\*\*\*\* Write a C++ program for drawing graphics primitives and color it.

#include <graphics.h>

#include <iostream>

using namespace std;

int main() {

// Initialize the graphics mode

int gd = DETECT, gm;

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

// Set background color

setbkcolor(WHITE);

cleardevice();

// Set color for drawing primitives

setcolor(RED);

setfillstyle(SOLID\_FILL, RED);

// Draw a rectangle and fill it with color

rectangle(100, 100, 200, 200);

floodfill(150, 150, RED); // Point inside the rectangle

// Draw a circle and fill it with color

setcolor(BLUE);

setfillstyle(SOLID\_FILL, BLUE);

circle(300, 150, 50);

floodfill(300, 150, BLUE); // Point inside the circle

// Draw a line

setcolor(GREEN);

line(50, 300, 400, 300);

// Pause to view the output

cout << "Press any key to exit...";

getch(); // Wait for user input

// Close the graphics window

closegraph();

return 0;

}

1. Draw a concave polygon and fill it with desired color using scan fill algorithm. Apply the concept of inheritance.

#include<iostream>

#include<graphics.h>

#include<conio.h>

#include<algorithm>

using namespace std;

class Shape {

public:

virtual void draw() = 0;

};

class Polygon : public Shape {

public:

int n;

int \*x, \*y;

Polygon(int numVertices) {

n = numVertices;

x = new int[n];

y = new int[n];

}

~Polygon() {

delete[] x;

delete[] y;

}

void inputVertices() {

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

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

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

}

}

void draw() {

for (int i = 0; i < n - 1; i++) {

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

}

line(x[n - 1], y[n - 1], x[0], y[0]);

}

void scanFill(int color) {

for (int yScan = 0; yScan < getmaxy(); yScan++) {

vector<int> intersections;

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

int next = (i + 1) % n;

if ((y[i] <= yScan && y[next] > yScan) || (y[i] > yScan && y[next] <= yScan)) {

int xIntersect = x[i] + (yScan - y[i]) \* (x[next] - x[i]) / (y[next] - y[i]);

intersections.push\_back(xIntersect);

}

}

sort(intersections.begin(), intersections.end());

for (size\_t i = 0; i < intersections.size(); i += 2) {

for (int xScan = intersections[i]; xScan < intersections[i + 1]; xScan++) {

putpixel(xScan, yScan, color);

} } } }};

int main() {

int gd = DETECT, gm;

initgraph(&gd, &gm, "");

int n;

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

cin >> n;

Polygon poly(n);

poly.inputVertices();

poly.draw();

poly.scanFill(RED);

getch();

closegraph();

return 0;

}

2. Write C++ program to implement Cohen Southerland line clipping algorithm.

#include <iostream>

#include <graphics.h>

using namespace std;

// Region Codes for the Cohen-Sutherland Algorithm

const int INSIDE = 0; // 0000

const int LEFT = 1; // 0001

const int RIGHT = 2; // 0010

const int BOTTOM = 4; // 0100

const int TOP = 8; // 1000

// Clipping window boundaries

int xmin = 100, ymin = 100, xmax = 400, ymax = 300;

// Function to compute region code for a point (x, y)

int computeCode(int x, int y) {

int code = INSIDE; // Initial region is inside

if (x < xmin) code |= LEFT;

if (x > xmax) code |= RIGHT;

if (y < ymin) code |= BOTTOM;

if (y > ymax) code |= TOP;

return code;

}

// Function to clip the line from (x1, y1) to (x2, y2)

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

int code1 = computeCode(x1, y1); // Compute region code for (x1, y1)

int code2 = computeCode(x2, y2); // Compute region code for (x2, y2)

bool accept = false;

while (true) {

if ((code1 == 0) && (code2 == 0)) { // Both points inside

accept = true;

break;

} else if (code1 & code2) { // Both points outside (same region)

break;

} else {

int codeOut;

int x, y;

// Pick the endpoint that is outside the window

if (code1 != 0) codeOut = code1;

else codeOut = code2;

// Find intersection point

if (codeOut & TOP) { // Line intersects with top

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

y = ymax;

} else if (codeOut & BOTTOM) { // Line intersects with bottom

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

y = ymin;

} else if (codeOut & RIGHT) { // Line intersects with right

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

x = xmax;

} else if (codeOut & LEFT) { // Line intersects with left

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

x = xmin;

}

// Replace the point outside the window with the intersection point

if (codeOut == code1) {

x1 = x;

y1 = y;

code1 = computeCode(x1, y1);

} else {

x2 = x;

y2 = y;

code2 = computeCode(x2, y2);

}

}

}

if (accept) {

// Draw the clipped line in green

setcolor(GREEN);

line(x1, y1, x2, y2);

}}

int main() {

// Initialize graphics mode

int gd = DETECT, gm;

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

// Draw the clipping window (rectangular boundary)

setcolor(WHITE);

rectangle(xmin, ymin, xmax, ymax);

// Input line endpoints

int x1, y1, x2, y2;

cout << "Enter the coordinates of the line (x1, y1, x2, y2): ";

cin >> x1 >> y1 >> x2 >> y2;

// Draw the original line in red (for reference)

setcolor(RED);

line(x1, y1, x2, y2);

// Clip the line using the Cohen-Sutherland algorithm

cohenSutherlandClip(x1, y1, x2, y2);

// Wait for user input and close graphics window

getch();

closegraph();

return 0;

}

3. Write a c++ program for drawing a line using DDA and Bresahnam’s Line Drawing Algorithm

#include <iostream>

#include <graphics.h>

#include <cmath>

using namespace std;

class Graphics {

public:

void DDA\_Line(int x1, int y1, int x2, int y2) {

int dx = x2 - x1;

int dy = y2 - y1;

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

float xInc = dx / float(steps);

float yInc = dy / float(steps);

float x = x1, y = y1;

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

putpixel(round(x), round(y), WHITE);

x += xInc;

y += yInc;

}

}

void Bresenham\_Circle(int xc, int yc, int r) {

int x = 0, y = r;

int d = 3 - 2 \* r;

while (x <= y) {

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

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

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

if (d <= 0) {

d = d + 4 \* x + 6;

} else {

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

y--;

}

x++;

}

}

};

int main() {

int gd = DETECT, gm;

initgraph(&gd, &gm, "");

Graphics g;

g.DDA\_Line(100, 100, 300, 300); // Drawing line using DDA

g.Bresenham\_Circle(300, 300, 50); // Drawing circle using Bresenham

getch();

closegraph();

return 0;

}

4. Write C++/Java program to draw 2-D object and perform following basic transformations,

a) Scaling

b) Translation

c) Rotation

Use operator overloading.

#include <iostream>

#include <cmath>

using namespace std;

// Define a class for a 2D point

class Point {

public:

float x, y;

// Constructor to initialize a point

Point(float x\_val = 0, float y\_val = 0) : x(x\_val), y(y\_val) {}

// Operator overloading for Scaling

Point operator\*(float scale) {

return Point(x \* scale, y \* scale);

}

// Operator overloading for Translation

Point operator+(const Point& p) {

return Point(x + p.x, y + p.y);

}

// Operator overloading for Rotation (counterclockwise)

Point operator()(float angle) {

float rad = angle \* M\_PI / 180; // Convert angle to radians

float new\_x = x \* cos(rad) - y \* sin(rad);

float new\_y = x \* sin(rad) + y \* cos(rad);

return Point(new\_x, new\_y);

}

// Method to display the point

void display() {

cout << "(" << x << ", " << y << ")\n";

}};

int main() {

Point p1(2, 3); // Initial point (2, 3)

cout << "Original point: ";

p1.display();

// Scaling the point by a factor of 2

Point p2 = p1 \* 2;

cout << "After scaling by 2: ";

p2.display();

// Translating the point by (3, 4)

Point p3 = p1 + Point(3, 4);

cout << "After translation by (3, 4): ";

p3.display();

// Rotating the point by 90 degrees

Point p4 = p1(90);

cout << "After rotating by 90 degrees: ";

p4.display();

return 0;

}

5. Write C++ program to generate Hilbert curve using concept of fractals.

#include<iostream>

#include<stdlib.h>

#include<graphics.h>

using namespace std;

void move(int j, int h, int &x, int &y)

{

if (j == 1) // Up

y -= h;

else if (j == 2) // Right

x += h;

else if (j == 3) // Down

y += h;

else if (j == 4) // Left

x -= h;

lineto(x, y); // Drawing line to new point (x, y)

}

void hilbert(int r, int d, int l, int u, int i, int h, int &x, int &y)

{

if (i > 0)

{

i--;

hilbert(d, r, u, l, i, h, x, y); // Recursive call for first part

move(r, h, x, y); // Move to next position

hilbert(r, d, l, u, i, h, x, y); // Recursive call for second part

move(d, h, x, y); // Move to next position

hilbert(r, d, l, u, i, h, x, y); // Recursive call for third part

move(l, h, x, y); // Move to next position

hilbert(u, l, d, r, i, h, x, y); // Recursive call for fourth part

}

}

int main()

{

int n;

int x0 = 50, y0 = 150, x, y;

int h = 10, r = 2, d = 3, l = 4, u = 1;

cout << "Enter value of n (order of Hilbert curve): ";

cin >> n;

x = x0;

y = y0;

int gd = DETECT, gm;

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

moveto(x, y); // Move to initial position

hilbert(r, d, l, u, n, h, x, y); // Generate Hilbert curve

delay(10000); // Wait for 10 seconds

closegraph(); // Close graphics window

return 0;

}

6. 3D Cube Transformation for scaling, transformation, scaling using OpenGL.

#include <GL/glut.h>

// Initial cube position

float angle = 0.0; // Rotation angle

float scale = 1.0; // Scaling factor

float translateX = 0.0, translateY = 0.0, translateZ = -5.0; // Translation values

// Function to draw the cube

void drawCube() {

glBegin(GL\_QUADS);

// Front face

glColor3f(1.0, 0.0, 0.0);

glVertex3f(-1.0, -1.0, 1.0);

glVertex3f( 1.0, -1.0, 1.0);

glVertex3f( 1.0, 1.0, 1.0);

glVertex3f(-1.0, 1.0, 1.0);

// Back face

glColor3f(0.0, 1.0, 0.0);

glVertex3f(-1.0, -1.0, -1.0);

glVertex3f(-1.0, 1.0, -1.0);

glVertex3f( 1.0, 1.0, -1.0);

glVertex3f( 1.0, -1.0, -1.0);

// Top face

glColor3f(0.0, 0.0, 1.0);

glVertex3f(-1.0, 1.0, -1.0);

glVertex3f(-1.0, 1.0, 1.0);

glVertex3f( 1.0, 1.0, 1.0);

glVertex3f( 1.0, 1.0, -1.0);

// Bottom face

glColor3f(1.0, 1.0, 0.0);

glVertex3f(-1.0, -1.0, -1.0);

glVertex3f( 1.0, -1.0, -1.0);

glVertex3f( 1.0, -1.0, 1.0);

glVertex3f(-1.0, -1.0, 1.0);

// Right face

glColor3f(1.0, 0.0, 1.0);

glVertex3f( 1.0, -1.0, -1.0);

glVertex3f( 1.0, 1.0, -1.0);

glVertex3f( 1.0, 1.0, 1.0);

glVertex3f( 1.0, -1.0, 1.0);

// Left face

glColor3f(0.0, 1.0, 1.0);

glVertex3f(-1.0, -1.0, -1.0);

glVertex3f(-1.0, -1.0, 1.0);

glVertex3f(-1.0, 1.0, 1.0);

glVertex3f(-1.0, 1.0, -1.0);

glEnd();

}

// Function to apply transformations and draw the cube

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); // Clear the color and depth buffers

glLoadIdentity(); // Reset transformations

// Apply translation

glTranslatef(translateX, translateY, translateZ);

// Apply scaling

glScalef(scale, scale, scale);

// Apply rotation

glRotatef(angle, 1.0, 1.0, 0.0); // Rotate the cube around the X and Y axes

drawCube(); // Draw the cube

glutSwapBuffers(); // Swap the front and back buffers

}

// Function to handle key presses for transformation control

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

if (key == 'w') translateY += 0.1; // Move up

if (key == 's') translateY -= 0.1; // Move down

if (key == 'a') translateX -= 0.1; // Move left

if (key == 'd') translateX += 0.1; // Move right

if (key == 'q') translateZ += 0.1; // Move forward

if (key == 'e') translateZ -= 0.1; // Move backward

if (key == '+') scale += 0.1; // Scale up

if (key == '-') scale -= 0.1; // Scale down

if (key == 'r') angle += 5.0; // Rotate clockwise

if (key == 'l') angle -= 5.0; // Rotate counterclockwise

glutPostRedisplay(); // Redraw the scene

}

// Function to initialize OpenGL settings

void initOpenGL() {

glClearColor(0.0, 0.0, 0.0, 1.0); // Set background color to black

glEnable(GL\_DEPTH\_TEST); // Enable depth testing for 3D

glMatrixMode(GL\_PROJECTION); // Set the projection matrix mode

gluPerspective(45.0, 1.0, 0.1, 50.0); // Set perspective view

glMatrixMode(GL\_MODELVIEW); // Set the modelview matrix mode

}

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

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH); // Set display mode

glutInitWindowSize(500, 500); // Set window size

glutCreateWindow("3D Cube Transformation"); // Create window

initOpenGL(); // Initialize OpenGL settings

glutDisplayFunc(display); // Register display function

glutKeyboardFunc(keyboard); // Register keyboard input function

glutMainLoop(); // Enter the GLUT main loop

return 0;

}

7. Write C++ program to draw man walking in the rain with an umbrella. Apply the concept of polymorphism.

#include <graphics.h>

#include <conio.h>

#include <stdlib.h>

#include <dos.h>

#include <iostream>

using namespace std;

class WalkingMan {

int rhx, rhy;

public:

void drawRain(int i); // Function to draw rain

void drawMan(int i); // Function to draw walking man

};

// Function to draw rain (simple lines falling)

void WalkingMan::drawRain(int i) {

for (int j = 0; j < 5; j++) {

line(30 + 20 \* j, i, 30 + 20 \* j, i + 10);

}

}

// Function to draw walking man

void WalkingMan::drawMan(int i) {

// Platform

line(20, 380, 580, 380); // Platform

// Walking man

if (i % 2 == 0) {

// Left leg and right leg while walking

line(25 + i, 380, 35 + i, 340); // Left leg

line(45 + i, 380, 35 + i, 340); // Right leg

// Left hand and right hand while walking

line(35 + i, 310, 25 + i, 330); // Left hand

} else {

line(35 + i, 380, 35 + i, 340); // Left leg (standing)

line(35 + i, 310, 40 + i, 330); // Right hand (raising umbrella)

}

// Body

line(35 + i, 340, 35 + i, 310); // Body

// Head

circle(35 + i, 300, 10); // Head (circle)

// Right hand holding the umbrella

line(35 + i, 310, 50 + i, 330); // Right hand

// Umbrella stick

line(50 + i, 330, 50 + i, 280); // Umbrella stick

// Umbrella body (umbrella shape)

line(15 + i, 280, 85 + i, 280); // Umbrella body (horizontal line)

// Umbrella arc (arc of the umbrella)

arc(50 + i, 280, 0, 180, 35); // Umbrella body (half-circle)

// Umbrella handle (arc)

arc(55 + i, 330, 180, 360, 5); // Umbrella handle (small arc)

}

// Main program

int main() {

int gd = DETECT, gm;

initgraph(&gd, &gm, "");

WalkingMan man;

int i = 0;

// Animate the walking man with umbrella in the rain

while (!kbhit()) {

for (i = 0; i < 100; i++) {

cleardevice(); // Clear the screen

// Draw the rain

man.drawRain(i);

// Draw the walking man

man.drawMan(i);

delay(50); // Delay for a while to simulate animation

}

}

getch();

closegraph(); // Close graphics window

return 0;

}