**UCS1712 – GRAPHICS AND MULTIMEDIA LAB**

**Lab Exercise 1**: Basic Output Primitives in C++ using OpenGL

**Aim:**

To create primitive shapes and lines using OpenGl and test out different options available. Build a Checker board and a house.

**CODE:**

#include<GL/glut.h>

void myInit()

{

glClearColor(1.0, 1.0, 1.0, 1.0);

glColor3f(0.0f, 0.0f, 0.0f);

glPointSize(5);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0, 1440.0, 0.0, 480.0);

}

void myDispA() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_POLYGON);

glVertex2d(100, 100);

glVertex2d(150, 230);

glVertex2d(170, 130);

glVertex2d(300, 350);

glEnd();

glFlush();

}

void myDispB()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_LINES);

for (int i = 0; i <= 8; i++)

{

glVertex2d(120 + i \* 50, 40);

glVertex2d(120 + i \* 50, 440);

}

for (int i = 0; i <= 8; i++)

{

glVertex2d(120 , 40 + i \* 50);

glVertex2d(520 , 40 + i\*50);

}

glEnd();

glBegin(GL\_QUADS);

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 4; j++)

{

glVertex2d(120 + j \* 100, 40 + i \* 100);

glVertex2d(170 + j \* 100, 40 + i \* 100);

glVertex2d(170 + j \* 100, 90 + i \* 100);

glVertex2d(120 + j \* 100, 90 + i \* 100);

}

for (int j = 0; j < 4; j++)

{

glVertex2d(170 + j \* 100, 90 + i \* 100);

glVertex2d(220 + j \* 100, 90 + i \* 100);

glVertex2d(220 + j \* 100, 140 + i \* 100);

glVertex2d(170 + j \* 100, 140 + i \* 100);

}

}

glEnd();

glFlush();

}

void myDispC() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_TRIANGLES);

glVertex2d(320, 440);

glVertex2d(120, 280);

glVertex2d(520, 280);

glEnd();

glBegin(GL\_LINE\_STRIP);

glVertex2d(170, 280);

glVertex2d(170, 40);

glVertex2d(470, 40);

glVertex2d(470, 280);

glEnd();

glBegin(GL\_QUADS);

glVertex2d(220, 200);

glVertex2d(300, 200);

glVertex2d(300, 40);

glVertex2d(220, 40);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(360, 230);

glVertex2d(420, 230);

glVertex2d(420, 170);

glVertex2d(360, 170);

glEnd();

glBegin(GL\_QUADS);

glVertex2d(360, 230);

glVertex2d(375, 220);

glVertex2d(375, 180);

glVertex2d(360, 170);

glVertex2d(420, 230);

glVertex2d(405, 220);

glVertex2d(405, 180);

glVertex2d(420, 170);

glEnd();

glFlush();

}

void myDispD()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.45f, 0.45f, 0.45f);

glBegin(GL\_TRIANGLES);

glVertex2d(120, 390);

glVertex2d(430, 390);

glVertex2d(275, 460);

glVertex2d(1320, 210);

glVertex2d(1170, 210);

glVertex2d(1170, 290);

glEnd();

glBegin(GL\_QUADS);

glVertex2d(275, 390);

glVertex2d(275, 440);

glVertex2d(480, 440);

glVertex2d(480, 390);

glVertex2d(470, 210);

glVertex2d(470, 300);

glVertex2d(720, 300);

glVertex2d(720, 210);

glVertex2d(720, 210);

glVertex2d(720, 290);

glVertex2d(1170, 290);

glVertex2d(1170, 210);

glVertex2d(720, 200);

glVertex2d(720, 210);

glVertex2d(1320, 210);

glVertex2d(1320, 200);

glEnd();

glColor3f(1.0f, 0.9f, 0.6f);

glBegin(GL\_QUADS);

glVertex2d(120, 40);

glVertex2d(120, 90);

glVertex2d(480, 90);

glVertex2d(480, 40);

glVertex2d(180, 90);

glVertex2d(180, 190);

glVertex2d(480, 190);

glVertex2d(480, 90);

glVertex2d(120, 190);

glVertex2d(120, 240);

glVertex2d(480, 240);

glVertex2d(480, 190);

glVertex2d(480, 210);

glVertex2d(720, 210);

glVertex2d(720, 40);

glVertex2d(480, 40);

glVertex2d(720, 45);

glVertex2d(1020, 45);

glVertex2d(1020, 210);

glVertex2d(720, 210);

glVertex2d(1020, 50);

glVertex2d(1020, 60);

glVertex2d(1320, 60);

glVertex2d(1320, 50);

glVertex2d(1020, 60);

glVertex2d(1020, 190);

glVertex2d(1280, 190);

glVertex2d(1280, 60);

glVertex2d(1020, 190);

glVertex2d(1020, 200);

glVertex2d(1320, 200);

glVertex2d(1320, 190);

glVertex2d(150, 240);

glVertex2d(150, 350);

glVertex2d(470, 350);

glVertex2d(470, 240);

glVertex2d(120, 350);

glVertex2d(120, 390);

glVertex2d(470, 390);

glVertex2d(470, 350);

glEnd();

glBegin(GL\_TRIANGLES);

glVertex2d(720, 210);

glVertex2d(720, 300);

glVertex2d(1020, 210);

glEnd();

glColor3f(1.0f, 1.0f, 1.0f);

glBegin(GL\_QUADS);

glVertex2d(210, 90);

glVertex2d(210, 180);

glVertex2d(240, 180);

glVertex2d(240, 90);

glVertex2d(250, 90);

glVertex2d(250, 180);

glVertex2d(280, 180);

glVertex2d(280, 90);

glVertex2d(290, 90);

glVertex2d(290, 180);

glVertex2d(320, 180);

glVertex2d(320, 90);

glVertex2d(330, 90);

glVertex2d(330, 180);

glVertex2d(360, 180);

glVertex2d(360, 90);

glVertex2d(490, 70);

glVertex2d(490, 180);

glVertex2d(650, 180);

glVertex2d(650, 70);

glVertex2d(655, 70);

glVertex2d(655, 180);

glVertex2d(680, 180);

glVertex2d(680, 70);

glVertex2d(690, 60);

glVertex2d(690, 190);

glVertex2d(700, 190);

glVertex2d(700, 60);

glVertex2d(750, 70);

glVertex2d(750, 180);

glVertex2d(990, 180);

glVertex2d(990, 70);

glVertex2d(1030, 80);

glVertex2d(1030, 170);

glVertex2d(1200, 170);

glVertex2d(1200, 80);

glVertex2d(170, 240);

glVertex2d(170, 340);

glVertex2d(410, 340);

glVertex2d(410, 240);

glEnd();

glColor3f(0.9f, 0.6f, 0.5f);

glBegin(GL\_QUADS);

glVertex2d(1210, 80);

glVertex2d(1210, 170);

glVertex2d(1270, 170);

glVertex2d(1270, 80);

glEnd();

glColor3f(0.0f, 0.0f, 0.0f);

glBegin(GL\_LINE\_LOOP);

glVertex2d(120,40);

glVertex2d(120,90);

glVertex2d(180,90);

glVertex2d(180,190);

glVertex2d(120,190);

glVertex2d(120,240);

glVertex2d(480,240);

glVertex2d(480,210);

glVertex2d(720,210);

glVertex2d(720,190);

glVertex2d(480,190);

glVertex2d(480,60);

glVertex2d(720,60);

glVertex2d(720,40);

glEnd();

glBegin(GL\_LINES);

glVertex2d(1020, 60);

glVertex2d(1320, 60);

glVertex2d(470, 300);

glVertex2d(720, 300);

glVertex2d(470, 390);

glVertex2d(480, 390);

glVertex2d(480, 390);

glVertex2d(480, 440);

glVertex2d(480, 440);

glVertex2d(318, 440);

glVertex2d(1170, 290);

glVertex2d(750, 290);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(210, 90);

glVertex2d(210, 180);

glVertex2d(240, 180);

glVertex2d(240, 90);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(250, 90);

glVertex2d(250, 180);

glVertex2d(280, 180);

glVertex2d(280, 90);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(290, 90);

glVertex2d(290, 180);

glVertex2d(320, 180);

glVertex2d(320, 90);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(330, 90);

glVertex2d(330, 180);

glVertex2d(360, 180);

glVertex2d(360, 90);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(490, 70);

glVertex2d(490, 180);

glVertex2d(650, 180);

glVertex2d(650, 70);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(655, 70);

glVertex2d(655, 180);

glVertex2d(680, 180);

glVertex2d(680, 70);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(690, 60);

glVertex2d(690, 190);

glVertex2d(700, 190);

glVertex2d(700, 60);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(750, 70);

glVertex2d(750, 180);

glVertex2d(990, 180);

glVertex2d(990, 70);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(1030, 80);

glVertex2d(1030, 170);

glVertex2d(1200, 170);

glVertex2d(1200, 80);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(170, 240);

glVertex2d(170, 340);

glVertex2d(410, 340);

glVertex2d(410, 240);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(1210, 80);

glVertex2d(1210, 170);

glVertex2d(1270, 170);

glVertex2d(1270, 80);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(150,240);

glVertex2d(150,350);

glVertex2d(430,350);

glVertex2d(430,240);

glVertex2d(410,240);

glVertex2d(410,340);

glVertex2d(170,340);

glVertex2d(170,240);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(120,350);

glVertex2d(120,390);

glVertex2d(470,390);

glVertex2d(470,240);

glVertex2d(430,240);

glVertex2d(430,350);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(720, 45);

glVertex2d(720, 300);

glVertex2d(1020, 210);

glVertex2d(1020, 45);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(1020, 200);

glVertex2d(1320, 200);

glVertex2d(1320, 190);

glVertex2d(1280, 190);

glVertex2d(1280, 60);

glVertex2d(1320, 60);

glVertex2d(1320, 50);

glVertex2d(1020, 50);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(120, 390);

glVertex2d(430, 390);

glVertex2d(275, 460);

glEnd();

glBegin(GL\_LINE\_LOOP);

glVertex2d(1320, 210);

glVertex2d(1020, 210);

glVertex2d(1170, 290);

glEnd();

glBegin(GL\_QUADS);

glVertex2d(120, 390);

glVertex2d(130, 390);

glVertex2d(275, 455);

glVertex2d(275, 460);

glVertex2d(430, 390);

glVertex2d(420, 390);

glVertex2d(275, 455);

glVertex2d(275, 460);

glVertex2d(1020, 210);

glVertex2d(1030, 210);

glVertex2d(1170, 285);

glVertex2d(1170, 290);

glVertex2d(1320, 210);

glVertex2d(1310, 210);

glVertex2d(1170, 285);

glVertex2d(1170, 290);

glVertex2d(130, 240);

glVertex2d(135, 240);

glVertex2d(135, 350);

glVertex2d(130, 350);

glVertex2d(430, 240);

glVertex2d(425, 240);

glVertex2d(425, 350);

glVertex2d(430, 350);

glVertex2d(130, 270);

glVertex2d(130, 267);

glVertex2d(430, 267);

glVertex2d(430, 270);

for (int i = 1; i < 10; i++)

{

glVertex2d(130 + i \* 30, 240);

glVertex2d(132 + i \* 30, 240);

glVertex2d(132 + i \* 30, 270);

glVertex2d(130 + i \* 30, 270);

}

glVertex2d(1290, 60);

glVertex2d(1290, 190);

glVertex2d(1292, 190);

glVertex2d(1292, 60);

glVertex2d(1310, 60);

glVertex2d(1310, 190);

glVertex2d(1312, 190);

glVertex2d(1312, 60);

glVertex2d(140, 90);

glVertex2d(140, 190);

glVertex2d(142, 190);

glVertex2d(142, 90);

glVertex2d(160, 90);

glVertex2d(160, 190);

glVertex2d(162, 190);

glVertex2d(162, 90);

glEnd();

glBegin(GL\_POINTS);

glVertex2d(1220, 120);

glEnd();

glFlush();

}

int main(int argc, char\* argv[])

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(1440, 480);

glutCreateWindow("Exercise1");

glutDisplayFunc(myDispD); //To be changed for each function

myInit();

glutMainLoop();

return 1;

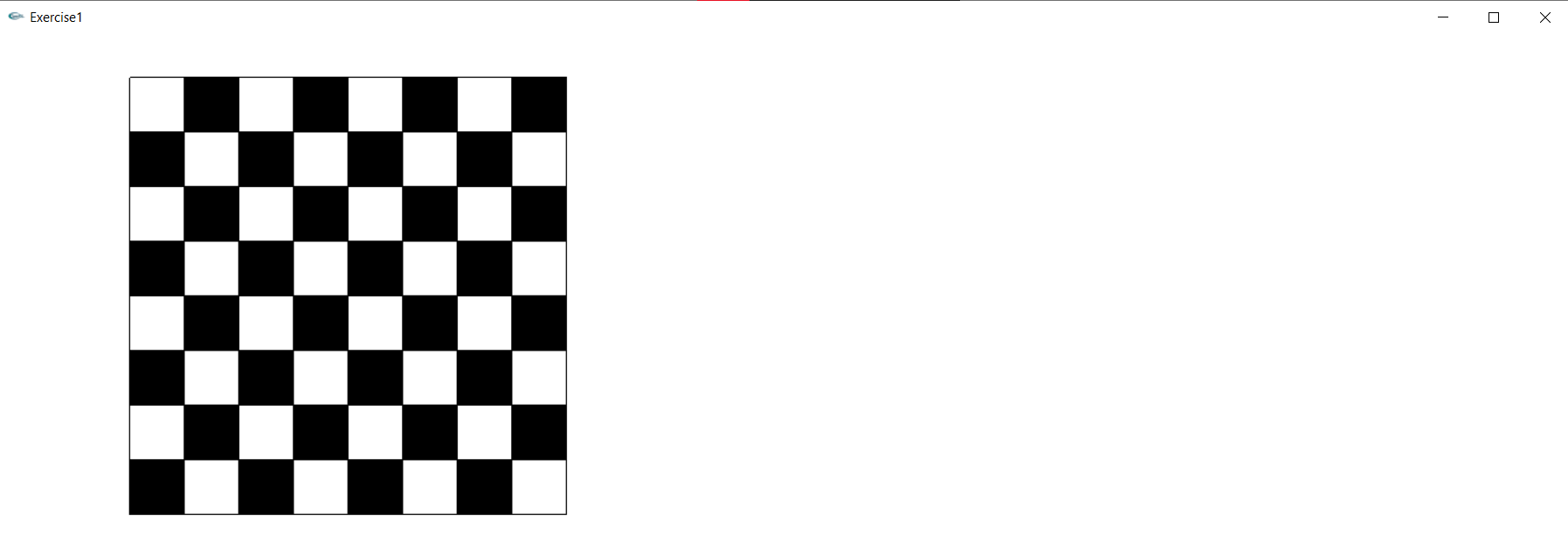
}

**Output:**

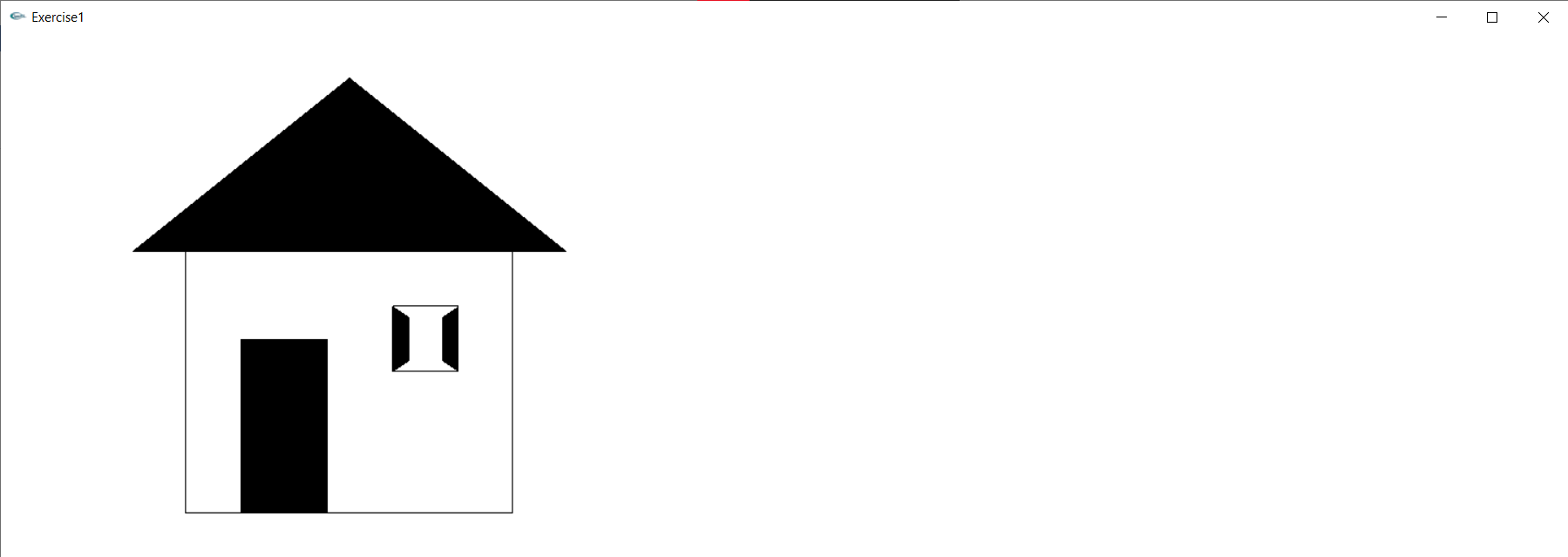
Polygon:

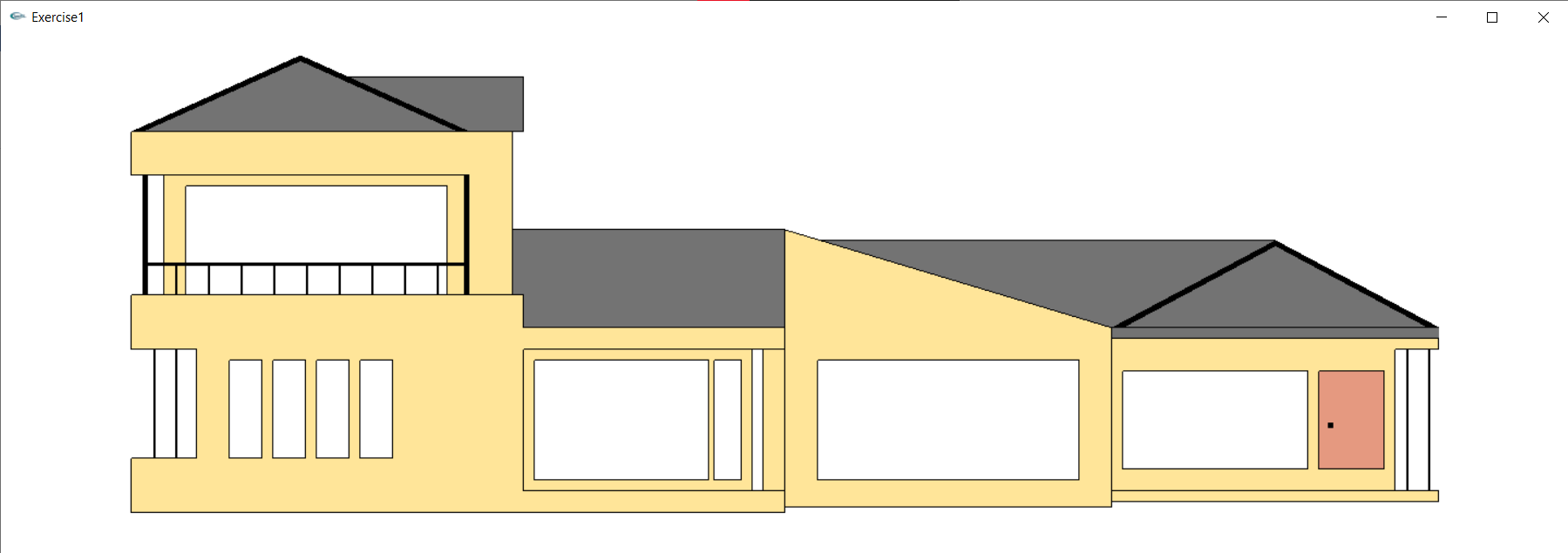


Checker Board:



House:





**Lab Exercise 2 : DDA Line Drawing Algorithm in C++ using OpenGL**

**Aim:**

To plot points that make up the line with endpoints (x0,y0) and (xn,yn) using DDA line drawing algorithm.

**CODE:**

#include<bits/stdc++.h>

#include <GL/glut.h>

using namespace std;

double X1, Y1, X2, Y2;

double arrx1[4], arry1[4], arrx2[4], arry2[4];

float round\_value(float v)

{

return floor(v + 0.5);

}

void myInit() {

glClearColor(1.0, 1.0, 1.0, 0.0);

glColor3f(0.0f, 0.0f, 0.0f);

glPointSize(4);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0,1000.0, 0.0, 1000.0);

}

void LineDDA(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

int j = 0;

while (j < 4)

{

X1 = arrx1[j];

Y1 = arry1[j];

X2 = arrx2[j];

Y2 = arry2[j];

double dx = (X2 - X1);

double dy = (Y2 - Y1);

double steps;

float xInc, yInc, x = X1, y = Y1;

steps = (fabs(dx) > fabs(dy)) ? (fabs(dx)) : (fabs(dy));

float m = dy / dx;

xInc = dx / (float)steps;

yInc = dy / (float)steps;

glBegin(GL\_POINTS);

glVertex2d(x, y);

int k;

for (k = 0; k < steps; k++)

{

x += xInc;

y += yInc;

glVertex2d(round\_value(x), round\_value(y));

}

glEnd();

j++;

glFlush();

}

}

int main(int argc, char\*\* argv)

{

int i = 0;

while (i < 4)

{

cout<<"Enter two end points of the line to be drawn:"<<endl;

cout<<endl<<"Case "<<i + 1<<":";

cout<<endl<<"Enter Point1( X1 , Y1):"<<endl;

cin >> X1;

cin >> Y1;

arrx1[i] = X1;

arry1[i] = Y1;

cout << endl;

cout << endl << "Case " << i + 1 << ":";

cout << endl << "Enter Point2( X2 , Y2):" << endl;

cin >> X2;

cin >> Y2;

arrx2[i] = X2;

arry2[i] = Y2;

i++;

}

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(1000, 1000);

glutCreateWindow("Ex2 DDA Line Drawing");

glutDisplayFunc(LineDDA);

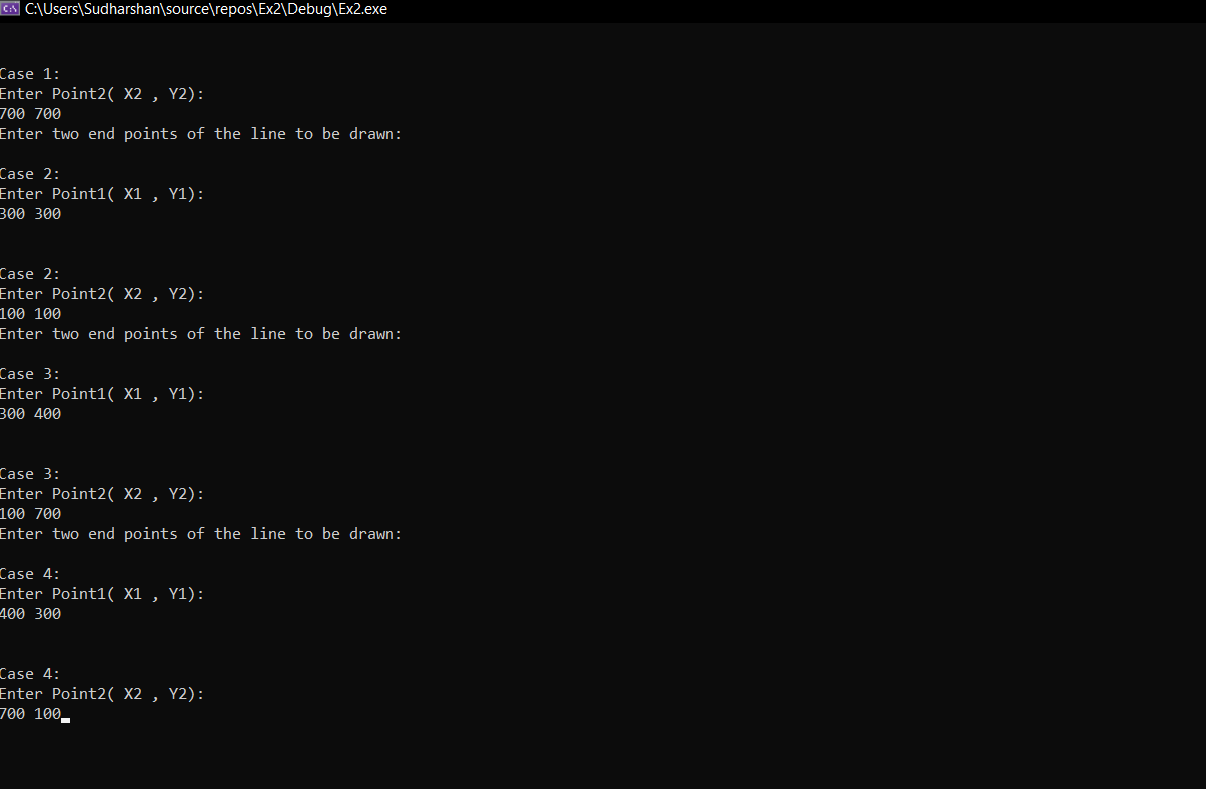
myInit();

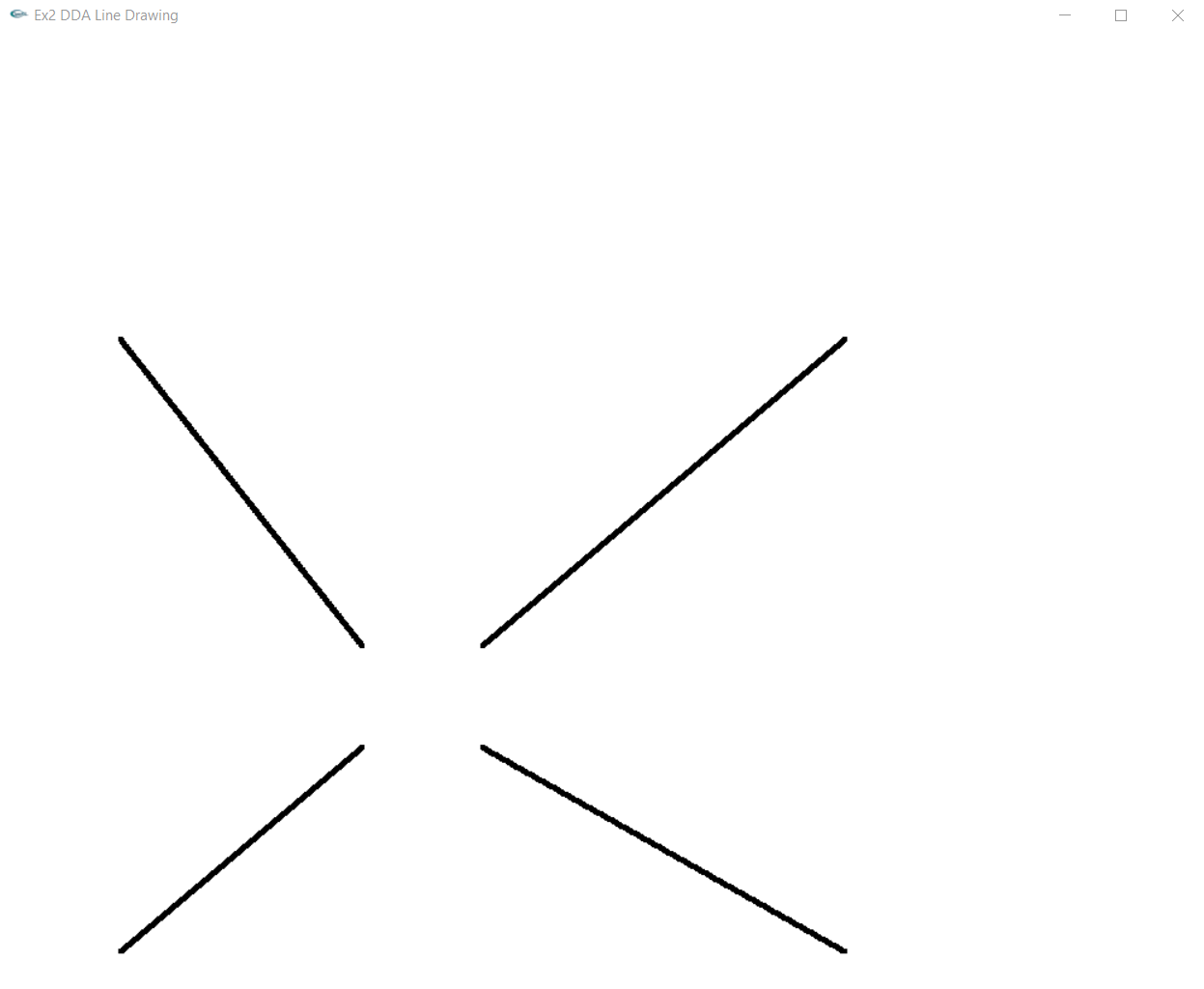
glutMainLoop();

return 1;

}

**OUTPUT:**





**Lab Exercise 3 : Bresenham’s Line Drawing Algorithm in C++ using OpenGL**

**Aim:**

To plot points that make up the line with endpoints (x0,y0) and (xn,yn) using DDA line drawing algorithm.

**CODE:**

#include<GL/glut.h>

#include<bits/stdc++.h>

using namespace std;

double X1, Y1, X2, Y2;

double arrx1[4], arry1[4], arrx2[4], arry2[4];

void myInit() {

glClearColor(1.0, 1.0, 1.0, 0.0);

glColor3f(0.0f, 0.0f, 0.0f);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0, 1000.0, 0.0, 1000.0);

}

void myDisplay() {

glClear(GL\_COLOR\_BUFFER\_BIT);

int j = 0;

while (j < 4) {

X1 = arrx1[j];

Y1 = arry1[j];

X2 = arrx2[j];

Y2 = arry2[j];

double dx = X2 - X1;

double dy = Y2 - Y1;

double d = 2\*dy - dx;

int x, y, Xend;

if (dx < 0) {

x = X2;

y = Y2;

Xend = X1;

}

else {

x = X1;

y = Y1;

Xend = X2;

}

glBegin(GL\_POINTS);

while (x <= Xend) {

glVertex2d(x, y);

if (d < 0) {

d = d + 2\*dy;

x++;

}

else {

d = d + 2\*dy-2\*dx;

x++;

y++;

}

}

glEnd();

j++;

}

glFlush();

}

int main(int argc, char\* argv[]) {

int i = 0;

while (i < 4)

{

cout << "Enter two end points of the line to be drawn:" << endl;

cout << endl << "Case " << i + 1 << ":";

cout << endl << "Enter Point1( X1 , Y1):" << endl;

cin >> X1;

cin >> Y1;

arrx1[i] = X1;

arry1[i] = Y1;

cout << endl;

cout << endl << "Case " << i + 1 << ":";

cout << endl << "Enter Point2( X2 , Y2):" << endl;

cin >> X2;

cin >> Y2;

arrx2[i] = X2;

arry2[i] = Y2;

i++;

}

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(1000, 1000);

glutCreateWindow("Ex2 Bresenham's Line");

glutDisplayFunc(myDisplay);

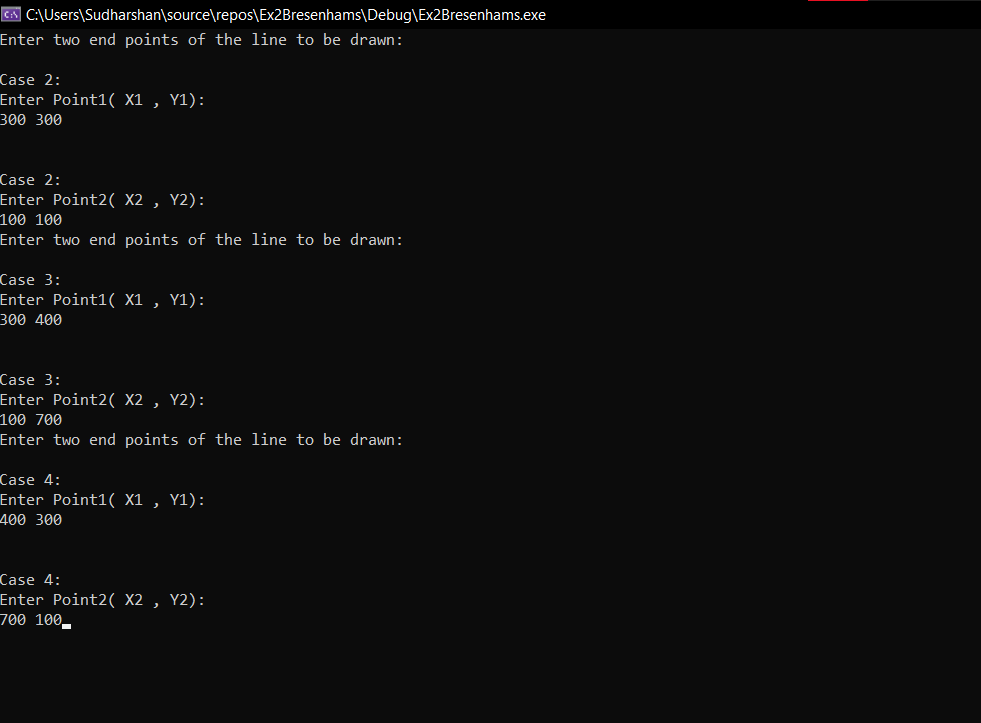
myInit();

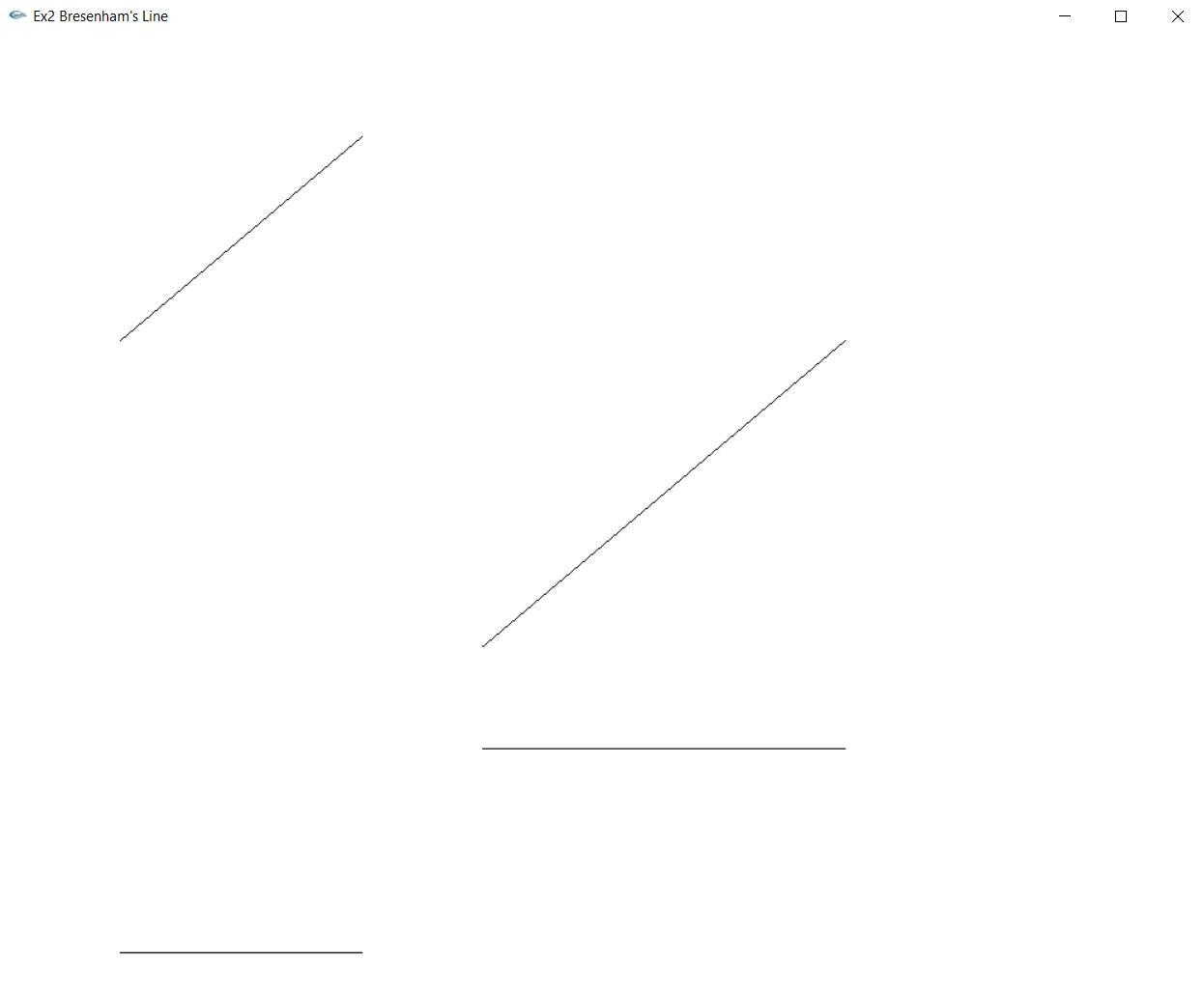
glutMainLoop();

return 1;

}

**OUTPUT:**

****

****

**Lab Exercise 4 : Midpoint Circle Drawing Algorithm in C++ using OpenGL**

**Aim:**

To plot points that make up the circle with center (xc,yc) and radius r using Midpoint circle drawing algorithm.

**Code:**

#include <stdio.h>

#include <iostream>

#include <GL/glut.h>

using namespace std;

int pntX1, pntY1, r;

void plot(int x, int y)

{

glBegin(GL\_POINTS);

glVertex2i(x + pntX1, y + pntY1);

glEnd();

}

void myInit(void)

{

glClearColor(1.0, 1.0, 1.0, 1.0);

glColor3f(0.0f, 0.0f, 0.0f);

glPointSize(4.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0, 1000.0, 0.0, 1000.0);

}

void midPointCircleAlgo()

{

int x = 0;

int y = r;

float decision = 5 / 4 - r;

plot(x, y);

while (y > x)

{

if (decision < 0)

{

x++;

decision += 2 \* x + 1;

}

else

{

y--;

x++;

decision += 2 \* (x - y) + 1;

}

plot(x, y);

plot(x, -y);

plot(-x, y);

plot(-x, -y);

plot(y, x);

plot(-y, x);

plot(y, -x);

plot(-y, -x);

}

}

void myDisplay(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0, 0.5, 0.7);

glPointSize(2.0);

midPointCircleAlgo();

glFlush();

}

int main(int argc, char\*\* argv)

{

cout << "Enter the coordinates of the center: " << endl;

cout << "X-coordinate : "; cin >> pntX1;

cout << "\nY-coordinate : "; cin >> pntY1;

cout << "\nEnter radius : "; cin >> r;

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(640, 480);

glutInitWindowPosition(100, 150);

glutCreateWindow("Circle");

glutDisplayFunc(myDisplay);

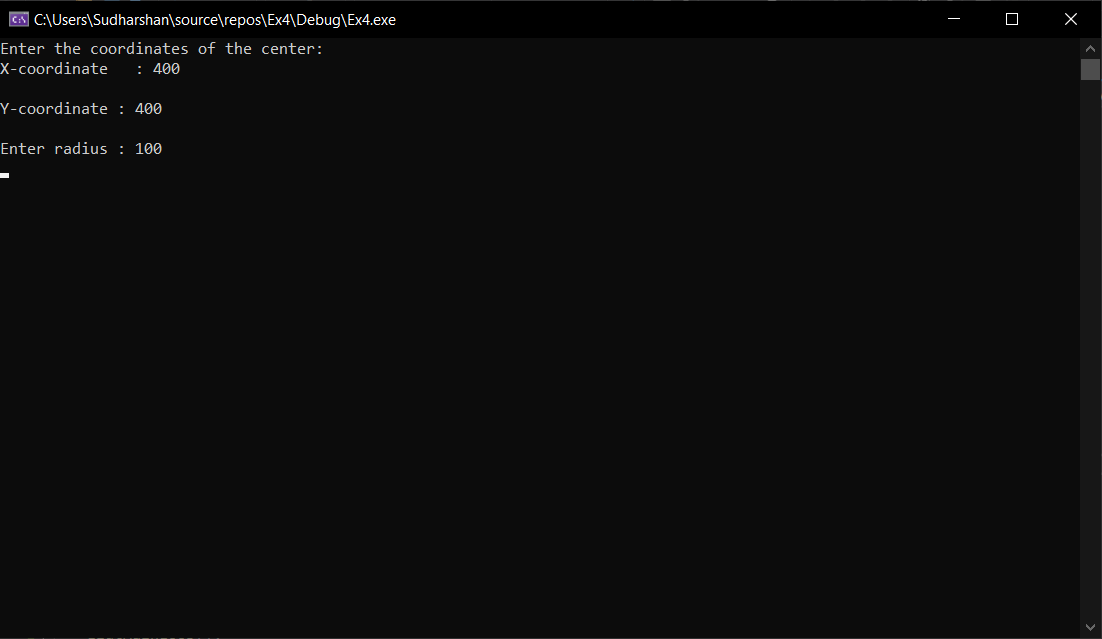
myInit();

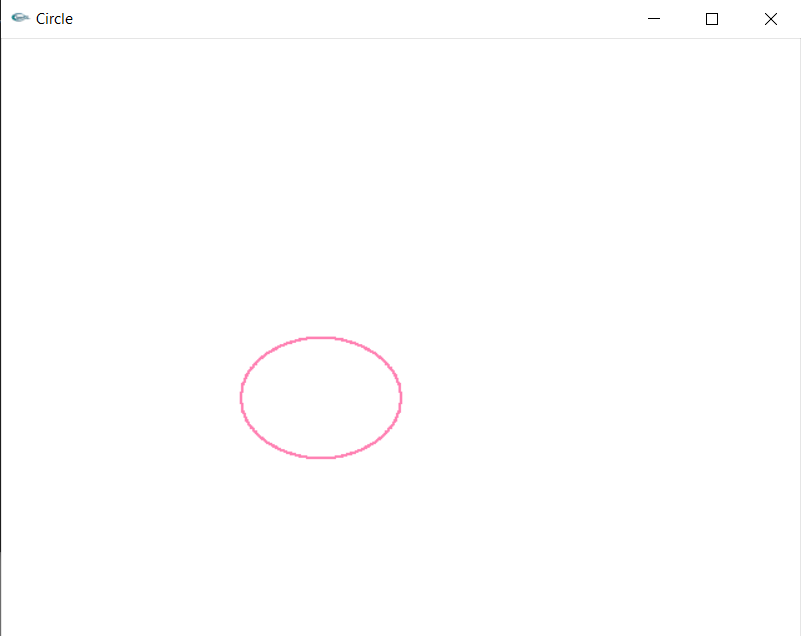
glutMainLoop();

return 0;

}

**Output:**

****

****

**Lab Exercise 5: 2D Transformations in C++ using OpenGL**

**Aim:**

To apply the 2D transformations on objects and to render the final output along with the original object.

**CODE:**

#include<bits/stdc++.h>

#include <GL/glut.h>

constexpr auto PI = 3.14;

using namespace std;

vector<pair<int, int>> coords;

int n;

int tx, ty;

int xr, yr;

int xf, yf;

double sx, sy;

double ang, angRad;

double shx, shy;

void myInit(void)

{

glClearColor(1.0, 1.0, 1.0, 1.0);

glColor3f(0.0f, 0.0f, 0.0f);

glPointSize(4.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(-500, 500, -500, 500);

}

void DrawCartesianPlane() {

glBegin(GL\_LINES);

glColor3f(0.0, 0.0, 0.0);

glVertex2d(-500, 0);

glVertex2d(500, 0);

glVertex2d(0, -500);

glVertex2d(0, 500);

glEnd();

}

void drawPolygon()

{

glBegin(GL\_LINE\_LOOP);

glColor3f(0.0, 0.0, 0.0);

for (int i = 0; i < n; i++)

{

glVertex2d(coords[i].first, coords[i].second);

}

glEnd();

}

void drawPolygonTrans()

{

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 0.0, 0.0);

for (int i = 0; i < n; i++)

{

glVertex2d(coords[i].first+tx, coords[i].second+ty);

}

glEnd();

}

void rotatePolygonOrigin() {

glBegin(GL\_LINE\_LOOP);

glColor3f(0.0, 0.0, 1.0);

for (int i = 0; i < n; i++)

{

glVertex2d(round(coords[i].first \* cos(angRad)-coords[i].second\*sin(angRad)), round(coords[i].first \* sin(angRad)+ coords[i].second \* cos(angRad)));

}

glEnd();

}

void rotatePolygonFixed() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 0.0, 1.0);

vector<pair<int, int>> newCoords;

vector<vector<double>> rotatingMatrix(3, vector<double>(3, 0));

vector<double> curpoint(3, 0), matProduct(3, 0);

rotatingMatrix[0][0] = cos(angRad);

rotatingMatrix[1][1] = cos(angRad);

rotatingMatrix[2][2] = 1;

rotatingMatrix[1][0] = sin(angRad);

rotatingMatrix[0][1] = -1\*sin(angRad);

rotatingMatrix[0][2] = xr \* (1 - cos(angRad)) + yr \* sin(angRad);

rotatingMatrix[1][2] = yr \* (1 - cos(angRad)) - xr \* sin(angRad);

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += rotatingMatrix[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void scalePolygonOrigin() {

glBegin(GL\_LINE\_LOOP);

glColor3f(0.0, 1.0, 0.0);

for (int i = 0; i < n; i++)

{

glVertex2d(round(coords[i].first\*sx), round(coords[i].second\*sy));

}

glEnd();

}

void scalePolygonFixed() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 1.0, 0.0);

vector<pair<int,int>> newCoords;

vector<vector<double>> scalingMatrix(3, vector<double>(3, 0));

vector<double> curpoint(3,0), matProduct(3, 0);

scalingMatrix[0][0] = sx;

scalingMatrix[1][1] = sy;

scalingMatrix[2][2] = 1;

scalingMatrix[0][2] = xf \* (1 - sx);

scalingMatrix[1][2] = yf \* (1 - sy);

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += scalingMatrix[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void reflection\_Xaxis() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 0.5, 0.0);

vector<pair<int, int>> newCoords;

vector<vector<double>> reflectMatrix(3, vector<double>(3, 0));

vector<double> curpoint(3, 0), matProduct(3, 0);

reflectMatrix[0][0] = 1;

reflectMatrix[1][1] = -1;

reflectMatrix[2][2] = 1;

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += reflectMatrix[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void reflection\_Yaxis() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 0.5, 0.0);

vector<pair<int, int>> newCoords;

vector<vector<double>> reflectMatrix(3, vector<double>(3, 0));

vector<double> curpoint(3, 0), matProduct(3, 0);

reflectMatrix[0][0] = -1;

reflectMatrix[1][1] = 1;

reflectMatrix[2][2] = 1;

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += reflectMatrix[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void reflection\_origin() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 0.5, 0.0);

vector<pair<int, int>> newCoords;

vector<vector<double>> reflectMatrix(3, vector<double>(3, 0));

vector<double> curpoint(3, 0), matProduct(3, 0);

reflectMatrix[0][0] = -1;

reflectMatrix[1][1] = -1;

reflectMatrix[2][2] = 1;

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += reflectMatrix[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void reflection\_XeqYline() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 0.5, 0.0);

vector<pair<int, int>> newCoords;

vector<vector<double>> reflectMatrix(3, vector<double>(3, 0));

vector<double> curpoint(3, 0), matProduct(3, 0);

reflectMatrix[0][1] = 1;

reflectMatrix[1][0] = 1;

reflectMatrix[2][2] = 1;

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += reflectMatrix[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void x\_directionShear() {

glBegin(GL\_LINE\_LOOP);

glColor3f(0.0, 1.0, 1.0);

vector<pair<int, int>> newCoords;

vector<vector<double>> shearMatrix(3, vector<double>(3, 0));

vector<double> curpoint(3, 0), matProduct(3, 0);

shearMatrix[0][0] = 1;

shearMatrix[1][1] = 1;

shearMatrix[2][2] = 1;

shearMatrix[0][1] = shx;

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += shearMatrix[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void y\_directionShear() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 0.0, 0.5);

vector<pair<int, int>> newCoords;

vector<vector<double>> shearMatrix(3, vector<double>(3, 0));

vector<double> curpoint(3, 0), matProduct(3, 0);

shearMatrix[0][0] = 1;

shearMatrix[1][1] = 1;

shearMatrix[2][2] = 1;

shearMatrix[1][0] = shy;

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += shearMatrix[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void myDisplay()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0, 0.0, 0.0);

DrawCartesianPlane();

drawPolygon();

drawPolygonTrans();

scalePolygonOrigin();

scalePolygonFixed();

rotatePolygonOrigin();

rotatePolygonFixed();

reflection\_Xaxis();

reflection\_Yaxis();

reflection\_origin();

reflection\_XeqYline();

x\_directionShear();

y\_directionShear();

glFlush();

}

int main(int argc, char\*\* argv)

{

cout << "Specify polygon Dimensions" << endl;

cout << "Enter the number of vertices : ";

cin >> n;

int x, y;

for (int i = 0; i < n; i++)

{

cout << "Enter vertex " << i + 1 << "(x,y) : ";

cin >> x >> y;

coords.push\_back(make\_pair(x,y));

}

cout << "Enter the translation factor for X and Y: ";

cin >> tx >> ty;

cout << "Enter the angle of rotation: ";

cin >> ang;

angRad = ang \* PI / 180;

cout << "Enter the point to rotate about: ";

cin >> xr >> yr;

cout << "Enter the Scaling factor for X and Y : ";

cin >> sx >> sy;

cout << "Enter the point to scale (x,y): ";

cin >> xf >> yf;

cout << "Enter shear parameter for x-direction shear: ";

cin >> shx;

cout << "Enter shear parameter for y-direction shear: ";

cin >> shy;

cout << "Original polygon -> Black Color" << endl;

cout << "Translated polygon -> Red Color" << endl;

cout << "Rotated Polygon : " << endl;

cout << " about origin -> Dark Blue Color" << endl;

cout << " about fixed point -> Purple Color" << endl;

cout << "Scaled Polygon : " << endl;

cout << " about origin -> Green Color" << endl;

cout << " about fixed point -> Yellow Color" << endl;

cout << "Reflection Polygons -> Orange Color" << endl;

cout << "X-Direction Shear -> Light Blue Color" << endl;

cout << "Y-Direction Shear -> Pink Color" << endl;

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(1000, 1000);

glutCreateWindow("Transformation");

glutDisplayFunc(myDisplay);

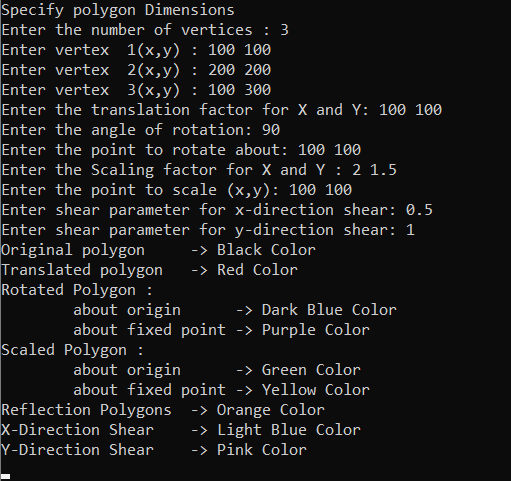
myInit();

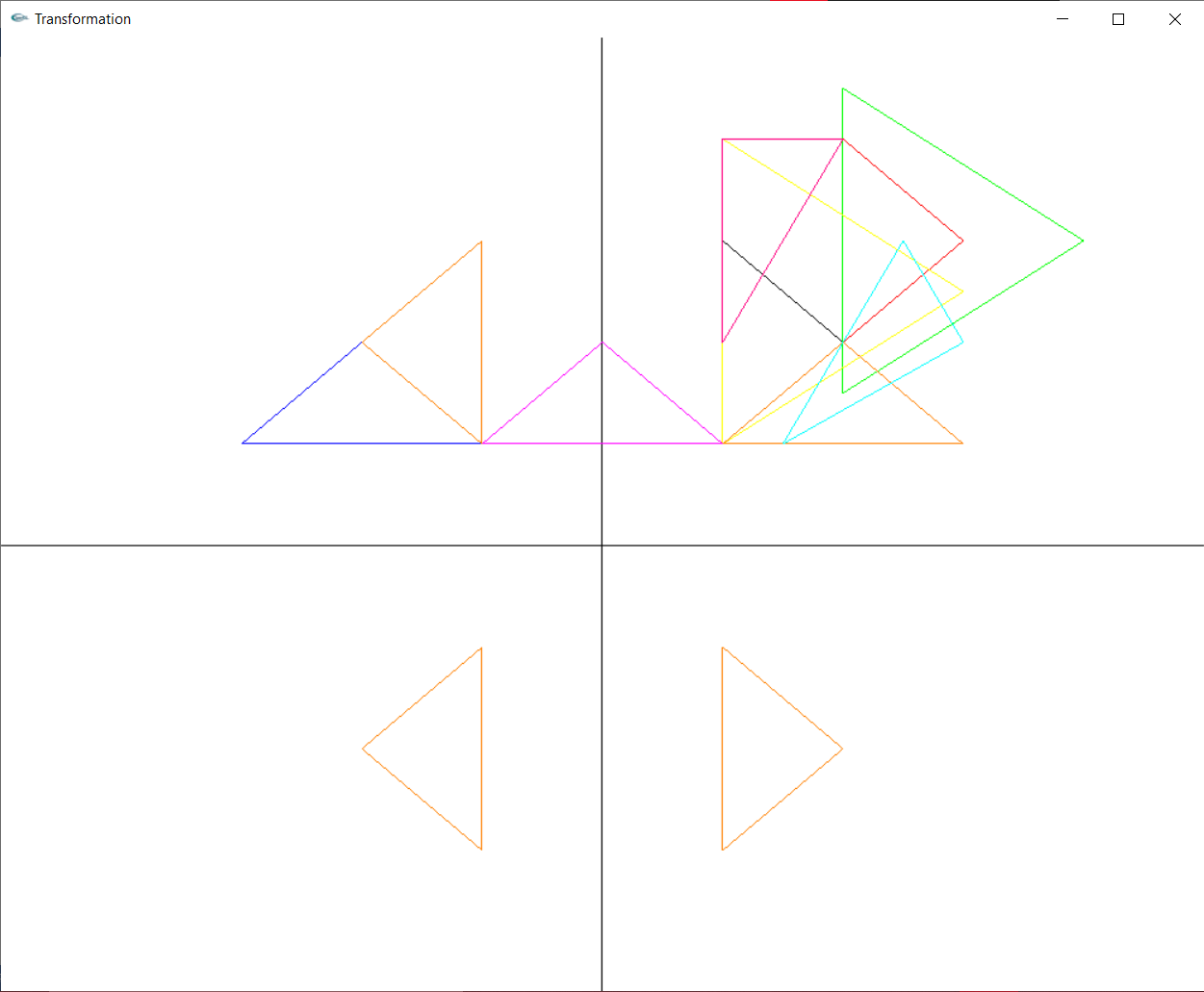
glutMainLoop();

return 0;

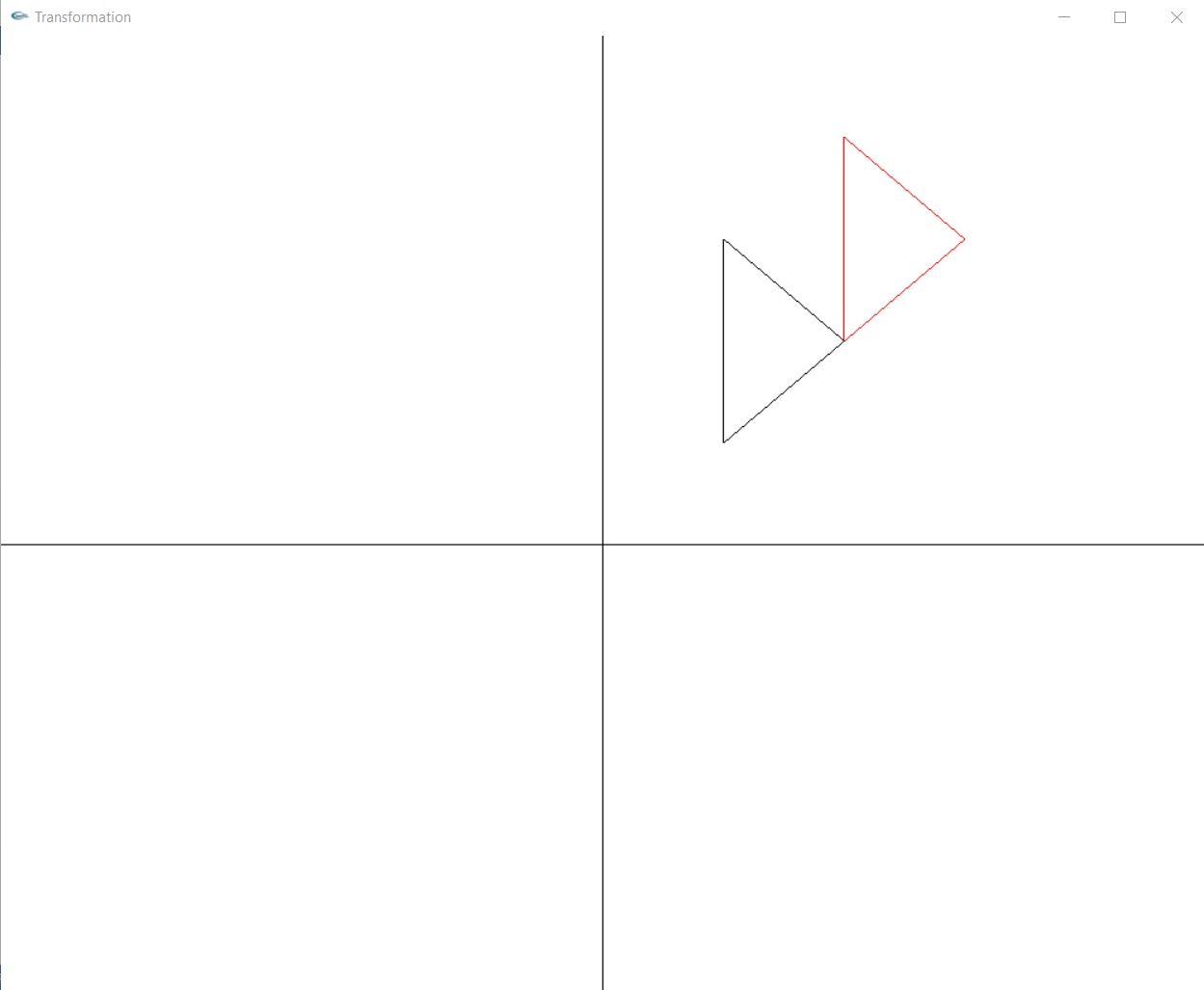
}

**OUTPUT:**

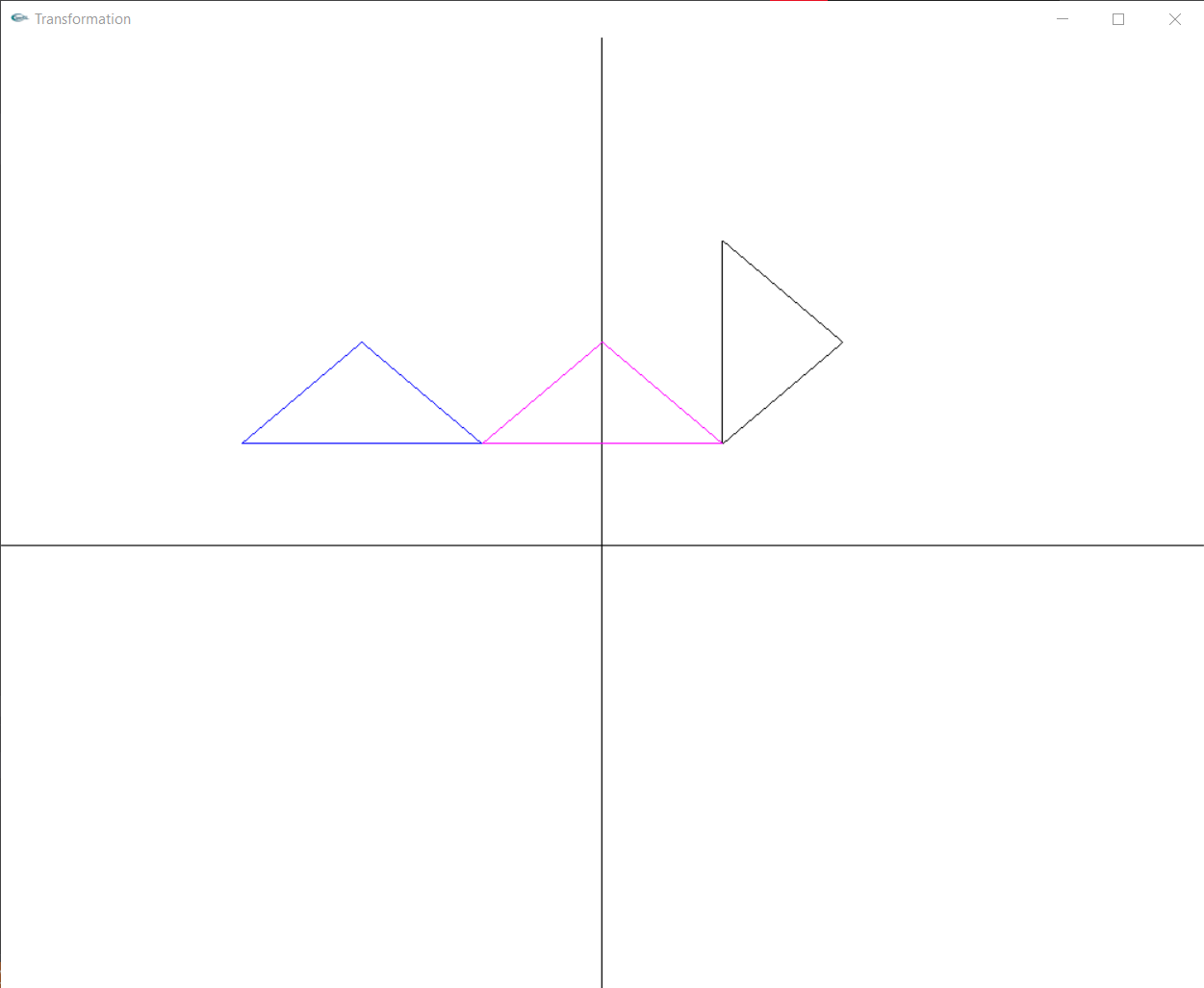
****

****

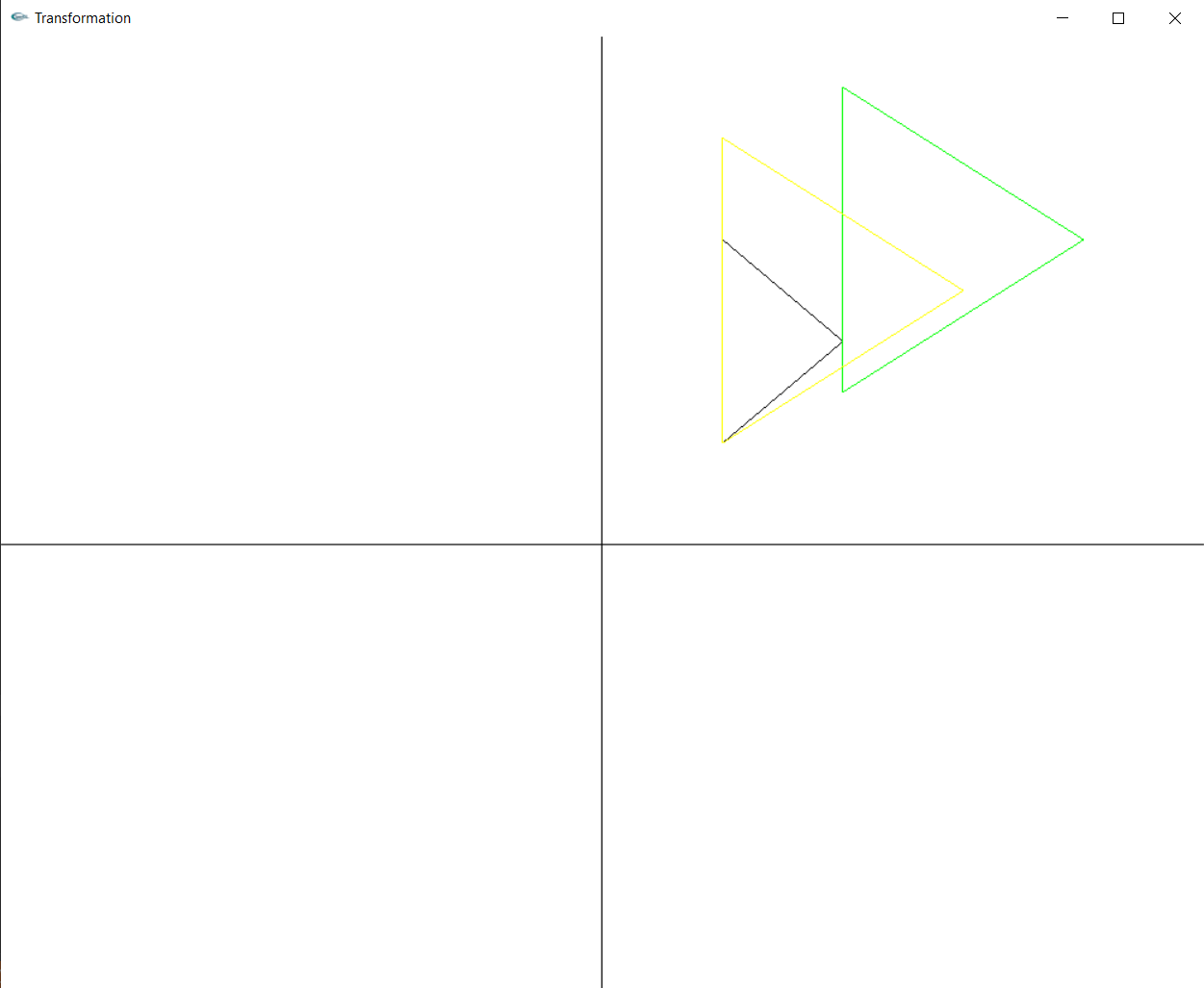
Translation:



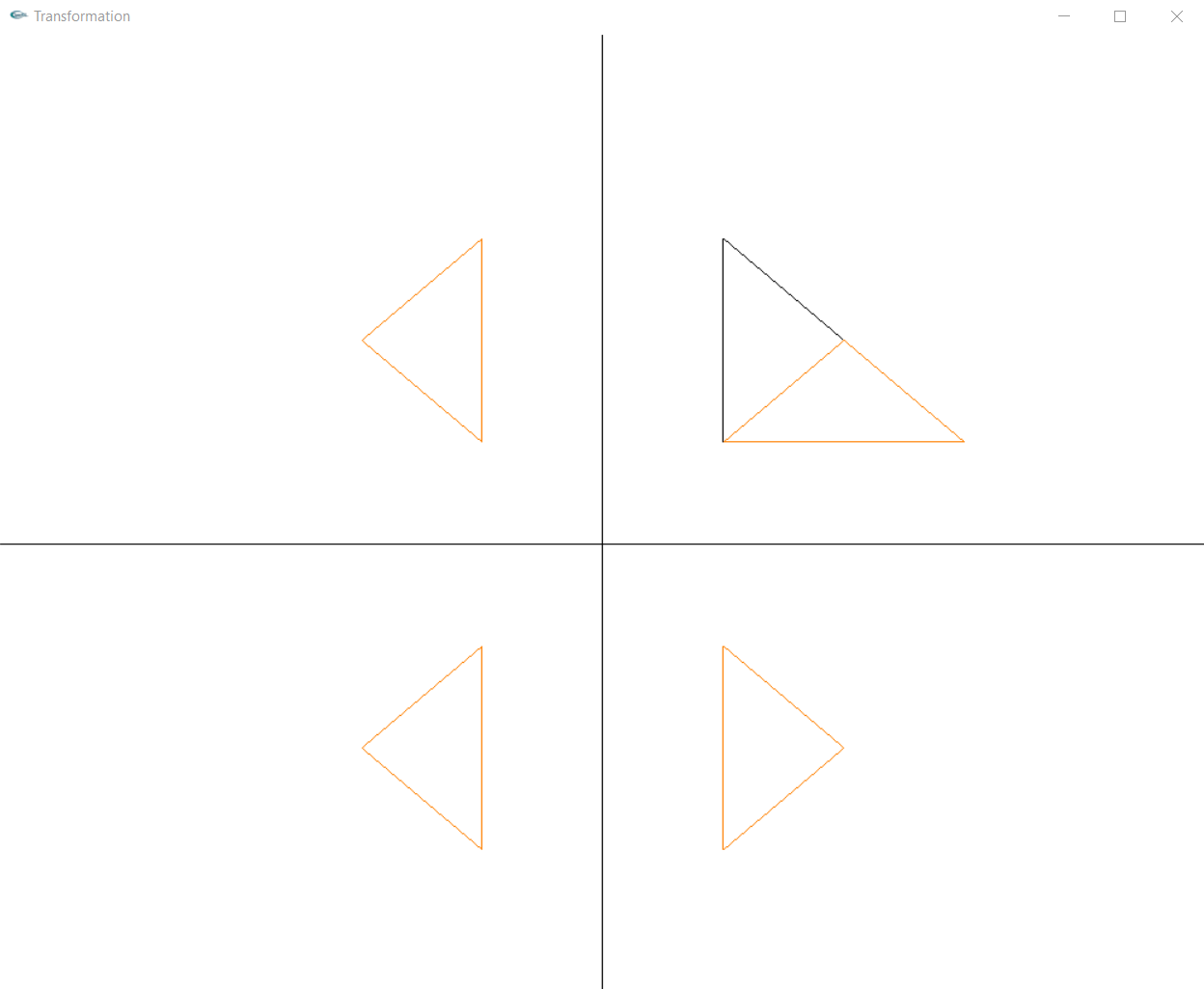
Rotation:



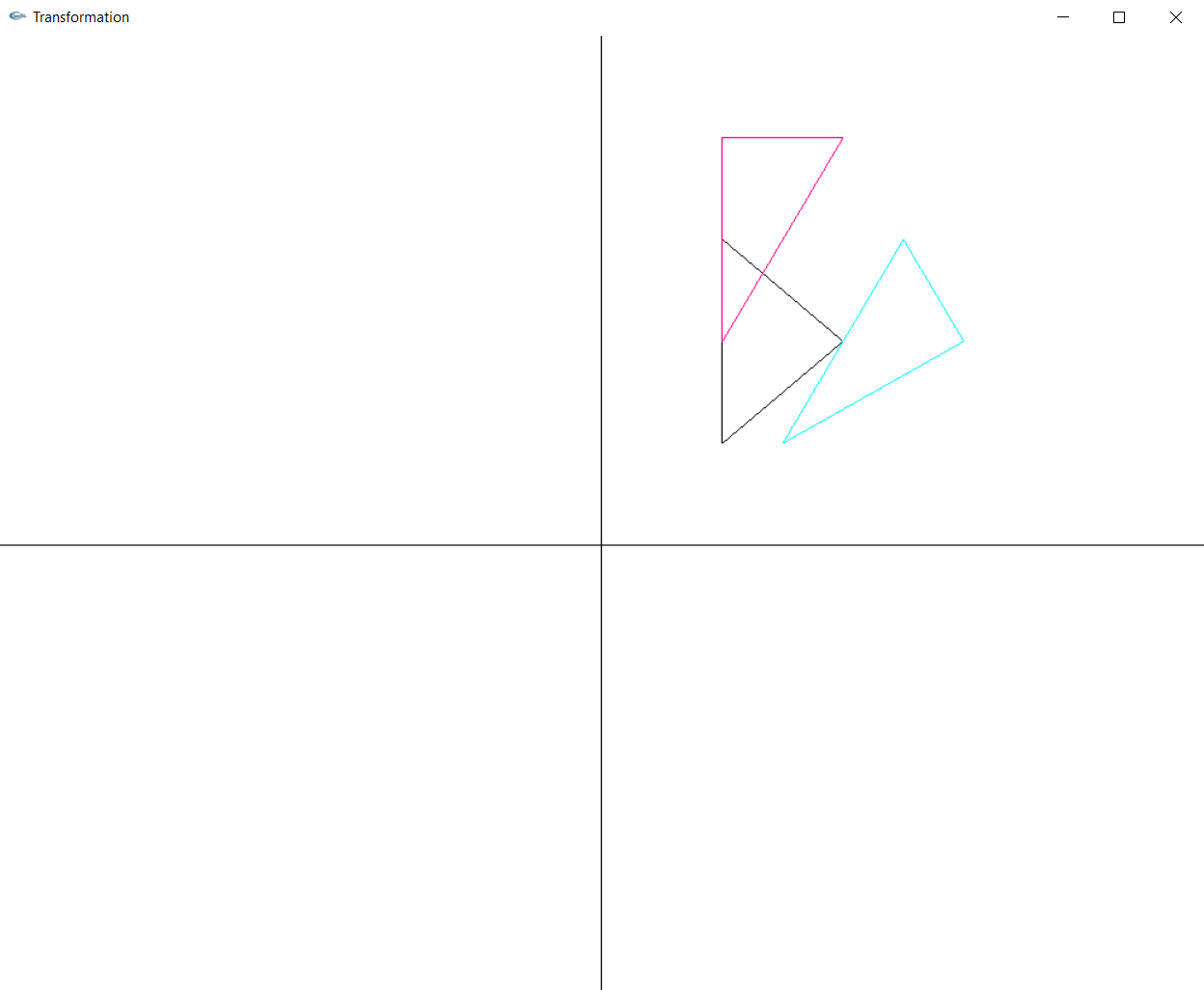
Scaling:



Reflection:



Shearing:



**Lab Exercise 6: 2D Composite Transformations and Windowing**

**Aim:**

To apply 2d composite transformations and window to viewport transformation on any 2d object.

**Part A: 2D composite Transformations**

**Code:**

#include<gl/glut.h>

#include<bits/stdc++.h>

using namespace std;

constexpr auto PI = 3.14;

int n;

vector<pair<int, int>> coords;

int tx, ty;

int xr, yr;

int xf, yf;

double sx, sy;

double ang, angRad;

double shx, shy;

int opr1, opr2, rfl, sh, shd;

vector<vector<double>> T(3, vector<double>(3, 0));

void myInit(void) {

glClearColor(1.0, 1.0, 1.0, 1.0);

glColor3f(0.0f, 0.0f, 0.0f);

glPointSize(4.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(-500, 500, -500, 500);

}

void DrawCartesianPlane() {

glBegin(GL\_LINES);

glColor3f(0.0, 0.0, 0.0);

glVertex2d(-500, 0);

glVertex2d(500, 0);

glVertex2d(0, -500);

glVertex2d(0, 500);

glEnd();

}

void drawPolygon()

{

glBegin(GL\_LINE\_LOOP);

glColor3f(0.0, 0.0, 0.0);

for (int i = 0; i < n; i++)

{

glVertex2d(coords[i].first, coords[i].second);

}

glEnd();

}

vector<vector<double>> drawPolygonTrans()

{

vector<vector<double>> transMatrix(3,vector<double>(3,0));

transMatrix[0][0] = 1;

transMatrix[1][1] = 1;

transMatrix[2][2] = 1;

transMatrix[0][2] = tx;

transMatrix[1][2] = ty;

return transMatrix;

}

vector<vector<double>> rotatePolygonFixed() {

vector<vector<double>> rotatingMatrix(3, vector<double>(3, 0));

rotatingMatrix[0][0] = cos(angRad);

rotatingMatrix[1][1] = cos(angRad);

rotatingMatrix[2][2] = 1;

rotatingMatrix[1][0] = sin(angRad);

rotatingMatrix[0][1] = -1 \* sin(angRad);

rotatingMatrix[0][2] = xr \* (1 - cos(angRad)) + yr \* sin(angRad);

rotatingMatrix[1][2] = yr \* (1 - cos(angRad)) - xr \* sin(angRad);

return rotatingMatrix;

}

vector<vector<double>> scalePolygonFixed() {

vector<vector<double>> scalingMatrix(3, vector<double>(3, 0));

scalingMatrix[0][0] = sx;

scalingMatrix[1][1] = sy;

scalingMatrix[2][2] = 1;

scalingMatrix[0][2] = xf \* (1 - sx);

scalingMatrix[1][2] = yf \* (1 - sy);

return scalingMatrix;

}

vector<vector<double>> reflection\_Xaxis() {

vector<vector<double>> reflectMatrix(3, vector<double>(3, 0));

reflectMatrix[0][0] = 1;

reflectMatrix[1][1] = -1;

reflectMatrix[2][2] = 1;

return reflectMatrix;

}

vector<vector<double>> reflection\_Yaxis() {

vector<vector<double>> reflectMatrix(3, vector<double>(3, 0));

reflectMatrix[0][0] = -1;

reflectMatrix[1][1] = 1;

reflectMatrix[2][2] = 1;

return reflectMatrix;

}

vector<vector<double>> reflection\_origin() {

vector<vector<double>> reflectMatrix(3, vector<double>(3, 0));

reflectMatrix[0][0] = -1;

reflectMatrix[1][1] = -1;

reflectMatrix[2][2] = 1;

return reflectMatrix;

}

vector<vector<double>> reflection\_XeqYline() {

vector<vector<double>> reflectMatrix(3, vector<double>(3, 0));

reflectMatrix[0][1] = 1;

reflectMatrix[1][0] = 1;

reflectMatrix[2][2] = 1;

return reflectMatrix;

}

vector<vector<double>> x\_directionShear() {

vector<vector<double>> shearMatrix(3, vector<double>(3, 0));

shearMatrix[0][0] = 1;

shearMatrix[1][1] = 1;

shearMatrix[2][2] = 1;

shearMatrix[0][1] = shx;

return shearMatrix;

}

vector<vector<double>> y\_directionShear() {

vector<vector<double>> shearMatrix(3, vector<double>(3, 0));

shearMatrix[0][0] = 1;

shearMatrix[1][1] = 1;

shearMatrix[2][2] = 1;

shearMatrix[1][0] = shy;

return shearMatrix;

}

void executeTransformMatrix(int opr,int oprn) {

vector<vector<double>> mat;

if (opr == 1) {

mat = drawPolygonTrans();

}

else if (opr == 2) {

mat = rotatePolygonFixed();

}

else if (opr == 3) {

mat = scalePolygonFixed();

}

else if (opr == 4) {

if (rfl == 1) mat = reflection\_Xaxis();

else if (rfl == 2) mat = reflection\_Yaxis();

else if (rfl == 3) mat = reflection\_origin();

else if (rfl == 4) mat = reflection\_XeqYline();

}

else {

if (shd == 1) mat = x\_directionShear();

else mat = y\_directionShear();

}

if (oprn == 1) T = mat;

else {

vector<vector<double>> res(3,vector<double>(3,0));

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

res[i][j] += T[i][k] \* mat[k][j];

}

}

}

T = res;

}

}

void drawTransformedPolygon() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0, 0.0, 0.5);

vector<pair<int, int>> newCoords;

vector<double> curpoint(3, 0), matProduct(3, 0);

for (int i = 0; i < n; i++) {

curpoint[0] = coords[i].first;

curpoint[1] = coords[i].second;

curpoint[2] = 1;

matProduct[0] = 0;

matProduct[1] = 0;

matProduct[2] = 0;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

matProduct[j] += T[j][k] \* curpoint[k];

}

}

newCoords.push\_back(make\_pair(round(matProduct[0]), round(matProduct[1])));

}

for (int i = 0; i < n; i++)

{

glVertex2d(newCoords[i].first, newCoords[i].second);

}

glEnd();

}

void myDisplay() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0, 0.0, 0.0);

DrawCartesianPlane();

drawPolygon();

executeTransformMatrix(opr1, 1);

executeTransformMatrix(opr2, 2);

drawTransformedPolygon();

glFlush();

}

void getDetails(int opr) {

if (opr == 1) {

cout << "Enter the translation factor for X and Y: ";

cin >> tx >> ty;

}

else if (opr == 2) {

cout << "Enter the angle of rotation: ";

cin >> ang;

angRad = ang \* PI / 180;

cout << "Enter the point to rotate about: ";

cin >> xr >> yr;

}

else if (opr == 3) {

cout << "Enter the Scaling factor for X and Y : ";

cin >> sx >> sy;

cout << "Enter the point to scale (x,y): ";

cin >> xf >> yf;

}

else if (opr == 4) {

cout << "Enter axis about reflection"<<endl;

cout << "1. X-axis" << endl;

cout << "2. Y-axis" << endl;

cout << "3. origin" << endl;

cout << "4. X=Y line" << endl;

cin >> rfl;

}

else {

cout << "Which direction do you want to shear about?" << endl;

cout << "1.X-direction" << endl;

cout << "2.Y-direction" << endl;

cin >> shd;

cout << "Enter shear parameter: ";

cin >> sh;

if (shd == 1) sh = shx;

else sh = shy;

}

}

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

cout << "Specify Polygon Details" << endl;

cout << "Enter number of vertices : ";

cin >> n;

int x, y;

for (int i = 0; i < n; i++) {

cout << "Enter vertex " << i + 1 << " : ";

cin >> x >> y;

coords.push\_back(make\_pair(x, y));

}

cout << "What 2 operations do you want to perform?" << endl;

cout << "1.Translate" << endl;

cout << "2.Rotation" << endl;

cout << "3.Scaling" << endl;

cout << "4.Reflection" << endl;

cout << "5.Shearing" << endl;

cout << endl << "Enter operation 1: ";

cin >> opr1;

getDetails(opr1);

cout << "Enter operation 2: ";

cin >> opr2;

getDetails(opr2);

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(1000, 1000);

glutCreateWindow("Transformation");

glutDisplayFunc(myDisplay);

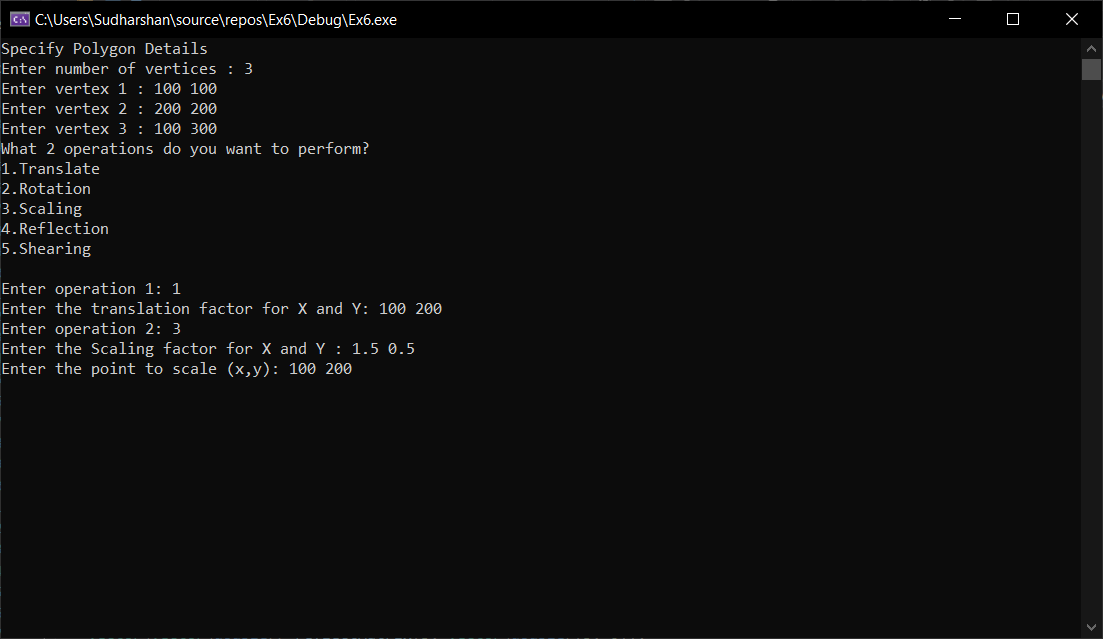
myInit();

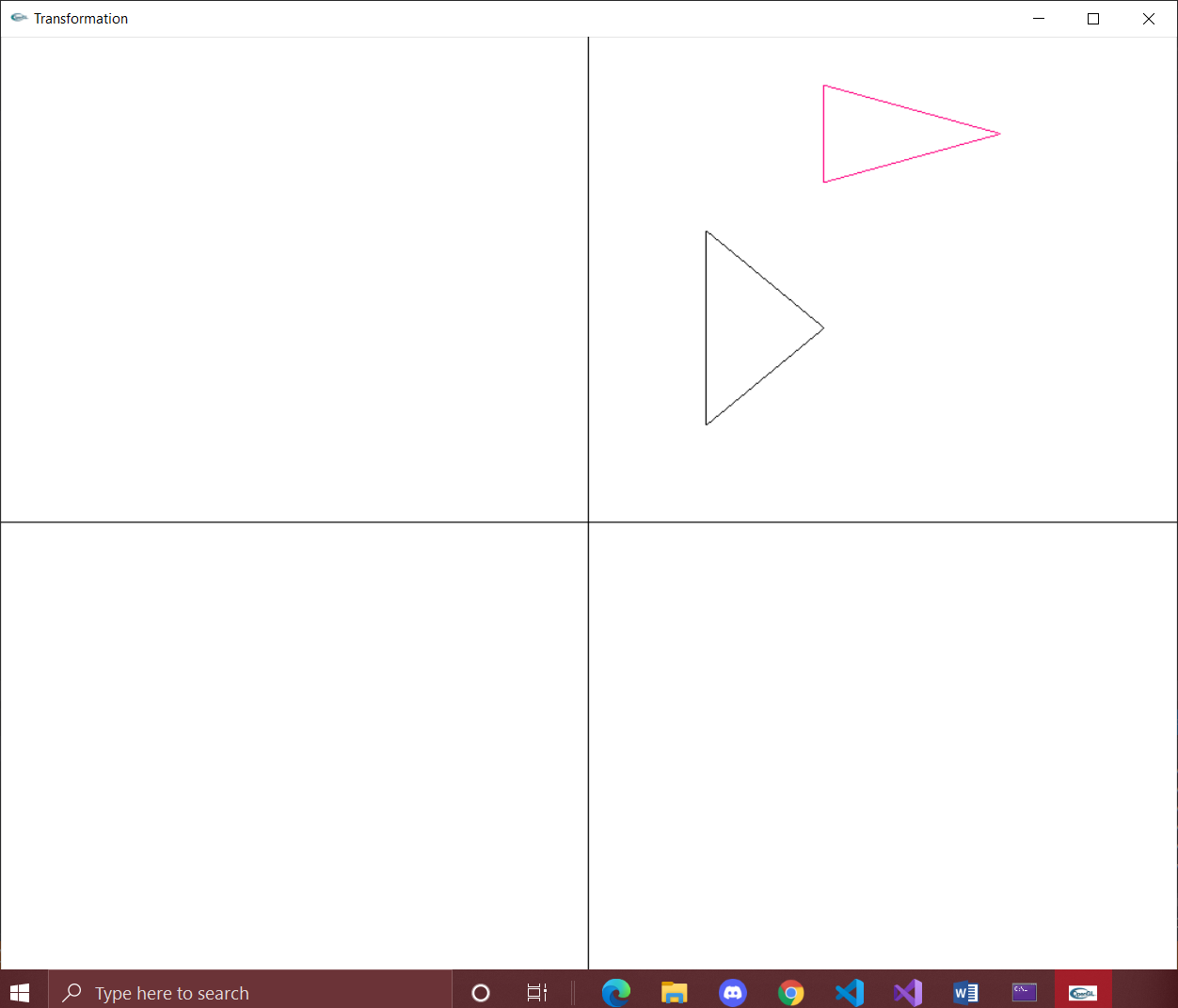
glutMainLoop();

return 0;

}

**Output:**





**PartB: Window to Viewport**

**Code:**

#include<gl/glut.h>

#include<vector>

#include<utility>

#include<iostream>

#include<math.h>

using namespace std;

int n;

vector<pair<int, int>> coords;

int xv\_min, xv\_max, yv\_min, yv\_max;

int xw\_min = 0, yw\_min = 0, xw\_max = 1000, yw\_max = 1000;

double sx, sy;

void myInit(void) {

glClearColor(1.0, 1.0, 1.0, 1.0);

glColor3f(0.0f, 0.0f, 0.0f);

glPointSize(4.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, 1000, 0, 1000);

}

void drawVWPort() {

glBegin(GL\_LINE\_LOOP);

glColor3f(0.0f, 1.0f, 0.0f);

glVertex2d(xv\_min, yv\_min);

glVertex2d(xv\_min, yv\_max);

glVertex2d(xv\_max, yv\_max);

glVertex2d(xv\_max, yv\_min);

glEnd();

}

void drawWPolygon() {

glBegin(GL\_LINE\_LOOP);

for (int i = 0; i < n; i++) {

glVertex2d(coords[i].first, coords[i].second);

}

glEnd();

}

void drawVPolygon() {

glBegin(GL\_LINE\_LOOP);

glColor3f(1.0f, 0.0f, 0.0f);

for (int i = 0; i < n; i++) {

glVertex2d(xv\_min + (coords[i].first - xw\_min) \* sx, yv\_min + (coords[i].second - yw\_min) \* sy);

}

glEnd();

}

void myDisplay() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0, 0.0, 0.0);

drawWPolygon();

drawVPolygon();

drawVWPort();

glFlush();

}

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

cout << "Enter Ploygon Dimensions" << endl;

cout << "Enter number of vertices: ";

cin >> n;

int x, y;

for (int i = 0; i < n; i++) {

cout << "Enter vertex " << i + 1 << " : ";

cin >> x >> y;

coords.push\_back(make\_pair(x, y));

}

cout << "Enter Viewport details" << endl;

cout << "Enter min x and max x : ";

cin >> xv\_min >> xv\_max;

cout << "Enter min y and max y : ";

cin >> yv\_min >> yv\_max;

sx = (xv\_max - xv\_min) \* 1.0 / (xw\_max - xw\_min);

sy = (yv\_max - yv\_min) \* 1.0 / (yw\_max - yw\_min);

cout << "Black -> original polygon " << endl;

cout << "Red -> transformed polygon" << endl;

cout << "Green -> view port" << endl;

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(1000, 1000);

glutCreateWindow("Transformation");

glutDisplayFunc(myDisplay);

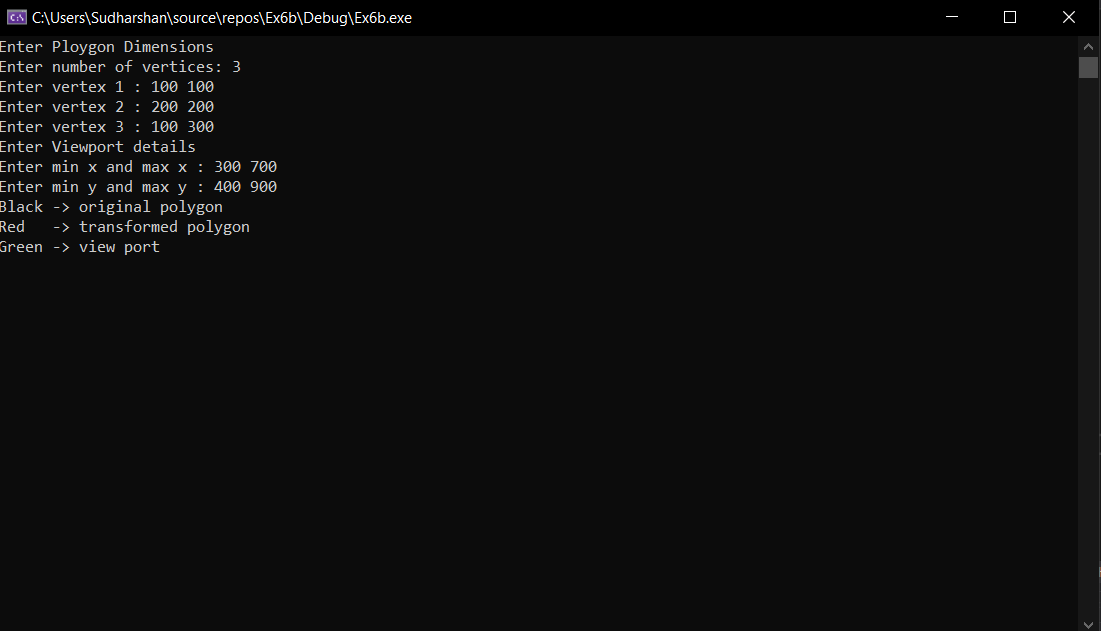
myInit();

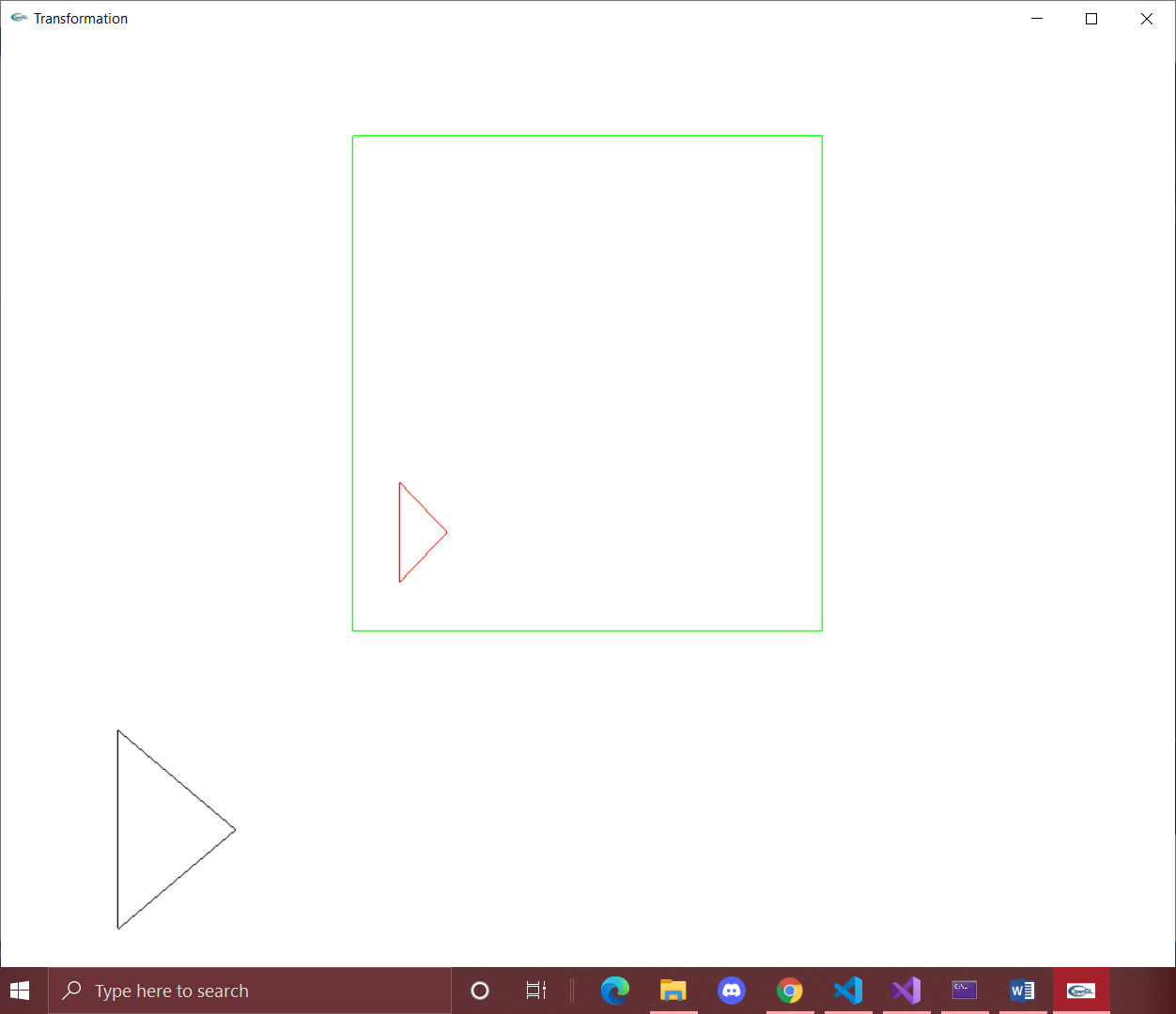
glutMainLoop();

return 0;

}

**Output:**

****



**Lab Exercise 7: Cohen Sutherland Line clipping in C++ using OpenGL**

**Aim:**

To simulate Cohen-Sutherland Line Clipping algorithm and apply it to a line.

**CODE:**

#include<gl/glut.h>

#include<iostream>

#include<utility>

using namespace std;

pair<int, int> P1, P2;

int X1, X2, Y1, Y2;

int xwmin, xwmax, ywmin, ywmax;

void myInit()

{

glClearColor(1.0, 1.0, 1.0, 0.0);

glColor3f(0.0f, 0.0f, 0.0f);

glPointSize(10);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0.0, 640.0, 0.0, 480.0);

}

void drawWindow() {

glBegin(GL\_LINE\_LOOP);

glVertex2d(xwmin, ywmin);

glVertex2d(xwmax, ywmin);

glVertex2d(xwmax, ywmax);

glVertex2d(xwmin, ywmax);

glEnd();

}

void drawOriginal() {

glBegin(GL\_LINES);

glVertex2d(P1.first, P1.second);

glVertex2d(P2.first, P2.second);

glEnd();

}

int getRC(pair<int, int>& P)

{

int rc = 0;

if (P.first < xwmin) rc |= 1;

else if (P.first > xwmax) rc |= 1 << 1;

if (P.second < ywmin) rc |= 1 << 2;

else if (P.second > ywmax) rc |= 1 << 3;

return rc;

}

void findIntersection(pair<int, int>& P, double m, int rc) {

if (rc == 0) return;

// y = ywmax

if ((rc >> 3)&1) {

//x =X1 + (y-Y1)/m

P.second = ywmax;

P.first = X1 + (ywmax - Y1) / m;

return;

}

//y = ywmin

if ((rc >> 2 )& 1) {

//x =X1 + (y-Y1)/m

P.second = ywmin;

P.first = X1 + (ywmin - Y1) / m;

return;

}

// x= xwmax

if ((rc >> 1) & 1) {

//y =Y1 + (x-X1)\*m

P.first = xwmax;

P.second = Y1 + (xwmax - X1) \* m;

return;

}

// x= xwmin

if (rc & 1) {

//y =Y1 + (x-X1)\*m

P.first = xwmin;

P.second = Y1 + (xwmin - X1) \* m;

return;

}

}

void PerformClipping(pair<int, int>& P1, pair<int, int>& P2)

{

int rc1 = getRC(P1), rc2 = getRC(P2);

//Checking for trivial OR

if (int(rc1 | rc2) == 0) {

glBegin(GL\_LINES);

glVertex2d(P1.first, P1.second);

glVertex2d(P2.first, P2.second);

glEnd();

return;

}

else if (int(rc1 & rc2) != 0) return;

double m = (Y2-Y1) \* 1.0 / (X2-X1);

findIntersection(P1, m, rc1);

findIntersection(P2, m, rc2);

PerformClipping(P1, P2);

}

void myDisplay()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0f, 0.0f, 1.0f);

drawWindow();

glColor3f(0.0f, 0.0f, 0.0f);

drawOriginal();

glColor3f(1.0f, 0.0f, 0.0f);

PerformClipping(P1, P2);

glFlush();

}

int main(int argc, char\* argv[])

{

cout << "Enter window properties:" << endl;

cout << "xwmin:";

cin >> xwmin;

cout << "xwmax:";

cin >> xwmax;

cout << "ywmin:";

cin >> ywmin;

cout << "ywmax:";

cin >> ywmax;

int x, y;

cout << endl << "Enter point p1(x,y) :";

cin >> x >> y;

P1.first = x;

P1.second = y;

X1 = x;

Y1 = y;

cout << "Enter point p2(x,y) :";

cin >> x >> y;

P2.first = x;

P2.second = y;

X2 = x;

Y2 = y;

cout << "Blue -> Clipping Window" << endl;

cout << "Black -> Original Line" << endl;

cout << "Red -> Clipped Line" << endl;

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(640, 480);

glutCreateWindow("Cohen Sutherland");

glutDisplayFunc(myDisplay);

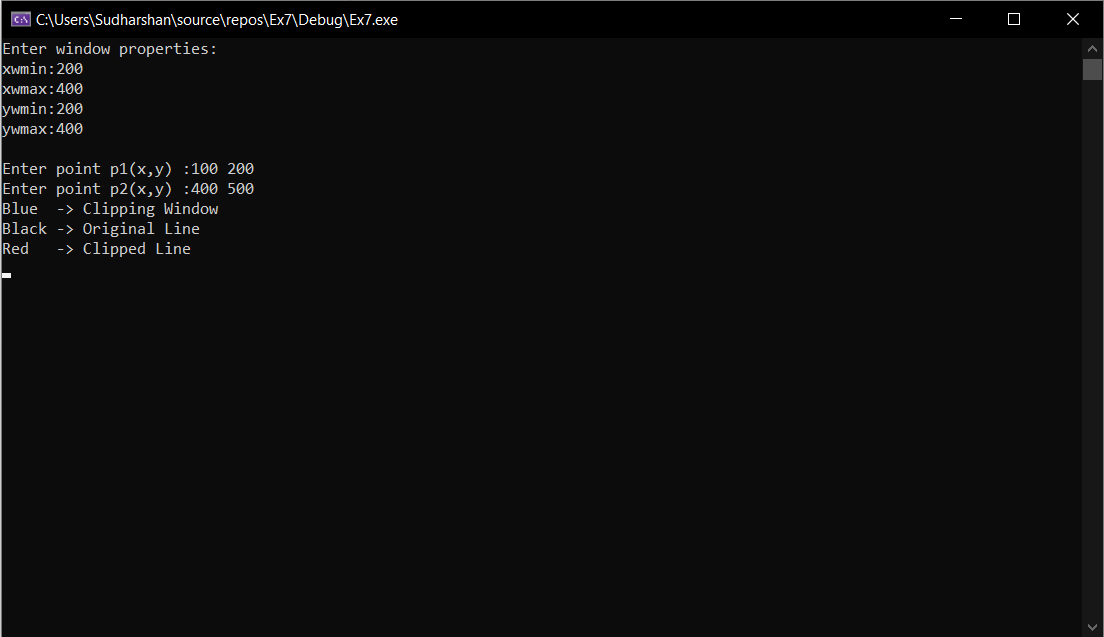
myInit();

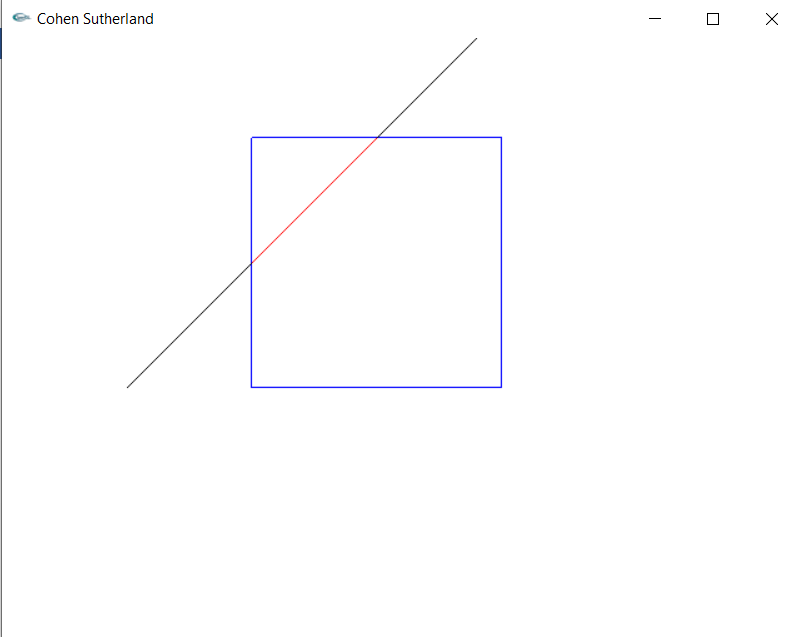
glutMainLoop();

return 1;

}

**OUTPUT:**





**Lab Exercise 8: 3-Dimensional Transformations in C++ using OpenGL**

**Aim:**

To create a 3D object and apply transformations to it using matrix.

**CODE:**

#include<gl/glut.h>

#include<iostream>

#include<utility>

#include<vector>

#include<math.h>

constexpr auto PI = 3.14;

using namespace std;

vector<vector<GLfloat>> coords(8,vector<GLfloat>(3));

int tx, ty, tz;

int ch;

double sx, sy, sz;

int xf, yf, zf;

int RotAxis;

int ang;

float rad;

void myInit()

{

glClearColor(1.0, 1.0, 1.0, 1.0);

glLoadIdentity();

glOrtho(-500.0, 500.0, -500.0, 500.0, -500.0, 500.0);

glEnable(GL\_DEPTH\_TEST);

}

void drawCube() {

glBegin(GL\_QUADS);

glColor3f(1, 0, 0);

glVertex3f(coords[0][0], coords[0][1], coords[0][2]);

glVertex3f(coords[1][0], coords[1][1], coords[1][2]);

glVertex3f(coords[2][0], coords[2][1], coords[2][2]);

glVertex3f(coords[3][0], coords[3][1], coords[3][2]);

glColor3f(0, 1, 0);

glVertex3f(coords[4][0], coords[4][1], coords[4][2]);

glVertex3f(coords[5][0], coords[5][1], coords[5][2]);

glVertex3f(coords[6][0], coords[6][1], coords[6][2]);

glVertex3f(coords[7][0], coords[7][1], coords[7][2]);

glColor3f(0, 0, 1);

glVertex3f(coords[0][0], coords[0][1], coords[0][2]);

glVertex3f(coords[1][0], coords[1][1], coords[1][2]);

glVertex3f(coords[5][0], coords[5][1], coords[5][2]);

glVertex3f(coords[4][0], coords[4][1], coords[4][2]);

glColor3f(1, 0, 1);

glVertex3f(coords[0][0], coords[0][1], coords[0][2]);

glVertex3f(coords[4][0], coords[4][1], coords[4][2]);

glVertex3f(coords[7][0], coords[7][1], coords[7][2]);

glVertex3f(coords[3][0], coords[3][1], coords[3][2]);

glColor3f(0, 1, 1);

glVertex3f(coords[1][0], coords[1][1], coords[1][2]);

glVertex3f(coords[2][0], coords[2][1], coords[2][2]);

glVertex3f(coords[6][0], coords[6][1], coords[6][2]);

glVertex3f(coords[5][0], coords[5][1], coords[5][2]);

glColor3f(1, 1, 0);

glVertex3f(coords[2][0], coords[2][1], coords[2][2]);

glVertex3f(coords[3][0], coords[3][1], coords[3][2]);

glVertex3f(coords[7][0], coords[7][1], coords[7][2]);

glVertex3f(coords[6][0], coords[6][1], coords[6][2]);

glEnd();

}

void Axis() {

glBegin(GL\_LINES);

glColor3f(0, 0, 1);

glVertex3f(0, 0, 0);

glVertex3f(0, 0, 500);

glColor3f(1, 0, 0);

glVertex3f(0, 0, 0);

glVertex3f(500, 0, 0);

glColor3f(0, 1, 0);

glVertex3f(0, 0, 0);

glVertex3f(0, 500, 0);

glEnd();

}

void translate() {

vector<vector<GLfloat>> T(4, vector<GLfloat>(4, 0));

T[0][0] = 1;

T[1][1] = 1;

T[2][2] = 1;

T[3][3] = 1;

T[0][3] = tx;

T[1][3] = ty;

T[2][3] = tz;

for (int c = 0; c < coords.size(); c++)

{

vector<GLfloat> P(4),N(4,0);

P[0] = coords[c][0];

P[1] = coords[c][1];

P[2] = coords[c][2];

P[3] = 1;

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 1; j++) {

N[i] = 0;

for (int k = 0; k < 4; k++) {

N[i] += T[i][k] \* P[k];

}

}

}

coords[c][0] = N[0];

coords[c][1] = N[1];

coords[c][2] = N[2];

}

}

void scale() {

vector<vector<GLfloat>> T(4, vector<GLfloat>(4, 0));

T[0][0] = sx;

T[1][1] = sy;

T[2][2] = sz;

T[3][3] = 1;

T[0][3] = (1 - sx) \* xf;

T[1][3] = (1 - sy) \* yf;

T[2][3] = (1 - sz) \* zf;

for (int c = 0; c < coords.size(); c++)

{

vector<GLfloat> P(4), N(4, 0);

P[0] = coords[c][0];

P[1] = coords[c][1];

P[2] = coords[c][2];

P[3] = 1;

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 1; j++) {

N[i] = 0;

for (int k = 0; k < 4; k++) {

N[i] += T[i][k] \* P[k];

}

}

}

coords[c][0] = N[0];

coords[c][1] = N[1];

coords[c][2] = N[2];

}

}

void rotate() {

vector<vector<GLfloat>> T(4, vector<GLfloat>(4, 0));

switch (RotAxis) {

case 1: T = {

{1, 0, 0, 0},

{0, cos(rad), -1\*sin(rad), 0},

{0, sin(rad), cos(rad), 0},

{0, 0, 0, 1}

};

break;

case 2: T = {

{cos(rad), 0, sin(rad), 0},

{0, 1, 0, 0},

{-1\*sin(rad), 0, cos(rad), 0},

{0, 0, 0, 1}

};

break;

case 3: T = {

{cos(rad), -1\*sin(rad), 0, 0},

{sin(rad), cos(rad), 0, 0},

{0, 0, 1, 0},

{0, 0, 0, 1}

};

break;

default: T = {

{1,0,0,0},

{0,1,0,0},

{0,0,1,0},

{0,0,0,1}

};

break;

}

for (int c = 0; c < coords.size(); c++)

{

vector<GLfloat> P(4), N(4, 0);

P[0] = coords[c][0];

P[1] = coords[c][1];

P[2] = coords[c][2];

P[3] = 1;

for (int i = 0; i < 4; i++) {

for (int j = 0; j < 1; j++) {

N[i] = 0;

for (int k = 0; k < 4; k++) {

N[i] += T[i][k] \* P[k];

}

}

}

coords[c][0] = N[0];

coords[c][1] = N[1];

coords[c][2] = N[2];

}

}

void myDisplay()

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glColor3f(0.0f, 0.0f, 0.0f);

glRotatef(-45, 0, 1, 0);

glRotatef(45, 1, 0, 0);

glRotatef(-30, 0, 0, 1);

glTranslatef(-100, 0, 0);

Axis();

drawCube();

switch (ch) {

case 1:translate();

drawCube();

break;

case 2: rotate();

drawCube();

break;

case 3:scale();

drawCube();

break;

}

glFlush();

}

int main(int argc, char\* argv[])

{

GLfloat x1=100, Y1=100, z1=100,x2=200,y2=200,z2=200;

cout << "Enter cube dimensions:" << endl;

cout << "Enter min x,y,z: ";

cin >> x1>> Y1>> z1;

cout << "Enter max x,y,z: ";

cin >> x2>> y2>> z2;

cout << "Enter Transformation Operation" <<endl<< "1.Translate" << endl << "2.Rotate" << endl << "3.Scale" << endl;

cout << "Choice: ";

cin >> ch;

switch (ch) {

case 1: cout << "Enter tx,ty,tz: ";

cin >> tx >> ty >> tz;

break;

case 2: cout << "Enter Axis to rotate about X(1), Y(2), Z(3): ";

cin >> RotAxis;

cout << "Enter rotation ang: ";

cin >> ang;

rad = ang \* PI / 180;

break;

case 3: cout << "Enter sx,sy,sz: ";

cin >> sx >> sy >> sz;

cout << "Enter point to scale about x,y,z: ";

cin >> xf >> yf >> zf;

break;

default: cout << "invalid";

}

coords[0][0] = x1;

coords[0][1] = Y1;

coords[0][2] = z1;

coords[1][0] = x1;

coords[1][1] = y2;

coords[1][2] = z1;

coords[2][0] = x2;

coords[2][1] = y2;

coords[2][2] = z1;

coords[3][0] = x2;

coords[3][1] = Y1;

coords[3][2] = z1;

coords[4][0] = x1;

coords[4][1] = Y1;

coords[4][2] = z2;

coords[5][0] = x1;

coords[5][1] = y2;

coords[5][2] = z2;

coords[6][0] = x2;

coords[6][1] = y2;

coords[6][2] = z2;

coords[7][0] = x2;

coords[7][1] = Y1;

coords[7][2] = z2;

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB | GLUT\_DEPTH);

glutInitWindowSize(1000, 1000);

glutCreateWindow("3d cube");

glutDisplayFunc(myDisplay);

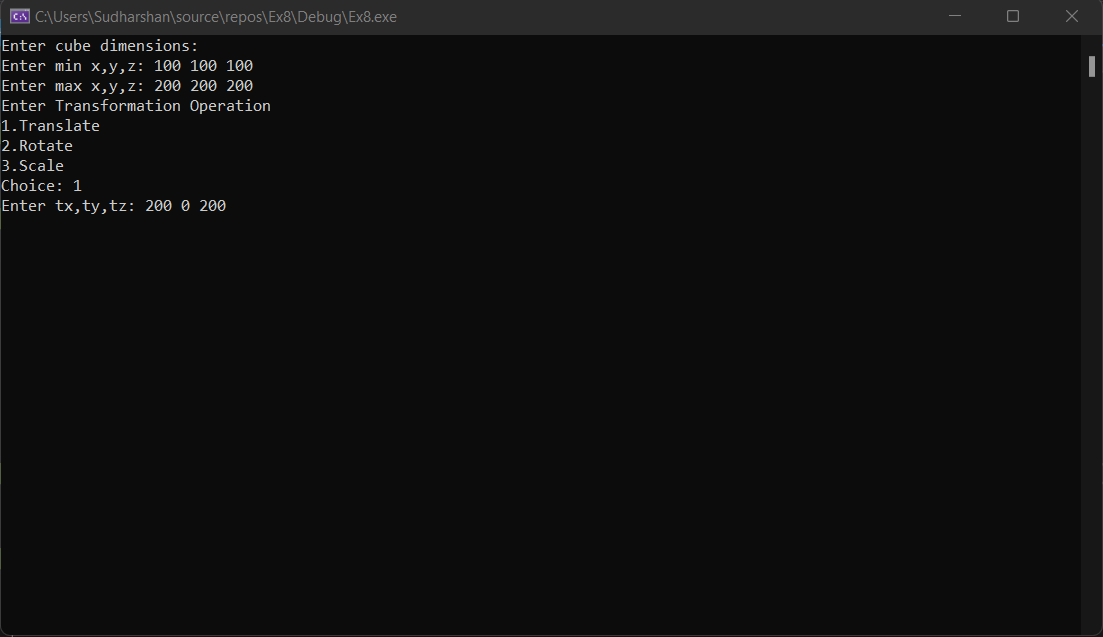
myInit();

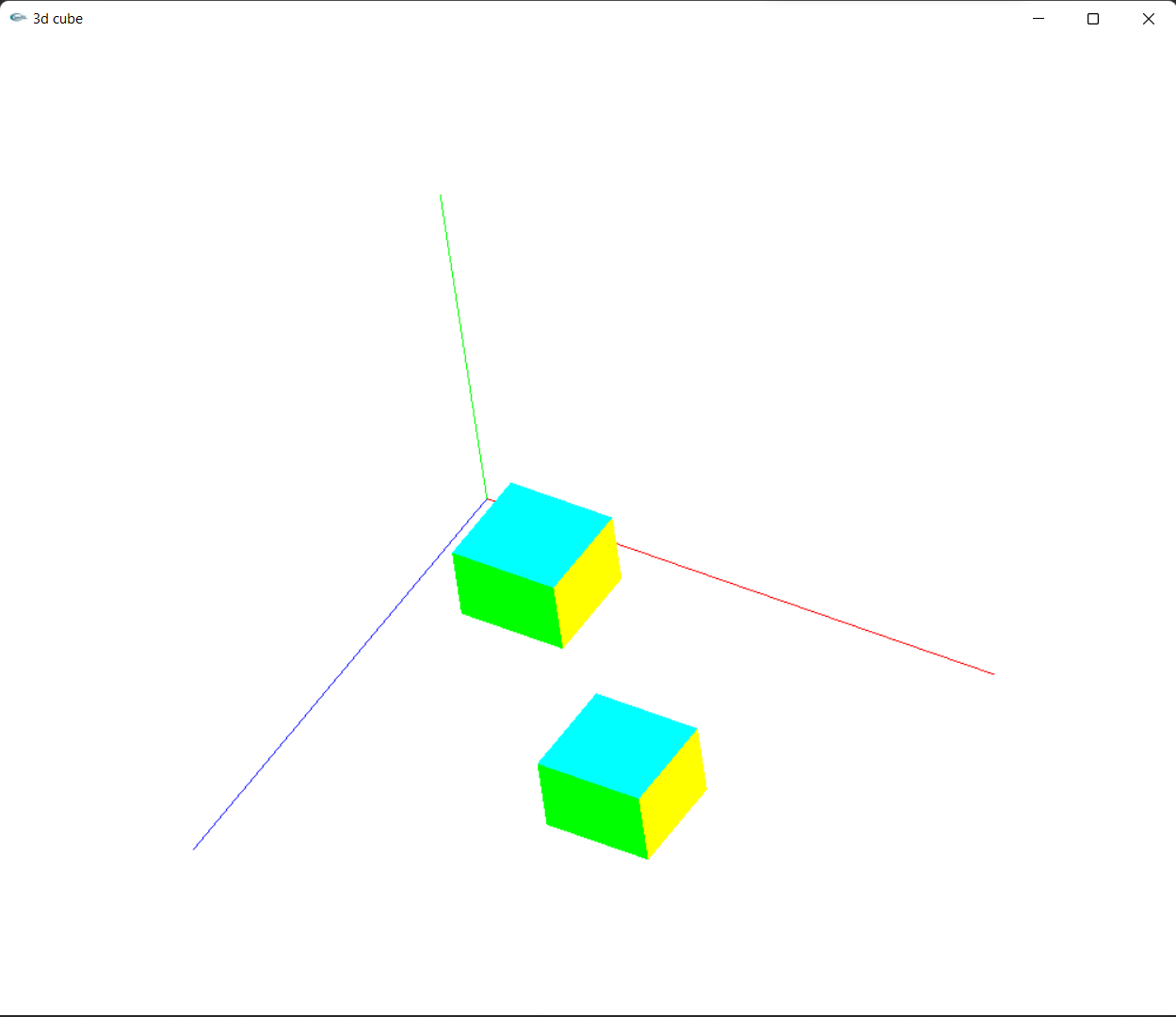
glutMainLoop();

return 1;

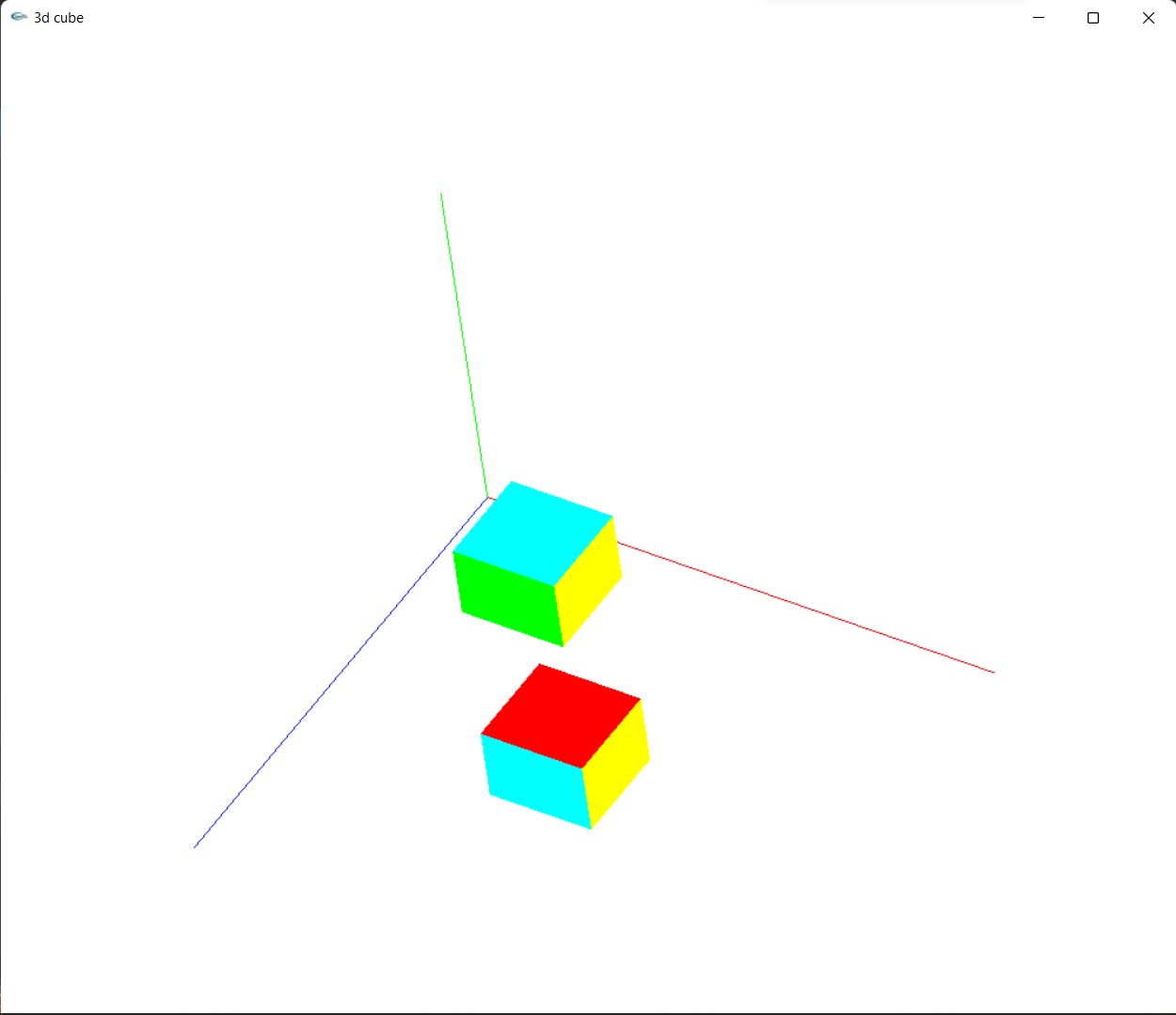
}

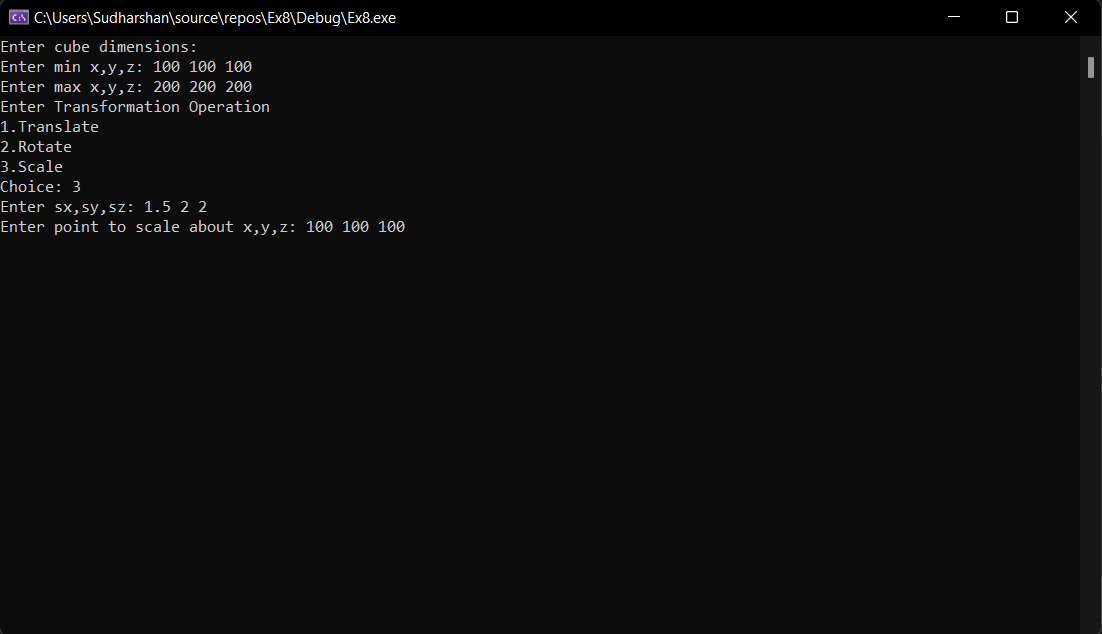
**OUTPUT:**

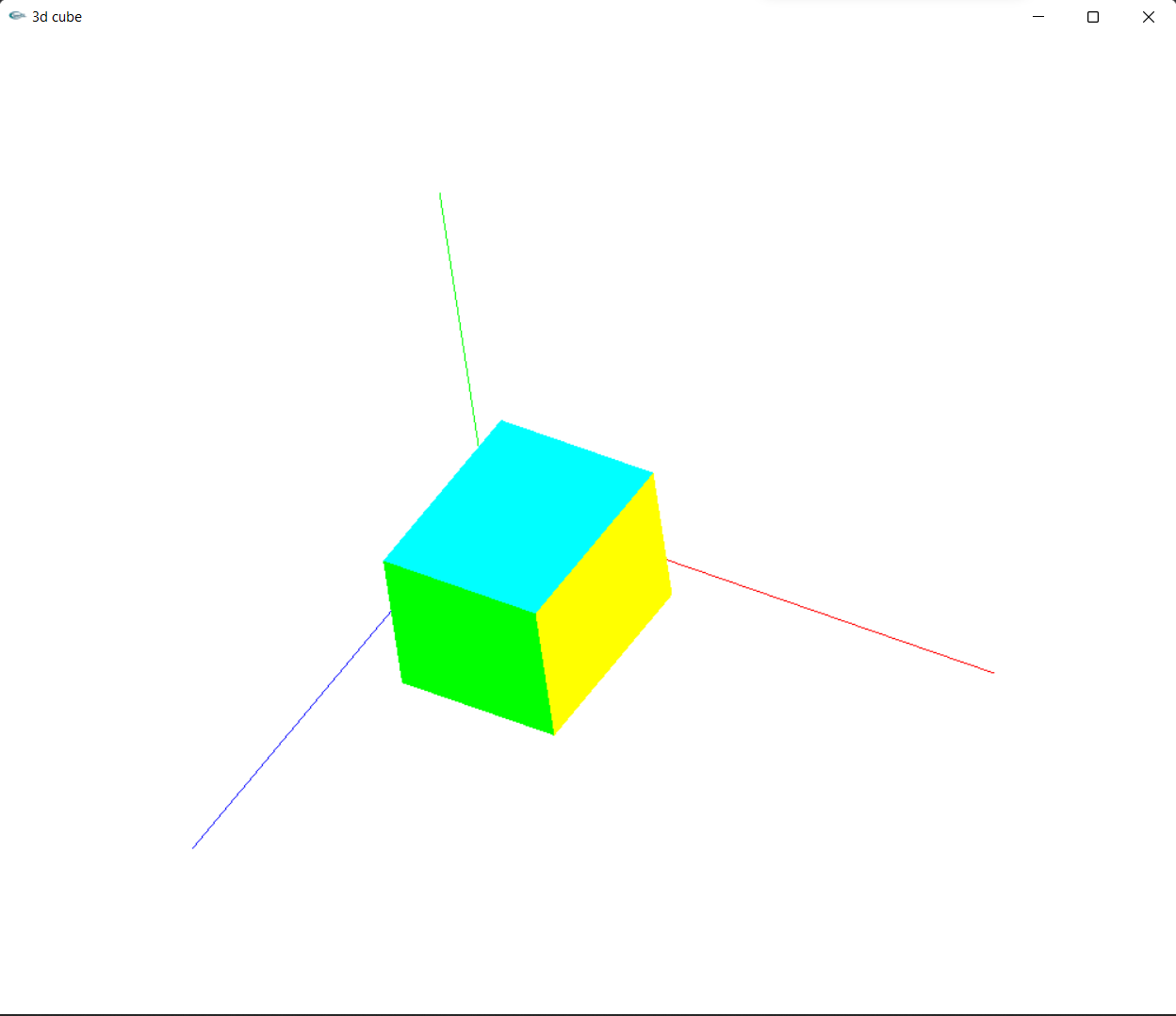
****











**Lab Exercise 9: 3-Dimensional Projections in C++ using OpenGL**

**Aim:**

To create ortho parallel projection and perspective projection on a 3D object.

**CODE:**

#include<gl/glut.h>

#include<iostream>

using namespace std;

int alpha = 0, theta = 0;

bool flag = true;

void init() {

glClearColor(1.0, 1.0, 1.0, 1.0);

glEnable(GL\_DEPTH\_TEST);

}

void keyPress(int key, int x, int y) {

switch (key) {

case GLUT\_KEY\_RIGHT: alpha++;

break;

case GLUT\_KEY\_LEFT:

alpha--;

break;

case GLUT\_KEY\_UP:

theta++;

break;

case GLUT\_KEY\_DOWN:

theta--;

break;

case GLUT\_KEY\_HOME: flag = !flag;

break;

}

glutPostRedisplay();

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

if (flag) glOrtho(-500, 500, -500, 500, -500, 500);

else gluPerspective(100, 1, 0.1, 1000);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

gluLookAt(0, 0, 300, 0, 0, 0, 0, 1, 0);

glRotatef(alpha, 0, 1, 0);

glRotatef(theta, 1, 0, 0);

glColor3f(0.0, 0.0, 0.0);

glutWireTorus(50, 150, 20, 20);

glFlush();

}

int main(int argc, char\* argv[]) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB | GLUT\_DEPTH);

glutInitWindowSize(1000, 1000);

glutCreateWindow("Parallel and Perspective Projections");

init();

glutDisplayFunc(display);

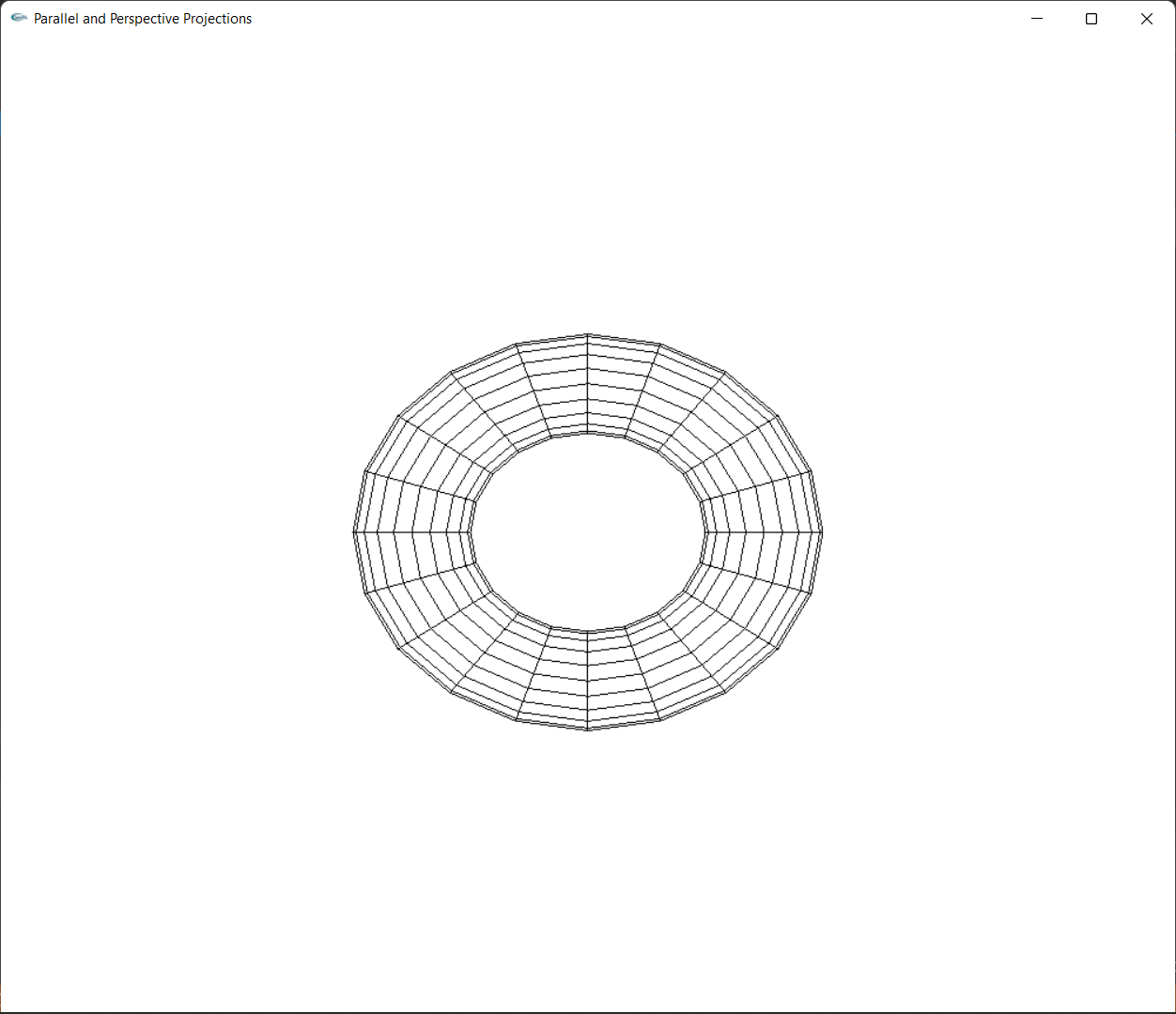
glutSpecialFunc(keyPress);

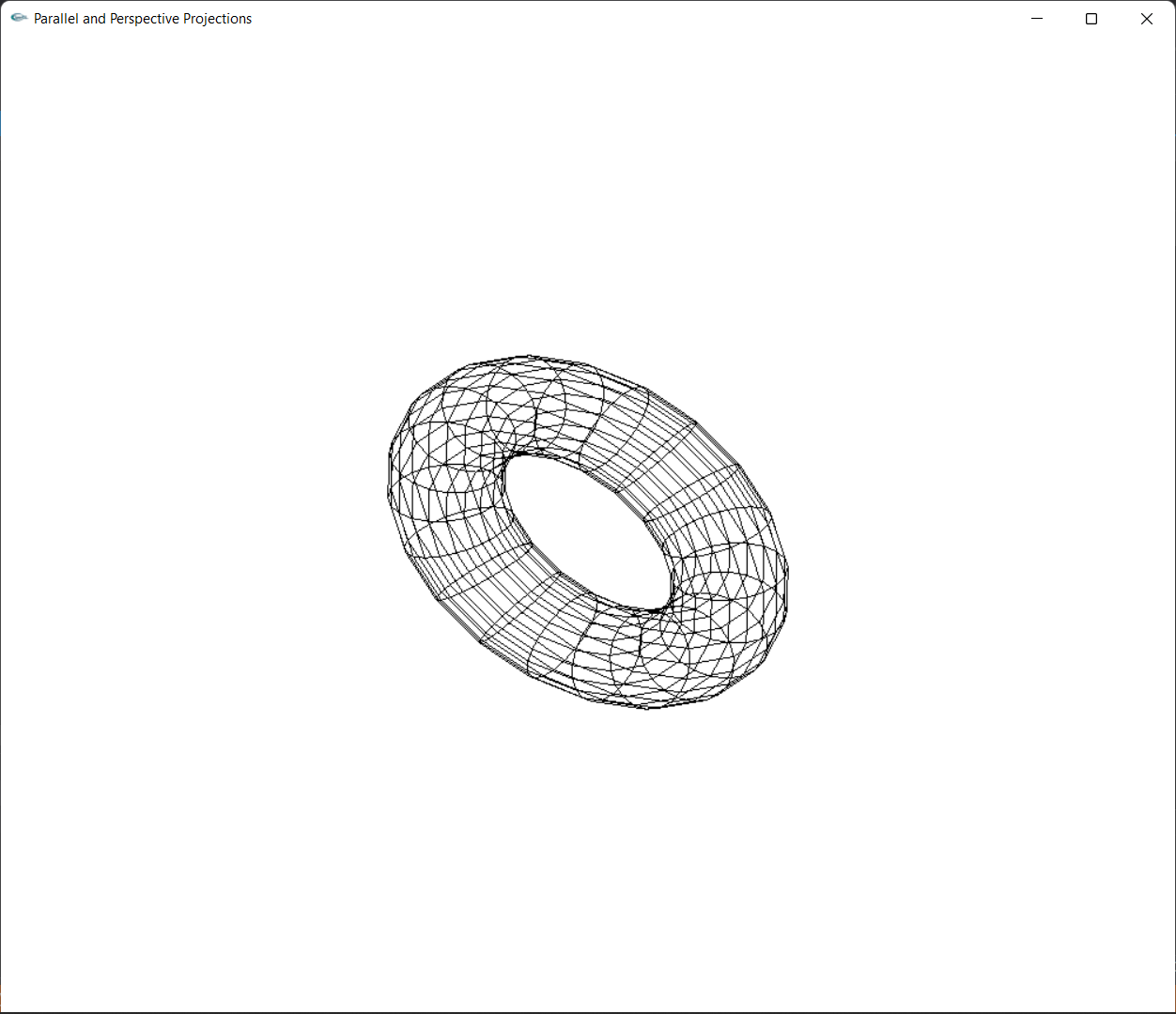
glutMainLoop();

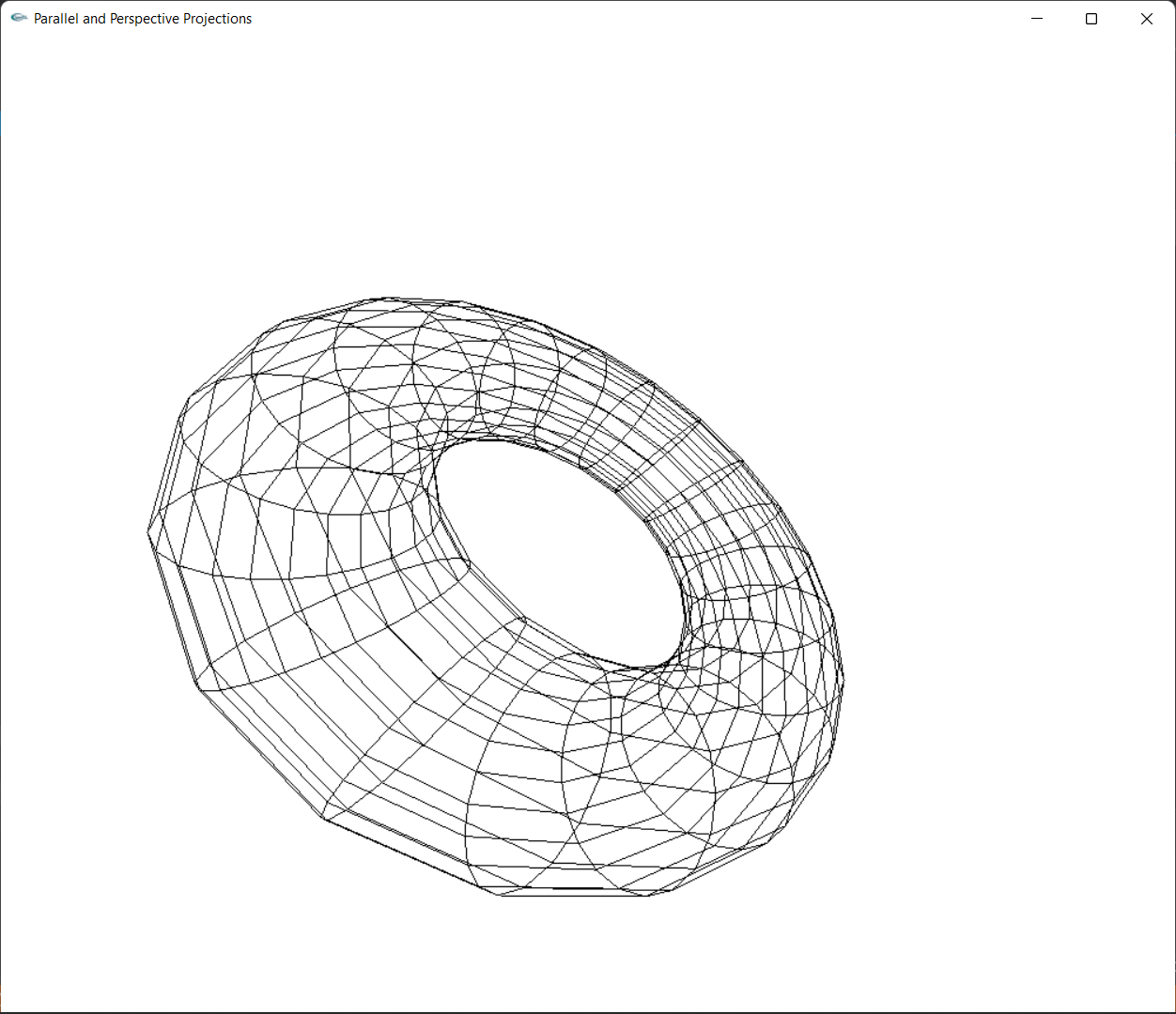
return 0;

}

**OUTPUT:**







**Lab Exercise 10: Creating a 3D Scene in C++ using OpenGL**

**Aim:**

To write a C++ program using Opengl to draw at least four 3D objects. Apply lighting and texture and render the scene. Apply transformations to create a simple 3D animation.

**CODE:**

#include<gl/glut.h>

#include<iostream>

#include<vector>

using namespace std;

GLfloat alpha = 300, theta = 300, gamma = 300;

int index=0;

bool rev=false;

vector<vector<GLfloat>> coords(8, vector<GLfloat>(3));

GLfloat light\_position[] = { 1.0, 1.0, 1.0, 0.0 };

void init(void)

{

GLfloat mat\_specular[] = { 1.0, 1.0, 1.0, 1.0 };

GLfloat mat\_shininess[] = { 50.0 };

glClearColor(0.0, 0.0, 0.0, 0.0);

glShadeModel(GL\_SMOOTH);

glMaterialfv(GL\_FRONT, GL\_SPECULAR, mat\_specular);

glMaterialfv(GL\_FRONT, GL\_SHININESS, mat\_shininess);

glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position);

glEnable(GL\_LIGHTING);

glEnable(GL\_LIGHT0);

glEnable(GL\_DEPTH\_TEST);

}

void display(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluPerspective(100, 1, 0.1, 10000);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glRotatef(30, 0, 1, 0);

gluLookAt(gamma, alpha, theta, -500, 0, -500, 0, 1, 0);

glutSolidCube(100);

gluLookAt(gamma, alpha, theta, 500, 0, -500, 0, 1, 0);

glutSolidTeapot(100);

gluLookAt(gamma, alpha, theta, -500, 0, 500, 0, 1, 0);

glutSolidSphere(100,20,20);

gluLookAt(gamma, alpha, theta, 500, 0, 500, 0, 1, 0);

glutSolidTorus(50,100,20,20);

gluLookAt(gamma, alpha, theta, 1000, 0, 0, 0, 1, 0);

glFlush();

}

void timer(int v)

{

if (!rev) {

alpha += 1;

theta += 1;

gamma += 1;

if (alpha == 500) rev = true;

}

else {

alpha -= 1;

theta -= 1;

gamma -= 1;

if (alpha == 200) rev = false;

}

glutPostRedisplay();

glutTimerFunc(10, timer, v);

}

int main(int argc, char\*\* argv)

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB | GLUT\_DEPTH);

glutInitWindowSize(1000, 1000);

glutCreateWindow("3D Scene");

init();

glutDisplayFunc(display);

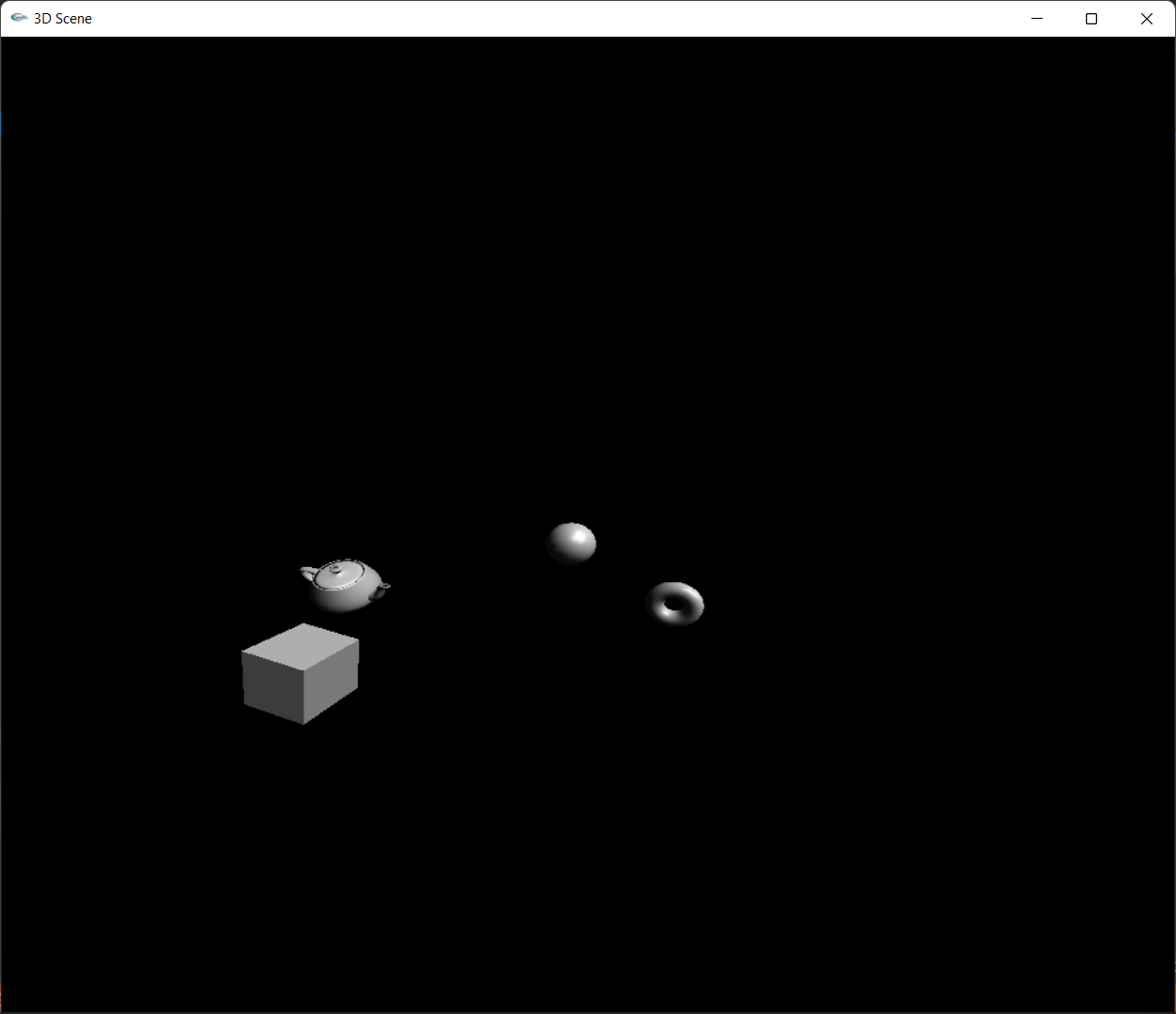
glutTimerFunc(10,timer,0);

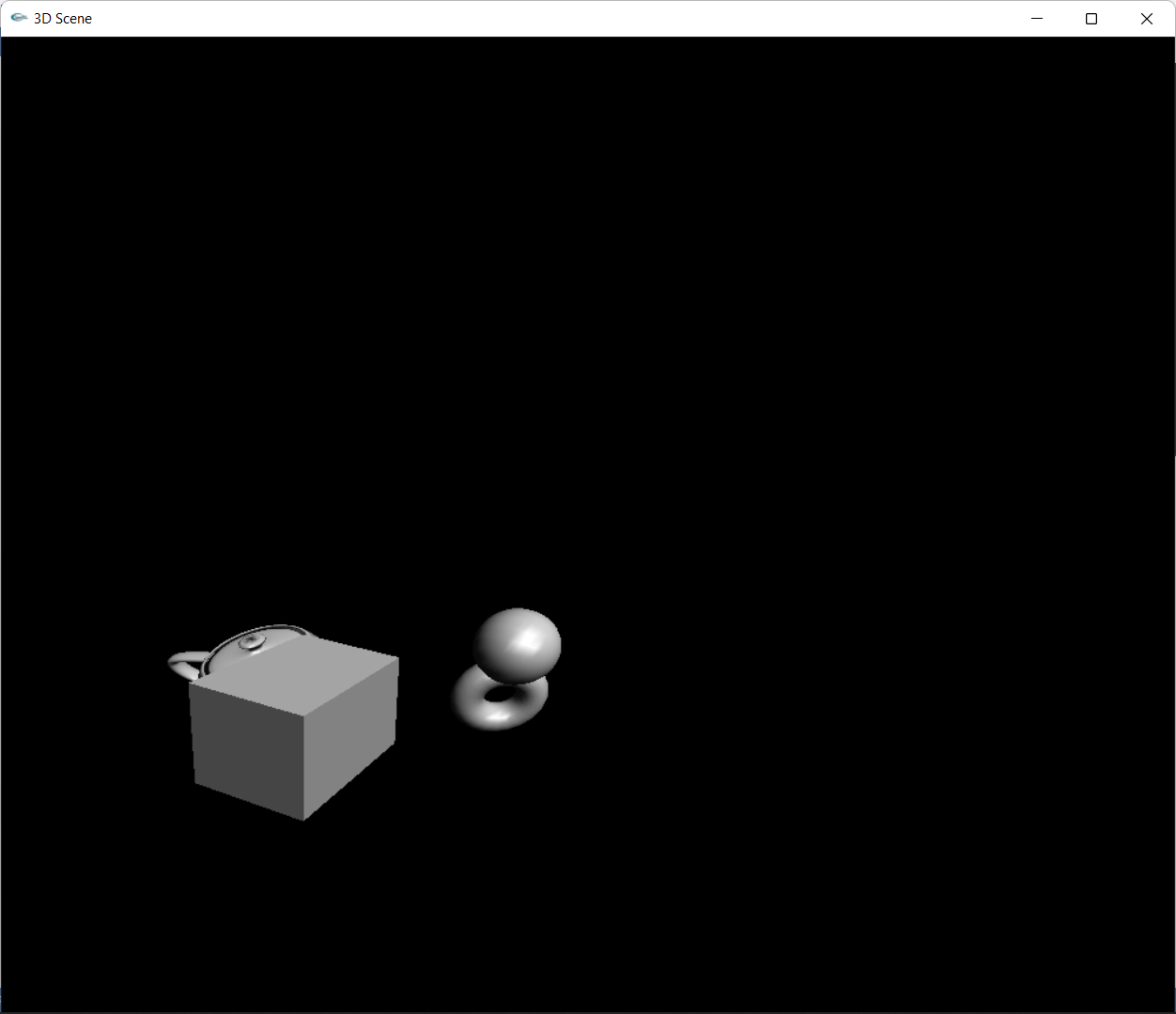
glutMainLoop();

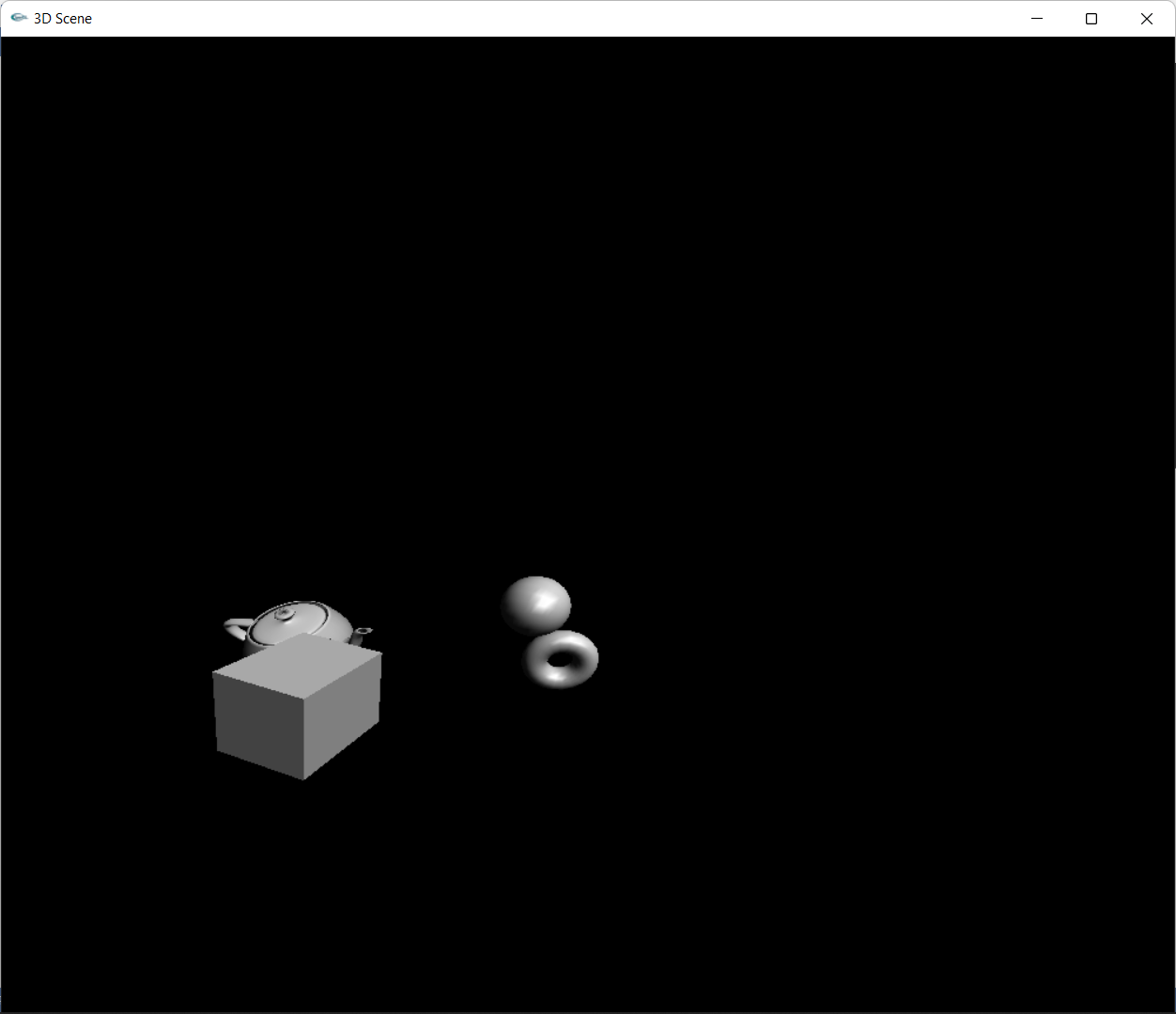
return 0;

}

OUTPUT:







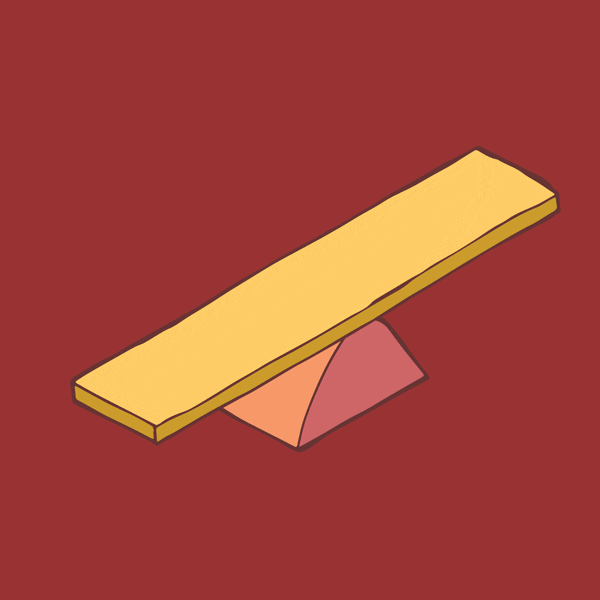
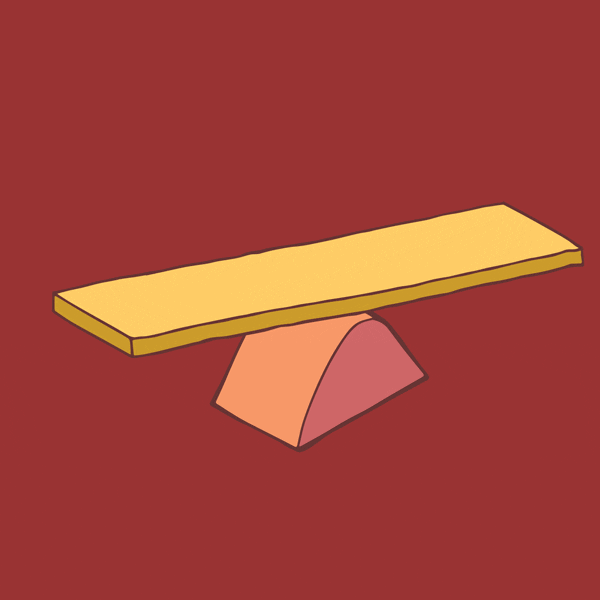
**Lab Exercise 11: Image Editing and Manipulation**

**Aim:**

To perform the below operations using GIMP, include an image and apply filters, noise and masks. To create a GIF animated image.

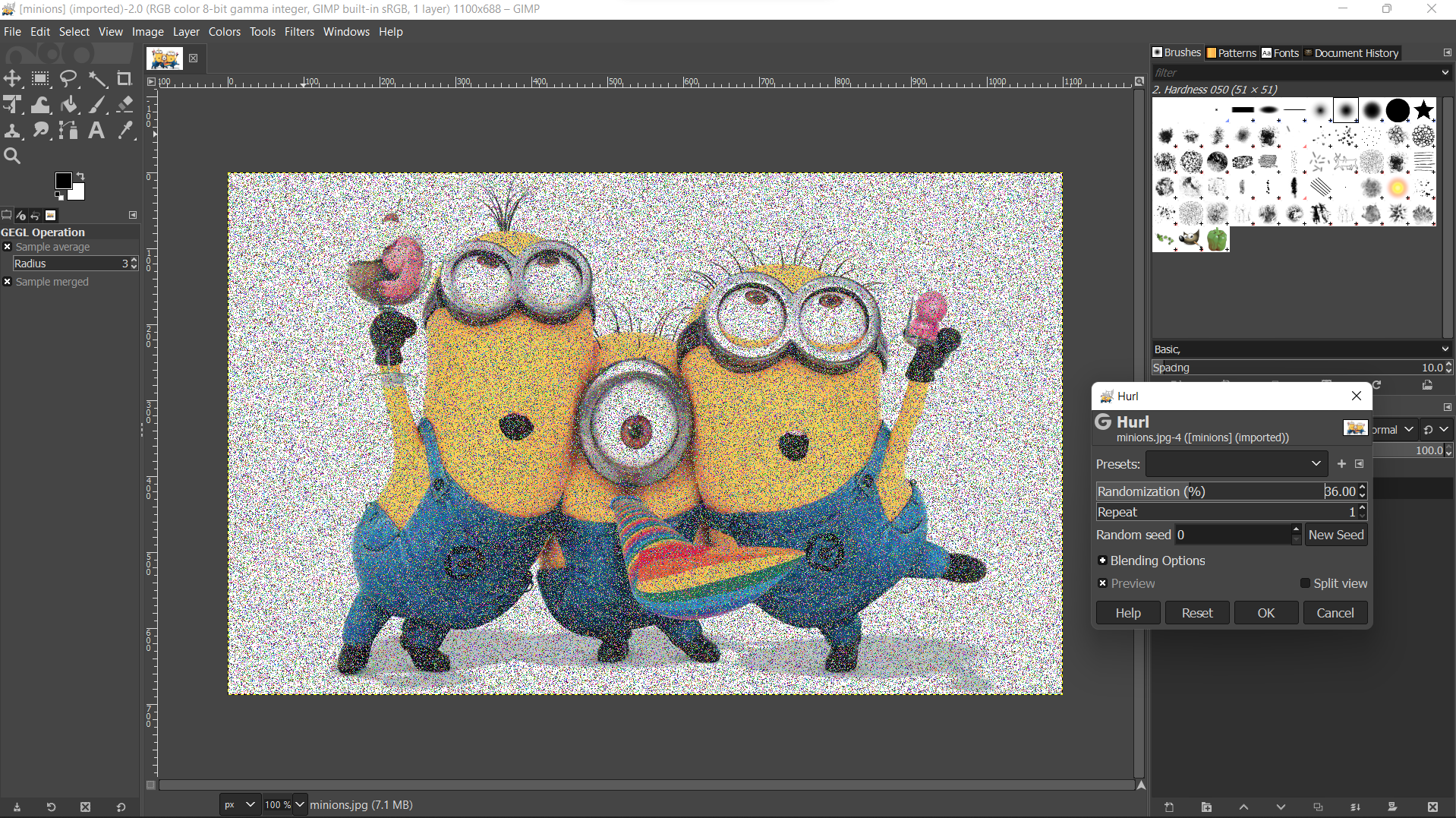
**Data:**

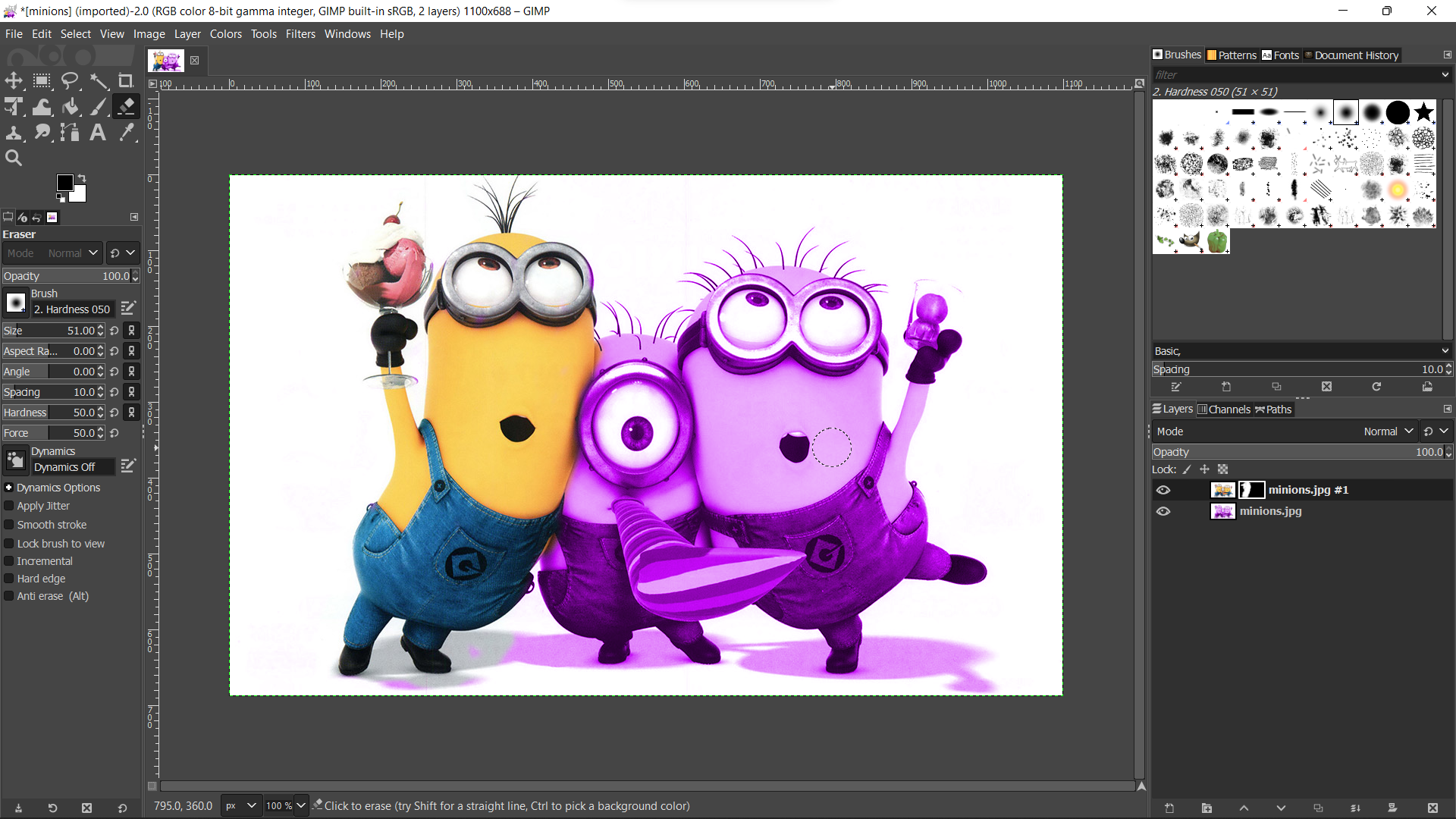


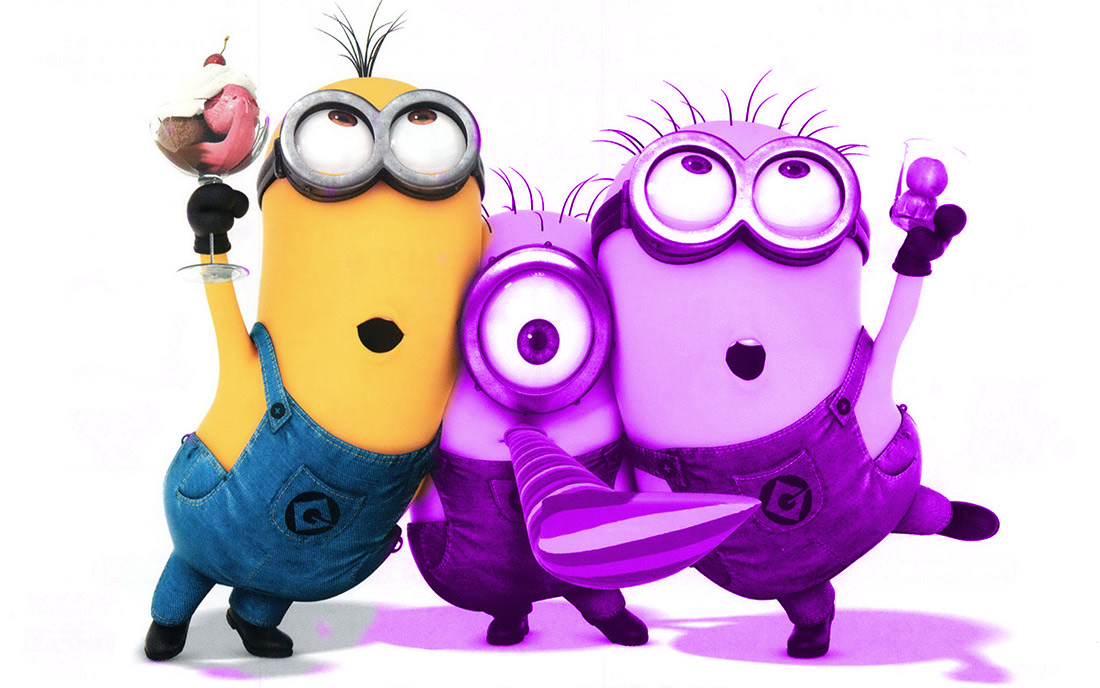
** **

**Output:**

Noise and Masks

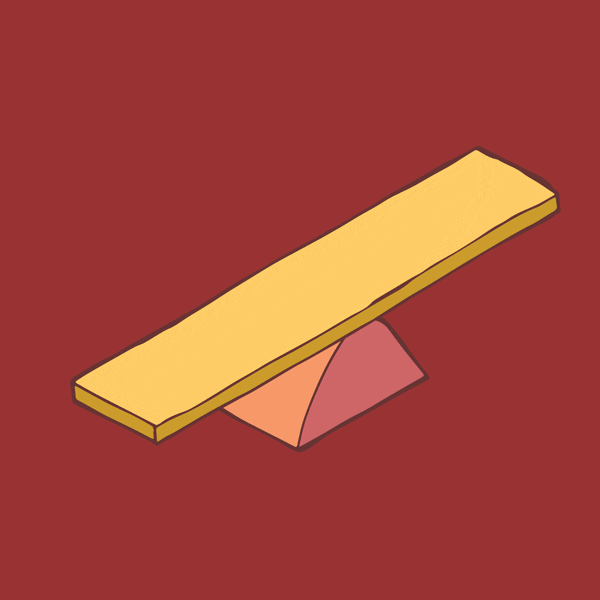




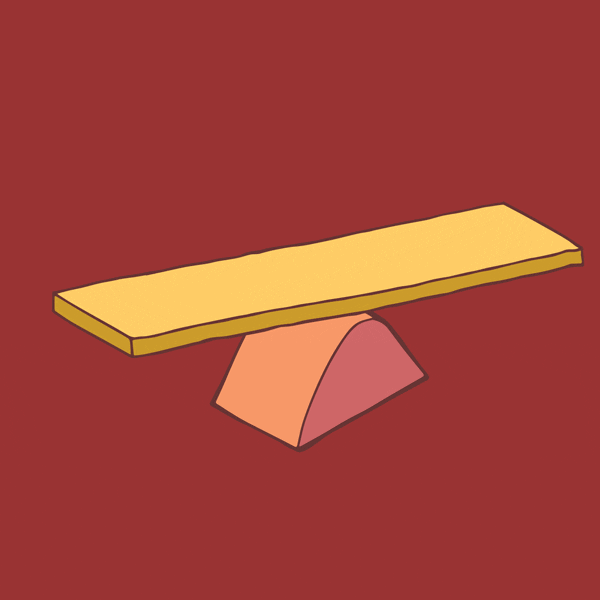


GIF Animated Image

Frame 0

****

Frame 1

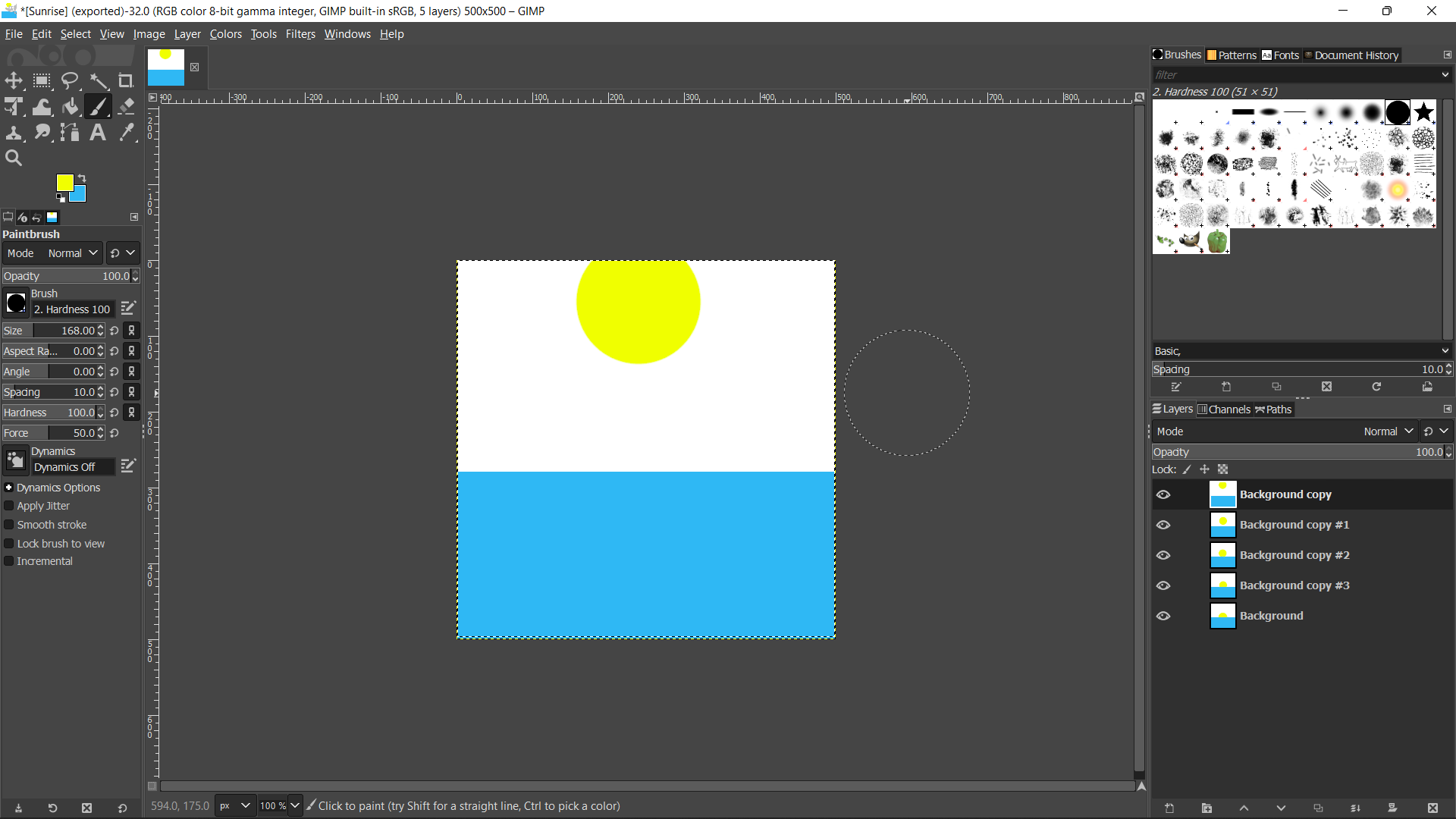
****

**Lab Exercise 12: Creating 2D animation**

**Aim:**

Using GIMP to include layers and create a simple animation.

**Process:**



**Output:**



