**-----------------COMPUTER GRAPHICS PRACTICAL FILE------------**

**NAME : TAPAN**

**ROLL NO. : 10876**

**COURSE: BSC COMPUTER SCIENCE HONS.**

**SUBJECT : COMPUTER GRAPHICS**

**SECTION: B**

**-----------------------------------------------------------------------------------------------**

**Practical 1: Write a program to implement Bresenham’s line drawing algorithm.**

**CODE: #include <iostream>**

**#include <iomanip>**

**#include <math.h>**

**#include <graphics.h>**

**using namespace std;**

**void BresenhamLine(int x1, int y1, int x2, int y2)**

**{**

**// Setup**

**int win = initwindow(700, 500, "Bresenham Line Drawing Algorithm Example");**

**setcurrentwindow(win);**

**// Get middle of window as adjusted origin**

**int x\_origin = getwindowwidth() / 2;**

**int y\_origin = getwindowheight() / 2;**

**// Calculate dy, dx, a, b**

**double dx = x2 - x1;**

**double dy = y2 - y1;**

**double a = 2\*dy;**

**double b = -2\*dx;**

**// Initial value of d**

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

**// Draw initial pixel**

**putpixel(x1 + x\_origin, -y1 + y\_origin, 15);**

**// Output to terminal**

**cout << "\ni\tPixel\td\tPlotted Values" << endl;**

**cout << "-------------------------------------" << endl;**

**cout << "0\t \t \t" << "(" << round(x1) << "," << round(y1) << ")" << endl;**

**double x = x1;**

**double y = y1;**

**string pixel = "";**

**for(int i = 1; ; i++) {**

**double d\_temp = d;**

**// Choose NE pixel**

**if (d>0){**

**d = d + a + b;**

**x = x + 1;**

**y = y + 1;**

**pixel = "NE";**

**}**

**// Choose E pixel**

**else {**

**d = d + a;**

**x = x + 1;**

**pixel = "E";**

**}**

**// Exit condition**

**if (x > x2 || y > y2) break;**

**// draw pixel**

**putpixel(x + x\_origin, -y + y\_origin, 15);**

**// Output to terminal**

**cout << i << "\t" << pixel << "\t" << d\_temp << "\t" << "(" << round(x) << "," << round(y) << ")" << endl;**

**}**

**// Clean up**

**getch();**

**closegraph();**

**cout << endl;**

**}**

**int main()**

**{**

**int x1 = -200;**

**int y1 = -100;**

**int x2 = 200;**

**int y2 = 180;**

**BresenhamLine(x1, y1, x2, y2);**

**return 0;**

**}**

****

**PRACTICAL 2:** Write a program to implement mid-point circle drawing algorithm .

#include <cmath>

#include <cstdlib>

#include <graphics.h>

#include <iostream>

using namespace std;

void drawCirclePixels(int c\_x, int c\_y, int x, int y, int val)

{

putpixel(c\_x + x, c\_y + y, val);

putpixel(c\_x + y, c\_y + x, val);

putpixel(c\_x + y, c\_y + -x, val);

putpixel(c\_x + x, c\_y + -y, val);

putpixel(c\_x + -x, c\_y + -y, val);

putpixel(c\_x + -y, c\_y + -x, val);

putpixel(c\_x + -y, c\_y + x, val);

putpixel(c\_x + -x, c\_y + y, val);

return;

}

void BresenhamCircle(int x1, int y1, int r)

{

// Setup

int win = initwindow(400, 300, "Bresenham Circle Drawing Algorithm Example");

setcurrentwindow(win);

// Get middle of window + given value as centre

int x\_c = round(x1 + getwindowwidth()/2);

int y\_c = round(-y1 + getwindowheight()/2);

// Initial value of d

int d = round(5/4 - r);

// Draw initial pixel

drawCirclePixels(x\_c, y\_c, 0, -r, 15);

// Output to terminal

cout << "\nIst OCTANT\n-------------" << endl;

cout << "\ni\tPixel\td\tPlotted Values" << endl;

cout << "-------------------------------------" << endl;

cout << "0\t \t \t" << "(" << x1 << "," << y1+r << ")" << endl;

int i = 0;

string pixel = "";

int x = 0;

int y = r;

while (y >= x)

{

i = i + 1;

int d\_temp = d;

// Choose E pixel

if (d < 0)

{

d += 2 \* x + 3;

x += 1;

pixel = 'E';

}

// Choose SE pixel

else

{

d += 2 \* (x - y) + 5;

x += 1;

y -= 1;

pixel = "SE";

}

drawCirclePixels(x\_c, y\_c, x, -y, 15);

// Output to terminal

cout << i << "\t" << pixel << "\t" << d\_temp << "\t" << "(" << x << "," << y << ")" << endl;

}

cout << endl;

// Clean up

getch();

closegraph();

}

int main(void)

{

int x, y, r;

cout << "Enter Centre (x y): ";

cin >> x >> y;

cout << "Enter Radius (r): ";

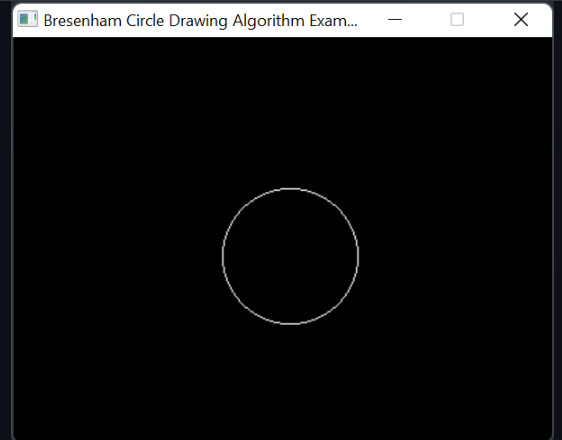
cin >> r;

BresenhamCircle(x, y, r);

return 0;

}

**OUTPUT:**

****

**PRACTICAL 3:** Write a program to clip a line using Cohen and Sutherland line clipping algorithm.

// for line clipping.

// including libraries

#include <bits/stdc++.h>

#include <graphics.h>

using namespace std;

// Global Variables

int xmin, xmax, ymin, ymax;

// Lines where co-ordinates are (x1, y1) and (x2, y2)

struct lines {

int x1, y1, x2, y2;

};

// This will return the sign required.

int sign(int x)

{

if (x > 0)

return 1;

else

return 0;

}

// CohenSutherLand LineClipping Algorithm As Described in theory.

// This will clip the lines as per window boundaries.

void clip(struct lines mylines)

{

// arrays will store bits

// Here bits implies initial Point whereas bite implies end points

int bits[4], bite[4], i, var;

// setting color of graphics to be RED

setcolor(RED);

// Finding Bits

bits[0] = sign(xmin - mylines.x1);

bite[0] = sign(xmin - mylines.x2);

bits[1] = sign(mylines.x1 - xmax);

bite[1] = sign(mylines.x2 - xmax);

bits[2] = sign(ymin - mylines.y1);

bite[2] = sign(ymin - mylines.y2);

bits[3] = sign(mylines.y1 - ymax);

bite[3] = sign(mylines.y2 - ymax);

// initial will used for initial coordinates and end for final

string initial = "", end = "", temp = "";

// convert bits to string

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

if (bits[i] == 0)

initial += '0';

else

initial += '1';

}

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

if (bite[i] == 0)

end += '0';

else

end += '1';

}

// finding slope of line y=mx+c as (y-y1)=m(x-x1)+c

// where m is slope m=dy/dx;

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

float c = mylines.y1 - m \* mylines.x1;

// if both points are inside the Accept the line and draw

if (initial == end && end == "0000") {

// inbuild function to draw the line from(x1, y1) to (x2, y2)

line(mylines.x1, mylines.y1, mylines.x2, mylines.y2);

return;

}

// this will contain cases where line maybe totally outside for partially inside

else {

// taking bitwise end of every value

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

int val = (bits[i] & bite[i]);

if (val == 0)

temp += '0';

else

temp += '1';

}

// as per algo if AND is not 0000 means line is completely outside hence draw nothing and return

if (temp != "0000")

return;

// Here contain cases of partial inside or outside

// So check for every boundary one by one

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

// if boths bit are same hence we cannot find any intersection with boundary so continue

if (bits[i] == bite[i])

continue;

// Otherwise there exist a intersection

// Case when initial point is in left xmin

if (i == 0 && bits[i] == 1) {

var = round(m \* xmin + c);

mylines.y1 = var;

mylines.x1 = xmin;

}

// Case when final point is in left xmin

if (i == 0 && bite[i] == 1) {

var = round(m \* xmin + c);

mylines.y2 = var;

mylines.x2 = xmin;

}

// Case when initial point is in right of xmax

if (i == 1 && bits[i] == 1) {

var = round(m \* xmax + c);

mylines.y1 = var;

mylines.x1 = xmax;

}

// Case when final point is in right of xmax

if (i == 1 && bite[i] == 1) {

var = round(m \* xmax + c);

mylines.y2 = var;

mylines.x2 = xmax;

}

// Case when initial point is in top of ymin

if (i == 2 && bits[i] == 1) {

var = round((float)(ymin - c) / m);

mylines.y1 = ymin;

mylines.x1 = var;

}

// Case when final point is in top of ymin

if (i == 2 && bite[i] == 1) {

var = round((float)(ymin - c) / m);

mylines.y2 = ymin;

mylines.x2 = var;

}

// Case when initial point is in bottom of ymax

if (i == 3 && bits[i] == 1) {

var = round((float)(ymax - c) / m);

mylines.y1 = ymax;

mylines.x1 = var;

}

// Case when final point is in bottom of ymax

if (i == 3 && bite[i] == 1) {

var = round((float)(ymax - c) / m);

mylines.y2 = ymax;

mylines.x2 = var;

}

// Updating Bits at every point

bits[0] = sign(xmin - mylines.x1);

bite[0] = sign(xmin - mylines.x2);

bits[1] = sign(mylines.x1 - xmax);

bite[1] = sign(mylines.x2 - xmax);

bits[2] = sign(ymin - mylines.y1);

bite[2] = sign(ymin - mylines.y2);

bits[3] = sign(mylines.y1 - ymax);

bite[3] = sign(mylines.y2 - ymax);

} // end of for loop

// Initialize initial and end to NULL

initial = "", end = "";

// Updating strings again by bit

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

if (bits[i] == 0)

initial += '0';

else

initial += '1';

}

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

if (bite[i] == 0)

end += '0';

else

end += '1';

}

// If now both points lie inside or on boundary then simply draw the updated line

if (initial == end && end == "0000") {

line(mylines.x1, mylines.y1, mylines.x2, mylines.y2);

return;

}

// else line was completely outside hence rejected

else

return;

}

}

// Driver Function

int main()

{

int gd = DETECT, gm;

// Setting values of Clipping window

xmin = 80;

xmax = 200;

ymin = 80;

ymax = 160;

// Setup

int win = initwindow(400, 300, "Line Clipping Example");

setcurrentwindow(win);

// Drawing Window using Lines

line(xmin, ymin, xmax, ymin);

line(xmax, ymin, xmax, ymax);

line(xmax, ymax, xmin, ymax);

line(xmin, ymax, xmin, ymin);

// Assume 4 lines to be clipped

struct lines mylines[4];

// Setting the coordinated of 4 lines

mylines[0].x1 = 60;

mylines[0].y1 = 130;

mylines[0].x2 = 110;

mylines[0].y2 = 60;

mylines[1].x1 = 120;

mylines[1].y1 = 40;

mylines[1].x2 = 200;

mylines[1].y2 = 180;

mylines[2].x1 = 120;

mylines[2].y1 = 200;

mylines[2].x2 = 160;

mylines[2].y2 = 140;

mylines[3].x1 = 170;

mylines[3].y1 = 100;

mylines[3].x2 = 240;

mylines[3].y2 = 150;

// Drawing Initial Lines without clipping

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

line(mylines[i].x1, mylines[i].y1,

mylines[i].x2, mylines[i].y2);

delay(1000);

}

// Drawing clipped Line

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

// Calling clip() which in term clip the line as per window and draw it

clip(mylines[i]);

delay(1000);

}

delay(4000);

getch();

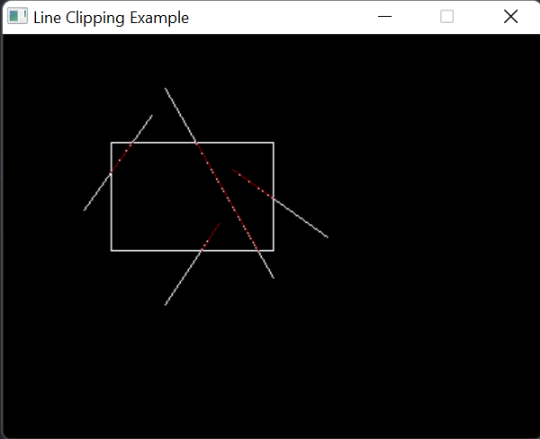
// For Closing the graph.

closegraph();

return 0;

}

**OUTPUT:**



**PRACTICAL 4:** Write a program to clip a polygon using Sutherland Hodgeman algorithm.

#include<iostream>

#include <graphics.h>

using namespace std;

const int MAX\_POINTS = 20;

// Returns x-value of point of intersection of two

// lines

int x\_intersect(int x1, int y1, int x2, int y2,

int x3, int y3, int x4, int y4)

{

int num = (x1\*y2 - y1\*x2) \* (x3-x4) -

(x1-x2) \* (x3\*y4 - y3\*x4);

int den = (x1-x2) \* (y3-y4) - (y1-y2) \* (x3-x4);

return num/den;

}

// Returns y-value of point of intersection of

// two lines

int y\_intersect(int x1, int y1, int x2, int y2,

int x3, int y3, int x4, int y4)

{

int num = (x1\*y2 - y1\*x2) \* (y3-y4) -

(y1-y2) \* (x3\*y4 - y3\*x4);

int den = (x1-x2) \* (y3-y4) - (y1-y2) \* (x3-x4);

return num/den;

}

// This functions clips all the edges w.r.t one clip

// edge of clipping area

void clip(int poly\_points[][2], int &poly\_size,

int x1, int y1, int x2, int y2)

{

int new\_points[MAX\_POINTS][2], new\_poly\_size = 0;

// (ix,iy),(kx,ky) are the co-ordinate values of

// the points

for (int i = 0; i < poly\_size; i++)

{

// i and k form a line in polygon

int k = (i+1) % poly\_size;

int ix = poly\_points[i][0], iy = poly\_points[i][1];

int kx = poly\_points[k][0], ky = poly\_points[k][1];

// Calculating position of first point

// w.r.t. clipper line

int i\_pos = (x2-x1) \* (iy-y1) - (y2-y1) \* (ix-x1);

// Calculating position of second point

// w.r.t. clipper line

int k\_pos = (x2-x1) \* (ky-y1) - (y2-y1) \* (kx-x1);

// Case 1 : When both points are inside

if (i\_pos < 0 && k\_pos < 0)

{

//Only second point is added

new\_points[new\_poly\_size][0] = kx;

new\_points[new\_poly\_size][1] = ky;

new\_poly\_size++;

}

// Case 2: When only first point is outside

else if (i\_pos >= 0 && k\_pos < 0)

{

// Point of intersection with edge

// and the second point is added

new\_points[new\_poly\_size][0] = x\_intersect(x1,

y1, x2, y2, ix, iy, kx, ky);

new\_points[new\_poly\_size][1] = y\_intersect(x1,

y1, x2, y2, ix, iy, kx, ky);

new\_poly\_size++;

new\_points[new\_poly\_size][0] = kx;

new\_points[new\_poly\_size][1] = ky;

new\_poly\_size++;

}

// Case 3: When only second point is outside

else if (i\_pos < 0 && k\_pos >= 0)

{

//Only point of intersection with edge is added

new\_points[new\_poly\_size][0] = x\_intersect(x1,

y1, x2, y2, ix, iy, kx, ky);

new\_points[new\_poly\_size][1] = y\_intersect(x1,

y1, x2, y2, ix, iy, kx, ky);

new\_poly\_size++;

}

// Case 4: When both points are outside

else

{

//No points are added

}

}

// Copying new points into original array

// and changing the no. of vertices

poly\_size = new\_poly\_size;

for (int i = 0; i < poly\_size; i++)

{

poly\_points[i][0] = new\_points[i][0];

poly\_points[i][1] = new\_points[i][1];

}

}

// Implements Sutherland–Hodgman algorithm

void suthHodgClip(int poly\_points[][2], int poly\_size,

int clipper\_points[][2], int clipper\_size)

{

//i and k are two consecutive indexes

for (int i=0; i<clipper\_size; i++)

{

int k = (i+1) % clipper\_size;

// We pass the current array of vertices, it's size

// and the end points of the selected clipper line

clip(poly\_points, poly\_size, clipper\_points[i][0],

clipper\_points[i][1], clipper\_points[k][0],

clipper\_points[k][1]);

}

// Printing vertices of clipped polygon

cout << "\nClipped Polygon : " << endl;

for (int i=0; i < poly\_size; i++)

cout << '(' << poly\_points[i][0] <<

", " << poly\_points[i][1] << ") ";

cout << endl << endl;

// Drawing Clipped Polygon

int poly\_clipped[50];

for (int q = 0; q < poly\_size; q++)

{

for (int t = 0; t < 2; t++)

{

poly\_clipped[q \* 2 + t] = poly\_points[q][t];

}

}

setcolor(BLUE);

poly\_clipped[2 \* poly\_size] = poly\_clipped[0];

poly\_clipped[2 \* poly\_size + 1] = poly\_clipped[1];

drawpoly(poly\_size + 1, poly\_clipped);

getch();

}

//Driver code

int main()

{

int gd = DETECT, gm, errorcode;

initgraph(&gd, &gm, NULL);

// Defining polygon vertices in clockwise order

int poly\_size = 3;

int poly\_points[20][2] = {{100,150}, {200,250},

{300,100}};

// Defining clipper polygon vertices in clockwise order

// 1st Example with square clipper

int clipper\_size = 4;

int clipper\_points[][2] = {{100,100}, {100,200},

{200,200}, {200,100} };

setcolor(RED);

rectangle(100, 100, 200, 200);

setcolor(YELLOW);

int poly[50];

for (int q = 0; q < poly\_size; q++)

{

for (int t = 0; t < 2; t++)

{

poly[q \* 2 + t] = poly\_points[q][t];

}

}

poly[2 \* poly\_size] = poly[0];

poly[2 \* poly\_size + 1] = poly[1];

drawpoly(poly\_size + 1, poly);

//Calling the clipping function

suthHodgClip(poly\_points, poly\_size, clipper\_points,

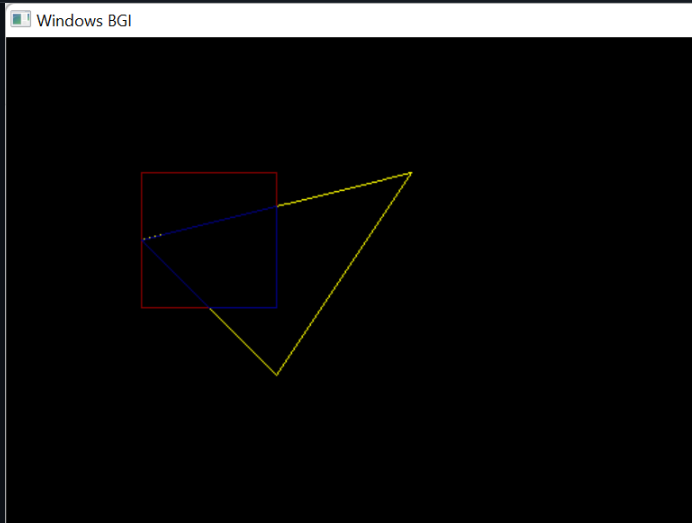
clipper\_size);

getch();

return 0;

}

**OUTPUT:**



**PRACTICAL 5: Write a program to fill a polygon using Scan line fill algorithm.**

#include <iostream>

#include <vector>

#include <utility> // for pair

#include <algorithm>

#include <math.h>

#include "dda.cpp"

#include <conio.h>

#include <math.h>

#include <cmath>

#include <graphics.h>

using namespace std;

// Structure to represent a node in the adjacency list

struct Node {

int vertex; // Index of the vertex

pair<int, int> point; // Coordinates of the vertex

Node(int v, pair<int, int> p) : vertex(v), point(p) {}

};

// Function to add an edge between vertices u and v in the adjacency list

void addEdge(vector<vector<Node>>& adjList, int u, int v, pair<int, int> point\_u, pair<int, int> point\_v) {

adjList[u].push\_back(Node(v, point\_v));

adjList[v].push\_back(Node(u, point\_u));

// For undirected graph

}

class EdgeNode

{

public:

int vertex1; // Index of the vertex

int vertex2;

double x;

double y;

double m\_inv;

EdgeNode \*ptr;

EdgeNode(){

this->x = 0.0;

this->y = 0.0;

this->m\_inv = 0.0;

ptr = NULL;

}

static bool compareX(const EdgeNode\* a, const EdgeNode\* b) {

return a->x < b->x;

}

};

class vertex\_ptr{

public:

EdgeNode \*ptr;

vertex\_ptr(){

ptr = NULL;

}

};

class array\_linked\_list

{

public:

vertex\_ptr \*arr;

int s;

array\_linked\_list(int n){

arr = new vertex\_ptr[n]; //

s = n;

}

};

void print\_global\_edge\_table(array\_linked\_list \*ged,int n){

cout<<"\n\nPRINTING GLOBAL EDGE TABLE :: "<<endl;

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

cout<<"for y is "<<i<<" "<<endl;

cout<<"----------------------------------------------------------------------"<<endl;

EdgeNode \*temp = ged->arr[i].ptr ;

while(temp != NULL){ // reached to last node

cout<<"for edge : "<<temp->vertex1<<" , "<<temp->vertex2<<" , x min : "<<temp->x<<" , y max : "<<temp->y<<" , 1/m is : "<<temp->m\_inv;

cout<<endl;

temp = temp->ptr;

}

cout<<"---------------------------------------------------------------------\n\n"<<endl;

}

}

void print\_edge(vector<pair<int, int>>& edges){

//cout<<"---------------edge list----------------------------"<<endl;

for(int i=0;i<edges.size();i++){

// cout<<edges[i].first<<" , "<<edges[i].second<<endl;

}

//cout<<"-----------------------------------------------------"<<endl;

}

void scan\_line(vector<vector<Node>>& adjList,int n,int y, array\_linked\_list \*ged, vector<pair<int, int>>& edges){

// find all vertex which has y =4 , and than find adjacent edges

// and store some flags that will be used to know that those edges, has already been used

vector <int> vertices;

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

const Node& node = adjList[i][0];

if(node.point.second == y){

vertices.push\_back(int(i));

}

}

// now creating GLOBAL EDGE TABLE for each vertex

// to keep the edges which are already covered;

int flag = 0;

// ged->arr[1] is a pointer

// y\_max will be used later

int i = y;

// filling according to increasing y , and increasing x

for(int j=0;j<vertices.size();j++){ // this access all vertex with y = 4, or i=4 , or some other

// when j = 0 than we are at 1st vertex whose y =4

// when j = 1 than we are at 2nd vertex whose y =4

// for each vertex iterate its adjacent vertex

vector<int> temp;

for(int k= 0; k < adjList[vertices[j]].size(); k++){ // this in 1 loop gives all adjacent to 1 vertex

const Node& node = adjList[vertices[j]][k];

// here we have all ajacent edges releated to some vertex j

// iteration is giving all vertex , wrt j

temp.push\_back(node.vertex); // contains the adjacent vertexs

}

// now for each edge we will make a node

// temp has all the vertex adjacent

// vertices[j] gives vertex for which we will find edge

int x1 = adjList[vertices[j]][0].point.first;

int y1 = adjList[vertices[j]][0].point.second;

// v1 is vertices[j]

// v2 inside ,, if ( v1,v2) is found in any pair in edge vectore than we will skip it

for(int p=0;p<temp.size();p++){

// this is v2

int x2 = adjList[temp[p]][0].point.first;

int y2 = adjList[temp[p]][0].point.second;

// check for complete list of edges for already done

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

if(edges[c].first == temp[p] && edges[c].second == vertices[j] ){

flag = -1;

break;

}

if(edges[c].first == vertices[j] && edges[c].second == temp[p] ){

flag = -1;

break;

}

}

if(flag == -1){

flag =0;

// calculate for next edge

continue;

}

// cout<<"working for :: (x1,x2) : ( "<<x1<<" , "<<y1<<" )"<<" and "<<" , ( "<<x2<<" , "<<y2<<" ) "<<endl;

int minX = (y1 < y2) ? x1 : y2; // for x which ever has minmum y

int maxY = (y1 > y2) ? y1 : y2;

if( ged->arr[i].ptr == NULL){

ged->arr[i].ptr = new EdgeNode();

ged->arr[i].ptr->x = minX;

ged->arr[i].ptr->y = maxY;

ged->arr[i].ptr->m\_inv = (x2\*1.0 - x1\*1.0)\*1.0 / (y2\*1.0 - y1\*1.0 )\*1.0 ;

ged->arr[i].ptr->vertex1 = vertices[j];

ged->arr[i].ptr->vertex2 = temp[p];

edges.push\_back(make\_pair(vertices[j], temp[p]));

}else{

EdgeNode \*tempp = ged->arr[i].ptr ;

while(tempp->ptr != NULL){ // reached to last node

tempp = tempp->ptr;

}

tempp->ptr = new EdgeNode();

tempp->ptr->x = minX;

tempp->ptr->y = maxY;

tempp->ptr->vertex1 = vertices[j];

tempp->ptr->vertex2 = temp[p];

tempp->ptr->m\_inv = (x2\*1.0 - x1\*1.0)\*1.0 / (y2\*1.0 - y1\*1.0 )\*1.0 ;

edges.push\_back(make\_pair( vertices[j], temp[p]));

}

}

}

}

void move\_edges(int y\_min, vector<vector<Node>>& adjList, vector<EdgeNode \*>& active\_edges , array\_linked\_list \*ged ){

//cout<<"============================================================\n";

EdgeNode \*temp = ged->arr[y\_min].ptr ;

while(temp != NULL){ // reached to last node

// cout<<temp->vertex1<<" , "<<temp->vertex2<<endl;

active\_edges.push\_back(temp);

temp = temp->ptr;

}

//cout<<"==============================================================\n";

}

void remove\_from\_active\_edge(vector<EdgeNode \*>& active\_edges,int y){

for(int i=0;i< active\_edges.size();i++){

// cout<<"v1 : "<<active\_edges[i]->vertex1<<" , "<<active\_edges[i]->vertex2<<endl;

if(active\_edges[i]->y == y){

// remove it

// cout<<"remove it : "<<endl;

int indexToRemove = i;

if (indexToRemove >= 0 && indexToRemove < active\_edges.size()) {

active\_edges.erase(active\_edges.begin() + indexToRemove);

} else {

// std::cout << "Index out of range" << std::endl;

}

}

}

}

void print\_active\_edge\_table(vector<EdgeNode \*>& active\_edges ){

cout<<"------------------------------------------ACTIVE EDGE TABLE--------------------------------------------\n"<<endl;

for(int i=0;i<active\_edges.size();i++){

cout<<"v1 : "<<active\_edges[i]->vertex1<<" , v2 is : "<<active\_edges[i]->vertex2<<endl;

}

cout<<"------------------------------------------------------------------------------------------------------\n"<<endl;

}

void make\_pair\_and\_print(vector<EdgeNode \*>& active\_edges,int y ){

int x1 = 0;

int y1 = y;

int x2 = 0;

int y2 = y;

for(int i =0 ;i<active\_edges.size(); i +=2){

x1 = active\_edges[i]->x;

x2 = active\_edges[i+1]->x;

cout<<"pair : "<<x1<<" , "<<y1<<" and "<<x2<<" , "<<y2<<endl;

dda(x1,y1,x2,y2,GREEN);

}

}

void increment\_x(vector<EdgeNode \*>& active\_edges){

for(int i =0 ;i<active\_edges.size(); i++){

if(!isinf(active\_edges[i]->m\_inv)){

active\_edges[i]->x = active\_edges[i]->x + active\_edges[i]->m\_inv ;

}

}

}

void scan\_algo(array\_linked\_list \*ged,int y\_min, int y\_max, vector<vector<Node>>& adjList){

int y = y\_min;

vector<EdgeNode \*> active\_edges ; // active edge table is empty

int get = 5;

int aet = 0;

while(y <= y\_max){ // repeat until aet and get is empty

//move from et to aet those y\_min = y

move\_edges(y,adjList, active\_edges,ged);

// remove those has y = y\_max

remove\_from\_active\_edge(active\_edges,y);

// sort aet on x basis

std::sort(active\_edges.begin(), active\_edges.end(), EdgeNode::compareX);

// cout<<"sorted list :============================\n";

// cout<<"=================check if there is any changes in GLOBAL EDGE TABLE==============\n";

// print\_global\_edge\_table(ged,y\_max);

// make pairs from aet using y

// print\_active\_edge\_table(active\_edges);

make\_pair\_and\_print(active\_edges,y);

y +=1;

// increment with slope

if(y< y\_max){

increment\_x(active\_edges);

}else{

break;

}

}

}

int main() {

int gd = DETECT, gm;

char pathtodriver[] = "";

initgraph(&gd, &gm, pathtodriver);

int numVertices = 6;

// Initialize adjacency list

vector<vector<Node>> adjList(numVertices);

// Store points associated with each vertex

vector<pair<int, int>> points;

int y\_min = 100;

int y\_max = 400;

points.push\_back(make\_pair(200,100));

points.push\_back(make\_pair(50,300));

points.push\_back(make\_pair(80,400));

points.push\_back(make\_pair(200,350));

points.push\_back(make\_pair(300,400));

points.push\_back(make\_pair(350,310));

// Add some edges

addEdge(adjList, 1, 0, points[0], points[1]);

addEdge(adjList, 2, 1, points[1], points[2]);

addEdge(adjList, 3, 2, points[2], points[3]);

addEdge(adjList, 4, 3, points[3], points[4]);

addEdge(adjList, 5, 4, points[4], points[5]);

addEdge(adjList, 0, 5, points[5], points[0]);

vector<pair<int, int>> edges;

array\_linked\_list \*ged = new array\_linked\_list(y\_max+1);

for(int i= y\_min ;i <= y\_max ;i++){

scan\_line(adjList,numVertices,i,ged,edges);

}

//print\_global\_edge\_table(ged,y\_max);

scan\_algo(ged,y\_min,y\_max,adjList);

cout<<"done"<<endl;

getch();

closegraph();

return 0;

}

**PRACTICAL 6: Write a program to apply various 2D transformations on a 2D object (use homogenous coordinates).**

#define \_USE\_MATH\_DEFINES

#include <cmath>

#include <cstdlib>

#include <graphics.h>

#include <iostream>

#define COORD\_SHIFT 100

using namespace std;

void clrscr()

{

#ifdef \_WIN32

system("cls");

#elif \_\_unix\_\_

system("clear");

#endif

}

double \*\*inputFigure(int n)

{

cout << "Enter the matrix for the 2-D shape (homogeneous):\n";

double \*\*figure = NULL;

figure = new double \*[n];

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

{

figure[i] = new double[3];

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

{

cin >> figure[i][j];

}

}

return figure;

}

void drawFigure(double \*\*points, int n)

{

setcolor(WHITE);

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

{

line(COORD\_SHIFT + points[i][0],

COORD\_SHIFT + points[i][1],

COORD\_SHIFT + points[(i + 1) % n][0],

COORD\_SHIFT + points[(i + 1) % n][1]);

}

delay(5e3);

cleardevice();

}

double \*\*translate(double \*\*figure, int dim, int m, int n)

{

double \*\*\_figure = NULL;

int T[dim][3] = {{1, 0, 0}, {0, 1, 0}, {m, n, 1}};

\_figure = new double \*[dim];

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

{

\_figure[i] = new double[3];

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

{

for (int k = 0; k < dim; k++)

{

\_figure[i][j] += figure[i][k] \* T[k][j];

}

}

}

return \_figure;

}

double \*\*rotate(double \*\*figure, int dim, double theta)

{

double \*\*\_figure = NULL;

double T[dim][3] = {{cos(theta \* M\_PI / 180.0), sin(theta \* M\_PI / 180.0), 0},

{-sin(theta \* M\_PI / 180.0), cos(theta \* M\_PI / 180.0), 0},

{0, 0, 1}};

\_figure = new double \*[dim];

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

{

\_figure[i] = new double[3];

for (int j = 0; j < 2; j++)

{

for (int k = 0; k < dim; k++)

{

\_figure[i][j] += figure[i][k] \* T[k][j];

}

}

}

return \_figure;

}

double \*\*scale(double \*\*figure, int dim, int m, int n)

{

double \*\*\_figure = NULL;

int T[dim][3] = {{m, 0, 0}, {0, n, 0}, {0, 0, 1}};

\_figure = new double \*[dim];

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

{

\_figure[i] = new double[3];

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

{

for (int k = 0; k < dim; k++)

{

\_figure[i][j] += figure[i][k] \* T[k][j];

}

}

}

return \_figure;

}

double \*\*reflect(double \*\*figure, int dim, int c)

{

double \*\*\_figure = NULL;

int T[dim][3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};

switch (c)

{

case 1:

T[1][1] = -1;

break;

case 2:

T[0][0] = -1;

break;

case 3:

T[0][0] = 0;

T[0][1] = 1;

T[1][0] = 1;

T[1][1] = 0;

break;

case 4:

T[0][0] = -1;

T[1][1] = -1;

break;

default:

return NULL;

break;

}

\_figure = new double \*[dim];

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

{

\_figure[i] = new double[3];

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

{

for (int k = 0; k < dim; k++)

{

\_figure[i][j] += figure[i][k] \* T[k][j];

}

}

}

return \_figure;

}

double \*\*shear(double \*\*figure, int dim, int m, int n)

{

double \*\*\_figure = NULL;

int T[dim][3] = {{1, n, 0}, {m, 1, 0}, {0, 0, 1}};

\_figure = new double \*[dim];

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

{

\_figure[i] = new double[3];

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

{

for (int k = 0; k < dim; k++)

{

\_figure[i][j] += figure[i][k] \* T[k][j];

}

}

}

return \_figure;

}

void menu(double \*\*figure, int dim)

{

int ch = 0;

double \*\*\_figure;

do

{

clrscr();

cout << "\nMenu\n-------\n(1) Translation\n(2) Rotation";

cout << "\n(3) Scaling\n(4) Reflection\n(5) Shearing";

cout << "\n(6) View Figure\n(7) Exit\n\nEnter Choice: ";

cin >> ch;

cout << endl;

switch (ch)

{

case 1:

int m, n;

cout << "Enter translation in x-axis: ";

cin >> m;

cout << "Enter translation in y-axis: ";

cin >> n;

\_figure = translate(figure, dim, m, n);

cout << "Drawing Original Figure...\n";

drawFigure(figure, dim);

cout << "Drawing Transformed Figure...\n";

drawFigure(\_figure, dim);

break;

case 2:

double theta;

cout << "Enter rotation angle (degrees): ";

cin >> theta;

\_figure = rotate(figure, dim, theta);

cout << "Drawing Original Figure...\n";

drawFigure(figure, dim);

cout << "Drawing Transformed Figure...\n";

drawFigure(\_figure, dim);

break;

case 3:

cout << "Enter scaling in x-axis: ";

cin >> m;

cout << "Enter scaling in y-axis: ";

cin >> n;

\_figure = scale(figure, dim, m, n);

cout << "Drawing Original Figure...\n";

drawFigure(figure, dim);

cout << "Drawing Transformed Figure...\n";

drawFigure(\_figure, dim);

break;

case 4:

cout << "Reflect along\n(1) x-axis\n(2) y-axis\n(3) y = x\n(4) y = -x\n"

<< "\nEnter Choice: ";

cin >> m;

\_figure = reflect(figure, dim, m);

cout << "Drawing Original Figure...\n";

drawFigure(figure, dim);

cout << "Drawing Transformed Figure...\n";

drawFigure(\_figure, dim);

break;

case 5:

cout << "Enter shearing in x-axis: ";

cin >> m;

cout << "Enter shearing in y-axis: ";

cin >> n;

\_figure = shear(figure, dim, m, n);

cout << "Drawing Original Figure...\n";

drawFigure(figure, dim);

cout << "Drawing Transformed Figure...\n";

drawFigure(\_figure, dim);

break;

case 6:

cout << "Drawing Original Figure...\n";

drawFigure(figure, dim);

break;

case 7:

default:

break;

}

delete \_figure;

cout << endl

<< "Finished..."

<< endl;

if (ch != 7)

{

cout << "\nPress Enter to continue ...\n";

cin.ignore();

cin.get();

}

} while (ch != 7);

};

int main(void)

{

int n;

double \*\*fig;

int gd = DETECT, gm;

initgraph(&gd, &gm, NULL);

cout << "Enter number of points in the figure: ";

cin >> n;

fig = inputFigure(n);

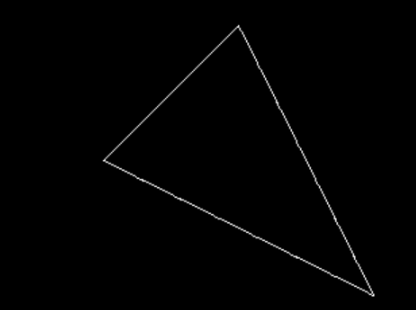
menu(fig, n);

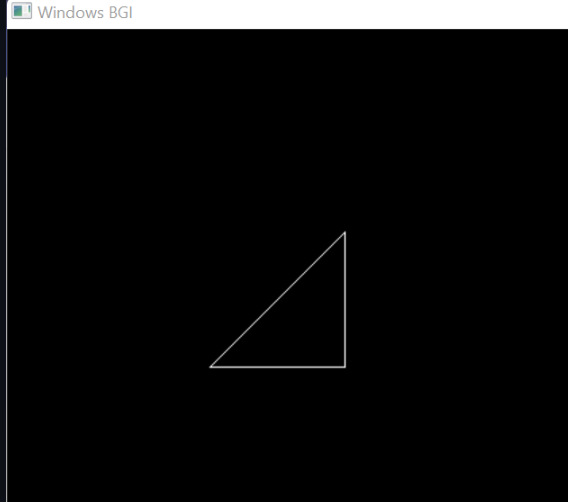
delete fig;

closegraph();

return 0;

**OUTPUT:**

****

****

**PRACTICAL 7: Write a program to apply various 3D transformations on a 3D object and then apply parallel and perspective projection on it.**

#include <iostream>

#include <direct.h>

#include <stdio.h>

#include <math.h>

#include <conio.h>

#include <graphics.h>

#include <process.h>

using namespace std;

int gd = DETECT, gm;

double x1, x2, y2;

void draw\_cube(double edge[20][3])

{

double y1;

initgraph(&gd, &gm, NULL);

int i;

clearviewport();

for (i = 0; i < 19; i++)

{

x1 = edge[i][0] + edge[i][2] \* (cos(2.3562));

y1 = edge[i][1] - edge[i][2] \* (sin(2.3562));

x2 = edge[i + 1][0] + edge[i + 1][2] \* (cos(2.3562));

y2 = edge[i + 1][1] - edge[i + 1][2] \* (sin(2.3562));

line(x1 + 320, 240 - y1, x2 + 320, 240 - y2);

}

line(320, 240, 320, 25);

line(320, 240, 550, 240);

line(320, 240, 150, 410);

getch();

closegraph();

}

void scale(double edge[20][3])

{

double a, b, c;

int i;

cout << "Enter The Scaling Factors: ";

cin >> a >> b >> c;

initgraph(&gd, &gm, NULL);

clearviewport();

for (i = 0; i < 20; i++)

{

edge[i][0] = edge[i][0] \* a;

edge[i][1] = edge[i][1] \* b;

edge[i][2] = edge[i][2] \* c;

}

draw\_cube(edge);

closegraph();

}

void translate(double edge[20][3])

{

int a, b, c;

int i;

cout << "Enter The Translation Factors: ";

cin >> a >> b >> c;

initgraph(&gd, &gm, NULL);

clearviewport();

for (i = 0; i < 20; i++)

{

edge[i][0] += a;

edge[i][0] += b;

edge[i][0] += c;

}

draw\_cube(edge);

closegraph();

}

void rotate(double edge[20][3])

{

int ch;

int i;

double temp, theta, temp1;

cout << "-=[ Rotation About ]=-" << endl;

cout << "1:==> X-Axis " << endl;

cout << "2:==> Y-Axis" << endl;

cout << "3:==> Z-Axis " << endl;

cout << "Enter Your Choice: ";

cin >> ch;

switch (ch)

{

case 1:

cout << "Enter The Angle: ";

cin >> theta;

theta = (theta \* 3.14) / 180;

for (i = 0; i < 20; i++)

{

edge[i][0] = edge[i][0];

temp = edge[i][1];

temp1 = edge[i][2];

edge[i][1] = temp \* cos(theta) - temp1 \* sin(theta);

edge[i][2] = temp \* sin(theta) + temp1 \* cos(theta);

}

draw\_cube(edge);

break;

case 2:

cout << "Enter The Angle: ";

cin >> theta;

theta = (theta \* 3.14) / 180;

for (i = 0; i < 20; i++)

{

edge[i][1] = edge[i][1];

temp = edge[i][0];

temp1 = edge[i][2];

edge[i][0] = temp \* cos(theta) + temp1 \* sin(theta);

edge[i][2] = -temp \* sin(theta) + temp1 \* cos(theta);

}

draw\_cube(edge);

break;

case 3:

cout << "Enter The Angle: ";

cin >> theta;

theta = (theta \* 3.14) / 180;

for (i = 0; i < 20; i++)

{

edge[i][2] = edge[i][2];

temp = edge[i][0];

temp1 = edge[i][1];

edge[i][0] = temp \* cos(theta) - temp1 \* sin(theta);

edge[i][1] = temp \* sin(theta) + temp1 \* cos(theta);

}

draw\_cube(edge);

break;

}

}

void reflect(double edge[20][3])

{

int ch;

int i;

cout << "-=[ Reflection About ]=-" << endl;

cout << "1:==> X-Axis" << endl;

cout << "2:==> Y-Axis " << endl;

cout << "3:==> Z-Axis " << endl;

cout << "Enter Your Choice: ";

cin >> ch;

switch (ch)

{

case 1:

for (i = 0; i < 20; i++)

{

edge[i][0] = edge[i][0];

edge[i][1] = -edge[i][1];

edge[i][2] = -edge[i][2];

}

draw\_cube(edge);

break;

case 2:

for (i = 0; i < 20; i++)

{

edge[i][1] = edge[i][1];

edge[i][0] = -edge[i][0];

edge[i][2] = -edge[i][2];

}

draw\_cube(edge);

break;

case 3:

for (i = 0; i < 20; i++)

{

edge[i][2] = edge[i][2];

edge[i][0] = -edge[i][0];

edge[i][1] = -edge[i][1];

}

draw\_cube(edge);

break;

}

}

void perspect(double edge[20][3])

{

int ch;

int i;

double p, q, r;

cout << "-=[ Perspective Projection About ]=-" << endl;

cout << "1:==> X-Axis " << endl;

cout << "2:==> Y-Axis " << endl;

cout << "3:==> Z-Axis" << endl;

cout << "Enter Your Choice := ";

cin >> ch;

switch (ch)

{

case 1:

cout << " Enter P := ";

cin >> p;

for (i = 0; i < 20; i++)

{

edge[i][0] = edge[i][0] / (p \* edge[i][0] + 1);

edge[i][1] = edge[i][1] / (p \* edge[i][0] + 1);

edge[i][2] = edge[i][2] / (p \* edge[i][0] + 1);

}

draw\_cube(edge);

break;

case 2:

cout << " Enter Q := ";

cin >> q;

for (i = 0; i < 20; i++)

{

edge[i][1] = edge[i][1] / (edge[i][1] \* q + 1);

edge[i][0] = edge[i][0] / (edge[i][1] \* q + 1);

edge[i][2] = edge[i][2] / (edge[i][1] \* q + 1);

}

draw\_cube(edge);

break;

case 3:

cout << " Enter R := ";

cin >> r;

for (i = 0; i < 20; i++)

{

edge[i][2] = edge[i][2] / (edge[i][2] \* r + 1);

edge[i][0] = edge[i][0] / (edge[i][2] \* r + 1);

edge[i][1] = edge[i][1] / (edge[i][2] \* r + 1);

}

draw\_cube(edge);

break;

}

closegraph();

}

int main()

{

int choice;

double edge[20][3] = {

100, 0, 0,

100, 100, 0,

0, 100, 0,

0, 100, 100,

0, 0, 100,

0, 0, 0,

100, 0, 0,

100, 0, 100,

100, 75, 100,

75, 100, 100,

100, 100, 75,

100, 100, 0,

100, 100, 75,

100, 75, 100,

75, 100, 100,

0, 100, 100,

0, 100, 0,

0, 0, 0,

0, 0, 100,

100, 0, 100};

while (1)

{

cout << "1:==> Draw Cube " << endl;

cout << "2:==> Scaling " << endl;

cout << "3:==> Rotation " << endl;

cout << "4:==> Reflection " << endl;

cout << "5:==> Translation " << endl;

cout << "6:==> Perspective Projection " << endl;

cout << "7:==> Exit " << endl;

cout << "Enter Your Choice := ";

cin >> choice;

switch (choice)

{

case 1:

draw\_cube(edge);

break;

case 2:

scale(edge);

break;

case 3:

rotate(edge);

break;

case 4:

reflect(edge);

break;

case 5:

translate(edge);

break;

case 6:

perspect(edge);

break;

case 7:

exit(0);

default:

cout << "\nPress A Valid Key...!!! ";

getch();

break;

}

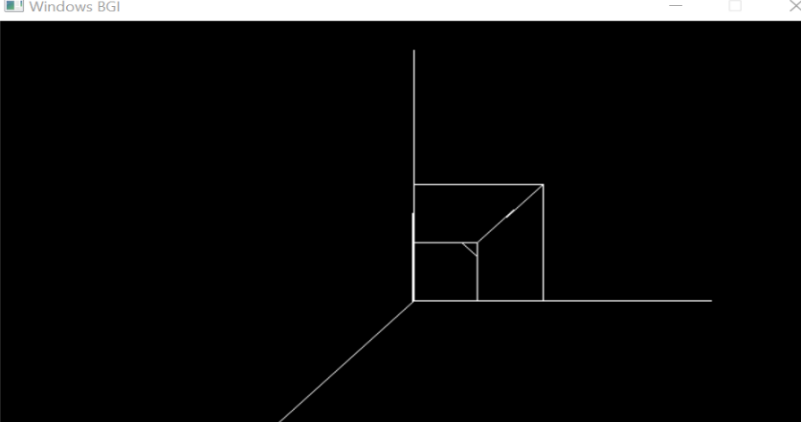
closegraph();

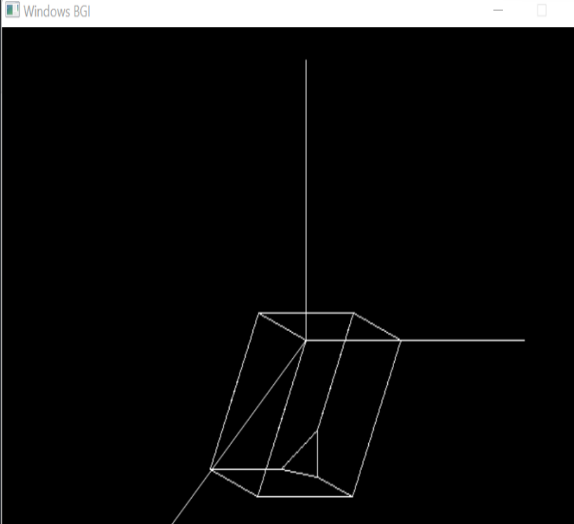
}

return 0;

}

**OUTPUT:**

****

****

**PRACTICAL 8(A): Write a program to draw Bezier curve.**

#include<graphics.h>

#include<math.h>

#include<conio.h>

#include<stdio.h>

int main()

{

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

double put\_x,put\_y,t;

int gr=DETECT,gm;

initgraph(&gr,&gm,NULL);

printf("\n\*\*\*\*\*\* Bezier Curve \*\*\*\*\*\*\*\*\*\*\*");

printf("\n Please enter x and y coordinates ");

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

{

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

putpixel(x[i],y[i],3); // Control Points

}

for(t=0.0;t<=1.0;t=t+0.001) // t always lies between 0 and 1

{

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]; // Formula to draw curve

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); // putting pixel

}

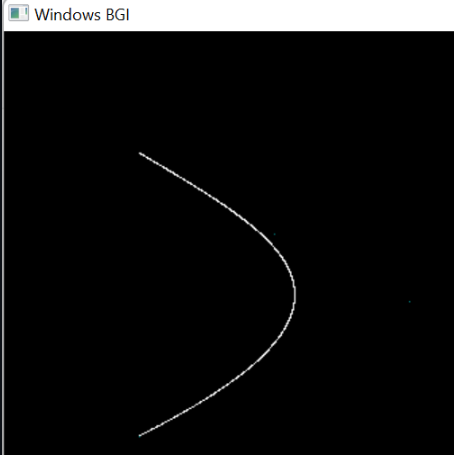
getch();

closegraph();

return 0;

}

**OUTPUT:**

****

**PRACTICAL 8(B): Write a program to draw Hermite curve.**

#include <iostream>

#include <graphics.h>

#include <conio.h>

#include <stdio.h>

#include <stdlib.h>

using namespace std;

struct point

{

int x, y;

};

void hermite(point p1, point p4, double r1, double r4)

{

float x, y, t;

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

{

x = (2 \* t \* t \* t - 3 \* t \* t + 1) \* p1.x + (-2 \* t \* t \* t + 3 \* t \* t) \* p4.x + (t \* t \* t - 2 \* t \* t + t) \* r1 + (t \* t \* t - t \* t) \* r4;

y = (2 \* t \* t \* t - 3 \* t \* t + 1) \* p1.y + (-2 \* t \* t \* t + 3 \* t \* t) \* p4.y + (t \* t \* t - 2 \* t \* t + 1) \* r1 + (t \* t \* t - t \* t) \* r4;

putpixel(x, y, YELLOW);

}

}

int main()

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, NULL);

/\* read result of initialization \*/

errorcode = graphresult();

/\* an error occurred \*/

if (errorcode != grOk)

{

printf("Graphics error: %s\n", grapherrormsg(errorcode));

printf("Press any key to halt:");

getch();

exit(1);

}

double r1, r4;

point p1, p2;

cout << "Enter 2 hermite points: " << endl;

cin >> p1.x >> p1.y >> p2.x >> p2.y;

cout << "Enter tangents at p1 and p4: " << endl;

cin >> r1 >> r4;

hermite(p1, p2, r1, r4);

putpixel(p1.x, p1.y, WHITE);

putpixel(p2.x, p2.y, WHITE);

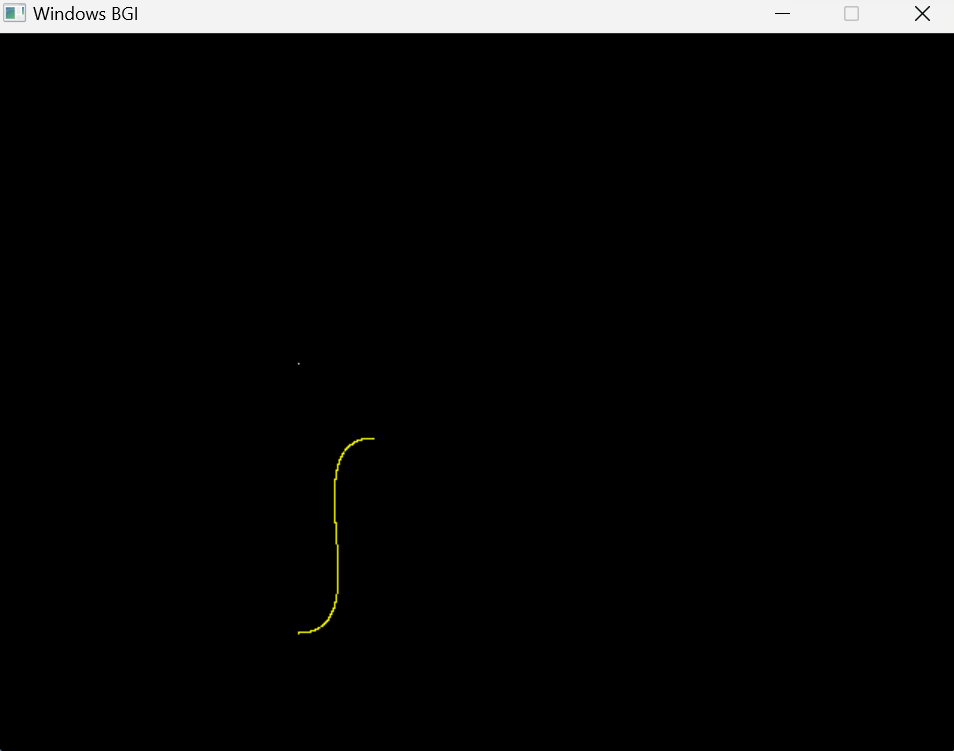
getch();

closegraph();

return 0;

}

**OUTPUT:**

****