**LAB REPORT**

**OF**

**Computer Graphics**

Subject Code: CSC 214

**NAGARJUNA COLLEGE OF IT**

(Affiliated to Tribhuvan University)

Sankhamul, Lalitpur

**Submitted By: Submitted To:**

NAME : Ashwini Kharel Sanjeet Devkota

ROLL NO : 4

SEMESTER : 3rd (Third)

PROGRAM : B.Sc.CSIT

**TABLE OF CONTENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **TITLE** | **PAGE** | **SIGNATURE** |
| 1 | DDA Line Drawing Algorithm | 1-3 |  |
| 2 | Bresenham’s Line Drawing Algorithm | 4-6 |  |
| 3 | Mid-Point Circle Drawing Algorithm | 7-10 |  |
| 4 | Mid-Point Ellipse Drawing Algorithm | 11-14 |  |
| 5 | Cohen-Sutherland Line Clipping Algorithm | 15-18 |  |
| 6 | 2D Transformation | 19-23 |  |
| 7 | 3D Transformation | 24-26 |  |
| 8 | Visible Surface Detection (Depth Buffer Algorithm) | 27-29 |  |
| 9 | Basic OpenGL Drawing | 30-32 |  |

**LAB 1**

**DDA LINE DRAWING ALGORITHM**

**THEORY :**

DDA (Digital Differential Analyzer) is a line drawing algorithm used in computer graphics to generate a line segment between two specified endpoints. It is a simple and efficient algorithm that works by using the incremental difference between the x-coordinates and y-coordinates of the two endpoints to plot the line.

1. **ALGORITHM :**
2. Input the two endpoints of the line segment, (x1,y1) and (x2,y2).
3. Compute dx = x2 - x1 and dy = y2 - y1.
4. Set the larger of dx and dy to steps.
5. Set the initial point of the line as (x1,y1).
6. Calculate xIncrement = dx / steps and yIncrement = dy / steps.
7. Increment the x-coordinate and y-coordinate as:

x += xIncrement;

y += yIncrement;

1. Plot the pixel at the calculated (x,y) coordinate.
2. Continue until the endpoint (x2,y2) is reached.
3. **SOURCE CODE :**

#include <GL/glut.h>

#include <iostream>

#include <cmath>

using namespace std;

// Function to plot a pixel

void plotPixel(int x, int y) {

glBegin(GL\_POINTS);

glVertex2i(x, y);

glEnd();

}

// DDA Line Drawing Algorithm

void drawLineDDA(int x1, int y1, int x2, int y2) {

int dx = x2 - x1;

int dy = y2 - y1;

int steps = max(abs(dx), abs(dy));

float xIncrement = dx / (float)steps;

float yIncrement = dy / (float)steps;

float x = x1;

float y = y1;

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

plotPixel(round(x), round(y));

x += xIncrement;

y += yIncrement;

}

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0, 1.0, 1.0); // White color for the line

drawLineDDA(50, 50, 200, 200);

glFlush();

}

void init() {

glClearColor(0.0, 0.0, 0.0, 0.0);

glMatrixMode(GL\_PROJECTION);

gluOrtho2D(0.0, 400.0, 0.0, 400.0);

}

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

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(800, 600);

glutInitWindowPosition(100, 100);

glutCreateWindow("Ashwini Kharel #04 (DDA)");

glPointSize(3.0);

init();

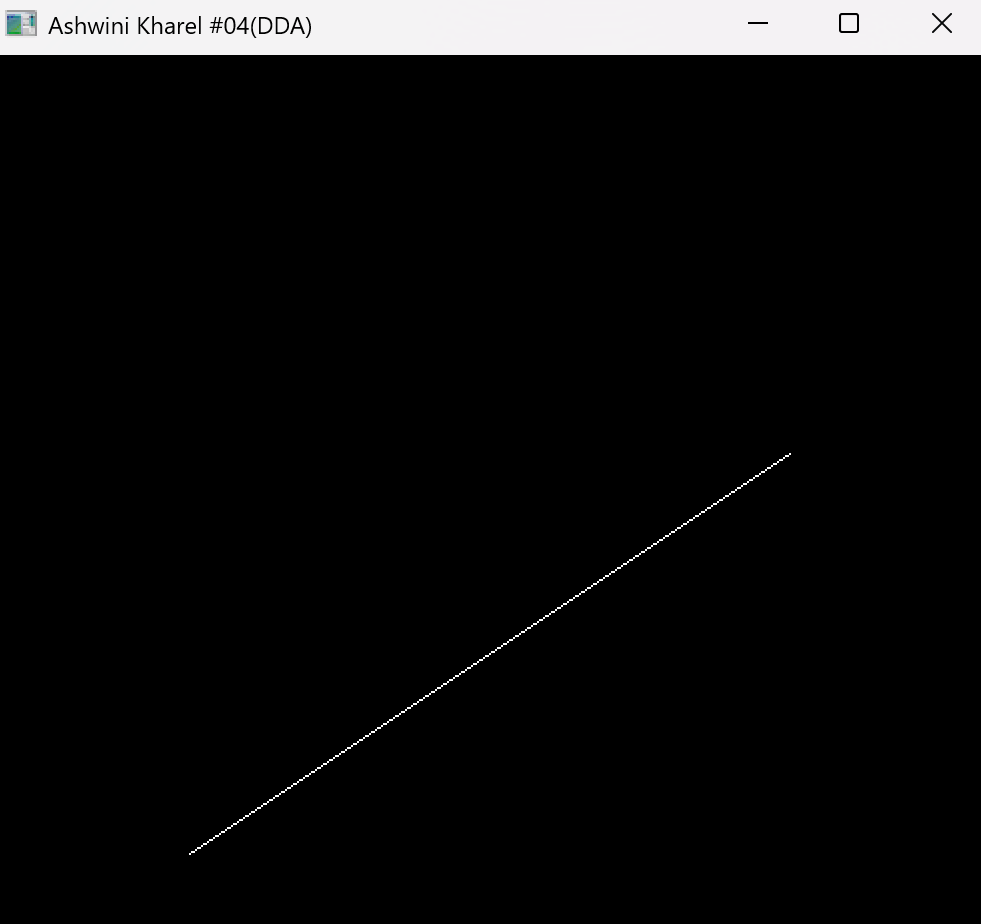
glutDisplayFunc(display);

glutMainLoop();

return 0;

}

1. **OUTPUT :**



**LAB 2**

**BRESENHAM’S LINE DRAWING ALGORITHM (BSA)**

**THEORY :**

Bresenham’s Line Algorithm is an efficient raster-based algorithm to draw a straight line between two points using only integer calculations. It is widely used in computer graphics because it avoids floating-point arithmetic, making it faster and more efficient. It uses a special value called the decision parameter (P-value) to decide whether the next pixel should be placed horizontally or diagonally.

* 1. **ALGORITHM :**

1. Input two endpoints (x0, y0) and (x1, y1).
2. Initialize (x0, y0) as first plot point.

3) Compute dx = x1 – x0 and dy = y1 – y0.

4) Calculate the slope as m = dy/dx.

5) If m < 1, compute as follows:

* If p < 0, x = x + 1 and p = p + 2dy
* If p >= 0, x = x + 1, y = y + 1 and p = p + 2dy - 2dx
* Compute initial decision parameter as p0 = 2dy -dx.

Else if m > 1, compute as follows:

* If p < 0, x = x + 1 and p = p + 2dx
* If p >= 0, x = x + 1, y = y - 1 and p = p + 2dx - 2dy
* Compute initial decision parameter: p0 = 2dx -dy.

6) End.

* 1. **SOURCE CODE :**

#include <GL/glut.h>

#include <iostream>

#include <cmath>

using namespace std;

//BSA Line drawing function

void drawBresenhamLine(int x0, int y0, int x1, int y1) {

int dx = x1 - x0;

int dy = y1 - y0;

int p = 2 \* dy - dx;

int x = x0, y = y0;

glBegin(GL\_POINTS);

glVertex2i(x, y);

while (x < x1) {

x++;

if (p < 0) {

p += 2 \* dy;

} else {

y++;

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

}

glVertex2i(x, y);

}

glEnd();

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

drawBresenhamLine(50, 50, 300, 200);

glFlush();

}

void init() {

glClearColor(0.0, 0.0, 0.0, 1.0);

glColor3f(1.0, 1.0, 1.0); // White color for the line

glPointSize(3.0);

gluOrtho2D(0.0, 400, 0.0, 400);

}

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

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(800, 600);

glutInitWindowPosition(100,100);

glutCreateWindow("Ashwini Kharel #04 (BSA)");

init();

glutDisplayFunc(display);

glutMainLoop();

return 0;

}

* 1. **OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**LAB 3**

**MID-POINT CIRCLE DRAWING ALGORITHM**

**THEORY :**

The mid-point circle drawing algorithm is an algorithm used to determine the points needed for rasterizing a circle. It determines the next pixel to plot by checking the midpoint between two possible pixel positions and updating a decision parameter accordingly. The algorithm exploits the eight-way symmetry of circles, meaning only one-eighth of the circle is calculated, and the rest is mirrored.

1. **ALGORITHM :**
   1. Input the radius of circle (r).
   2. Initialise at the point (0, r).
   3. Use the initial decision parameter

p = 1−r.

* 1. Compute p as:
* If p<0,

p+= 2\*x + 1

* If p>=0,

p+=2\*(x-y)+1

y = y-1

* 1. Reflect points across all 8 octants.
  2. Continue until x=y.

1. **SOURCE CODE :**

#include <GL/glut.h>

#include <iostream>

#include <cmath>

//function to plot a pixel

void setPixel(int x, int y){

glBegin(GL\_POINTS);

glVertex2i(x,y);

glEnd();

}

//Circle drawing function

void drawCircle(int xc, int yc, int r){

int x=0, y=r;

int p= 1-r;

//initial octants

while(x<=y){

//Symmetric points in octants

setPixel(xc +x, yc +y);

setPixel(xc -x, yc +y);

setPixel(xc +x, yc -y);

setPixel(xc -x, yc -y);

setPixel(xc +y, yc +x);

setPixel(xc -y, yc +x);

setPixel(xc +y, yc -x);

setPixel(xc -y, yc -x);

if (p<0){

p+=2\*x+1;

}else{

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

y--;

}

x++;

}

glFlush();

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0, 1.0, 1.0); // White color for the line

glPointSize(3.0f);

drawCircle(250, 250, 60);

glFlush();

}

// Initialization function

void init() {

glClearColor(0.0, 0.0, 0.0, 0.0);

glMatrixMode(GL\_PROJECTION);

gluOrtho2D(0.0, 400.0, 0.0, 400.0);

}

// Main function

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

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(800, 600);

glutInitWindowPosition(100, 100);

glutCreateWindow("Ashwini Kharel #04 (Circle)");

init();

glutDisplayFunc(display);

glutMainLoop();

return 0;

}

1. **OUTPUT :**

A black background with a white circle

AI-generated content may be incorrect.

**LAB 4**

**MID-POINT ELLIPSE DRAWING ALGORITHM**

**THEORY :**

The Midpoint Ellipse Algorithm is used in computer graphics to draw an ellipse efficiently by determining pixel positions based on a decision parameter. It takes advantage of the symmetry of the ellipse, allowing calculations to be performed in only one quadrant, and then reflecting the points into the other three quadrants.

1. **ALGORITHM :**

1) Input horizontal radius (rx), vertical radius (ry) and center (xc, yc).

2) Initialize with x = 0, y = ry.

3) Region 1 (until 2ry²x < 2rx²y):

* Set initial decision parameter as p1 = ry² - rx²ry + (0.25 \* rx²)
* If p1 < 0:

x = x + 1 and p1 = p1 + 2ry²x + ry²

Else if p1 >= 0:

x = x + 1, y = y – 1 and p1 = p1 + 2ry²x – 2rx²y + ry²

* Plot (xc + x, yc + y) and symmetric points.

4) Region 2 (until y >= 0):

* Set decision parameter as p2= ry²(x+0.5)² + rx² (y-1)² - (rx² \* ry²);
* If p2 > 0:

y = y – 1 and  p2 = p2 – 2rx²y + rx²

Else if p2 <= 0:

x = x + 1, y = y – 1 and p2 = p2 + 2ry²x – 2rx²y + rx²

* Plot (xc + x, yc + y) and symmetric points.

5) End.

1. **SOURCE CODE :**

#include <GL/glut.h>

#include <iostream>

#include <cmath>

using namespace std;

int rx,ry,xc,yc;

//function to plot pixel

void plot(int x, int y){

glBegin(GL\_POINTS);

glVertex2i(xc+x, yc+y);

glVertex2i(xc-x, yc+y);

glVertex2i(xc+x, yc-y);

glVertex2i(xc-x, yc-y);

glEnd();

}

//Ellipse drawing function

void Ellipse()

{

int x= 0, y=ry;

float p1= (ry\*ry)-(rx\*rx\*ry)+(0.25\*rx\*rx);

int c1 = 2\*ry\*ry\*x;

int c2 = 2\*rx\*rx\*y;

//for region 1

while(c1<c2)

{

plot(x,y);

if(p1<0){

x++;

c1 +=2\*ry\*ry;

p1+=c1+ry\*ry;

}

else

{

x++;

y--;

c1+=2\*ry\*ry;

c2-=2\*rx\*rx;

p1+=c1-c2+ry\*ry;

}

}

//initial decision parameter for region 2

float p2= ((ry\*ry)\*((x+0.5)\*(x+0.5)))+((rx\*rx)\*((y-1)\*(y-1)))-(rx\*rx\*ry\*ry);

while(y>=0)

{

plot(x,y);

if(p2>0)

{

y--;

c2 -=2\*rx\*rx;

p2+=rx\*rx-c2;

}

else

{

y--;

x++;

c1+=2\*ry\*ry;

c2-=2\*rx\*rx;

p2+=c1-c2+rx\*rx;

}

}

}

void display()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0, 1.0, 1.0);

glPointSize(3.0);

Ellipse();

glFlush();

}

// Initialization function

void init()

{

glClearColor(0.0, 0.0, 0.0, 0.0);

glMatrixMode(GL\_PROJECTION);

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

}

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

{

cout<<"Enter center of ellipse (xc,yc): ";

cin>>xc>>yc;

cout<<"Enter x-radius and y-radius (rx,ry): ";

cin>>rx>>ry;

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(800, 600);

glutCreateWindow("Krishtina Maharjan #15 ");

init();

glutDisplayFunc(display);

glutMainLoop();

return 0;

}

1. **OUTPUT :**

A screenshot of a computer

AI-generated content may be incorrect.

**LAB 5**

**COHEN-SUTHERLAND LINE CLIPPING ALGORITHM**

**THEORY:**

The Cohen-Sutherland algorithm is a line-clipping algorithm that efficiently clips a polygon or a line segment against a rectangular clipping window. It divides the space around the clipping window into nine regions and uses a code to quickly determine whether a line is entirely inside, outside, or intersects the window.

1. **ALGORITHM :**
2. Define region codes:

Inside = 0000 Left = 0001 Right = 0010

Bottom = 0100 Top = 1000

2) Compute region codes for both endpoints.

3) If both endpoints have code 0000 (fully inside), accept the line.

4) If the AND of both codes is not 0000 (fully outside), reject the line.

5) Else, find the intersection point with the clipping boundary:

* If point is left of window: x = xmin, compute y
* If point is right of window: x = xmax, compute y
* If point is below window: y = ymin, compute x
* If point is above window: y = ymax, compute x

6) Update the line and repeat until both endpoints are inside.

7)Draw the clipped line.

8)End

1. **SOURCE CODE :**

#include <GL/glut.h>

#include <iostream>

#include <cmath>

void display();

using namespace std;

float xmin = -100;

float ymin = -100;

float xmax = 100;

float ymax = 100;

float xd1, xd2, yd1,yd2;

void init(void){

glClearColor(0.0,0.0,0.0,0.0);

glMatrixMode(GL\_PROJECTION);

gluOrtho2D(-600,600,-600,600);

}

int code(float x,float y){

int c=0;

if(y>ymax)c=8;

if(y<ymin)c=4;

if(x>xmax)c=2;

if(x<xmin)c=1;

return c;

}

void cohen\_line(float x1, float y1 ,float x2, float y2){

int c1 = code(x1,y1);

int c2 = code(x2,y2);

float m= (y2-y1) / (x2 - x1);

bool accept=false;

while(true){

if((c1 | c2)==0){

accept=true;

break; }

else if((c1& c2)!=0) {

break; }

else {

float x,y;

int c= (c1!=0)? c1:c2; //choose one of the endpoints

if(c&8) { //top

x=x1+(x2-x1)\*(ymax-y1)/(y2-y1);

y=ymax; }

else if(c&4) { //bottom

x=x1+(x2-x1)\*(ymin-y1)/(y2-y1);

y=ymin; }

else if(c&2) { //right

y=y1+(y2-y1)\*(xmax-x1)/(x2-x1);

x=xmax; }

if(c&1) { //left

y=y1+(y2-y1)\*(xmin-x1)/(x2-x1);

x=xmin; }

if(c==c1) {

x1=x;

y1=y;

c1=code(x1,y1); }

else {

x2=x;

y2=y;

c2=code(x2,y2);

} } }

if(accept) {

xd1=x1;

yd1=y1;

xd2=x2;

yd2=y2; }

display(); //redraw display with clipped line

}

void mykey(unsigned char key, int x,int y) {

if(key == 's'){

cohen\_line(xd1, yd1, xd2, yd2);

glFlush();

}

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0,1.0,0.0);

glLineWidth(3.0f);

glBegin(GL\_LINE\_LOOP);

glVertex2i(xmin,ymin);

glVertex2i(xmin,ymax);

glVertex2i(xmax,ymax);

glVertex2i(xmax,ymin);

glEnd();

glColor3f(1.0,1.0,1.0);

glLineWidth(3.0f);

glBegin(GL\_LINES);

glVertex2i(xd1,yd1);

glVertex2i(xd2,yd2);

glEnd();

glFlush();

}

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

printf("Enter line coordinates: ");

cin>>xd1>>yd1>>xd2>>yd2;

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE|GLUT\_RGB);

glutInitWindowSize(800,600);

glutInitWindowPosition(0,0);

glutCreateWindow("Ashwini Kharel #04 (Polygon Clipping)");

glutDisplayFunc(display);

glutKeyboardFunc(mykey);

init();

glutMainLoop();

return 0;

}

1. **OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**LAB 6**

**2D TRANSFORMATION**

**THEORY :**

2D transformation involves changing the position, size, or orientation of an object in a two-dimensional space. The main types include translation (shifting an object), scaling (changing its size), rotation (rotating around a point), and shearing (distorting the shape). These transformations are typically represented using 2x3 matrices for efficient computation.

1. **ALGORITHM :**

1) Input the coordinates of triangle.

**2)** Choose transformation (Translation, Rotation or Scaling).

i) Translation:

x' = x + Tx

y' = y + Ty

ii) Rotation (angle θ):

x' = x \* cos(θ) - y \* sin(θ)

y' = x \* sin(θ) + y \* cos(θ)

iii) Scaling (Sx, Sy):

x' = x \* Sx

y' = y \* Sy

3)Apply transformations to each point and plot new positions.

4)End

1. **SOURCE CODE :**

#include <GL/glut.h>

#include <iostream>

#include <cmath>

using namespace std;

float triangle[3][2]={{100,100},{200,100},{150,250}};

float transformed[3][2];

void drawTriangle(float points[3][2]){

glBegin(GL\_TRIANGLES);

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

glVertex2f(points[i][0], points[i][1]);

glEnd();

}

void translate(float tx,float ty){

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

transformed[i][0]= triangle[i][0]+tx;

transformed[i][1]= triangle[i][1]+ty;

}

}

void rotate(float angle){

float rad = angle \*3.14159/180;

int i;

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

transformed[i][0] = triangle[i][0] \* cos(rad) - triangle[i][1] \* sin(rad);

transformed[i][1] = triangle[i][1] \* sin(rad) - triangle[i][1] \* cos(rad);

}

void scale(float sx,float sy){

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

transformed[i][0]= triangle[i][0]\*sx;

transformed[i][1]= triangle[i][1]\*sy;

}

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0, 0.0, 0.0);

glPointSize(4.0);

drawTriangle(triangle);

glColor3f(0.0, 1.0, 0.0);

drawTriangle(transformed);

glFlush();

}

// Initialization function

void init() {

glClearColor(0.0, 0.0, 0.0, 1.0);

gluOrtho2D(-300,300,-300, 300);

}

// Main function

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

int choice;

float tx,ty,angle,sx,sy;

cout<<"Choose transformation:\n";

cout<<"1. Translation\n2. Rotation\n3. Scaling\nEnter your choice:\n";

cin>>choice;

switch (choice) {

case 1:

cout<<"Enter translation values (tx,ty): ";

cin>>tx>>ty;

translate(tx, ty);

break;

case 2:

cout<<"Enter rotation angle: ";

cin>>angle;

rotate(angle);

break;

case 3:

cout<<"Enter scaling values (sx,sy): ";

cin>>sx>>sy;

scale(sx, sy);

break;

default:

cout<<"Invalid choice!"<<endl;

break;

}

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutInitWindowSize(800,600);

glutCreateWindow("Ashwini Kharel #04 (2D Transformation)");

init();

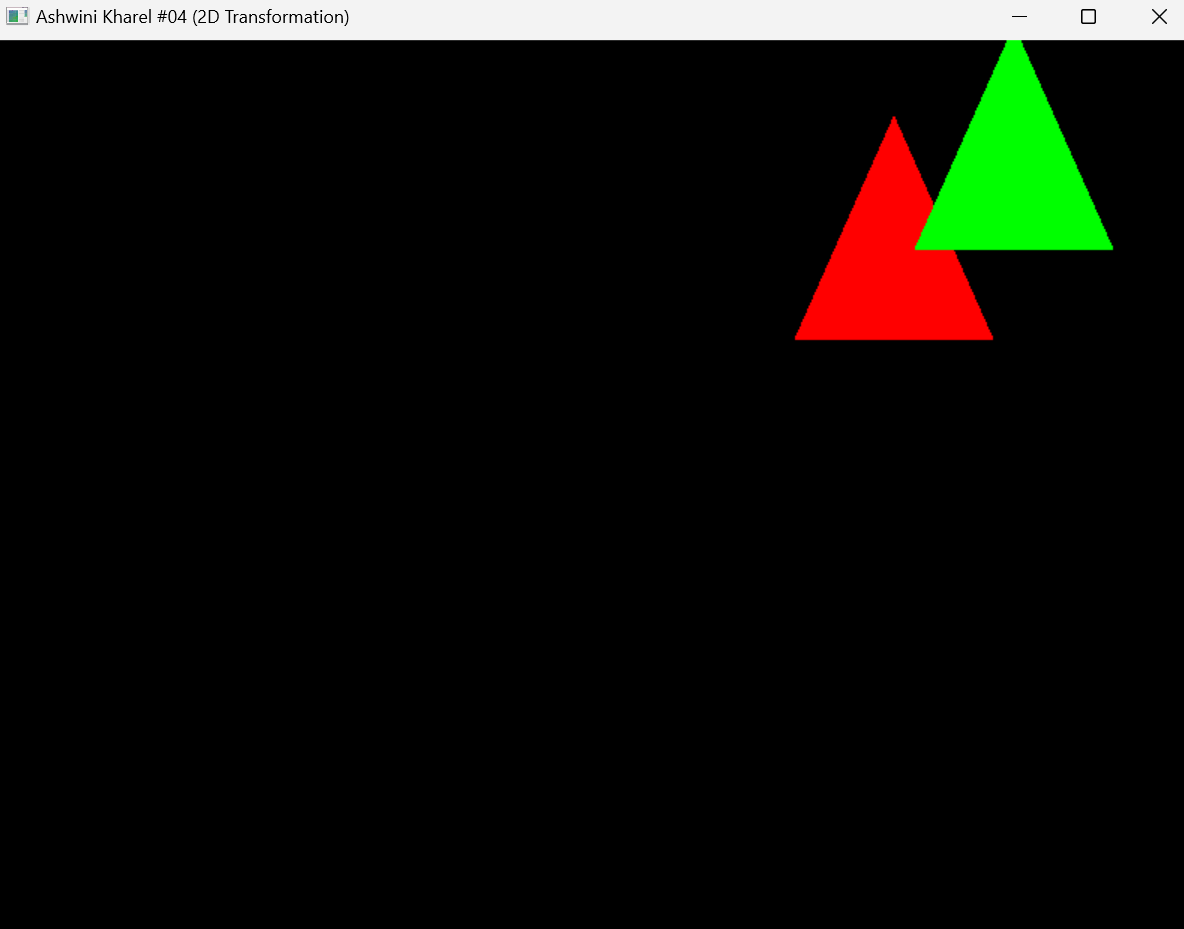
glutDisplayFunc(display);

glutMainLoop();

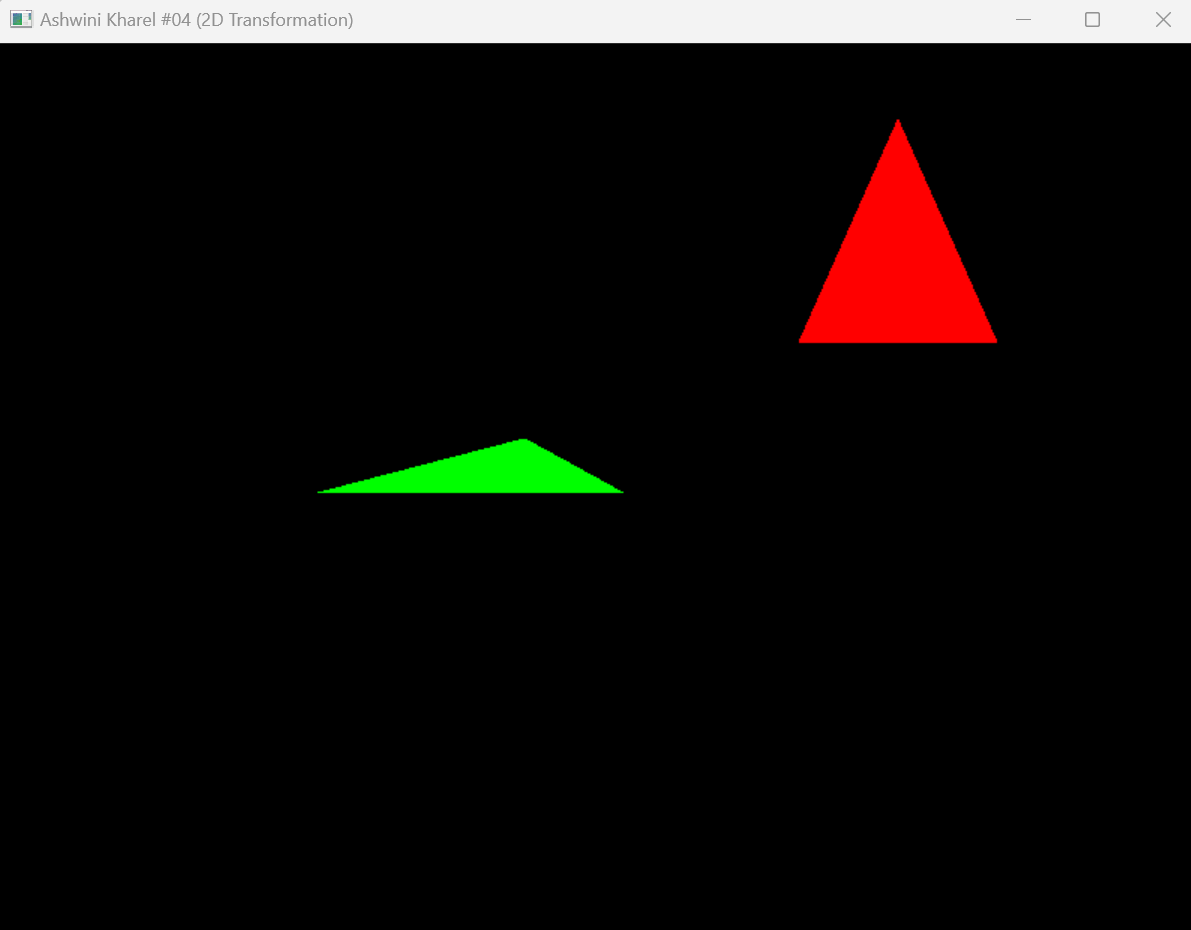
return 0;

}

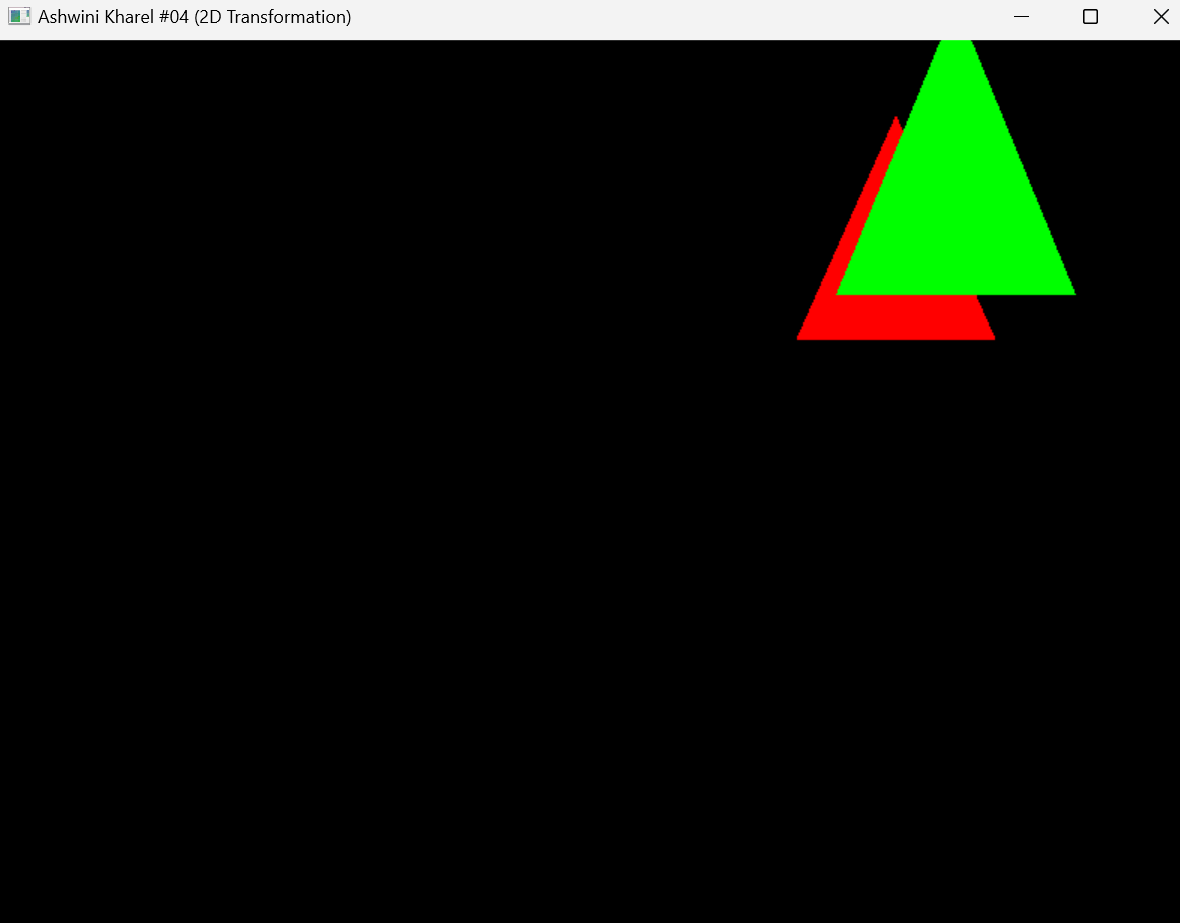
1. **OUTPUT :**
2. **2D-TRANSLATION :**



1. **2D-ROTATION :**



1. **2D-SCALING :**



**LAB 7**

**3D TRANSFORMATION**

**THEORY :**

3D transformations are operations that modify the position, size, orientation, or shape of 3D objects in a three-dimensional space. These transformations are represented using 4×4 matrices and applied using homogeneous coordinates. 3D transformations include translation (moving objects), scaling (changing size), rotation (turning around an axis), reflection (mirroring across a plane), and shearing (distorting the shape).

1. **ALGORITHM :**

1) Input the coordinates of front face and back face.

2) **Input the** transformation factors (Translation, Rotation and Scaling).

i) Translation:

x' = x + Tx y' = y + Ty z' = z + Tz

ii) Rotation:

* About X-axis (angle θ):

y' = y \* cos(θ) - z \* sin(θ)

z' = y \* sin(θ) + z \* cos(θ)

* About Y-axis:

x' = x \* cos(θ) + z \* sin(θ)

z' = -x \* sin(θ) + z \* cos(θ)

* About Z-axis:

x' = x \* cos(θ) - y \* sin(θ)

y' = x \* sin(θ) + y \* cos(θ)

iii) Scaling (Sx, Sy):

x' = x \* Sx

y' = y \* Sy

z' = z \* Sz

3)Apply transformations to each vertex and update the object.

4)End

1. **SOURCE CODE :**

#include <GL/glut.h>

#include <iostream>

#include <cmath>

using namespace std;

float angle = 0.0;

void display() {

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

glLoadIdentity();

glTranslatef(0.5f, -0.5f, -5.0f); //translation

glRotatef(angle, 0.0f, 1.0f, 0.0f); //rotation

glScalef(-0.5, 0.5f, 0.5f); //scaling

glBegin(GL\_QUADS);

//front face

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

glVertex3f(-0.5f, -0.5f, 0.5f);

glVertex3f(0.5f, -0.5f, 0.5f);

glVertex3f(0.5f, 0.5f, 0.5f);

glVertex3f(-0.5f, 0.5f, 0.5f);

//back face

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

glVertex3f(-0.5f, -0.5f, -0.5f);

glVertex3f(-0.5f, 0.5f, -0.5f);

glVertex3f(0.5f, 0.5f, -0.5f);

glVertex3f(0.5f, -0.5f, -0.5f);

glEnd();

glutSwapBuffers();

}

void update(int value) {

angle+=1.0f;

if(angle>360){

angle-=360;

}

glutPostRedisplay();

glutTimerFunc(16, update, 0);

}

// Initialization function

void init() {

glEnable(GL\_DEPTH\_TEST);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluPerspective(45.0f, 1.0f, 1.0f, 10.0f);

glMatrixMode(GL\_MODELVIEW);

}

// Main function

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

glutInit(&argc, argv);

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

glutInitWindowSize(800,600);

glutCreateWindow("Ashwini Kharel #04 (3D Transformation)");

init();

glutDisplayFunc(display);

glutTimerFunc(16, update, 0);

glutMainLoop();

return 0;

}

1. **OUTPUT :**

A computer screen shot of a red cube

AI-generated content may be incorrect.

**LAB 8**

**VISIBLE SURFACE DETECTION (DEPTH BUFFER ALGORITHM)**

**THEORY :**

Visible Surface Detection determines which surfaces of 3D objects are visible from a given viewpoint. Common methods include Back Face Detection, Z-Buffer method, A-buffer method, Scan line algorithm, Binary Space Partitioning (BSP), Painter’s algorithm and Area Sub-division method. The **Depth Buffer Algorithm**, also known as the **Z-buffer algorithm**, is a widely used technique in computer graphics for hidden surface removal. It works by maintaining a depth buffer (Z-buffer) alongside the color buffer, storing depth values for each pixel on the screen.

1. **ALGORITHM :**
2. Enable depth buffering for proper visibility based on depth.
3. Set the perspective projection with gluPerspective() to define the 3D view.
4. Draw the red cube at a farther position.
5. Draw the blue cube at a closer position.
6. Compares the depth of objects.
7. Clear depth buffer to farthest value.
8. Use glutSwapBuffers() to display the result.
9. **SOURCE CODE :**

//Hidden surface removal algorithm

#include<GL/glut.h>

//display function

void display() {

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

glLoadIdentity();

//draw red cube at a farther depth

glPushMatrix();

glTranslatef(-0.5f, 0.0f, -3.0f); //translation

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

glutSolidCube(0.7);

glPopMatrix();

//draw blue cube at a nearer depth

glPushMatrix();

glTranslatef(0.5f, 0.0f, -2.5f); //adjusted position

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

glutSolidCube(0.7);

glPopMatrix();

glutSwapBuffers();

}

// Initialization function

void init() {

glEnable(GL\_DEPTH\_TEST); //enable depth buffering

glDepthFunc(GL\_LESS); //Depth test function

glClearDepth(1.0f); //clear depth buffer to farthest value

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluPerspective(45.0, 1.0, 1.0, 10.0);

glMatrixMode(GL\_MODELVIEW);

}

// Main function //callback functions of OpenGL

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

glutInit(&argc, argv);

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

glutInitWindowSize(1000,600);

glutCreateWindow("Ashwini Kahrel #4 (Hidden Surface Removal)");

init();

glutDisplayFunc(display);

glutMainLoop();

return 0;

}

1. **OUTPUT :**

A screenshot of a computer

AI-generated content may be incorrect.

**LAB 9**

**BASIC OPENGL DRAWING**

**THEORY :**

OpenGL (Open Graphics Library) is a cross-platform API for rendering 2D and 3D graphics. Basic drawing in OpenGL involves setting up a rendering context, defining geometric shapes, and displaying them on the screen. Using libraries like **GLUT** or **GLFW**, an OpenGL window is initialized, where objects can be drawn using primitive functions like glBegin(GL\_TRIANGLES), glVertex(), and glEnd(). Colors are applied using glColor3f(), and transformations such as translation, rotation, and scaling can be performed with functions like glTranslatef(), glRotatef(), and glScalef(). After defining the shape, glFlush() or glutSwapBuffers() is used to render it on the screen.

1. **ALGORITHM :**
2. Initialize GLUT using glutInit(&argc, argv) to set up the environment.
3. Create three separate windows using glutCreateWindow() for drawing a line, triangle, and quadrilateral.
4. In the line() function, clear the screen, draw a line with glBegin(GL\_LINES), set endpoints with glVertex2f(), and call glFlush() to render it.
5. In the triangle() function, clear the screen, set the color to red, draw a triangle with glBegin(GL\_TRIANGLES), set the three vertices, and call glFlush() to render it.
6. In the quad() function, clear the screen, set the color to blue, draw a quadrilateral with glBegin(GL\_QUADS), set the four vertices, and call glFlush() to render it.
7. Call glutMainLoop() to display and maintain the rendered shapes in the windows.
8. **SOURCE CODE :**

#include <GL/glut.h>

int window1, window2, window3;

// Display function for the Line

void displayLine() {

glClear(GL\_COLOR\_BUFFER\_BIT);

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

glBegin(GL\_LINES);

glVertex2f(-0.8, 0.0);

glVertex2f(0.8, 0.0);

glEnd();

glFlush();

}

// Display function for the Triangle

void displayTriangle() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(1.0, 0.0, 0.0); // Red color

glBegin(GL\_TRIANGLES);

glVertex2f(-0.3, -0.5);

glVertex2f(0.3, -0.5);

glVertex2f(0.0, 0.3);

glEnd();

glFlush();

}

// Display function for the Quadrilateral

void displayQuadrilateral() {

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0, 0.0, 1.0);

glBegin(GL\_QUADS);

glVertex2f(-0.4, 0.4);

glVertex2f(-0.4, -0.4);

glVertex2f(0.4, -0.4);

glVertex2f(0.4, 0.4);

glEnd();

glFlush();

}

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

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

// Window 1: Line

glutInitWindowSize(400, 400);

glutInitWindowPosition(50, 50);

window1 = glutCreateWindow("Ashwini Kharel #04 Line");

glutDisplayFunc(displayLine);

// Window 2: Triangle

glutInitWindowSize(400, 400);

glutInitWindowPosition(400, 50);

window2 = glutCreateWindow("Ashwini Kharel #04 Triangle");

glutDisplayFunc(displayTriangle);

// Window 3: Quadrilateral

glutInitWindowSize(400, 400);

glutInitWindowPosition(750, 50);

window3 = glutCreateWindow("Ashwini Kharel #04 Quadrilateral");

glutDisplayFunc(displayQuadrilateral);

glutMainLoop();

return 0;

}

1. **OUTPUT :**

