1st December, 2025

Title

2D Implementation (Translation)

Introduction

In this lab task, I used OpenGL with GLUT to draw a Bus with windows and wheels using basic geometric primitives. The program demonstrates how to create 2D graphics by combining Circles and polygons with different colors and move them. Through this task, we learned how to initialize an OpenGL window, set background and object colors, and use functions like glBegin(), glVertex3f(), and glFlush() to display objects on the screen. This experiment helps in understanding the fundamentals of computer graphics and how shapes are formed using coordinates in OpenGL.

Contents

In this lab task :

1. Functions used

- ‘glClear()’ – clears the screen.
- ‘ glColor3f()’ – sets color.
- ‘glBegin()’/ ‘glEnd()’ – start and end shape drawing.
- ‘glVertex2f()’ – defines shape corners.
- ‘glFlush()’ – displays the drawing.

2. Shapes used:

- Circle – Bus wheels.
- Polygons (rectangles) – bus windows.
- Quads (rectangles) – bus body

Code

```
#include <GL/gl.h>
#include<windows.h>
#ifndef APPLE
#else
#include <GL/glut.h>
#endif
#include <stdlib.h>
#include <math.h>
float p=6.0;
```

```

void drawCircle(int h, int k, int rx, int ry)
{
    glColor3f(0.0, 1.0, 0.0);
    glBegin(GL_POLYGON);
    for (int i = 0; i <= 360; i++) {
        float angle = 3.14159 * i / 180.0;
        glVertex2f(h + rx * cos(angle), k + ry * sin(angle));
    }
    glEnd();
}

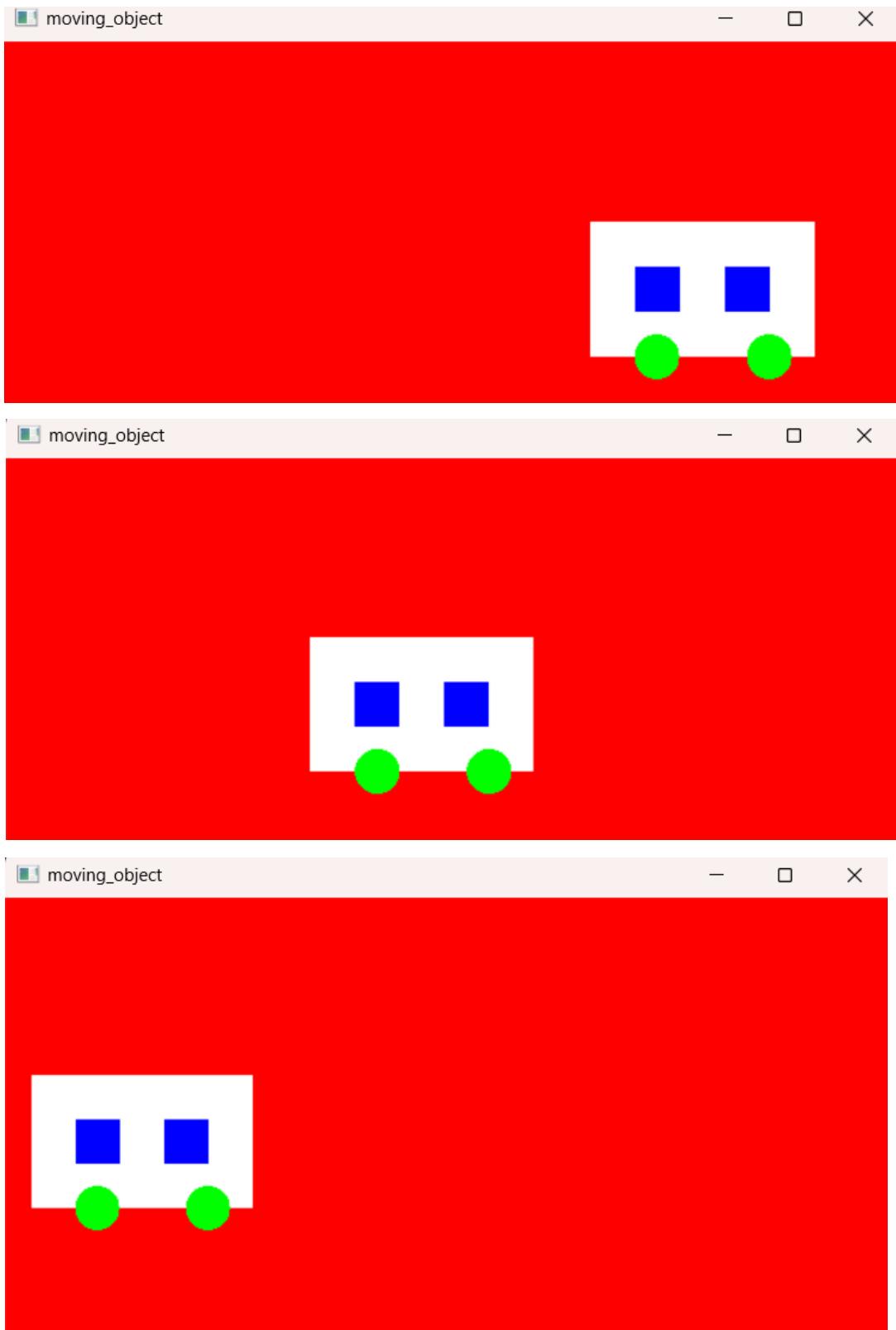
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    if(p<=10)
        p=p-.0005;
    else
        p=6;
    glutPostRedisplay();
    glBegin(GL_QUADS);
    glVertex2f(p-1,3);
    glVertex2f(p-1,6);
    glVertex2f(p+4,6);
    glVertex2f(p+4,3);
    glEnd();
    glColor3f(0.0, 1.0, 0.0);
    glBegin(GL_POLYGON);
    for (int i = 0; i <= 360; i++) {

```

```
float angle = 3.14159 * i / 180.0;  
glVertex2f(p+.5 + 0.5 * cos(angle), 3 + 0.5 * sin(angle));  
}  
glEnd();  
//drawCircle(p+3, 3, 1, 1);  
glBegin(GL_POLYGON);  
for (int i = 0; i <= 360; i++) {  
    float angle = 3.14159 * i / 180.0;  
    glVertex2f(p+3 + 0.5 * cos(angle), 3 + 0.5 * sin(angle));  
}  
glEnd();  
	glColor3f(0.0, 0.0, 1.0);  
glBegin(GL_QUADS);  
    glVertex2f(p,4);  
    glVertex2f(p,5);  
    glVertex2f(p+1,5);  
    glVertex2f(p+1,4);  
glEnd();  
	glColor3f(0.0, 0.0, 1.0);  
glBegin(GL_QUADS);  
    glVertex2f(p+2,4);  
    glVertex2f(p+2,5);  
    glVertex2f(p+3,5);  
    glVertex2f(p+3,4);  
glEnd();  
	glColor3f(1.0, 1.0, 1.0);
```

```
glFlush();  
}  
  
void init(void)  
{  
    glClearColor (1.0, 0.0, 0.0, 0.0);  
    glOrtho(-10.00,10.00,-10.00,10.00,-10.00,10.00);  
}  
  
int main(int argc, char** argv)  
{  
    glutInit(&argc, argv);  
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);  
    glutInitWindowSize (600, 600);  
    glutInitWindowPosition (100, 100);  
    glutCreateWindow ("moving_object");  
    init();  
    glutDisplayFunc(display);  
    glutMainLoop();  
    return 0;  
}
```

Output



Discussion

In this lab task, I successfully created a bus using basic OpenGL functions. The Scenery was drawn by combining simple shapes such as circles and rectangles for the wheels, body and body and move all of them. Each part was given a different color using glColor3f() to make the figure more visually clear. The program used glBegin() and glEnd() to define shapes, while glVertex3f() specified their coordinates. Overall, this lab demonstrated how OpenGL can be used to design graphical objects using coordinating geometry & color control.



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 08

Title: 2D Implementation (Scaling) and Translation for making Object smaller to bigger or bigger to smaller.

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 14th December, 2025

Title

2D Implementation (Scaling) and Translation for making object smaller to bigger or bigger to smaller.

Introduction

In this project, a simple 2D scaling transformation has been implemented using OpenGL and GLUT. The main objective is to modify a previously used translation-based program and extend it to allow an object to become **bigger** or **smaller** based on keyboard input. A circle shape is rendered at the center of the window, and interactive scaling is applied using the '+', '-', and 'r' keys. This project demonstrates how scaling transformations affect objects in a 2D coordinate system and how user input can be used to dynamically change the size of shapes in real time.

Contents

The following functions and shapes have been used in this project:

1. `display()`
 - This function handles all drawing operations.
 - Applies the scaling transformation using `glScalef(scaleFactor, scaleFactor, 1.0f)`.
 - A yellow-colored square is drawn at the center of the screen using `GL_TRIANGLE_FAN`.
2. `keyboard()`
 - Handles keyboard input from the user.
 - Pressing '+' increases the scale factor, making the object bigger.
 - Pressing '-' decreases the scale factor, making the object smaller (with a minimum limit).
 - Pressing 'r' resets the scale factor to its original value (1.0).
 - Finally calls `glutPostRedisplay()` to update the screen.
3. `main()`
 - Creates the window using GLUT functions.
 - Sets up the 2D orthographic projection using `gluOrtho2D(0, 800, 0, 600)`.
 - Registers the display and keyboard callback functions.
 - Starts the GLUT main loop.
4. Shape Used: Circle
 - A simple yellow circle is created with 100 segments using trigonometric functions.
 - Initially centered at the origin after translation.
 - Its size changes dynamically depending on the scaling factor.

Code

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

float scaleFactor = 1.0f;

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(400.0f, 300.0f, 0.0f);
    glScalef(scaleFactor, scaleFactor, 1.0f);

    glBegin(GL_TRIANGLE_FAN);
    glColor3f(1.0f, 1.0f, 0.0f);
    glVertex2f(0, 0);

    float radius = 50.0f;
    int segments = 100;
    for(int i = 0; i <= segments; i++)
    {
        float angle = 2.0f * 3.14159f * i / segments;
        glVertex2f(radius * cos(angle), radius * sin(angle));
    }
}
```

```
glEnd();
glutSwapBuffers();
}

void keyboard(unsigned char key, int x, int y)
{
    if (key == '+')
    {
        scaleFactor += 0.1f;
        std::cout << "Bigger: Scale = " << scaleFactor << std::endl;
    }
    else if (key == '-')
    {
        scaleFactor -= 0.1f;
        if (scaleFactor < 0.1f) scaleFactor = 0.1f;
        std::cout << "Smaller: Scale = " << scaleFactor << std::endl;
    }
    else if (key == 'r')
    {
        scaleFactor = 1.0f;
        std::cout << "Reset: Scale = 1.0" << std::endl;
    }
    glutPostRedisplay();
}

int main(int argc, char** argv)
{
```

```

glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize(800, 600);
glutCreateWindow("2D Scaling - Circle");
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, 800, 0, 600);
glMatrixMode(GL_MODELVIEW);
glutDisplayFunc(display);
glutKeyboardFunc(keyboard);
glutMainLoop();
return 0;
}

```

Output

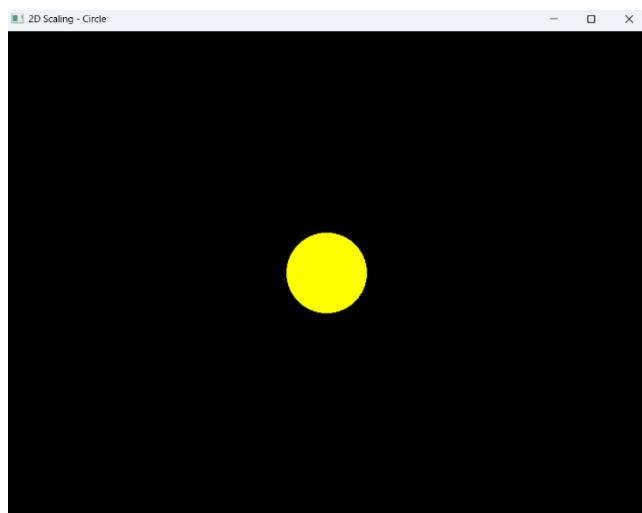


Figure 1: Initial Position and after pressing “r”

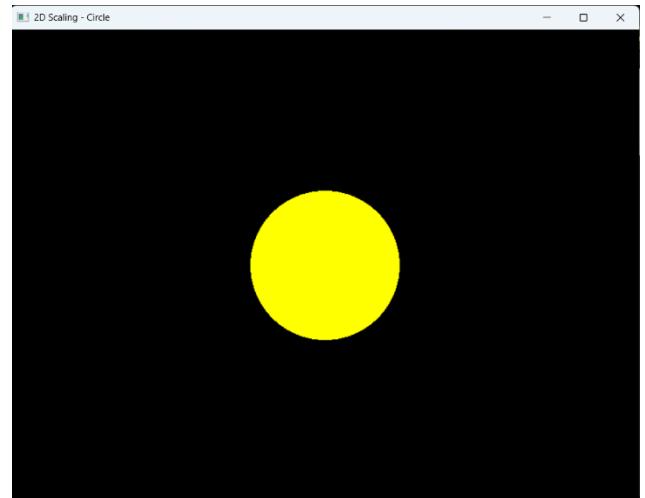
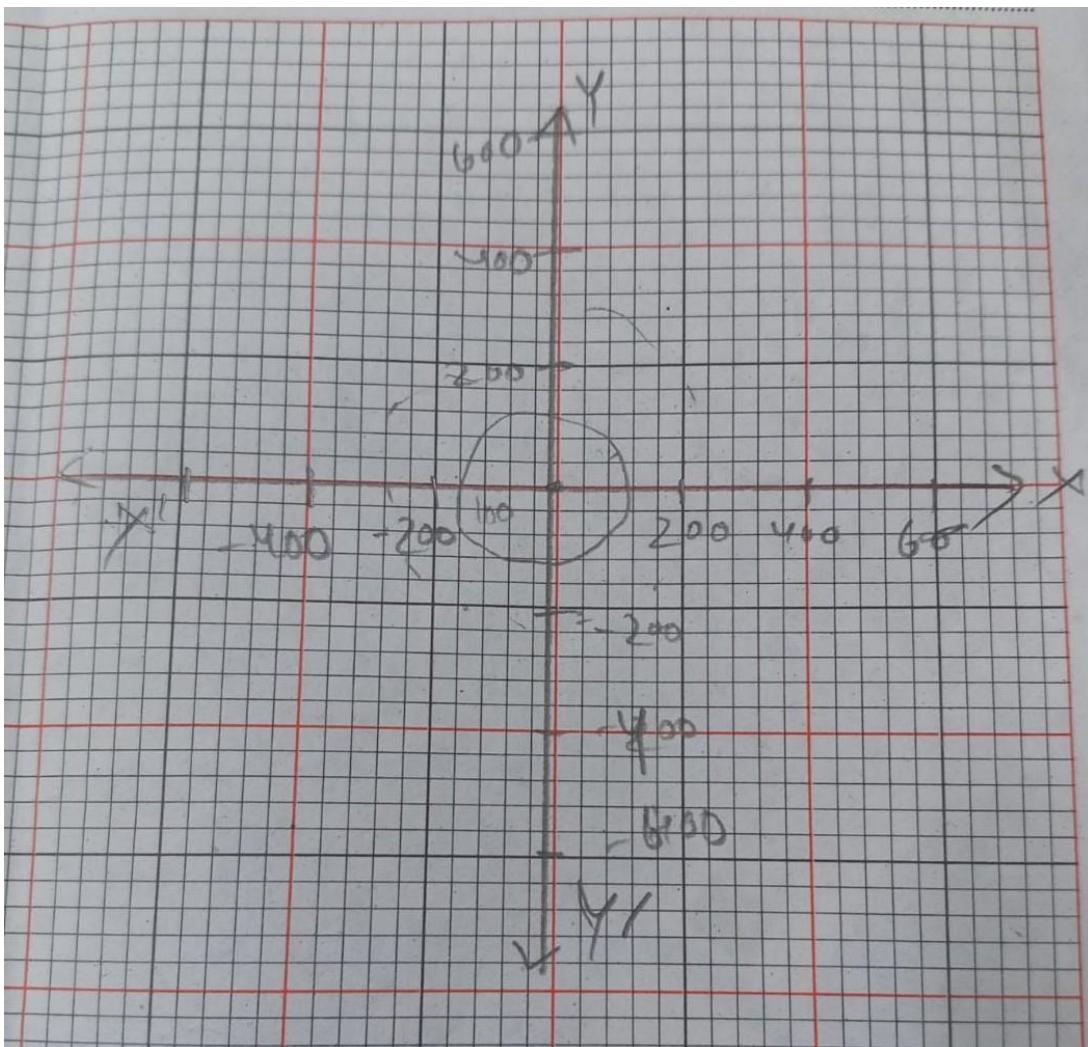


Figure 2: After pressing “+”.

Graph



Discussion

The main objective of the task is to modify a translation-based program and implement interactive 2D scaling. Scaling transforms the size of an object while keeping its shape proportional. In this program, scaling is applied around the center of the window by first translating the coordinate system to the center and then applying `glScalef()`.

Whenever the user presses the '+' key, the scale factor increases, making the square larger. Pressing the '-' key decreases the size but does not allow it to shrink below a safe minimum value. The 'r' key resets the object to its default size. This dynamic interaction demonstrates how transformations can be controlled in real time, which is useful in animations, simulations, and interactive graphics applications.



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 09

Title: Drawing different objects using different primitives,
circle and 2D Transformation.

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 15th December, 2025

Title

Drawing different objects using different primitives, circle and 2D Transformation.

Introduction

In this project, a complete train scenery has been created using various **OpenGL primitive shapes** such as rectangles, triangles, circles, and lines. The program demonstrates the application of **2D Transformations** including translation and scaling to position and size different objects such as the locomotive, train car, wheels, trees, clouds, and sun. Multiple helper functions were written to draw reusable objects like wheels, clouds, trees, and the locomotive components. This project shows how simple primitive shapes can be combined to form complex graphical scenes featuring a moving train.

Contents

The following functions and shapes have been used in this project:

1. **drawRectangle(x, y, w, h)**

- Draws a quadrilateral using GL_POLYGON.
- Used for sky background, locomotive body, train car body, wheels axles, windows, chimney, and tree components.

2. **drawTriangle(x1, y1, x2, y2, x3, y3)**

- Uses GL_TRIANGLES to draw triangular shapes.
- Applied for tree leaves, palm tree leaves, and smoke effects.

3. **drawCircle(cx, cy, r)**

- Approximates a circle using 360 small segments.
- Used for wheels, clouds, smoke, sun, and fruits on trees.

4. **drawSky()**

- Creates the blue sky background and green grass ground respectively.
- Fundamental components for setting the scene environment.

5. **drawCloud(x,y)**

- Composed of 3 overlapping circles to create a cloud shape.
- Positioned using translation transformations.

6. drawSunRays(x, y)

- Creates 3 diagonal yellow rays in the top-left corner.
- Uses drawLine() with varied positions for aesthetic effect.

7. drawSky() & drawGround()

- Creates the blue sky background and green grass ground respectively.
- Fundamental components for setting the scene environment

8. drawTree(x, y) & drawFruitTree(x, y)

- Regular trees: Composed of a brown rectangle (trunk) and green triangle (leaves).
- Fruit trees: Include red circles positioned on the tree leaves using translation.
- Uses translation (glTranslatef) to position trees in the scene

9. drawWheel(x, y, radius)

- Multi-layered wheel design with dark gray outer circle (tire), light gray middle circle (rim), and medium gray center (hub).
- Used for both locomotive and train car wheels with different scales.

10. drawLocomotive(x, y)

- Main locomotive body composed of:
 - Dark blue main body rectangle.
 - Blue cabin rectangle on top.
 - Red chimney rectangle.
 - Light blue window rectangle.
 - Dark front bumper.
 - Gray wheel axles.
 - Two wheels positioned below.
 - Gray coupling point for connecting to train car.
- Uses nested rectangles and glPushMatrix/glPopMatrix for organization.

11. drawTrainCar(x, y)

- Train cargo car composed of:
 - Red bottom body rectangle.
 - Yellow top cargo rectangle.
 - Red chimney rectangle.
 - Gray axles.
 - Two wheels with different scale than locomotive.
 - Gray coupling connection point.
- Demonstrates scaling of wheels to different sizes.

12. drawSmoke(x, y)

- Three overlapping circles of decreasing size positioned vertically.
- Creates smoke effect rising from the locomotive chimney.

13. 2D Transformations Used

- **Translation:** Positioning all objects (locomotive, train car, wheels, trees, clouds) using glTranslatef().
- **Scaling:** Creating different-sized trees and wheels using glScalef().
- **Matrix Operations:** Using glPushMatrix() and glPopMatrix() to manage transformation hierarchies.

Code

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

const float PI = 3.14159265359;

void drawRectangle(float x, float y, float w, float h) {
    glBegin(GL_POLYGON);
    glVertex2f(x, y);
    glVertex2f(x + w, y);
    glVertex2f(x + w, y + h);
    glVertex2f(x, y + h);
    glEnd();
}

void drawTriangle(float x1, float y1, float x2, float y2, float x3, float y3) {
```

```
glBegin(GL_TRIANGLES);
glVertex2f(x1, y1);
glVertex2f(x2, y2);
glVertex2f(x3, y3);
glEnd();
}
```

```
void drawCircle(float cx, float cy, float r) {
    glBegin(GL_POLYGON);
    for (int i = 0; i < 360; i++) {
        float theta = i * PI / 180;
        glVertex2f(cx + r * cos(theta), cy + r * sin(theta));
    }
    glEnd();
}
```

```
void drawWheel(float x, float y, float r) {
    glPushMatrix();
    glTranslatef(x, y, 0);
    glColor3f(0.6, 0.6, 0.6);
    drawCircle(0, 0, r);
    glColor3f(0.8, 0, 0);
    drawCircle(0, 0, r * 0.6);
    glPopMatrix();
}
```

```
void drawCloud(float x, float y) {
```

```
glPushMatrix();
glTranslatef(x, y, 0);
glColor3f(0.85, 0.85, 0.85);
drawCircle(0, 0, 0.05);
drawCircle(0.06, 0.01, 0.06);
drawCircle(0.12, 0, 0.05);
drawCircle(0.06, 0.04, 0.05);
glPopMatrix();
}
```

```
void drawCustomSun(float x, float y) {
    glPushMatrix();
    glTranslatef(x, y, 0);
    glColor3f(1, 1, 0);
    drawCircle(0, 0, 0.08);

    glLineWidth(3.0);
    glBegin(GL_LINES);
    for (int i = 0; i < 8; i++) {
        float theta = i * (2 * PI / 8);
        float r_inner = 0.1;
        float r_outer = 0.18;
        glVertex2f(r_inner * cos(theta), r_inner * sin(theta));
        glVertex2f(r_outer * cos(theta), r_outer * sin(theta));
    }
    glEnd();
    glLineWidth(1.0);
```

```
glPopMatrix();  
}  
  
void drawPineTree(float x, float y) {  
    glPushMatrix();  
    glTranslatef(x, y, 0);  
    glColor3f(0.4, 0.25, 0.1);  
    drawRectangle(-0.02, -0.2, 0.04, 0.2);  
    glColor3f(0.1, 0.6, 0.1);  
    drawTriangle(-0.12, -0.05, 0.12, -0.05, 0, 0.15);  
    drawTriangle(-0.1, 0.1, 0.1, 0.1, 0, 0.3);  
    drawTriangle(-0.08, 0.25, 0.08, 0.25, 0, 0.4);  
    glPopMatrix();  
}  
  
void drawAppleTree(float x, float y) {  
    glPushMatrix();  
    glTranslatef(x, y, 0);  
    glColor3f(0.4, 0.25, 0.1);  
    drawRectangle(-0.03, -0.2, 0.06, 0.3);  
    glColor3f(0.1, 0.7, 0.1);  
    drawCircle(0, 0.2, 0.18);  
    glColor3f(0.9, 0, 0);  
    drawCircle(-0.08, 0.15, 0.02);  
    drawCircle(0.05, 0.12, 0.02);  
    drawCircle(-0.02, 0.25, 0.02);  
    drawCircle(0.1, 0.22, 0.02);  
}
```

```
drawCircle(-0.06, 0.3, 0.02);

glPopMatrix();

}

void drawEngineBody(float x, float y) {

    glPushMatrix();
    glTranslatef(x, y, 0);

    glColor3f(0.1, 0.1, 0.7);
    drawRectangle(0, 0, 0.45, 0.25);

    glColor3f(0.8, 0, 0);
    drawRectangle(0.25, 0.25, 0.2, 0.2);
    drawRectangle(0.23, 0.45, 0.24, 0.03);

    glColor3f(0.5, 0.8, 1.0);
    drawRectangle(0.28, 0.28, 0.14, 0.14);

    glColor3f(0.8, 0, 0);
    glBegin(GL_POLYGON);
    glVertex2f(0.05, 0.25);
    glVertex2f(0.15, 0.25);
    glVertex2f(0.18, 0.45);
    glVertex2f(0.02, 0.45);
    glEnd();

    glColor3f(0.8, 0, 0);
```

```
drawTriangle(0, 0, 0, 0.15, -0.1, 0);

glColor3f(1, 1, 0);
drawCircle(0, 0.18, 0.04);

glPopMatrix();
}

void drawWagonBody(float x, float y) {
    glPushMatrix();
    glTranslatef(x, y, 0);

    glColor3f(0.1, 0.1, 0.7);
    drawRectangle(0.05, 0, 0.3, 0.1);

    glColor3f(0.8, 0, 0);
    glBegin(GL_POLYGON);
    glVertex2f(0, 0.1);
    glVertex2f(0.4, 0.1);
    glVertex2f(0.38, 0.3);
    glVertex2f(0.02, 0.3);
    glEnd();

    glColor3f(1, 1, 0);
    drawTriangle(0.05, 0.3, 0.35, 0.3, 0.2, 0.45);

    glPopMatrix();
}
```

```
}
```

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(0.3, 0.7, 1.0);
    drawRectangle(-1, -1, 2, 2);

    glColor3f(0.3, 0.8, 0.3);
    drawCircle(0, -1.3, 1.5);

    drawCustomSun(-0.75, 0.75);
    drawCloud(-0.4, 0.6);
    drawCloud(0.0, 0.7);
    drawCloud(0.4, 0.6);

    drawPineTree(0.4, 0.1);
    drawAppleTree(0.75, 0.05);

    float trainY = -0.25;

    glColor3f(0.6, 0.6, 0.6);
    drawRectangle(-0.1, trainY + 0.05, 0.2, 0.02);

    drawEngineBody(-0.55, trainY);
    drawWheel(-0.45, trainY, 0.08);
    drawWheel(-0.2, trainY, 0.08);
```

```
drawWagonBody(0.1, trainY);
drawWheel(0.2, trainY - 0.02, 0.06);
drawWheel(0.4, trainY - 0.02, 0.06);

glFlush();
}

void init() {
    glClearColor(1, 1, 1, 1);
    gluOrtho2D(-1, 1, -1, 1);
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Train Scenery");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Output



Discussion

This task required drawing a complete train scenery using only primitive shapes and applying 2D transformations to position and scale each object correctly. To achieve this:

- The **background** (sky and ground) was created using large rectangles with different colors.
- The **sun** was created using sun rays with appropriate positioning and coloring.
- **Clouds** were formed by overlapping circles arranged in a symmetric pattern.
- The **locomotive** was built using multiple rectangles layered together (body, cabin, chimney) with a complex color scheme to represent the vehicle structure.
- The **train car** was similarly constructed with rectangles for the body and top cargo section.
- **Wheels** were created using concentric circles to show tire, rim, and hub details, with scaling applied to create wheels of different sizes.
- **Trees** were positioned across the scene using translation transformations, with fruit trees including red circles for apples/fruits using positioned circles.
- **Smoke** rising from the chimney was created using three circles of decreasing size.

By modularizing the drawing functions (wheel, locomotive, train car, cloud, tree, smoke), the entire scenery becomes structured, easier to understand, and reusable. The use of 2D transformations (translation, scaling) demonstrates how complex compositions can be formed from simple primitive shapes. The hierarchical use of `glPushMatrix()` and `glPopMatrix()` ensures that transformations applied to parent objects (like the locomotive's position) correctly affect all child objects (like its wheels).



Daffodil
International
University

Lab Report

Course Code: CSE 422

Course Title: Computer Graphics Lab

Report No: 10

Title: Drawing different 3D objects using OpenGL primitives with 3D transformations (translation, rotation, and scaling).

Submitted To:

Syada Tasmia Alvi

Senior Lecturer

Department of Computer Science and Engineering (C.S.E)

Submitted by:

Name: Mahmudul Hasan Piash

ID: 221-15-5606

Section: 61_D1

Department of Computer Science and Engineering (C.S.E)

Date of submission: 15th December, 2025

Title

Drawing different 3D objects using OpenGL primitives with 3D transformations (translation, rotation, and scaling).

Introduction

In this project, multiple 3D objects have been created using OpenGL primitive shapes such as cubes, pyramids, tetrahedrons, and spheres. The program demonstrates the application of 3D Transformations including translation, rotation, and scaling to manipulate these objects interactively. Multiple helper functions were written to draw reusable 3D objects. This project shows how fundamental 3D geometric shapes can be rendered and transformed in real-time using OpenGL.

Contents

The following functions and shapes have been used in this project:

1. drawCubeFromPoints(x1, y1, z1, x2, y2, z2)

- Draws a cube using two diagonal corner points.
- Uses GL_QUADS to draw all six faces with different colors.
- Each face has 4 vertices specified in counter-clockwise order.

2. drawPyramid(x1, y1, z1, x2, y2, z2)

- Draws a pyramid with a rectangular base and triangular sides.
- Uses GL_TRIANGLES for all four triangular faces and the base.
- The apex is calculated as the center point at the top.

3. drawTetrahedron(float size)

- Draws a tetrahedron with four triangular faces.
- Uses GL_TRIANGLES for all four faces with different colors.
- Each face is a triangle connecting three vertices.

4. drawSphere(float radius, int slices, int stacks)

- Draws a sphere using latitude and longitude segments.
- Uses GL_QUAD_STRIP to create smooth curved surfaces.
- Radius, slices, and stacks control the sphere's detail level.

5. 3D Transformations Used

- Translation: Positioning objects using `glTranslatef()` with keyboard controls (W/A/S/D/Q/E).
- Rotation: Rotating objects around X, Y, and Z axes using `glRotatef()` with keyboard controls (X/x, Y/y, Z/z).
- Scaling: Resizing objects uniformly using `glScalef()` with keyboard controls (+/-).

Code

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

using namespace std;

// Global variables for cube coordinates
float X1, Y1, Z1;
float X2, Y2, Z2;

// Global variables for transformations
float transX = 0, transY = 0, transZ = 0;
float angleX = 0.0f, angleY = 0.0f, angleZ = 0.0f;
float scaleX = 1.0f, scaleY = 1.0f, scaleZ = 1.0f;

// Transformation step values
float stepX = 5, stepY = 5, stepZ = 5;
float Tx = 0.5f, Ty = 0.5f, Tz = 0.5f;
float Sx = 0.1f, Sy = 0.1f, Sz = 0.1f;

// ===== Function to draw a Cube =====
```

```
void drawCubeFromPoints(float x1, float y1, float z1, float x2, float y2, float z2)
{
    glBegin(GL_QUADS);

    // Front face (Red)
    glColor3f(1, 0, 0);
    glVertex3f(x1, y1, z1);
    glVertex3f(x2, y1, z1);
    glVertex3f(x2, y2, z1);
    glVertex3f(x1, y2, z1);

    // Back face (Green)
    glColor3f(0, 1, 0);
    glVertex3f(x1, y1, z2);
    glVertex3f(x1, y2, z2);
    glVertex3f(x2, y2, z2);
    glVertex3f(x2, y1, z2);

    // Left face (Blue)
    glColor3f(0, 0, 1);
    glVertex3f(x1, y1, z1);
    glVertex3f(x1, y1, z2);
    glVertex3f(x1, y2, z2);
    glVertex3f(x1, y2, z1);

    // Right face (Yellow)
    glColor3f(1, 1, 0);
}
```

```

glVertex3f(x2, y1, z1);
glVertex3f(x2, y1, z2);
glVertex3f(x2, y2, z2);
glVertex3f(x2, y2, z1);

// Top face (Cyan)
glColor3f(0, 1, 1);
glVertex3f(x1, y2, z1);
glVertex3f(x1, y2, z2);
glVertex3f(x2, y2, z2);
glVertex3f(x2, y2, z1);

// Bottom face (Magenta)
glColor3f(1, 0, 1);
glVertex3f(x1, y1, z1);
glVertex3f(x2, y1, z1);
glVertex3f(x2, y1, z2);
glVertex3f(x1, y1, z2);

glEnd();
}

// ===== Function to draw a Pyramid =====
void drawPyramid(float x1, float y1, float z1, float x2, float y2, float z2)
{
    float centerX = (x1 + x2) / 2.0f;
    float centerY = y2;
}

```

```
float centerZ = (z1 + z2) / 2.0f;

glBegin(GL_TRIANGLES);

// Front face
glColor3f(1, 0.5, 0);
glVertex3f(x1, y1, z1);
glVertex3f(x2, y1, z1);
glVertex3f(centerX, centerY, centerZ);

// Right face
glColor3f(1, 0, 1);
glVertex3f(x2, y1, z1);
glVertex3f(x2, y1, z2);
glVertex3f(centerX, centerY, centerZ);

// Back face
glColor3f(0, 1, 1);
glVertex3f(x2, y1, z2);
glVertex3f(x1, y1, z2);
glVertex3f(centerX, centerY, centerZ);

// Left face
glColor3f(0.5, 1, 0);
glVertex3f(x1, y1, z2);
glVertex3f(x1, y1, z1);
glVertex3f(centerX, centerY, centerZ);
```

```
// Base
glColor3f(0.5, 0.5, 0.5);
glVertex3f(x1, y1, z1);
glVertex3f(x2, y1, z1);
glVertex3f(x2, y1, z2);

glVertex3f(x1, y1, z1);
glVertex3f(x2, y1, z2);
glVertex3f(x1, y1, z2);

glEnd();
}

// ===== Function to draw a Tetrahedron =====
void drawTetrahedron(float size)
{
    glBegin(GL_TRIANGLES);

    // Face 1
    glColor3f(1, 0, 0);
    glVertex3f(0, 0, 0);
    glVertex3f(size, 0, 0);
    glVertex3f(size/2, size, 0);

    // Face 2
    glColor3f(0, 1, 0);
```

```

glVertex3f(0, 0, 0);
glVertex3f(size, 0, 0);
glVertex3f(size/2, size/2, size);

// Face 3
glColor3f(0, 0, 1);
glVertex3f(size, 0, 0);
glVertex3f(size/2, size, 0);
glVertex3f(size/2, size/2, size);

// Face 4
glColor3f(1, 1, 0);
glVertex3f(0, 0, 0);
glVertex3f(size/2, size, 0);
glVertex3f(size/2, size/2, size);

glEnd();
}

// ===== Function to draw a Sphere =====
void drawSphere(float radius, int slices, int stacks)
{
    for(int i = 0; i < stacks; i++)
    {
        float lat0 = M_PI * (-0.5 + (float)i / stacks);
        float lat1 = M_PI * (-0.5 + (float)(i+1) / stacks);

```

```

glBegin(GL_QUAD_STRIP);

for(int j = 0; j <= slices; j++)
{
    float lng = 2 * M_PI * (float)j / slices;

    float x0 = cos(lat0) * cos(lng);
    float y0 = sin(lat0);
    float z0 = cos(lat0) * sin(lng);

    float x1 = cos(lat1) * cos(lng);
    float y1 = sin(lat1);
    float z1 = cos(lat1) * sin(lng);

    glColor3f(0.2f + (i * 0.7f / stacks), 0.3f, 0.9f - (i * 0.5f / stacks));
    glVertex3f(x0 * radius, y0 * radius, z0 * radius);
    glVertex3f(x1 * radius, y1 * radius, z1 * radius);
}

glEnd();
}

// ===== Display Function =====
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

```

```
glLoadIdentity();

gluLookAt(20, 20, 30,
          0, 0, 0,
          0, 1, 0);

// Apply transformations
glTranslatef(transX, transY, transZ);
glRotatef(angleX, 1, 0, 0);
glRotatef(angleY, 0, 1, 0);
glRotatef(angleZ, 0, 0, 1);

// ===== Draw Cube =====
glPushMatrix();
glScalef(scaleX, scaleY, scaleZ);
drawCubeFromPoints(X1, Y1, Z1, X2, Y2, Z2);
glPopMatrix();

// ===== Draw Pyramid =====
glPushMatrix();
glTranslatef(8, 0, 0);
glScalef(scaleX, scaleY, scaleZ);
drawPyramid(X1, Y1, Z1, X2, Y2, Z2);
glPopMatrix();

// ===== Draw Tetrahedron =====
glPushMatrix();
```

```

glTranslatef(-8, 0, 0);
glScalef(scaleX, scaleY, scaleZ);
drawTetrahedron(3.0f);
glPopMatrix();

// ===== Draw Sphere =====
glPushMatrix();
glTranslatef(0, 0, 8);
drawSphere(2.0f, 20, 20);
glPopMatrix();

glutSwapBuffers();
}

// ===== Keyboard Function =====
void keyboard(unsigned char key, int x, int y)
{
    switch(key)
    {
        // Rotation controls
        case 'X': angleX += stepX; break;
        case 'x': angleX -= stepX; break;
        case 'Y': angleY += stepY; break;
        case 'y': angleY -= stepY; break;
        case 'Z': angleZ += stepZ; break;
        case 'z': angleZ -= stepZ; break;
    }
}

```

```
// Translation controls  
case 'w': transY += Ty; break;  
case 's': transY -= Ty; break;  
case 'd': transX += Tx; break;  
case 'a': transX -= Tx; break;  
case 'q': transZ += Tz; break;  
case 'e': transZ -= Tz; break;  
  
// Scaling controls  
case '+':  
    scaleX += Sx;  
    scaleY += Sy;  
    scaleZ += Sz;  
    break;  
case '-':  
    scaleX -= Sx;  
    scaleY -= Sy;  
    scaleZ -= Sz;  
    if(scaleX < 0.1f) scaleX = 0.1f;  
    if(scaleY < 0.1f) scaleY = 0.1f;  
    if(scaleZ < 0.1f) scaleZ = 0.1f;  
    break;  
  
// Reset  
case 'r':  
    transX = 0; transY = 0; transZ = 0;  
    angleX = 0; angleY = 0; angleZ = 0;
```

```
    scaleX = 1.0f; scaleY = 1.0f; scaleZ = 1.0f;  
    break;
```

```
case 27: exit(0); // ESC to exit
```

```
}
```

```
glutPostRedisplay();
```

```
}
```

```
// ===== Reshape Function =====
```

```
void reshape(int w, int h)
```

```
{
```

```
    glViewport(0, 0, w, h);
```

```
    glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();
```

```
    gluPerspective(60, (float)w / (float)h, 1.0, 100.0);
```

```
    glMatrixMode(GL_MODELVIEW);
```

```
}
```

```
// ===== Initialization Function =====
```

```
void init()
```

```
{
```

```
    glEnable(GL_DEPTH_TEST);
```

```
    glClearColor(0.1f, 0.1f, 0.2f, 1.0f);
```

```
    glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();
```

```

gluPerspective(60, 1.0, 1.0, 100.0);

glMatrixMode(GL_MODELVIEW);
}

// ===== Main Function =====

int main(int argc, char** argv)
{
    cout << "===== \n";
    cout << " 3D Objects with Transformations\n";
    cout << "===== \n\n";

    cout << "Enter first point (x1 y1 z1): ";
    cin >> X1 >> Y1 >> Z1;

    cout << "Enter second point (x2 y2 z2): ";
    cin >> X2 >> Y2 >> Z2;

    cout << "\n===== CONTROLS =====\n";
    cout << "Rotation:  X/x, Y/y, Z/z\n";
    cout << "Translation: W/A/S/D/Q/E\n";
    cout << "Scale:      +/-\n";
    cout << "Reset:      R\n";
    cout << "Exit:       ESC\n";
    cout << "===== \n\n";

    glutInit(&argc, argv);
}

```

```

glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize(800, 600);
glutCreateWindow("3D Objects - Cube, Pyramid, Tetrahedron, Sphere");

init();

glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);

glutMainLoop();

return 0;
}

```

Explanation

3D Objects:

- The cube is drawn from two diagonal corner points, making it flexible and easy to resize.
- The pyramid is constructed with a rectangular base and four triangular sides meeting at an apex.
- The tetrahedron is a simple shape with four triangular faces, serving as a basic 3D primitive.
- The sphere is generated using trigonometric functions to create latitude and longitude segments.

3D Transformations:

The `display()` function applies transformations in a specific order: first translation, then rotation around all three axes, and finally scaling. This order is important because matrix multiplication in OpenGL is not commutative—changing the order of transformations produces different results.

Keyboard Controls:

Rotation:

- 'X' / 'x' - rotate in positive/negative direction around X-axis
- 'Y' / 'y' - rotate in positive/negative direction around Y-axis
- 'Z' / 'z' - rotate in positive/negative direction around Z-axis

Translation:

- 'W' - move up (positive Y-axis)
- 'S' - move down (negative Y-axis)
- 'D' - move right (positive X-axis)
- 'A' - move left (negative X-axis)
- 'Q' - move forward (positive Z-axis)
- 'E' - move backward (negative Z-axis)

Scaling:

- '+' - increase size uniformly in all dimensions
- '-' - decrease size uniformly in all dimensions (minimum 0.1)

Other:

- 'R' - reset all transformations
- 'ESC' - exit the program

Input Example:

- $(x_1, y_1, z_1) = (-2, -2, -2)$
- $(x_2, y_2, z_2) = (2, 2, 2)$

Output

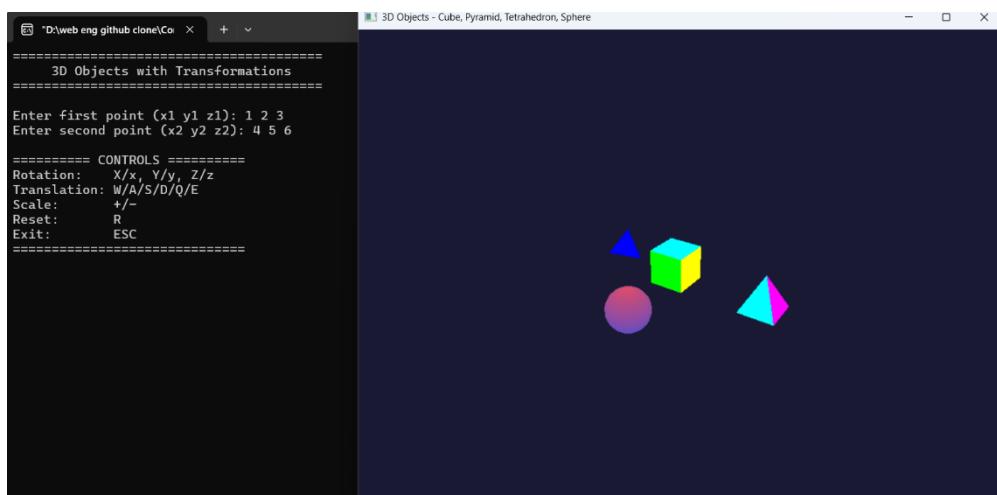


Figure 1: Initial Position with value and also preeing “r”.

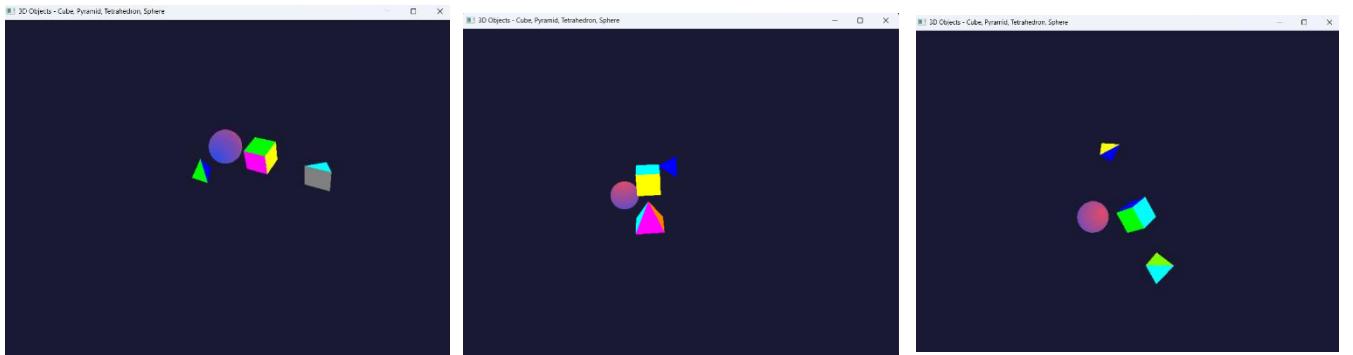


Figure 2,3,4: X-axis, Y-axis, Z-axis movement by pressing “x”, “y” and “z”.

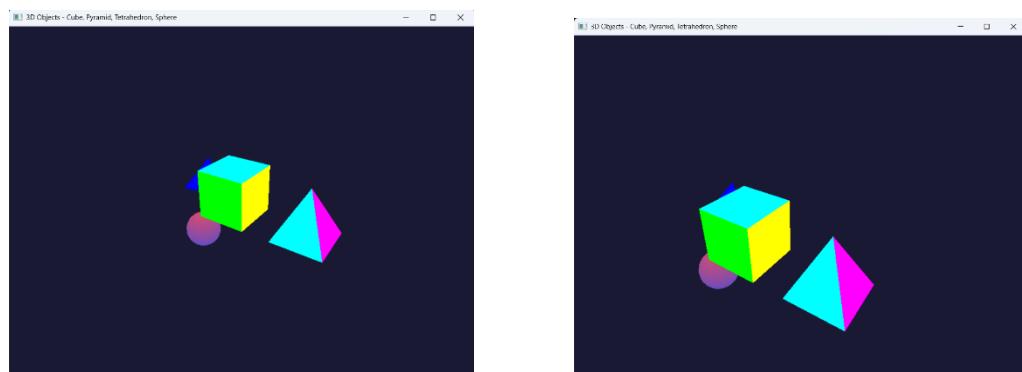


Figure 5,6: Figure Scaling by pressing "+/-", Translation by pressing "W/D/A/S/Q/E"

Discussion

The key learnings include:

1. **3D Object Rendering:** Multiple 3D geometric shapes (cube, pyramid, tetrahedron, sphere) are rendered using different primitive types (GL_QUADS and GL_TRIANGLES).
2. **Coordinate Systems:** Understanding 3D Cartesian coordinates and how to specify vertices in 3D space using (x, y, z) coordinates.
3. **Transformation Matrices:** The application of glTranslatef(), glRotatef(), and glScalef() demonstrates how transformation matrices are applied to modify object positions, orientations, and sizes.
4. **Matrix Stack:** Use of glPushMatrix() and glPopMatrix() allows independent transformation of multiple objects without affecting each other.
5. **Viewing Pipeline:** The gluLookAt() function positions the camera in 3D space, and gluPerspective() defines the projection matrix for proper 3D perspective rendering.

6. Depth Testing: Enabling GL_DEPTH_TEST ensures proper rendering order of overlapping objects based on their distance from the camera.

This program demonstrates fundamental 3D graphics concepts by rendering four different 3D objects (cube, pyramid, tetrahedron, and sphere) and allowing interactive manipulation through real-time transformations.