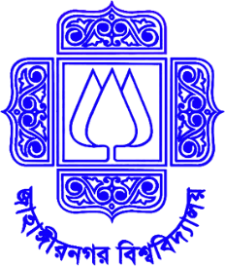
***Course Title:*** *Computer Graphics Laboratory*

***Course code:*** *CSE-304*

*3rd year 1st semester*

**Date of Submission**: 04/06/2023



###### **Submitted to-**

###### Dr. Mohammad Shorif Uddin

###### **Professor**

###### and

###### Dr. Morium Akter

###### **Associate Professor**

*Department of Computer Science and Engineering*

*Jahangirnagar University*

*Savar, Dhaka-1342*

| **Sl** | Class Roll | Exam Roll | Name |
| --- | --- | --- | --- |
| 01 |  |  |  |

**Experiment NO: 0 1**

**Name of Experiment: Scan conversion of a point.**

**Source Code:**

#include<graphics.h>

#include<conio.h>

#include<bits/stdc++.h>

using namespace std;

int main()

{

int gd = DETECT, gm;

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

float x = 200, y = 200;

putpixel (x, y, WHITE);

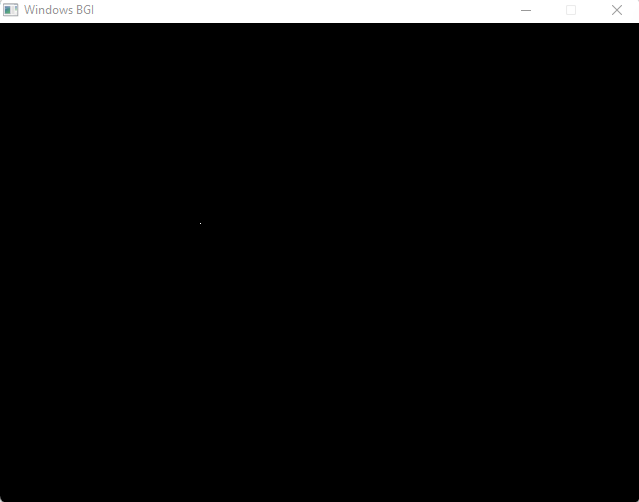
getch();

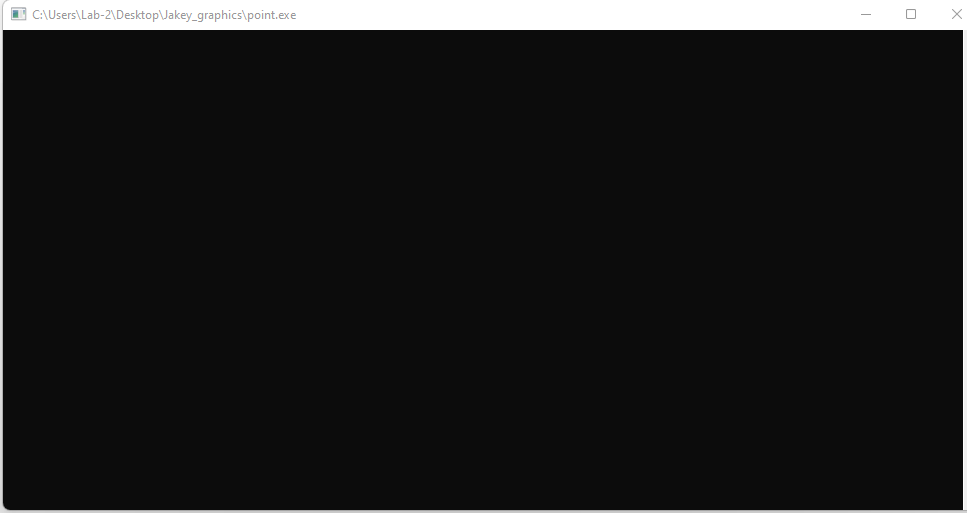
closegraph();

return 0;

}

**Output:**





**Experiment N0 : 0 2**

**Name of Experiment: Scan Conversion of a line using DDA Algorithm**

**Source Code:**

#include<graphics.h>

#include<conio.h>

#include<stdio.h>

int main()

{

int gd = DETECT ,gm, i;

float x, y,dx,dy,steps;

int x0, x1, y0, y1;

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

setbkcolor(WHITE);

x0 = 100 , y0 = 200, x1 = 500, y1 = 300;

dx = (float)(x1 - x0);

dy = (float)(y1 - y0);

if(dx>=dy)

{

steps = dx;

}

else

{

steps = dy;

}

dx = dx/steps;

dy = dy/steps;

x = x0;

y = y0;

i = 1;

while(i<= steps)

{

putpixel(x, y, WHITE);

x += dx;

y += dy;

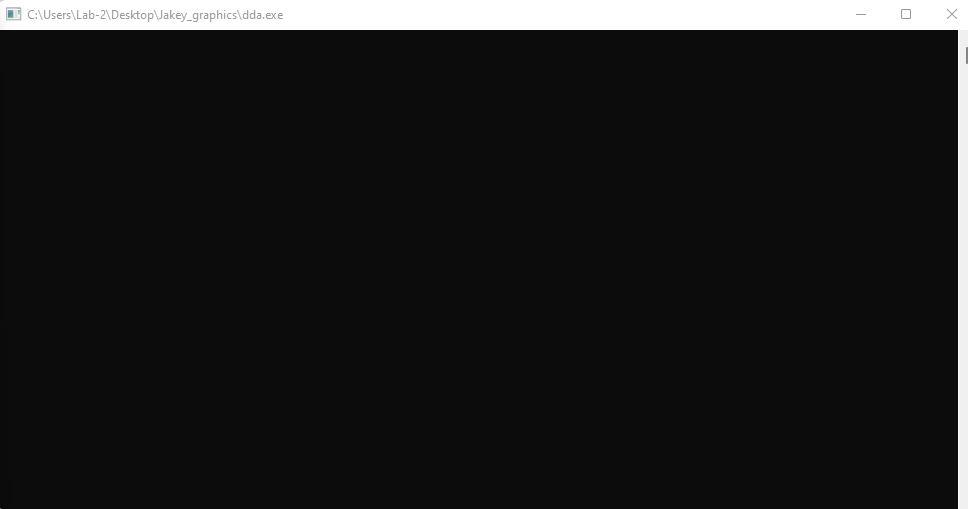
i=i+1;

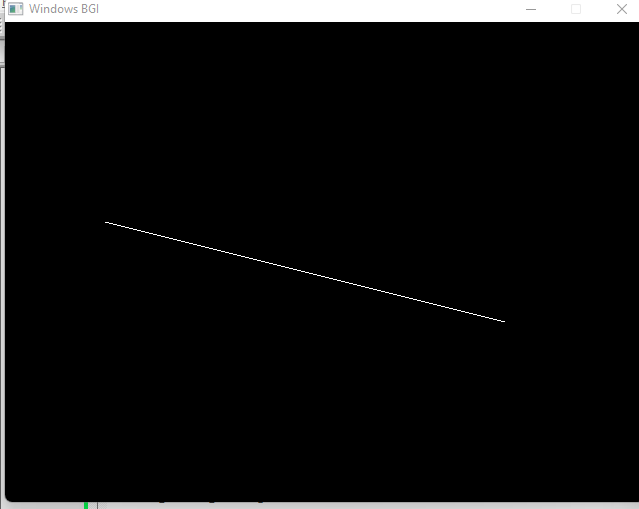
}

getch();

closegraph();

}

**Output: **

****

**Experiment NO : 03**

**Name of Experiment: Scan Conversion of line using Bresenham Algorithm**

**Source Code:**

#include<stdio.h>

#include<graphics.h>

#include<bits/stdc++.h>

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

{

int dx, dy, p, x, y;

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 gd = DETECT, gm;

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

int error, x0, y0, x1, y1;

// initgraph(&gdriver, &gmode, "c:\\turboc3\\bgi");

printf("Enter co-ordinates of first point: ");

scanf("%d%d", &x0, &y0);

printf("Enter co-ordinates of second point: ");

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

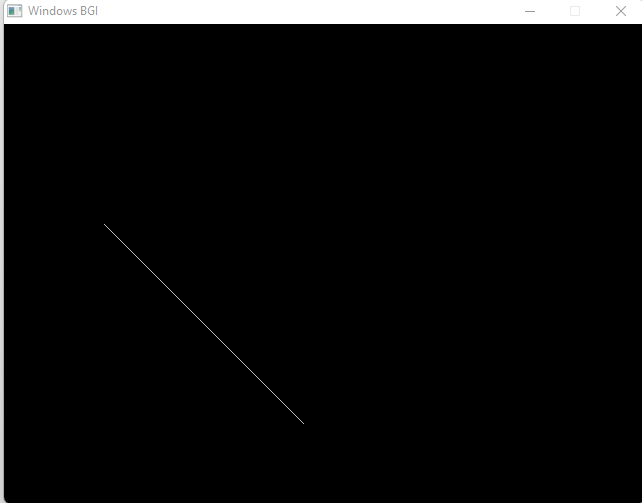
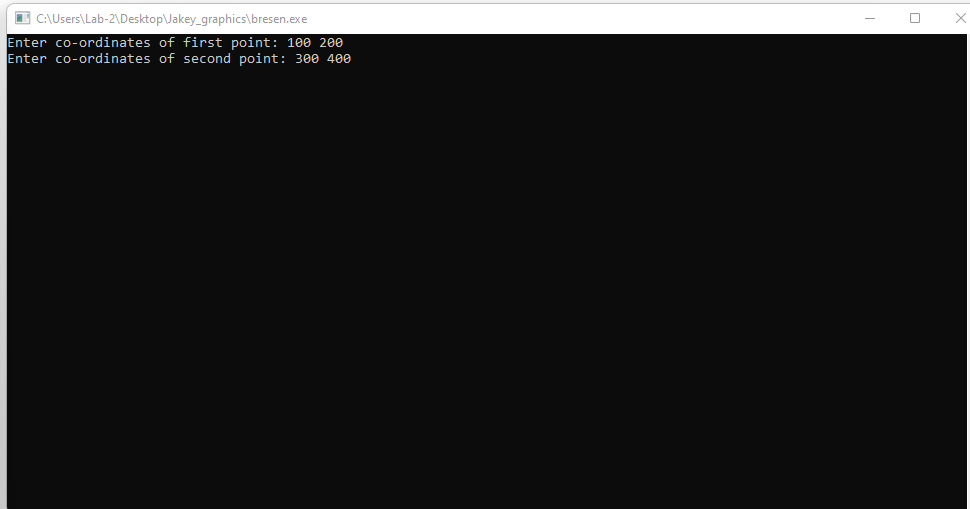
drawline(x0, y0, x1, y1);

getch();

return 0;

}

**Output:**

****

**Experiment No : 04**

**Name of Experiment: Scan Conversion of circle using bresenhen Algorithm**

**Source Code:**

#include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

#include <math.h>

void EightWaySymmetricPlot(int xc,int yc,int x,int y)

{

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

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

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

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

putpixel(y+xc,x+yc,12);

putpixel(y+xc,-x+yc,14);

putpixel(-y+xc,-x+yc,15);

putpixel(-y+xc,x+yc,6);

}

void BresenhamCircle(int xc,int yc,int r)

{

int x=0,y=r,d=3-(2\*r);

EightWaySymmetricPlot(xc,yc,x,y);

while(x<=y)

{

if(d<=0)

{

d=d+(4\*x)+6;

}

else

{

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

y=y-1;

}

x=x+1;

EightWaySymmetricPlot(xc,yc,x,y);

}

}

int main(void)

{

/\* request auto detection \*/

int xc,yc,r,gdriver = DETECT, gmode, errorcode;

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "C:\\TURBOC3\\BGI");

/\* read result of initialization \*/

errorcode = graphresult();

if (errorcode != grOk) /\* an error occurred \*/

{

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

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

getch();

exit(1); /\* terminate with an error code \*/

}

printf("Enter the values of xc and yc :");

scanf("%d%d",&xc,&yc);

printf("Enter the value of radius :");

scanf("%d",&r);

BresenhamCircle(xc,yc,r);

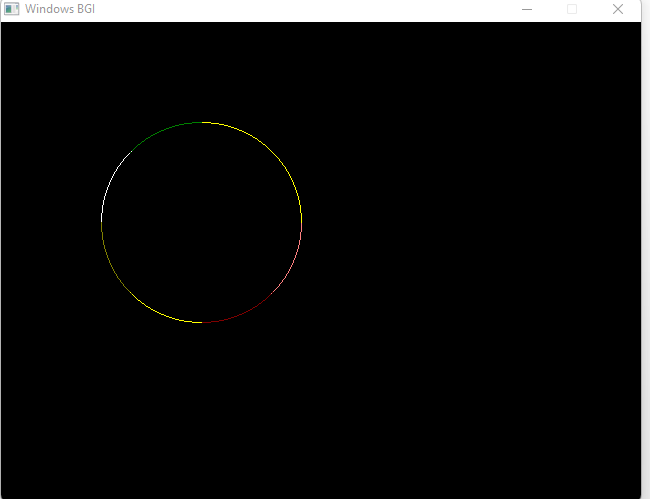
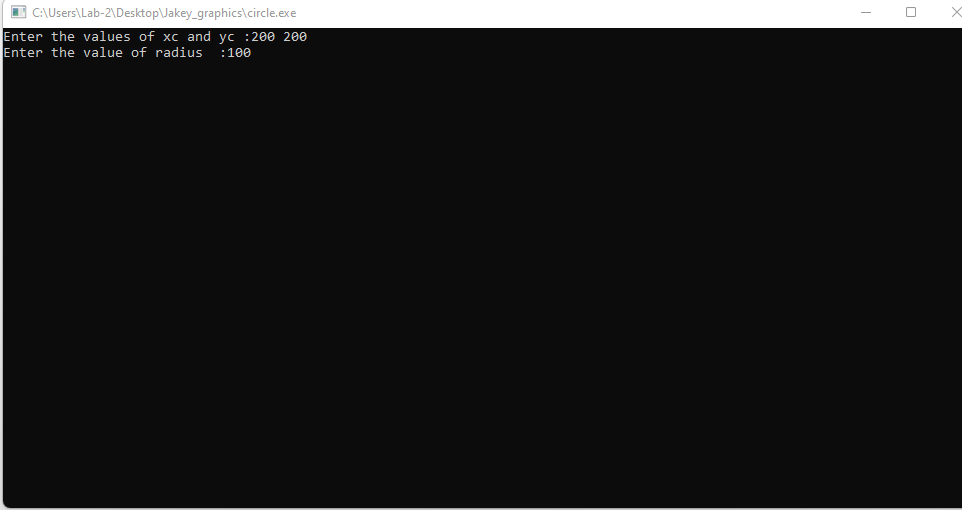
getch();

closegraph();

return 0;

}

**Output:**

****

**Experiment No : 05**

**Name OF Experiment : Scan Conversion OF an Ellipse**

**Source Code of Ellipse:**

#include <iostream>

#include <graphics.h>

void plotEllipsePoints(int xc, int yc, int x, int 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);

}

void drawEllipse(int xc, int yc, int rx, int ry)

{

int x = 0;

int y = ry;

int rxSq = rx \* rx;

int rySq = ry \* ry;

int twoRxSq = 2 \* rxSq;

int twoRySq = 2 \* rySq;

int p;

int px = 0;

int py = twoRxSq \* y;

plotEllipsePoints(xc, yc, x, y);

p = rySq - (rxSq \* ry) + (0.25 \* rxSq);

while (px < py)

{

x++;

px += twoRySq;

if (p < 0)

{

p += rySq + px;

}

else

{

y--;

py -= twoRxSq;

p += rySq + px - py;

}

plotEllipsePoints(xc, yc, x, y);

}

p = rySq \* (x + 0.5) \* (x + 0.5) + rxSq \* (y - 1) \* (y - 1) - rxSq \* rySq;

while (y > 0)

{

y--;

py -= twoRxSq;

if (p > 0)

{

p += rxSq - py;

}

else

{

x++;

px += twoRySq;

p += rxSq - py + px;

}

plotEllipsePoints(xc, yc, x, y);

}

}

int main()

{

int gd = DETECT, gm;

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

int xc = 250; // X-coordinate of the center of the ellipse

int yc = 250; // Y-coordinate of the center of the ellipse

int rx = 150; // X-radius of the ellipse

int ry = 100; // Y-radius of the ellipse

drawEllipse(xc, yc, rx, ry);

delay(5000);

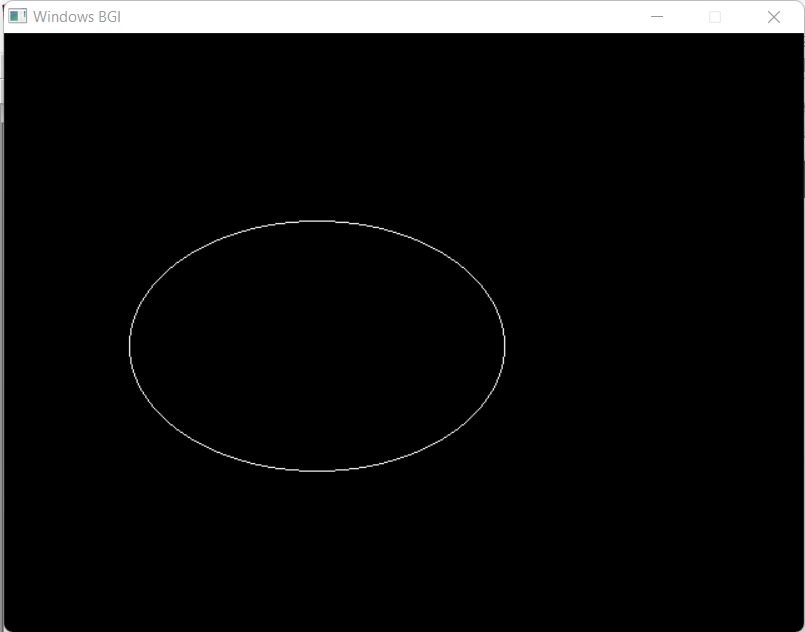
getch();

closegraph();

return 0;

}

**Output :**



**Experiment No : 06**

**Name OF Experiment : Scan Conversion a Circle using Mid-Point Algorithm**

**Source Code of Circle(Mid-Point):**

#include <iostream>

#include <graphics.h>

using namespace std;

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

int x = 0;

int y = radius;

int p = 1 - radius;

while (x <= y) {

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

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

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

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

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

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

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

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

x++;

if (p < 0) {

p += 2 \* x + 1;

} else {

y--;

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

}

}

}

int main() {

int xc, yc, radius;

cout << "Enter center coordinates (xc and yc): ";

cin >> xc >> yc;

cout << "Enter radius: ";

cin >> radius;

int gd = DETECT, gm;

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

drawCircle(xc, yc, radius);

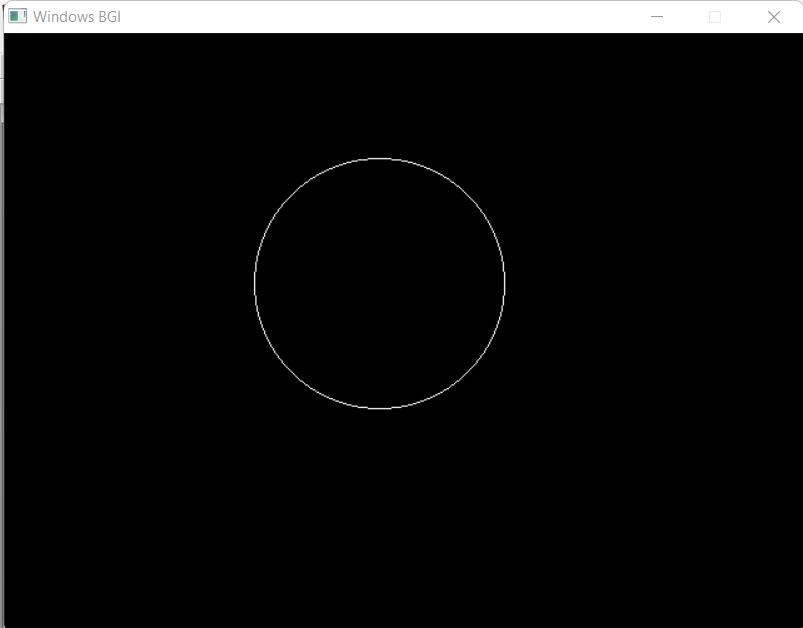
getch();

closegraph();

return 0;

}

**Output:**



**Experiment No : 07**

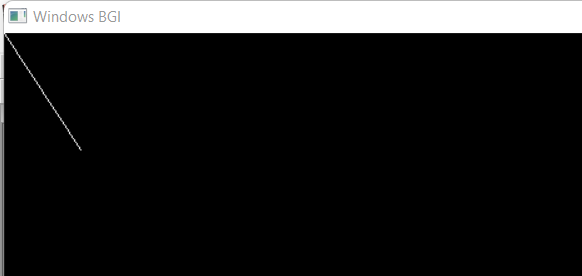
**Name of Experiment: *Scan convert a Line From p1(0,0) To p2(100,50) and perform specific task…***

**Problem 1:** *Rotating canned line at 30 degree angle*

**Source Code:**

| #include <graphics.h>  #include <cmath>  void drawLine(int x1, int y1, int x2, int y2) {  int dx = abs(x2 - x1);  int dy = abs(y2 - y1);  int sx = (x1 < x2) ? 1 : -1;  int sy = (y1 < y2) ? 1 : -1;  int err = dx - dy;  while (true) {  putpixel(x1, y1, WHITE);  if (x1 == x2 && y1 == y2)  break;  int e2 = 2 \* err;  if (e2 > -dy) {  err -= dy;  x1 += sx;  }  if (e2 < dx) {  err += dx;  y1 += sy;  }  }  }  } | int main() {  int gd = DETECT, gm;  initgraph(&gd, &gm, "");  int x1 = 0; // Starting X-coordinate  int y1 = 0; // Starting Y-coordinate  int x2 = 100; // Ending X-coordinate  int y2 = 50; // Ending Y-coordinate  // Rotate the line by 30 degrees  double angle = 30 \* (3.14159 / 180.0);  int rotatedX2 = static\_cast<int>(x1 + (x2 - x1) \* cos(angle) - (y2 - y1) \* sin(angle));  int rotatedY2 = static\_cast<int>(y1 + (x2 - x1) \*  sin(angle) + (y2 - y1) \* cos(angle));  drawLine(x1, y1, rotatedX2, rotatedY2);  getch();  closegraph();  return 0; |
| --- | --- |

**Output:**

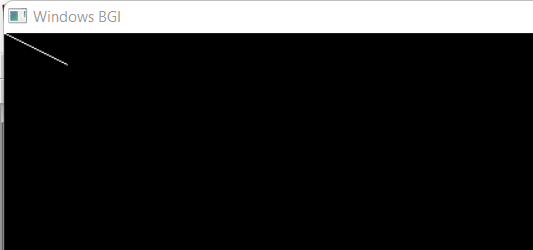


**Problem 02:** *Scaling scanned line at 50%*

**Source Code:**

| #include <graphics.h>  void drawLine(int x1, int y1, int x2, int y2) {  int dx = abs(x2 - x1);  int dy = abs(y2 - y1);  int sx = (x1 < x2) ? 1 : -1;  int sy = (y1 < y2) ? 1 : -1;  int err = dx - dy;  while (true) {  putpixel(x1, y1, WHITE);  if (x1 == x2 && y1 == y2)  break;  int e2 = 2 \* err;  if (e2 > -dy) {  err -= dy;  x1 += sx;  }  if (e2 < dx) {  err += dx;  y1 += sy;  }  }  } | int main() {  int gd = DETECT, gm;  initgraph(&gd, &gm, "");  int x1 = 0; // Starting X-coordinate  int y1 = 0; // Starting Y-coordinate  int x2 = 100; // Ending X-coordinate  int y2 = 50; // Ending Y-coordinate  // Scale the line by 50%  double scaleFactor = 0.5;  int scaledX2 = static\_cast<int>(x1 + (x2 - x1) \* scaleFactor);  int scaledY2 = static\_cast<int>(y1 + (y2 - y1) \* scaleFactor);  drawLine(x1, y1, scaledX2, scaledY2);  getch();  closegraph();  return 0;  } |
| --- | --- |

**Output:**



**Problem 03:** *Translate scanned line at 75 pixel*

**Source Code:**

| #include <graphics.h>  void drawLine(int x1, int y1, int x2, int y2) {  int dx = abs(x2 - x1);  int dy = abs(y2 - y1);  int sx = (x1 < x2) ? 1 : -1;  int sy = (y1 < y2) ? 1 : -1;  int err = dx - dy;  while (true) {  putpixel(x1, y1, WHITE);  if (x1 == x2 && y1 == y2)  break;  int e2 = 2 \* err;  if (e2 > -dy) {  err -= dy;  x1 += sx;  }  if (e2 < dx) {  err += dx;  y1 += sy;  }  }  } | int main() {  int gd = DETECT, gm;  initgraph(&gd, &gm, "");  int x1 = 0; // Starting X-coordinate  int y1 = 0; // Starting Y-coordinate  int x2 = 100; // Ending X-coordinate  int y2 = 50; // Ending Y-coordinate  // Translate the line by 75 pixels in both x and y directions  int translatedX1 = x1 + 75;  int translatedY1 = y1 + 75;  int translatedX2 = x2 + 75;  int translatedY2 = y2 + 75;  drawLine(translatedX1, translatedY1, translatedX2, translatedY2);  getch();  closegraph();  return 0;  } |
| --- | --- |

**Output:**



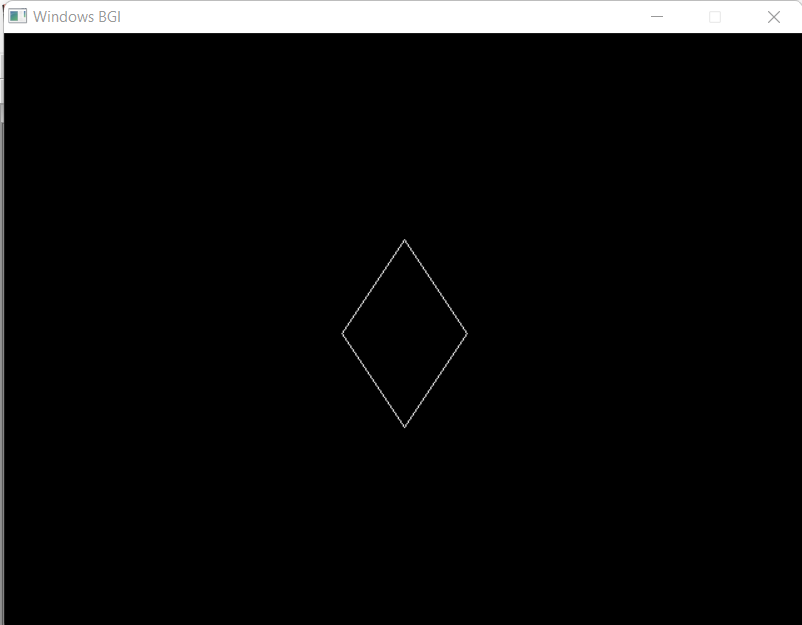
**Experiment No:08**

**Name OF Experiment:** *Scan Conversion A Kite using Bresenham’s Line algorithm*

**Source Code:**

| #include <graphics.h>  void drawLine(int x1, int y1, int x2, int y2) {  int dx = abs(x2 - x1);  int dy = abs(y2 - y1);  int sx = (x1 < x2) ? 1 : -1;  int sy = (y1 < y2) ? 1 : -1;  int err = dx - dy;  while (true) {  putpixel(x1, y1, WHITE);  if (x1 == x2 && y1 == y2)  break;  int e2 = 2 \* err;  if (e2 > -dy) {  err -= dy;  x1 += sx;  }  if (e2 < dx) {  err += dx;  y1 += sy;  }  }  }  void drawKite() {  int centerX = 320; // X-coordinate of the kite's center  int centerY = 240; // Y-coordinate of the kite's center  int width = 100; // Width of the kite  int height = 150; // Height of the kite  int topX = centerX;  int topY = centerY - (height / 2); | int bottomX = centerX;  int bottomY = centerY + (height / 2);  int leftX = centerX - (width / 2);  int leftY = centerY;  int rightX = centerX + (width / 2);  int rightY = centerY;  drawLine(topX, topY, leftX, leftY); // Top to left  drawLine(topX, topY, rightX, rightY); // Top to right  drawLine(bottomX, bottomY, leftX, leftY); // Bottom to left  drawLine(bottomX, bottomY, rightX, rightY); // Bottom to right  }  int main() {  int gd = DETECT, gm;  initgraph(&gd, &gm, "");  drawKite();  getch();  closegraph();  return 0;  } |
| --- | --- |

**Output:**



**Experiment No : 09**

**Name of Experiment: Line Clipping using Cohen-Sutherland Algorithm.**

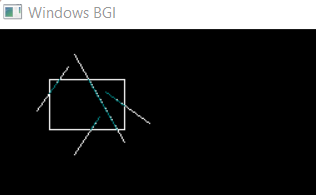
**Introduction:**

The Cohen-Sutherland algorithm is a fundamental technique in 2D computer graphics used for line clipping within a specified rectangular area, known as a clipping window. Developed by Ivan Sutherland and Gary Cohen in 1967, this algorithm efficiently identifies visible parts of a line segment by dividing the 2D space into different regions and utilizing three-bit codes to classify points in relation to the clipping window. By swiftly discarding or partially clipping line segments based on these codes, the algorithm optimizes the rendering process, making it a crucial tool for real-time graphics and graphical user interfaces. This report delves into the Cohen-Sutherland algorithm, its principles, implementation, and practical implications in computer graphics.

**Source Code:**

| #include <bits/stdc++.h>  #include <graphics.h>  using namespace std;  int xmin, xmax, ymin, ymax;  struct lines  {  int x1, y1, x2, y2;  };  {  if (x > 0)  return 1;  else  return 0;  }  void clip(struct lines mylines)  {  int bits[4], byte[4], i, var;  setcolor(RED);  bits[0] = sign(xmin - mylines.x1);  byte[0] = sign(xmin - mylines.x2);  bits[1] = sign(mylines.x1 - xmax);  byte[1] = sign(mylines.x2 - xmax);  bits[2] = sign(ymin - mylines.y1);  byte[2] = sign(ymin - mylines.y2);  bits[3] = sign(mylines.y1 - ymax);  byte[3] = sign(mylines.y2 - ymax);  string initial = "", end = "", temp = "";  for (i = 0; i < 4; i++)  {  if (bits[i] == 0)  initial += '0';  else  initial += '1';  }  for (i = 0; i < 4; i++)  {  if (byte[i] == 0)  end += '0';  else  end += '1';  }  m=dy/dx;  float m = (mylines.y2 - mylines.y1) / (float)(mylines.x2 - mylines.x1);  float c = mylines.y1 - m \* mylines.x1;  if (initial == end && end == "0000") {    line(mylines.x1, mylines.y1, mylines.x2, mylines.y2);  return;  }  else {  for (i = 0; i < 4; i++) {  int val = (bits[i] & byte[i]);  if (val == 0)  temp += '0';  else  temp += '1';  }    if (temp != "0000")  return;  for (i = 0; i < 4; i++) {    if (bits[i] == byte[i])  continue;    if (i == 0 && bits[i] == 1) {  var = round(m \* xmin + c);  mylines.y1 = var;  mylines.x1 = xmin;  }    if (i == 0 && byte[i] == 1) {  var = round(m \* xmin + c);  mylines.y2 = var;  mylines.x2 = xmin;  }    if (i == 1 && bits[i] == 1) {  var = round(m \* xmax + c);  mylines.y1 = var;  mylines.x1 = xmax;  }    if (i == 1 && byte[i] == 1) {  var = round(m \* xmax + c);  mylines.y2 = var;  mylines.x2 = xmax;  } | if (i == 2 && bits[i] == 1)  {  var = round((float)(ymin - c) / m);  mylines.y1 = ymin;  mylines.x1 = var;  }    if (i == 2 && byte[i] == 1)  {  var = round((float)(ymin - c) / m);  mylines.y2 = ymin;  mylines.x2 = var;  }    if (i == 3 && bits[i] == 1)  {  var = round((float)(ymax - c) / m);  mylines.y1 = ymax;  mylines.x1 = var;  }    if (i == 3 && byte[i] == 1)  {  var = round((float)(ymax - c) / m);  mylines.y2 = ymax;  mylines.x2 = var;  }  bits[0] = sign(xmin - mylines.x1);  byte[0] = sign(xmin - mylines.x2);  bits[1] = sign(mylines.x1 - xmax);  byte[1] = sign(mylines.x2 - xmax);  bits[2] = sign(ymin - mylines.y1);  byte[2] = sign(ymin - mylines.y2);  bits[3] = sign(mylines.y1 - ymax);  byte[3] = sign(mylines.y2 - ymax);    }  initial = "", end = "";    for (i = 0; i < 4; i++)  {  if (bits[i] == 0)  initial += '0';  else  initial += '1';  }  for (i = 0; i < 4; i++)  {  if (byte[i] == 0)  end += '0';  else  end += '1';  }  if (initial == end && end == "0000")  {  line(mylines.x1, mylines.y1, mylines.x2, mylines.y2);  return;  }  else  return;  }  }  int main()  {  int gd = DETECT, gm;  xmin = 40;  xmax = 100;  ymin = 40;  ymax = 80;  initgraph(&gd, &gm, NULL);  line(xmin, ymin, xmax, ymin);  line(xmax, ymin, xmax, ymax);  line(xmax, ymax, xmin, ymax);  line(xmin, ymax, xmin, ymin);  struct lines mylines[4];  mylines[0].x1 = 30;  mylines[0].y1 = 65;  mylines[0].x2 = 55;  mylines[0].y2 = 30;  mylines[1].x1 = 60;  mylines[1].y1 = 20;  mylines[1].x2 = 100;  mylines[1].y2 = 90;  mylines[2].x1 = 60;  mylines[2].y1 = 100;  mylines[2].x2 = 80;  mylines[2].y2 = 70;  mylines[3].x1 = 85;  mylines[3].y1 = 50;  mylines[3].x2 = 120;  mylines[3].y2 = 75;  for (int i = 0; i < 4; i++)  {  line(mylines[i].x1, mylines[i].y1,  mylines[i].x2, mylines[i].y2);  delay(1000);  }  for (int i = 0; i < 4; i++)  {  clip(mylines[i]);  delay(1000);  }  delay(4000);  getch();  closegraph();  return 0;  } |
| --- | --- |

**Output:**



**Discussion:**

The Cohen-Sutherland algorithm demonstrates its efficiency in line clipping by efficiently categorizing points and quickly determining their visibility with respect to a clipping window. Its strength lies in the rapid elimination of line segments that are entirely outside the window, which is vital for real-time graphics and rendering. However, this efficiency comes at the cost of potential precision issues and increased complexity in handling partial clipping scenarios.

While the algorithm's basic implementation is straightforward, converting coordinates to region codes introduces computational overhead. The trade-off between efficiency and precision becomes evident when comparing the algorithm to alternatives like the Liang-Barsky algorithm.

In practical terms, the Cohen-Sutherland algorithm is suitable for static clipping windows and scenarios where speed is critical. Nevertheless, for dynamic scenes or intricate shapes, alternative algorithms like Cyrus-Beck might be more appropriate.

This report highlights the algorithm's significance in 2D graphics, showcasing its advantages and limitations. Further exploration could involve refining its performance for specific applications or investigating hybrid approaches for line clipping in dynamic environments.

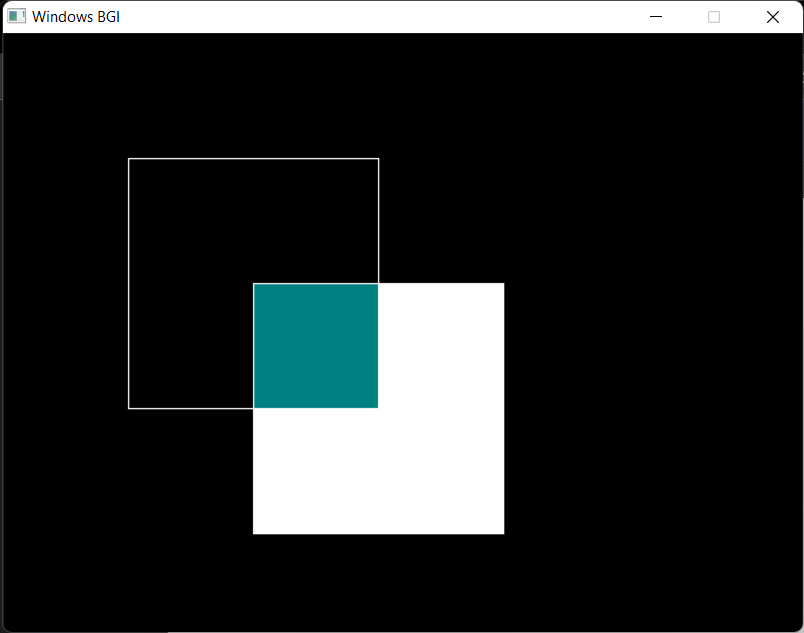
**Experiment No : 10**

**Experiment Name:** *Boundary and flood filing of an object.*

**Source code:**

| #include <graphics.h>  // Function to perform boundary filling  void boundaryFill(int x, int y, int fillColor, int boundaryColor) {  if (getpixel(x, y) != fillColor && getpixel(x, y) != boundaryColor) {  putpixel(x, y, fillColor);  // Perform boundary fill recursively in all 4 directions  boundaryFill(x + 1, y, fillColor, boundaryColor); // Right  boundaryFill(x - 1, y, fillColor, boundaryColor); // Left  boundaryFill(x, y + 1, fillColor, boundaryColor); // Up  boundaryFill(x, y - 1, fillColor, boundaryColor); // Down  }  }  floodFill(x, y - 1, fillColor, oldColor); // Down  }  } | // Function to perform flood filling  void floodFill(int x, int y, int fillColor, int oldColor) {  if (getpixel(x, y) == oldColor) {  putpixel(x, y, fillColor);  // Perform flood fill recursively in all 4 directions  floodFill(x + 1, y, fillColor, oldColor); // Right  floodFill(x - 1, y, fillColor, oldColor); // Left  floodFill(x, y + 1, fillColor, oldColor); // Up  int main() {  int gd = DETECT, gm;  initgraph(&gd, &gm, "");  // Draw the object  rectangle(100, 100, 300, 300);  rectangle(200, 200, 400, 400);  // Boundary filling  boundaryFill(250, 250, CYAN, WHITE);  // Flood filling  floodFill(350, 350, WHITE, BLACK);  getch();  closegraph();  return 0;  } |
| --- | --- |

**Output:**



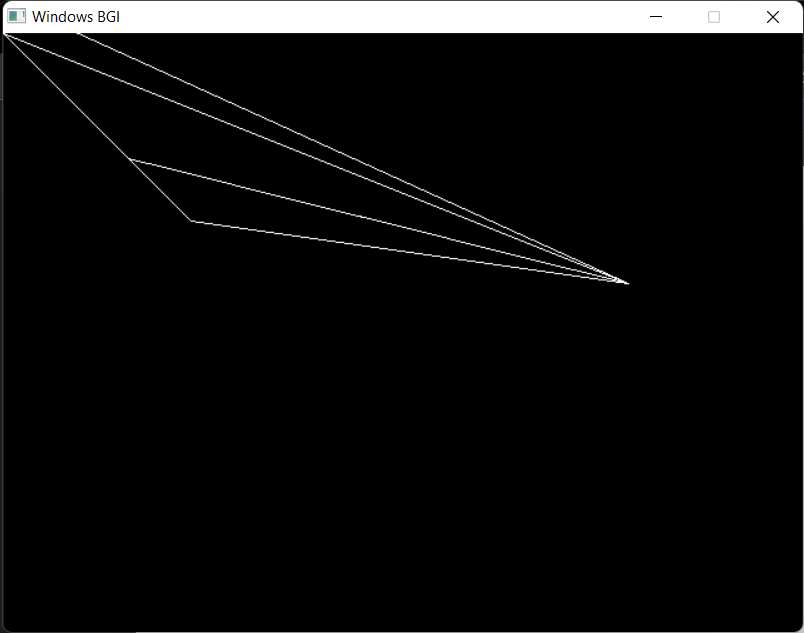
**Experiment No: 11**

**Experiment Name:** *Resize to double of a triangle whose vertices are A(0,0), B(1,1), C(5,2). Keep the vertex C(5,2) is fixed.*

**Source Code:**

| #include <iostream>  #include <graphics.h>  void drawTriangle(int x1, int y1, int x2, int y2, int x3, int y3) {  line(x1, y1, x2, y2);  line(x2, y2, x3, y3);  line(x3, y3, x1, y1);  }  void resizeTriangle(int& x1, int& y1, int& x2, int& y2, int x3, int y3) {  int centerX = (x1 + x2) / 2;  int centerY = (y1 + y2) / 2;  x1 = 2 \* (x1 - centerX) + centerX;  y1 = 2 \* (y1 - centerY) + centerY;  x2 = 2 \* (x2 - centerX) + centerX;  y2 = 2 \* (y2 - centerY) + centerY;  } | int main() {  int gd = DETECT, gm;  initgraph(&gd, &gm, "");  int x1 = 0, y1 = 0;  int x2 = 100, y2 = 100;  int x3 = 500, y3 = 200;  // Draw the initial triangle  drawTriangle(x1, y1, x2, y2, x3, y3);  // Delay to see the initial triangle  delay(1000);  // Resize the triangle  resizeTriangle(x1, y1, x2, y2, x3, y3);  // Draw the resized triangle  drawTriangle(x1, y1, x2, y2, x3, y3);  // Close the graphics window  delay(2000);  //closegraph();  return 0;  } |
| --- | --- |

**Output:**



**Experiment No : 12**

**Name of Experiment: Liang-Barsky Algorithm**

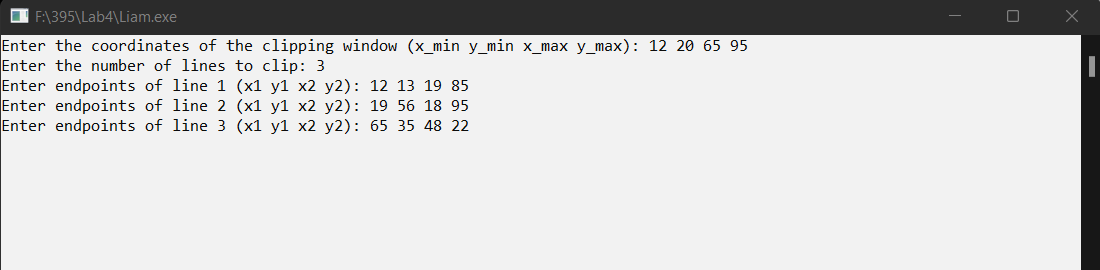
**Introduction:**

The Liang-Barsky algorithm is a technique used for quickly determining the visible portion of a line segment within a given clipping window. Clipping is a fundamental concept in computer graphics used to determine which parts of an object are visible within a specified viewing area. The Liang-Barsky algorithm is particularly useful for line clipping and helps optimize the process of deciding whether a line lies completely within the viewing area, partially intersects it, or lies outside.This lab report aims to present an implementation of the Liang-Barsky algorithm, provide example code, showcase the output, and discuss its performance and advantages.

**Code:**

| #include <iostream>  #include <cmath>  #include <graphics.h>  using namespace std;  const int LEFT = 1, RIGHT = 2, BOTTOM = 4, TOP = 8;  int x\_min, y\_min, x\_max, y\_max;  int computeCode(int x, int y) {  int code = 0;  if (x < x\_min)  code |= LEFT;  if (x > x\_max)  code |= RIGHT;  if (y < y\_min)  code |= BOTTOM;  if (y > y\_max)  code |= TOP;  return code;  }  void liangBarsky(int x1, int y1, int x2, int y2) {  int code1 = computeCode(x1, y1);  int code2 = computeCode(x2, y2);  bool accept = false;  while (true) {  if (!(code1 | code2)) {  accept = true;  break;  } else if (code1 & code2) {  break;  } else {  int codeOut = code1 ? code1 : code2;  int x, y;  if (codeOut & TOP) {  x = x1 + (x2 - x1) \* (y\_max - y1) / (y2 - y1);  y = y\_max;  } else if (codeOut & BOTTOM) {  x = x1 + (x2 - x1) \* (y\_min - y1) / (y2 - y1);  y = y\_min;  } else if (codeOut & RIGHT) {  y = y1 + (y2 - y1) \* (x\_max - x1) / (x2 - x1); | x = x\_max;  } else if (codeOut & LEFT) {  y = y1 + (y2 - y1) \* (x\_min - x1) / (x2 - x1);  x = x\_min;  }  if (codeOut == code1) {  x1 = x;  y1 = y;  code1 = computeCode(x1, y1);  } else {  x2 = x;  y2 = y;  code2 = computeCode(x2, y2);  }  }  }  if (accept) {  line(x1, y1, x2, y2);  }  }  int main() {  int gd = DETECT, gm;  initgraph(&gd, &gm, "");  cout << "Enter the coordinates of the clipping window (x\_min y\_min x\_max y\_max): ";  cin >> x\_min >> y\_min >> x\_max >> y\_max;  rectangle(x\_min, y\_min, x\_max, y\_max);  int numLines;  cout << "Enter the number of lines to clip: ";  cin >> numLines;  for (int i = 0; i < numLines; i++) {  int x1, y1, x2, y2;  cout << "Enter endpoints of line " << (i + 1) << " (x1 y1 x2 y2): ";  cin >> x1 >> y1 >> x2 >> y2;  liangBarsky(x1, y1, x2, y2);  }  delay(10000);  closegraph();  return 0;  } |
| --- | --- |

**Output:**



|  |  |
| --- | --- |

**Discussion:**

The Liang-Barsky algorithm is a valuable technique in computer graphics for efficiently determining the visible portion of a line segment within a specified clipping window. This algorithm plays a crucial role in optimizing the rendering process by identifying the portions of objects that are within the viewing area and need to be displayed.In the presented code example, the Liang-Barsky algorithm is implemented to clip a line segment against a given rectangular clipping window. The algorithm involves iteratively calculating intersection points with the clipping window's sides and determining whether the line segment lies completely inside, partially intersects, or lies entirely outside the window. The parameter values u1 and u2 indicate the points of intersection along the line.The key advantages of the Liang-Barsky algorithm include its efficiency and ease of implementation. It operates using precomputed parameters and simple arithmetic calculations, avoiding the need for complex floating-point operations. This efficiency makes it a preferred choice over other algorithms, such as the Cohen-Sutherland algorithm, when processing a large number of line segments.However, like any algorithm, Liang-Barsky has its limitations. It might encounter issues when dealing with line segments that are nearly parallel to the clipping window's sides. This situation can result in small parameter ranges, potentially leading to inaccuracies in determining the visible portion. Additionally, when the clipping window is significantly smaller than the line segment, the algorithm's performance might degrade, as the majority of the line lies outside the window.In conclusion, the Liang-Barsky algorithm is a valuable tool in computer graphics for efficient line clipping. Its ability to quickly identify visible segments of lines within a specified window helps optimize rendering processes. While it may have limitations in certain scenarios, its overall benefits in terms of performance and simplicity make it a favorable choice for many applications requiring line clipping.

**Experiment No: 13**

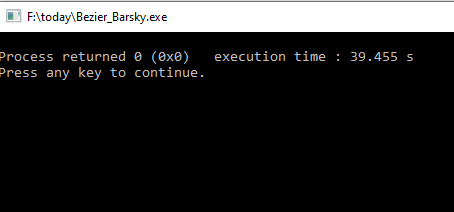
**Experiment Name:** Line clipping using Bezier Barsky polynomial approximations.

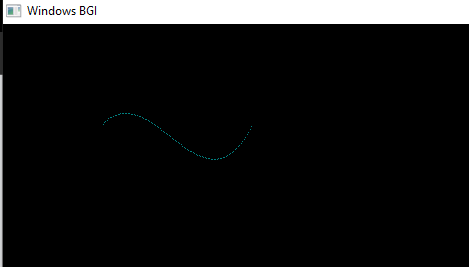
**Introduction:**In this lab report, we explore how the Barsky-Bezier algorithm works. We'll see how it tweaks Cohen-Sutherland for curves and uses clever math to reshape curves. A hands-on demo using the graphics.h library will illustrate how we can practically draw and clip Bézier curves, opening a window to the world of graphic manipulation.Through this report, we aim to introduce you to the art of Bézier curve clipping, the magic of Barsky-Bezier, and how these concepts can jazz up your visual creations.

**Source Code:**

| #include <iostream>  #include <cmath>  #include <graphics.h>  // Define a structure to represent a 2D point  struct Point {  int x, y;  };  // Function to calculate the Bézier curve point using polynomial approximation  Point calculateBezierPoint(Point p[], double t) {  double u = 1.0 - t;  double tt = t \* t;  double uu = u \* u;  double uuu = uu \* u;  double ttt = tt \* t;  Point pFinal;  pFinal.x = static\_cast<int>(p[0].x \* uuu + 3 \* p[1].x \* t \* uu + 3 \* p[2].x \* tt \* u + p[3].x \* ttt);  pFinal.y = static\_cast<int>(p[0].y \* uuu + 3 \* p[1].y \* t \* uu + 3 \* p[2].y \* tt \* u + p[3].y \* ttt);  return pFinal;  } | // Main function  int main() {  int gd = DETECT, gm;  initgraph(&gd, &gm, "C:\\Turboc3\\BGI");  // Define Bézier control points  Point controlPoints[4] = {{100, 100}, {150, 50}, {200, 200}, {250, 100}};  // Draw the Bézier curve using polynomial approximation  for (double t = 0.0; t <= 1.0; t += 0.01) {  Point p = calculateBezierPoint(controlPoints, t);  putpixel(p.x, p.y, WHITE);  }  delay(50000); // Pause for a few seconds before closing the graphics window  closegraph();  return 0;  } |
| --- | --- |

**Output:**

****

****

**Discussion:**

The Barsky-Bezier algorithm cleverly clips Bézier curves. It adapts Cohen-Sutherland for curves, preserving their shape while trimming. By finding curve-clip intersections, it uses polynomial equations to approximate clipped parts.

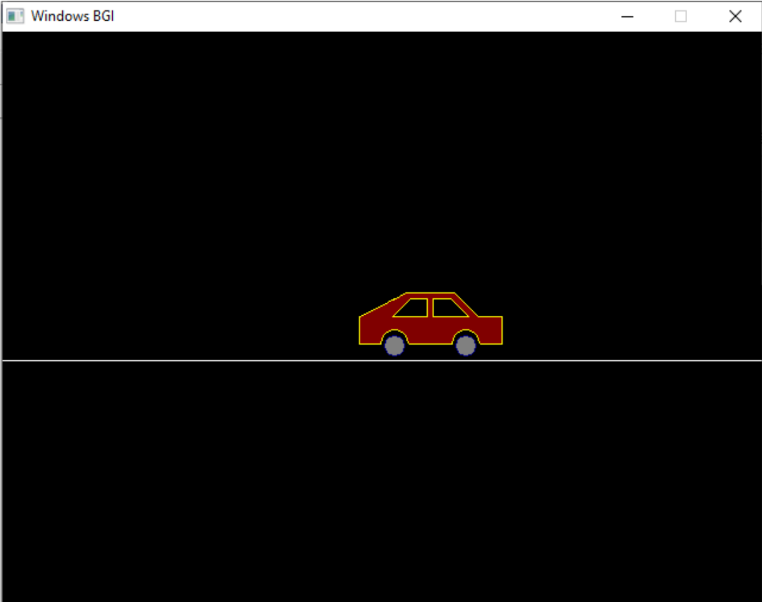
**Experiment No : 14**

**Name of Experiment :** Road Side Animation

**Source Code:**

| #include <stdio.h>  #include <graphics.h>  #include <dos.h>  int main() {  int gd = DETECT, gm;  int i, maxx, midy;  /\* initialize graphic mode \*/  initgraph(&gd, &gm, "X:\\TC\\BGI");  /\* maximum pixel in horizontal axis \*/  maxx = getmaxx();  /\* mid pixel in vertical axis \*/  midy = getmaxy()/2;  for (i=0; i < maxx-150; i=i+5) {  /\* clears screen \*/  cleardevice();  /\* draw a white road \*/  setcolor(WHITE);  line(0, midy + 37, maxx, midy + 37);  /\* Draw Car \*/  setcolor(YELLOW);  setfillstyle(SOLID\_FILL, RED);  line(i, midy + 23, i, midy);  line(i, midy, 40 + i, midy - 20);  line(40 + i, midy - 20, 80 + i, midy - 20);  line(80 + i, midy - 20, 100 + i, midy);  line(100 + i, midy, 120 + i, midy);  line(120 + i, midy, 120 + i, midy + 23);  line(0 + i, midy + 23, 18 + i, midy + 23);  arc(30 + i, midy + 23, 0, 180, 12); | line(42 + i, midy + 23, 78 + i, midy + 23);  arc(90 + i, midy + 23, 0, 180, 12);  line(102 + i, midy + 23, 120 + i, midy + 23);  line(28 + i, midy, 43 + i, midy - 15);  line(43 + i, midy - 15, 57 + i, midy - 15);  line(57 + i, midy - 15, 57 + i, midy);  line(57 + i, midy, 28 + i, midy);  line(62 + i, midy - 15, 77 + i, midy - 15);  line(77 + i, midy - 15, 92 + i, midy);  line(92 + i, midy, 62 + i, midy);  line(62 + i, midy, 62 + i, midy - 15);  floodfill(5 + i, midy + 22, YELLOW);  setcolor(BLUE);  setfillstyle(SOLID\_FILL, DARKGRAY);  /\* Draw Wheels \*/  circle(30 + i, midy + 25, 9);  circle(90 + i, midy + 25, 9);  floodfill(30 + i, midy + 25, BLUE);  floodfill(90 + i, midy + 25, BLUE);  /\* Add delay of 0.1 milli seconds \*/  delay(100);  }  closegraph();  return 0;  } |
| --- | --- |

**Output:**

****