LAB 1: **DDA line drawing algorithm to generate a line.**

#include <iostream>

#include <graphics.h>

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

int dx = x2 - x1;

int dy = y2 - y1;

int steps;

if (abs(dx) > abs(dy))

steps = abs(dx);

else

steps = 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++) {

putpixel((int)x, (int)y, WHITE); // Draw the pixel

x += xIncrement; // Increment x

y += yIncrement; // Increment y

}

}

int main() {

printf("Lab 1\n");

int gd = DETECT, gm;

initgraph(&gd, &gm, ""); // Initialize graphics mode

// Coordinates of the endpoints of the line

int x1 = 100, y1 = 100, x2 = 400, y2 = 300;

drawLineDDA(x1, y1, x2, y2);

getch(); // Wait for a key press

closegraph(); // Close graphics mode

return 0;

}

LAB 2: **Bresenham’s line drawing algorithm to generate a line.**

#include <iostream>

#include <graphics.h>

using namespace std;

void drawline(int x0, int y0, int x1, int y1)

{

int dx, dy, p, x, y;

int gdriver = DETECT, gmode, error;

initgraph(&gdriver, &gmode, "");

dx = x1 - x0;

dy = y1 - y0;

x = x0;

y = y0;

p = 2 \* dy - dx;

while (x < x1)

{

if (p >= 0)

{

putpixel(x, y, 7);

y = y + 1;

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

}

else

{

putpixel(x, y, 7);

p = p + 2 \* dy;

}

x = x + 1;

}

}

int main()

{

int x0, y0, x1, y1;

cout << "Enter co-ordinates of first point: ";

cin >> x0 >> y0;

cout << "Enter co-ordinates of second point: ";

cin >> x1 >> y1;

drawline(x0, y0, x1, y1);

getch();

return 0;

}

LAB 3: **Midpoint circle drawing algorithm for circle drawing.**

#include <graphics.h>

void drawCircleMidpoint(int xc, int yc, int radius) {

int x = radius;

int y = 0;

int err = 0;

while (x >= y) {

putpixel(xc + x, yc + y, WHITE);

putpixel(xc + y, yc + x, WHITE);

putpixel(xc - y, yc + x, WHITE);

putpixel(xc - x, yc + y, WHITE);

putpixel(xc - x, yc - y, WHITE);

putpixel(xc - y, yc - x, WHITE);

putpixel(xc + y, yc - x, WHITE);

putpixel(xc + x, yc - y, WHITE);

if (err <= 0) {

y += 1;

err += 2 \* y + 1;

}

if (err > 0) {

x -= 1;

err -= 2 \* x + 1;

}

}

}

int main() {

int gd = DETECT, gm;

initgraph(&gd, &gm, ""); // Initialize graphics mode

// Coordinates of the center of the circle and its radius

int xc = 300, yc = 200, radius = 100;

drawCircleMidpoint(xc, yc, radius);

getch(); // Wait for a key press

closegraph(); // Close graphics mode

return 0;

}

LAB 4: **Mid point circle drawing algorithm to draw ellipse.**

#include <iostream>

#include <graphics.h>

using namespace std;

void drawEllipseMidpoint(int xc, int yc, int rx, int ry) {

int x = 0, y = ry;

int rx2 = rx \* rx;

int ry2 = ry \* ry;

int twoRx2 = 2 \* rx2;

int twoRy2 = 2 \* ry2;

int p;

int px = 0;

int py = twoRx2 \* y;

// Region 1

p = ry2 - rx2 \* ry + 0.25 \* rx2;

while (px < py) {

putpixel(x + xc, y + yc, WHITE);

putpixel(-x + xc, y + yc, WHITE);

putpixel(x + xc, -y + yc, WHITE);

putpixel(-x + xc, -y + yc, WHITE);

x++;

px += twoRy2;

if (p < 0) {

p += ry2 + px;

} else {

y--;

py -= twoRx2;

p += ry2 + px - py;

}

}

// Region 2

p = ry2 \* (x + 0.5) \* (x + 0.5) + rx2 \* (y - 1) \* (y - 1) - rx2 \* ry2;

while (y >= 0) {

putpixel(x + xc, y + yc, WHITE);

putpixel(-x + xc, y + yc, WHITE);

putpixel(x + xc, -y + yc, WHITE);

putpixel(-x + xc, -y + yc, WHITE);

y--;

py -= twoRx2;

if (p > 0) {

p += rx2 - py;

} else {

x++;

px += twoRy2;

p += rx2 - py + px;

}

}

}

int main() {

int gd = DETECT, gm;

initgraph(&gd, &gm, ""); // Initialize graphics mode

// Coordinates of the center of the ellipse and its radii

int xc = 300, yc = 200, rx = 150, ry = 100;

drawEllipseMidpoint(xc, yc, rx, ry);

getch(); // Wait for a key press

closegraph(); // Close graphics mode

return 0;

}

LAB 5: **Implementation of 2-D transformation.**

#include <iostream>

#include <complex>

#include <conio.h>

#include <graphics.h>

#include <math.h>

using namespace std;

typedef complex<double> point;

#define x real()

#define y imag()

void displaymenu();

int drawpolygon(point, point, point, point);

point translation(point, int, int);

point scaling(point, int, int);

point rotation(point, int, int, float);

void reflectionmenu();

point reflectionthx(point);

point reflectionthy(point);

point reflectionthymx(point);

point reflectionthyx(point);

point reflectionthline(point, point, point);

void shearingmenu();

point shearingx(point, int);

point shearingy(point, int);

point shearingxy(point, int, int);

int main()

{

int gd = DETECT, gm;

point E, F, G, H;

initgraph(&gd, &gm, "C:\\TC\\BGI");

int x1, y1, x2, y2, x3, y3, x4, y4, choice, subchoice, i;

cout << "Enter four coordinates of polygon(one in a single line): ";

cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3 >> x4 >> y4;

point A(x1, y1);

point B(x2, y2);

point C(x3, y3);

point D(x4, y4);

drawpolygon(A, B, C, D);

displaymenu();

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

int a, b;

cout << "Enter translation distances: ";

cin >> a >> b;

E = translation(A, a, b);

F = translation(B, a, b);

G = translation(C, a, b);

H = translation(D, a, b);

setcolor(RED);

drawpolygon(E, F, G, H);

getch();

break;

case 2:

float angle, ang;

int c, d; // pivot

cout << "Enter pivot point for rotation: ";

cin >> c >> d;

cout << "Enter angle through which u want to rotate: ";

cin >> ang;

angle = (ang \* 3.14) / 180;

E = rotation(A, c, d, angle);

F = rotation(B, c, d, angle);

G = rotation(C, c, d, angle);

H = rotation(D, c, d, angle);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

case 3:

int sx, sy;

cout << "Enter scaling factors(Sx,Sy): ";

cin >> sx >> sy;

E = scaling(A, sx, sy);

F = scaling(B, sx, sy);

G = scaling(C, sx, sy);

H = scaling(D, sx, sy);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

case 4:

reflectionmenu();

cout << "Choose type of reflection..";

cin >> subchoice;

switch (subchoice)

{

case 1:

E = reflectionthx(A);

F = reflectionthx(B);

G = reflectionthx(C);

H = reflectionthx(D);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

case 2:

E = reflectionthy(A);

F = reflectionthy(B);

G = reflectionthy(C);

H = reflectionthy(D);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

case 3:

E = reflectionthyx(A);

F = reflectionthyx(B);

G = reflectionthyx(C);

H = reflectionthyx(D);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

case 4:

E = reflectionthymx(A);

F = reflectionthymx(B);

G = reflectionthymx(C);

H = reflectionthymx(D);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

case 5:

int a1, b1, a2, b2;

cout << "Enter starting and end coordinates of line:";

cin >> a1 >> b1 >> a2 >> b2;

point X(a1, b1);

point Y(a2, b2);

E = reflectionthline(A, X, Y);

F = reflectionthline(B, X, Y);

G = reflectionthline(C, X, Y);

H = reflectionthline(D, X, Y);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

}

break;

case 5:

shearingmenu();

int shx, shy;

cout << "Choose shearing option:";

cin >> subchoice;

switch (subchoice)

{

case 1:

cout << "enter shearing distance: ";

cin >> shx;

E = A;

F = shearingx(B, shx);

G = shearingx(C, shx);

H = D;

setcolor(RED);

drawpolygon(E, F, G, H);

break;

case 2:

cout << "Enter shearing distance: ";

cin >> shx;

E = shearingy(A, shy);

F = B;

G = C;

H = shearingy(D, shy);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

case 3:

cout << "Enter x and y shearing distance: ";

cin >> shx >> shy;

E = shearingxy(A, shx, shy);

F = shearingxy(B, shx, shy);

G = shearingxy(C, shx, shy);

H = shearingxy(D, shx, shy);

setcolor(RED);

drawpolygon(E, F, G, H);

break;

}

break;

default:

cout << "Invalid Choice..";

break;

}

getch();

closegraph();

}

void displaymenu()

{

cout << "Press 1 for translation." << endl;

cout << "Press 2 for rotation." << endl;

cout << "Press 3 for scaling." << endl;

cout << "Press 4 for reflecton." << endl;

cout << "Press 5 for shearing." << endl;

}

void reflectionmenu()

{

cout << "press 1 for reflection through x-axis." << endl;

cout << "press 2 for reflection through y-axis." << endl;

cout << "press 3 for reflection through line y=x." << endl;

cout << "press 4 for reflection through line y=-x." << endl;

cout << "press 5 for reftection through a line" << endl;

}

void shearingmenu()

{

cout << "press 1 for shearing through x-axis." << endl;

cout << "press 2 for shearing through y-axis." << endl;

cout << "press 3 for shearing through xy-axis." << endl;

}

point translation(point A, int a, int b)

{

point B(A.real() + a, A.imag() + b);

return B;

}

point rotation(point A, int a, int b, float angl)

{

point C = translation(A, -a, -b);

point B(((C.real() \* cos(angl)) - (C.imag() \* sin(angl))), ((C.real() \* sin(angl)) + (C.imag()) \* cos(angl)));

point D = translation(B, +a, +b);

return D;

}

point scaling(point A, int a, int b)

{

point B(A.real() \* a, A.imag() \* b);

return B;

}

point reflectionthx(point A)

{

point C(A.real(), -A.imag());

point B = translation(C, 0, 600);

return B;

}

point reflectionthy(point A)

{

point C(-A.real(), A.imag());

point B = translation(C, 600, 0);

return B;

}

point reflectionthyx(point A)

{

int p, q;

p = A.imag();

q = A.real();

point C(p, q);

return C;

}

point reflectionthymx(point A)

{

int p, q;

p = A.imag();

q = A.real();

point C(-p, -q);

point B = translation(C, 500, 500);

return B;

}

point reflectionthline(point P, point A, point B)

{

point Pt = P - A;

point Bt = B - A;

point Pr = Pt / Bt;

return conj(Pr) \* Bt + A;

}

point shearingx(point A, int shx)

{

point B(A.real() + shx \* A.imag(), A.imag());

return B;

}

point shearingy(point A, int shy)

{

point B(A.real(), A.imag() + shy \* A.real());

return B;

}

point shearingxy(point A, int shx, int shy)

{

point B((A.real() + shx \* A.imag()), (A.imag() + shy \* A.real()));

return B;

}

int drawpolygon(point W, point X, point Y, point Z)

{

line(W.real(), W.imag(), Z.real(), Z.imag());

delay(200);

line(Z.real(), Z.imag(), Y.real(), Y.imag());

delay(200);

line(Y.real(), Y.imag(), X.real(), X.imag());

delay(200);

line(X.real(), X.imag(), W.real(), W.imag());

delay(200);

return 0;

}

LAB 6: **Bezier curve implementation.**

#include <iostream>

#include <math.h>

#include <graphics.h>

using namespace std;

int main()

{

int x[4],y[4],i;

double put\_x,put\_y,t;

int gr=DETECT,gm;

initgraph (&gr,&gm,(char\*)"");

printf("Enter x and y coordinate:\n");

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

{

scanf("%d%d",&x[i],&y[i]);

}

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

{

line (x[i],y[i],x[i+1],y[i+1]);

}

for (t=0.0;t<=1.0;t=t+0.001)

{

put\_x=pow(1-t,3)\*x[0]+3\*t\*pow(1-t,2)\*x[1]+3\*t\*t\*(1-t)\*x[2]+pow(t,3)\*x[3];

put\_y=pow(1-t,3)\*y[0]+3\*t\*pow(1-t,2)\*y[1]+3\*t\*t\*(1-t)\*y[2]+pow(t,3)\*y[3];

putpixel(put\_x,put\_y,WHITE);

}

getch();

closegraph();

}

LAB 7: **Implementation of 3-D Transformation.**

#include<stdio.h>

#include<math.h>

#include<conio.h>

#include<graphics.h>

#include<stdlib.h>

int translation(int x, int tx) {

return (x + tx);

}

int scalar(int k, int x) {

return (k \* x);

}

int main() {

int gd = DETECT, gm;

initgraph(&gd, &gm, (char \*)" ");

printf("Enter the coordinates of cube (x,y,x2,y2)=");

int left, right, bottom, top, depth;

scanf("%d%d%d%d", &left, &top, &right, &bottom);

depth = fabs((right - left) / 2);

bar3d(left, top, right, bottom, depth, 1);

printf("1.Translation \n2.Scaling \n");

printf("Enter one of the options = ");

int option;

scanf("%d", &option);

switch(option) {

case 1: {

printf("Enter the x-translating factor = ");

int tx;

scanf("%d", &tx);

left = translation(left, tx);

right = translation(right, tx);

printf("Enter the y-translating factor = ");

int ty;

scanf("%d", &ty);

top = translation(top, ty);

bottom = translation(bottom, ty);

depth = fabs((right - left) / 2);

bar3d(left, top, right, bottom, depth, 1);

getch();

break;

}

case 2: {

printf("Assuming even scaling on all axes, enter the scalar factor = ");

int scale;

scanf("%d", &scale);

left = scalar(left, scale);

right = scalar(right, scale);

top = scalar(top, scale);

bottom = scalar(bottom, scale);

depth = fabs((right - left) / 2);

bar3d(left, top, right, bottom, depth, 1);

getch();

break;

}

default: {

printf("Invalid selection!\n");

break;

}

}

closegraph();

getch();

}

LAB 8: **Implementation Cohen-Sutherland line clipping algorithm.**

#include <graphics.h>

#include <conio.h>

#include <stdio.h>

#include <math.h>

int main()

{

int rcode\_begin[4] = {0, 0, 0, 0}, rcode\_end[4] = {0, 0, 0, 0}, region\_code[4];

int W\_xmax, W\_ymax, W\_xmin, W\_ymin, flag = 0;

float slope;

int x, y, x1, y1, i, xc, yc;

int gr = DETECT, gm;

initgraph(&gr, &gm, "C:\\TURBOC3\\BGI");

printf("\n\*\*\*\*\*\* Cohen Sutherlsnd Line Clipping algorithm \*\*\*\*\*\*\*\*\*\*\*");

printf("\n Now, enter XMin, YMin =");

scanf("%d %d", &W\_xmin, &W\_ymin);

printf("\n First enter XMax, YMax =");

scanf("%d %d", &W\_xmax, &W\_ymax);

printf("\n Please enter intial point x and y= ");

scanf("%d %d", &x, &y);

printf("\n Now, enter final point x1 and y1= ");

scanf("%d %d", &x1, &y1);

cleardevice();

rectangle(W\_xmin, W\_ymin, W\_xmax, W\_ymax);

line(x, y, x1, y1);

line(0, 0, 600, 0);

line(0, 0, 0, 600);

if (y > W\_ymax)

{

rcode\_begin[0] = 1; // Top

flag = 1;

}

if (y < W\_ymin)

{

rcode\_begin[1] = 1; // Bottom

flag = 1;

}

if (x > W\_xmax)

{

rcode\_begin[2] = 1; // Right

flag = 1;

}

if (x < W\_xmin)

{

rcode\_begin[3] = 1; // Left

flag = 1;

}

// end point of Line

if (y1 > W\_ymax)

{

rcode\_end[0] = 1; // Top

flag = 1;

}

if (y1 < W\_ymin)

{

rcode\_end[1] = 1; // Bottom

flag = 1;

}

if (x1 > W\_xmax)

{

rcode\_end[2] = 1; // Right

flag = 1;

}

if (x1 < W\_xmin)

{

rcode\_end[3] = 1; // Left

flag = 1;

}

if (flag == 0)

{

printf("No need of clipping as it is already in window");

}

flag = 1;

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

{

region\_code[i] = rcode\_begin[i] && rcode\_end[i];

if (region\_code[i] == 1)

flag = 0;

}

if (flag == 0)

{

printf("\n Line is completely outside the window");

}

else

{

slope = (float)(y1 - y) / (x1 - x);

if (rcode\_begin[2] == 0 && rcode\_begin[3] == 1) // left

{

y = y + (float)(W\_xmin - x) \* slope;

x = W\_xmin;

}

if (rcode\_begin[2] == 1 && rcode\_begin[3] == 0) // right

{

y = y + (float)(W\_xmax - x) \* slope;

x = W\_xmax;

}

if (rcode\_begin[0] == 1 && rcode\_begin[1] == 0) // top

{

x = x + (float)(W\_ymax - y) / slope;

y = W\_ymax;

}

if (rcode\_begin[0] == 0 && rcode\_begin[1] == 1) // bottom

{

x = x + (float)(W\_ymin - y) / slope;

y = W\_ymin;

}

// end points

if (rcode\_end[2] == 0 && rcode\_end[3] == 1) // left

{

y1 = y1 + (float)(W\_xmin - x1) \* slope;

x1 = W\_xmin;

}

if (rcode\_end[2] == 1 && rcode\_end[3] == 0) // right

{

y1 = y1 + (float)(W\_xmax - x1) \* slope;

x1 = W\_xmax;

}

if (rcode\_end[0] == 1 && rcode\_end[1] == 0) // top

{

x1 = x1 + (float)(W\_ymax - y1) / slope;

y1 = W\_ymax;

}

if (rcode\_end[0] == 0 && rcode\_end[1] == 1) // bottom

{

x1 = x1 + (float)(W\_ymin - y1) / slope;

y1 = W\_ymin;

}

}

delay(1000);

clearviewport();

rectangle(W\_xmin, W\_ymin, W\_xmax, W\_ymax);

line(0, 0, 600, 0);

line(0, 0, 0, 600);

setcolor(RED);

line(x, y, x1, y1);

getch();

closegraph();

}

LAB 9: **Implementation of Liang-Barsky algorithm.**

#include <stdio.h>

#include <graphics.h>

void liangBarsky(int x1, int y1, int x2, int y2, int xmin, int ymin, int xmax, int ymax) {

float t1 = 0, t2 = 1;

int dx = x2 - x1, dy = y2 - y1;

int p[4] = {-dx, dx, -dy, dy};

int q[4] = {x1 - xmin, xmax - x1, y1 - ymin, ymax - y1};

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

if (p[i] == 0 && q[i] < 0) {

printf("Line is parallel to clipping window and outside of it\n");

return;

}

float t = (float)q[i] / p[i];

if (p[i] < 0) {

if (t > t1) t1 = t;

} else if (p[i] > 0) {

if (t < t2) t2 = t;

}

}

if (t1 < t2) {

int x1\_new = x1 + t1 \* dx;

int y1\_new = y1 + t1 \* dy;

int x2\_new = x1 + t2 \* dx;

int y2\_new = y1 + t2 \* dy;

setcolor(WHITE);

line(x1\_new, y1\_new, x2\_new, y2\_new);

} else {

printf("Line lies completely outside the clipping window\n");

}

}

int main() {

int gd = DETECT, gm;

initgraph(&gd, &gm, NULL);

// Clipping window coordinates

int xmin = 50, ymin = 50, xmax = 200, ymax = 200;

rectangle(xmin, ymin, xmax, ymax);

// Line coordinatess

int x1, y1, x2, y2;

printf("Enter the coordinates of the line (x1 y1 x2 y2): ");

scanf("%d %d %d %d", &x1, &y1, &x2, &y2);

setcolor(YELLOW);

line(x1, y1, x2, y2);

liangBarsky(x1, y1, x2, y2, xmin, ymin, xmax, ymax);

getch();

closegraph();

return 0;

}

LAB 11: **Draw a line using OpenGL.**

#include <windows.h>

#include <GL/glut.h>

void init(void);

void lineSegment(void);

// driver program

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

{

glutInit(&argc, argv);

glutInitWindowSize(700, 700); // whole window size initialize

glutInitWindowPosition(100, 100); // window position initialize

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB);

glutCreateWindow("Line");

init();

glutDisplayFunc(lineSegment);

glutMainLoop();

return EXIT\_SUCCESS;

}

void init(void)

{

glClearColor(1.0, 10.0, 100.0, 0.0);

glMatrixMode(GL\_PROJECTION);

gluOrtho2D(0.0, 200.0, 0.0, 150.0);

}

void lineSegment(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glColor3f(0.0, 0.0, 0.0);

// GL\_POINTS, GL\_LINES, GL\_LINE\_STRIP, GL\_LINE\_LOOP, GL\_TRIANGLES, GL\_TRIANGLE\_STRIP, GL\_TRIANGLE\_FAN, GL\_QUADS, GL\_QUAD\_STRIP, and GL\_POLYGON

glBegin(GL\_LINES);

glVertex2i(180, 15);

glVertex2i(10, 145);

glEnd();

glFinish();

}

LAB 12: **Draw a triangle using OpenGL**

#include <GL/glut.h>

void init(void)

{

glClearColor(1.0, 1.0, 1.0, 0.0); // last value is alpha (transparency)

glMatrixMode(GL\_PROJECTION);

gluOrtho2D(0.0, 200.0, 0.0, 150.0); // orthographic projection

}

void triangle(void)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

glBegin(GL\_POLYGON);

glVertex3f(0.5, 0.0, 0.5);

glVertex3f(0.5, 0.0, 0.0);

glVertex3f(0.0, 0.5, 0.0);

glVertex3f(0.0, 0.0, 0.5);

glEnd();

glFlush();

}

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

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_RGB | GLUT\_SINGLE); // Single frame buffer

glutInitWindowSize(400, 300);

glutInitWindowPosition(100, 100);

glutCreateWindow("Triangle");

glutDisplayFunc(triangle);

glutMainLoop();

return 0;

}