# DDA LDA :

#include <GL/glut.h> #include <iostream> #include <cmath> using namespace std;

int xStart, yStart, xEnd, yEnd; int lineStyle = 1;

void plot(int x, int y) { glBegin(GL\_POINTS); glVertex2i(x, y); glEnd();

}

void drawLineDDA(int x1, int y1, int x2, int y2) { float dx = x2 - x1;

float dy = y2 - y1; float steps;

if (abs(dx) > abs(dy)) steps = abs(dx);

else

steps = abs(dy);

float xInc = dx / steps; float yInc = dy / steps;

float x = x1; float y = y1;

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

if (lineStyle == 1) draw = true;

else if (lineStyle == 2 && i % 2 == 0) draw = true; else if (lineStyle == 3 && i % 6 < 3) draw = true; else if (lineStyle == 4) draw = true;

if (draw)

plot(round(x), round(y));

x += xInc; y += yInc;

}

}

void drawAxes() { glColor3f(0.0, 0.0, 1.0);

for (int i = -500; i <= 500; i++) { plot(i, 0);

plot(0, i);

}

}

void display() { glClear(GL\_COLOR\_BUFFER\_BIT); drawAxes();

glColor3f(1.0, 0.0, 0.0); drawLineDDA(xStart, yStart, xEnd, yEnd);

glFlush();

}

void mouse(int button, int state, int x, int y) {

if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) { xStart = x - 250;

yStart = 250 - y;

}

if (button == GLUT\_RIGHT\_BUTTON && state == GLUT\_DOWN) { xEnd = x - 250;

yEnd = 250 - y; glutPostRedisplay();

}

}

void init() { glClearColor(1, 1, 1, 1);

gluOrtho2D(-250, 250, -250, 250);

glPointSize(2.0);

}

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

cout << "\nLINE DRAWING STYLES USING DDA ALGORITHM\n";

cout << "1. Simple Line\n"; cout << "2. Dotted Line\n"; cout << "3. Dashed Line\n"; cout << "4. Solid Line\n";

cout << "Enter your choice (1-4): "; cin >> lineStyle;

glutInit(&argc, argv); glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); glutInitWindowSize(500, 500); glutCreateWindow("DDA Line Drawing with Styles");

init(); glutDisplayFunc(display); glutMouseFunc(mouse); glutMainLoop();

return 0;

}

# Breshnam Circle :

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

using namespace std;

int centerX = 0, centerY = 0, radius = 50;

// Function to plot all eight symmetric points for the circle void plotCirclePoints(int x, int y) {

glBegin(GL\_POINTS); glVertex2i(centerX + x, centerY + y); glVertex2i(centerX - x, centerY + y); glVertex2i(centerX + x, centerY - y); glVertex2i(centerX - x, centerY - y); glVertex2i(centerX + y, centerY + x); glVertex2i(centerX - y, centerY + x); glVertex2i(centerX + y, centerY - x);

glVertex2i(centerX - y, centerY - x); glEnd();

}

// Function implementing Bresenham’s Circle Algorithm void drawCircle() {

int x = 0, y = radius; int d = 3 - 2 \* radius;

while (x <= y) { plotCirclePoints(x, y); x++;

if (d < 0) {

d += 4 \* x + 6;

} else { y--;

d += 4 \* (x - y) + 10;

}

}

}

// Function to draw coordinate axes with dividing points void drawAxes() {

glColor3f(0.0, 0.0, 0.0); // Black color for axes

// Draw X and Y Axes glBegin(GL\_LINES);

glVertex2i(-250, 0); glVertex2i(250, 0); // X-axis

glVertex2i(0, -250); glVertex2i(0, 250); // Y-axis glEnd();

// Draw dividing points on the axes glPointSize(5.0); // Make the points visible glBegin(GL\_POINTS);

for (int i = -250; i <= 250; i += 50) { glVertex2i(i, 0); // Points on the X-axis glVertex2i(0, i); // Points on the Y-axis

}

glEnd();

}

// Display function to render the scene void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

drawAxes(); // Draw coordinate axes with dividing points

glColor3f(1.0, 0.0, 0.0); // Set circle color to red drawCircle(); // Draw the circle using Bresenham's algorithm

glutSwapBuffers();

}

// Initialize OpenGL settings void init() {

glClearColor(1.0, 1.0, 1.0, 1.0); // White background glMatrixMode(GL\_PROJECTION); glLoadIdentity();

gluOrtho2D(-250, 250, -250, 250);

glPointSize(2.0);

}

// Main function to initialize OpenGL int main(int argc, char\*\* argv) {

cout << "Enter center coordinates (x y): "; cin >> centerX >> centerY;

cout << "Enter circle radius: "; cin >> radius;

glutInit(&argc, argv); glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB); glutInitWindowSize(500, 500);

glutCreateWindow("Bresenham's Circle Drawing Algorithm with Axes");

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

return 0;

}

# Polygon Transformation:

#include <GL/freeglut.h> #include <iostream>

#include <cmath> // For sin, cos using namespace std;

struct Point { float x, y;

} points[4]; // Original polygon vertices

// Transformation choice and parameters

int choice;

float tx = 0, ty = 0, sx = 1, sy = 1, angle = 0, shx = 0, shy = 0;

// Function to draw the solid X and Y axes with tick marks void drawAxesWithTicks() {

glColor3f(0, 1, 0); // green color for axes glLineWidth(2.0);

// Draw main X and Y axes glBegin(GL\_LINES);

glVertex2f(-100, 0); glVertex2f(100, 0); // X-axis

glVertex2f(0, -100); glVertex2f(0, 100); // Y-axis glEnd();

glLineWidth(1.0); // Reset line width

// Draw tick marks glBegin(GL\_LINES);

for (int i = -100; i <= 100; i += 10) { // Adjust step size for tick spacing if (i != 0) { // Skip center

// Vertical ticks on X-axis glVertex2f(i, -2);

glVertex2f(i, 2);

// Horizontal ticks on Y-axis glVertex2f(-2, i); glVertex2f(2, i);

}

}

glEnd();

}

// Function to draw a polygon outline given an array of points

void drawPolygon(Point pts[4], float r, float g, float b) { glColor3f(r, g, b);

glBegin(GL\_LINE\_LOOP); for (int i = 0; i < 4; i++) {

glVertex2f(pts[i].x, pts[i].y);

}

glEnd();

}

void display() { glClear(GL\_COLOR\_BUFFER\_BIT); glMatrixMode(GL\_MODELVIEW); glLoadIdentity();

// Draw axes with tick marks drawAxesWithTicks();

// Draw the original polygon (Before transformation) drawPolygon(points, 1, 0, 1);

// Create a copy of the original points to transform (After transformation) Point transformed[4];

for (int i = 0; i < 4; i++) { transformed[i] = points[i];

}

// Apply the chosen transformation to the copy if (choice == 1) {

// Translation

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

transformed[i].x += tx; transformed[i].y += ty;

}

} else if (choice == 2) {

// Scaling

for (int i = 0; i < 4; i++) { transformed[i].x \*= sx; transformed[i].y \*= sy;

}

} else if (choice == 3) {

// Rotation (about the origin)

float rad = angle \* 3.14159f / 180; // Convert degrees to radians for (int i = 0; i < 4; i++) {

float x = transformed[i].x; float y = transformed[i].y;

transformed[i].x = x \* cos(rad) - y \* sin(rad); transformed[i].y = x \* sin(rad) + y \* cos(rad);

}

} else if (choice == 4) {

// Shearing

for (int i = 0; i < 4; i++) { float x = transformed[i].x; float y = transformed[i].y;

transformed[i].x = x + y \* shx; transformed[i].y = y + x \* shy;

}

}

// Draw the transformed polygon (After transformation) drawPolygon(transformed, 1, 0.5,0 );

glFlush();

}

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

// Input the original polygon's four points cout << "Enter 4 points (x, y): ";

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

cin >> points[i].x >> points[i].y;

// Ask the user for the transformation type

cout << "Choose Transformation: 1-Translate, 2-Scale, 3-Rotate, 4-Shear\n"; cin >> choice;

// Input the transformation parameters based on the user's choice if (choice == 1) {

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

} else if (choice == 2) {

cout << "Enter scale factors (sx, sy): "; cin >> sx >> sy;

} else if (choice == 3) {

cout << "Enter rotation angle (in degrees): "; cin >> angle;

} else if (choice == 4) {

cout << "Enter shear factors (shx, shy): "; cin >> shx >> shy;

}

// Initialize GLUT

glutInit(&argc, argv); glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); glutInitWindowSize(500, 500);

glutCreateWindow("2D Transformations with Tick Mark Axes");

// Set the background color to white glClearColor(1.0, 1.0, 1.0, 1.0);

// Set the orthogonal projection to cover the coordinate range [-100, 100] glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(-100, 100, -100, 100);

// Register the display function and start the main loop glutDisplayFunc(display);

glutMainLoop(); return 0;

}

# Color fill algorithm:

#include <GL/glut.h> #include <stack> #include <iostream>

using namespace std;

// Window dimensions const int WIDTH = 800; const int HEIGHT = 600;

// Color definitions

float boundaryColor[3] = {1.0, 0.0, 0.0}; // Red for boundary float fillColor[3] = {0.0, 1.0, 0.0}; // Green for fill

float backgroundColor[3] = {0.0, 0.0, 0.0}; // Black background

// Function to draw a pixel

void drawPixel(int x, int y, float color[3]) { glColor3fv(color); glBegin(GL\_POINTS);

glVertex2i(x, y); glEnd(); glFlush();

}

// Function to get pixel color

void getPixelColor(int x, int y, float color[3]) { glReadPixels(x, y, 1, 1, GL\_RGB, GL\_FLOAT, color);

}

// Boundary Fill Algorithm

void boundaryFill(int x, int y, float fillColor[3], float boundaryColor[3]) { stack<pair<int, int> > pixelStack;

pixelStack.push(make\_pair(x, y));

while (!pixelStack.empty()) {

pair<int, int> current = pixelStack.top(); pixelStack.pop();

int currX = current.first; int currY = current.second;

float currentColor[3]; getPixelColor(currX, currY, currentColor);

bool isBoundary = (currentColor[0] == boundaryColor[0] && currentColor[1] == boundaryColor[1] && currentColor[2] == boundaryColor[2]);

bool isFilled = (currentColor[0] == fillColor[0] && currentColor[1] == fillColor[1] && currentColor[2] == fillColor[2]);

bool isBackground = (currentColor[0] == backgroundColor[0] && currentColor[1] == backgroundColor[1] && currentColor[2] == backgroundColor[2]);

if (!isBoundary && !isFilled && isBackground) { drawPixel(currX, currY, fillColor);

// Push neighboring pixels pixelStack.push(make\_pair(currX + 1, currY)); pixelStack.push(make\_pair(currX - 1, currY)); pixelStack.push(make\_pair(currX, currY + 1)); pixelStack.push(make\_pair(currX, currY - 1));

}

}

}

// Flood Fill Algorithm

void floodFill(int x, int y, float fillColor[3], float oldColor[3]) { stack<pair<int, int> > pixelStack; pixelStack.push(make\_pair(x, y));

while (!pixelStack.empty()) {

pair<int, int> current = pixelStack.top(); pixelStack.pop();

int currX = current.first; int currY = current.second;

float currentColor[3]; getPixelColor(currX, currY, currentColor);

bool isOldColor = (currentColor[0] == oldColor[0] && currentColor[1] == oldColor[1] && currentColor[2] == oldColor[2]);

if (isOldColor) {

drawPixel(currX, currY, fillColor);

// Push neighboring pixels pixelStack.push(make\_pair(currX + 1, currY)); pixelStack.push(make\_pair(currX - 1, currY)); pixelStack.push(make\_pair(currX, currY + 1)); pixelStack.push(make\_pair(currX, currY - 1));

}

}

}

// Draw two rectangles for demonstration void drawRectangles() {

// Left rectangle (for boundary fill) glColor3fv(boundaryColor); glBegin(GL\_LINE\_LOOP); glVertex2i(100, 100);

glVertex2i(300, 100);

glVertex2i(300, 400);

glVertex2i(100, 400); glEnd();

// Right rectangle (for flood fill) glColor3fv(boundaryColor); glBegin(GL\_LINE\_LOOP); glVertex2i(500, 100);

glVertex2i(700, 100);

glVertex2i(700, 400);

glVertex2i(500, 400); glEnd();

glFlush();

}

// Display callback function void display() {

glClear(GL\_COLOR\_BUFFER\_BIT); drawRectangles();

glFlush();

}

// Mouse callback function

void mouse(int button, int state, int x, int y) { y = HEIGHT - y; // Flip y coordinate

if (button == GLUT\_LEFT\_BUTTON && state == GLUT\_DOWN) {

// Boundary fill for left rectangle

if (x > 100 && x < 300 && y > 100 && y < 400) { boundaryFill(x, y, fillColor, boundaryColor);

}

// Flood fill for right rectangle

else if (x > 500 && x < 700 && y > 100 && y < 400) { floodFill(x, y, fillColor, backgroundColor);

}

}

}

// Initialize OpenGL void init() {

glClearColor(backgroundColor[0], backgroundColor[1], backgroundColor[2], 1.0); glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, WIDTH, 0, HEIGHT);

glMatrixMode(GL\_MODELVIEW);

}

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

glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); glutInitWindowSize(WIDTH, HEIGHT); glutCreateWindow("Simple Fill Algorithms Demo");

init();

glutDisplayFunc(display); glutMouseFunc(mouse);

cout << "Click inside the left rectangle for boundary fill" << endl; cout << "Click inside the right rectangle for flood fill" << endl;

glutMainLoop(); return 0;

}

# Bouncing Square animation:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Boucing square animation</title>

<style>

canvas{

border: 1px solid black;

background-color: rgb(212, 200, 200);

}

body{

margin: 0; display:flex;

flex-direction: column; align-items: center; justify-content: center; height: 100vh;

}

</style>

</head>

<body>

<h1>Bouncing square animation</h1>

<canvas id="mycanvas" width="500" height="500"></canvas>

<script>

const canvas = document.getElementById('mycanvas'); const ctx= canvas.getContext('2d');

let x=50; let y=50; let dx=2; let dy=3;

const boxsize=30;

function drawSquare(){ ctx.clearRect(0,0,canvas.width,canvas.height); ctx.fillStyle='black'; ctx.fillRect(x,y,boxsize,boxsize);

x=x+dx; y=y+dy;

if(x+boxsize>canvas.width||x<0) dx=dx\*-1; if(y+boxsize>canvas.height||y<0) dy=dy\*-1; requestAnimationFrame(drawSquare);

}

drawSquare();

</script>

</body>

</html>

# Moving car animation:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0"/>

<title>Car Driving with Mountains</title>

<style>

\* {

margin: 0;

padding: 0;

box-sizing: border-box;

}

html, body { height: 100%;

width: 100%; overflow: hidden;

background: linear-gradient(to top, #87CEEB 60%, #cbeeff 100%); position: relative;

font-family: sans-serif;

}

/\* Mountains \*/

.mountains { position: absolute; bottom: 120px; width: 100%;

height: 60%;

z-index: 1;

}

.mountain { position: absolute; bottom: 0;

width: 0;

height: 0;

border-left: 15vw solid transparent; border-right: 15vw solid transparent; border-bottom: 25vh solid #8B4513;

}

.mountain:nth-child(1) { left: 5vw; }

.mountain:nth-child(2) { left: 20vw; border-bottom-color: #A0522D; }

.mountain:nth-child(3) { left: 40vw; }

.mountain:nth-child(4) { left: 60vw; border-bottom-color: #A0522D; }

.mountain:nth-child(5) { left: 80vw; }

/\* Trees \*/

.trees {

position: absolute; bottom: 120px; width: 100%; height: 100px;

z-index: 2;

}

.tree {

position: absolute; bottom: 0;

width: 10px; height: 30px;

background: #8B4513;

}

.tree::after {

content: ''; position: absolute; bottom: 20px; left: -10px;

width: 30px; height: 30px;

background: #228B22; border-radius: 50%;

}

.tree:nth-child(1) { left: 10vw; }

.tree:nth-child(2) { left: 30vw; }

.tree:nth-child(3) { left: 50vw; }

.tree:nth-child(4) { left: 70vw; }

.tree:nth-child(5) { left: 90vw; }

/\* Road \*/

.road {

position: absolute; bottom: 0;

width: 100%; height: 120px; background: #333;

z-index: 3;

}

.line {

position: absolute; top: 55px;

width: 100%; height: 8px;

background: repeating-linear-gradient( to right,

#fff 0 30px, transparent 30px 60px

);

}

/\* Car \*/

.car {

position: absolute;

bottom: 120px; /\* Exactly on top of the road \*/ left: -200px;

width: 160px; height: 60px;

background: #e74c3c;

border-radius: 20px 20px 8px 8px; animation: moveCar 6s linear infinite; box-shadow: 0 4px 8px rgba(0,0,0,0.3); z-index: 4;

}

.car::before {

content: ''; position: absolute; width: 120px;

height: 40px; background: #c0392b; top: -30px;

left: 20px;

border-radius: 40px 40px 0 0;

}

.wheel {

position: absolute; bottom: -18px; width: 30px; height: 30px; background: #000;

border-radius: 50%; border: 4px solid #555;

}

.wheel.front { left: 20px;

}

.wheel.back { right: 20px;

}

@keyframes moveCar { 0% {

left: -200px;

} 100% {

left: 110%;

}

}

</style>

</head>

<body>

<!-- Mountains -->

<div class="mountains">

<div class="mountain"></div>

<div class="mountain"></div>

<div class="mountain"></div>

<div class="mountain"></div>

<div class="mountain"></div>

</div>

<!-- Trees -->

<div class="trees">

<div class="tree"></div>

<div class="tree"></div>

<div class="tree"></div>

<div class="tree"></div>

<div class="tree"></div>

</div>

<!-- Road -->

<div class="road">

<div class="line"></div>

</div>

<!-- Car -->

<div class="car">

<div class="wheel front"></div>

<div class="wheel back"></div>

</div>

</body>

</html>

# Coffee machine animation:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Simple Coffee Machine</title>

<style>

body {

display: flex;

justify-content: center; align-items: center; height: 100vh;

background-color: #dfbaba; font-family: Arial, sans-serif; margin: 0;

}

.coffee-machine { width: 250px; perspective: 1000px;

}

.machine-body {

background: linear-gradient(to right, #d3d3d3, #e0e0e0); padding: 20px;

border-radius: 15px;

box-shadow: 0 10px 25px rgba(0, 0, 0, 0.2); transform-style: preserve-3d;

transform: rotateX(5deg);

}

.display {

background-color: #4CAF50; color: white;

padding: 10px; text-align: center; border-radius: 5px;

margin-bottom: 15px; font-weight: bold;

box-shadow: inset 0 0 10px rgba(0, 0, 0, 0.3);

}

.coffee-btn { width: 100%; padding: 12px; border: none;

border-radius: 5px;

background: linear-gradient(to bottom, #8B4513, #A0522D); color: white;

font-weight: bold;

cursor: pointer; transition: all 0.2s;

box-shadow: 0 4px 0 #5D2906; margin-bottom: 20px;

}

.coffee-btn:hover {

background: linear-gradient(to bottom, #A0522D, #8B4513);

}

.coffee-btn:active { transform: translateY(2px);

box-shadow: 0 2px 0 #5D2906;

}

.coffee-outlet { position: relative; height: 180px; display: flex;

justify-content: center;

}

.cup {

width: 70px; height: 50px;

background-color: white; border: 2px solid #000000; border-radius: 0 0 15px 15px; position: absolute;

bottom: 0;

opacity: 0.9;

}

.coffee-stream { position: absolute; width: 25px; height: 0;

background: linear-gradient(to bottom, #4B371C, #6F4E37); border-radius: 0 0 5px 5px;

top: 20px;

transition: height 0.5s; z-index: 1;

}

</style>

</head>

<body>

<div class="coffee-machine">

<div class="machine-body">

<div class="display">Ready</div>

<button class="coffee-btn" id="make-coffee">Make Coffee</button>

<div class="coffee-outlet">

<div class="coffee-stream" id="stream"></div>

<div class="cup"></div>

</div>

</div>

</div>

<script>

document.addEventListener('DOMContentLoaded', function() { const display = document.querySelector('.display');

const coffeeBtn = document.getElementById('make-coffee'); const coffeeStream = document.getElementById('stream'); const cup = document.querySelector('.cup');

function makeCoffee() { display.textContent = "Brewing coffee..."; coffeeStream.style.height = '130px';

setTimeout(() => { coffeeStream.style.height = '0'; display.textContent = "Coffee ready!"; cup.style.backgroundColor = '#4B371C';

setTimeout(() => { display.textContent = 'Ready';

}, 2000);

}, 2500);

}

coffeeBtn.addEventListener('click', makeCoffee);

});

</script>

</body>

</html>