আন্তর্জাতিক ইসলামী বিশ্ববিদ্যালয় চট্টগ্রাম

الجامعة الإسلامية العالمية شيتاغونغ

International Islamic University Chittagong

Lab Report

Course Code: CSE-4742

Course Title: Computer Graphics Lab

Submitted By:

ID: C201242

Name: Farida Nusrat

Semester: 7th

Section: 7FA

Dept. of CSE, IIUC.

Submitted To:

Mrs. Israt Binteh Habib

Dept. Of CSE, IIUC.

Submission Date: 11.11.2023

Table of Content

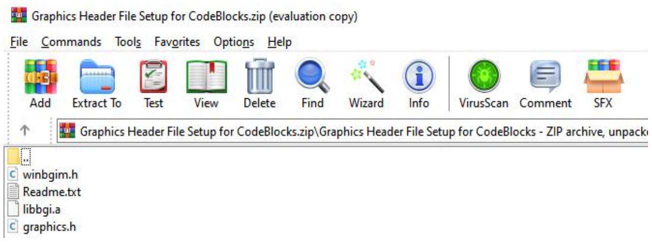
Experiment no: 01

Experiment name: Including graphics.h in codeblocks.

Introduction: Graphics programming is an essential skill for any aspiring programmer or computer scientist. The ability to create visual representations of data, designs, and interactive applications is highly valuable. To start working with graphics in Code::Blocks, we chose to use the graphics.h library, which is based on the Borland Graphics Interface (BGI).

Procedure: The following steps were taken to set up the graphics library in Code::Blocks:

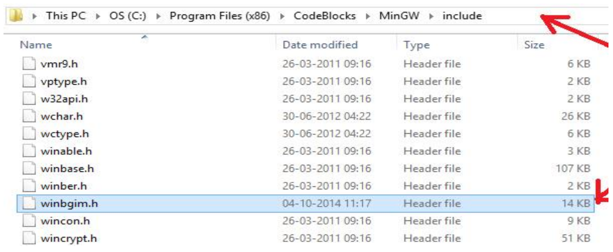
1. To setup "graphics.h" in CodeBlocks, first set up winBGIm graphics library. Download WinBGIm from http://winbgim.codecutter.org/ or use this link.
2. Extract the downloaded file. There will be three file : a) graphics.h, b) winbgim.h c) libbgi.a


Graphics Header File Setup for CodeBlocks.zip (evaluation copy)
File Commands Tools Favorites Options Help
Add Extract To Test View Delete Find Wizard Info VirusScan Comment SFX

Graphics Header File Setup for CodeBlocks.zip\Graphics Header File Setup for CodeBlocks - ZIP archive, unpack

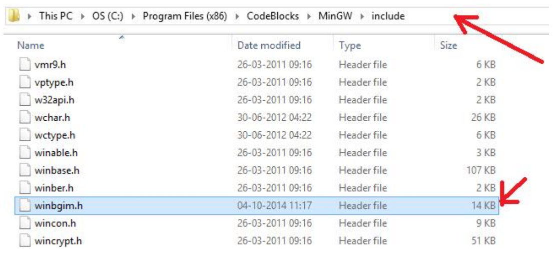
..
winbgim.h
Readme.txt
libbgi.a
graphics.h

3. Copy and paste graphics.h and winbgim.h files into the include folder of compiler directory. (If you have Code::Blocks installed in C drive of your computer, go through: Local C >> TDM-GCC-32 >> include. Paste these two files there.)



This PC › OS (C:) › Program Files (x86) › CodeBlocks › MinGW › include

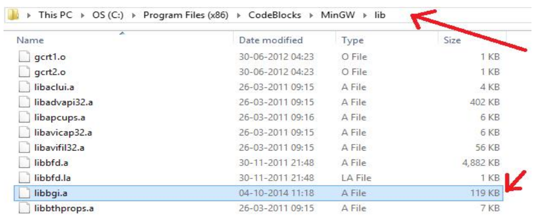
Name	Date modified	Type	Size
vmr9.h	26-03-2011 09:16	Header file	6 KB
vptype.h	26-03-2011 09:16	Header file	2 KB
w32api.h	26-03-2011 09:16	Header file	2 KB
wchar.h	30-06-2012 04:22	Header file	26 KB
wctype.h	30-06-2012 04:22	Header file	6 KB
winable.h	26-03-2011 09:16	Header file	3 KB
winbase.h	26-03-2011 09:16	Header file	107 KB
winber.h	26-03-2011 09:16	Header file	2 KB
winbgim.h	04-10-2014 11:17	Header file	14 KB
wincon.h	26-03-2011 09:16	Header file	9 KB
wincrypt.h	26-03-2011 09:16	Header file	51 KB



This PC › OS (C:) › Program Files (x86) › CodeBlocks › MinGW › include

Name	Date modified	Type	Size
vmr9.h	26-03-2011 09:16	Header file	6 KB
vptype.h	26-03-2011 09:16	Header file	2 KB
w32api.h	26-03-2011 09:16	Header file	2 KB
wchar.h	30-06-2012 04:22	Header file	26 KB
wctype.h	30-06-2012 04:22	Header file	6 KB
winable.h	26-03-2011 09:16	Header file	3 KB
winbase.h	26-03-2011 09:16	Header file	107 KB
winber.h	26-03-2011 09:16	Header file	2 KB
winbgim.h	04-10-2014 11:17	Header file	14 KB
wincon.h	26-03-2011 09:16	Header file	9 KB
wincrypt.h	26-03-2011 09:16	Header file	51 KB

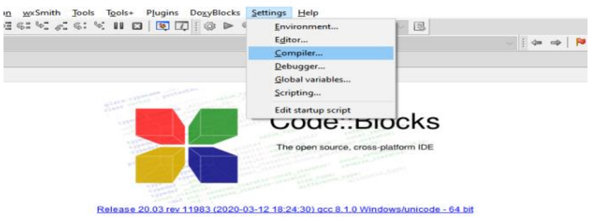
4. Copy and paste **libbgi.a** to the **lib** folder of compiler directory.

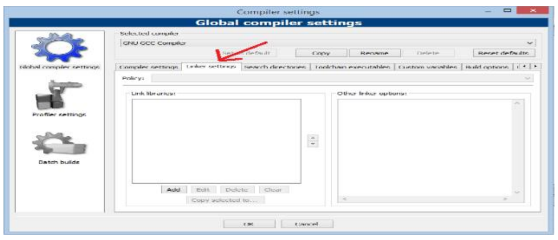


This PC › OS (C:) › Program Files (x86) › CodeBlocks › MinGW › lib

Name	Date modified	Type	Size
gcrt1.o	30-06-2012 04:23	O File	1 KB
gcrt2.o	30-06-2012 04:23	O File	1 KB
libaclui.a	26-03-2011 09:15	A File	4 KB
libadvapi32.a	26-03-2011 09:15	A File	402 KB
libapcups.a	26-03-2011 09:16	A File	6 KB
libavicap32.a	26-03-2011 09:15	A File	6 KB
libavifil32.a	26-03-2011 09:15	A File	56 KB
libbfd.a	30-11-2011 21:48	A File	4,882 KB
libbfd.la	30-11-2011 21:48	LA File	1 KB
libbgi.a	04-10-2014 11:18	A File	119 KB
libbthprops.a	26-03-2011 09:15	A File	7 KB

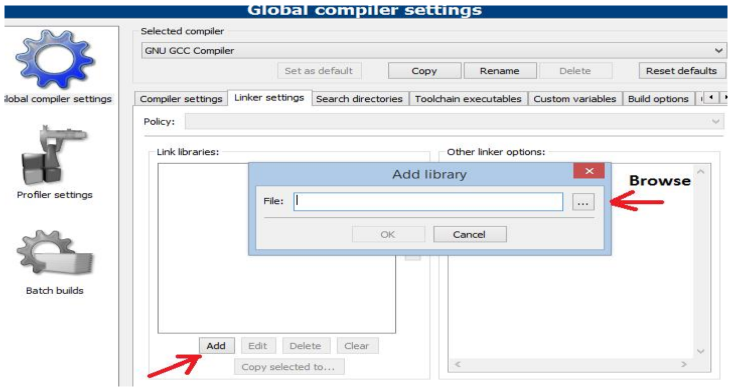
5. Open Code::Blocks. Go to Settings >> Compiler >> Linker settings.


wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Environment...
Editor...
Compiler...
Debugger...
Global variables...
Scripting...
Edit startup script
Code::Blocks
The open source, cross-platform IDE
Release 20.03 rev 11983 (2020-03-12 18:24:30) gcc 8.1.0 Windows/unicode - 64 bit

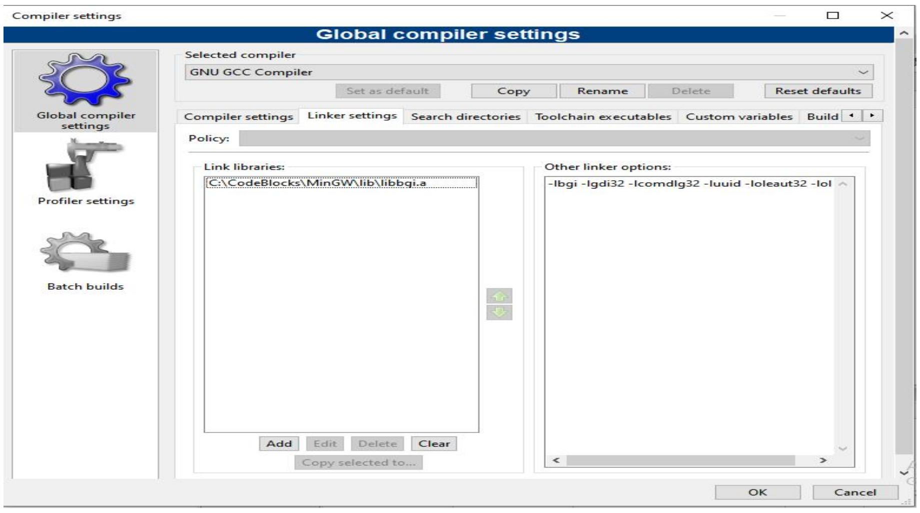


Selected compiler: GNU GCC Compiler

6. In that window, click the Add button under the “Link libraries” part, and browse.



7. Select the libbgi.a file copied to the lib folder and In right part (ie. other linker options) paste commands
-lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32


Selected compiler: GNU GCC Compiler

Link libraries: C:\CodeBlocks\MinGW\lib\libbgi.a

Other linker options: -lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lol

Output: Our Graphics Setup Lab was a success, and we were able to create various graphical applications using the graphics.h library in Code::Blocks. The initial test application displayed a circle on the screen, confirming the proper installation and configuration of the graphics library.

Conclusion: In this lab, we successfully set up the graphics.h library in Code::Blocks and created a basic graphics program. Graphics programming opens up a world of possibilities to create visually engaging applications and games. Understanding graphics libraries and their setup is an essential skill for aspiring computer scientists and developers. With this lab, we have taken the first step towards mastering graphics programming in C/C++ using Code::Blocks.

Experiment no: 02

Experiment name: Scan the conversion of a line using a direct line equation.

Introduction: Scan conversion is a fundamental technique in computer graphics used to convert geometric objects into pixel representations for display on a screen. Drawing a line is a common task in graphics programming, and the direct line equation method offers a straightforward approach to achieve this. The direct line equation method calculates the slope and y-intercept of the line, allowing us to draw the line pixel by pixel.

Algorithm: It is the simplest form of conversion. First of all scan P1 and P2 points. P1 has coordinates (x1',y1') and P2 has coordinates (x2' y2'). Then m = (y2' - y1')/(x2' - x1') and b = $y_1^1 + mx_1^1$, If value of |m|≤1 for each integer value of x. But do not consider x_1^1 and x_2^2, the If value of |m|>1 for each integer value of y. But do not consider y_1^1 and y_2^2.

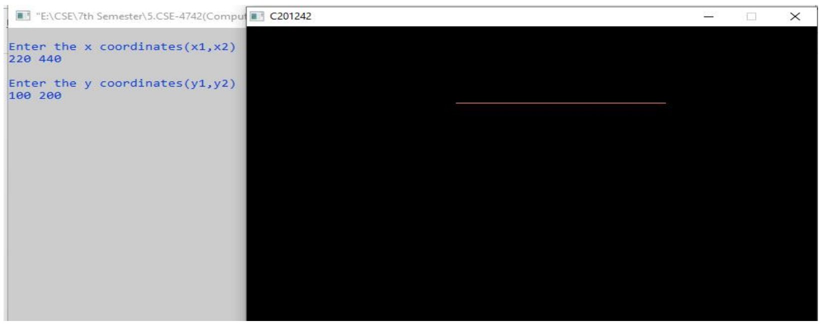
```
Step 1: Start Algorithm
Step 2: Declare variables x1, x2, y1, y2, dx, dy, m, b.
Step 3: Enter values of $x_1$, $x_2$, $y_1$, $y_2$.
        The ($x_1$, $y_1$) are coordinates of a starting point of the line.
        The ($x_2$, $y_2$) are coordinates of an ending point of the line.
Step 4: Calculate dx = $x_2$- $x_1$
Step 5: Calculate dy = $y_2$-$y_1$
Step 6: Calculate m = dy/dx
Step 7: Calculate b = $y_1$-m* $x_1$
Step 8: Set (x, y) equal to starting point, i.e., lowest point and $x_{end}$ equal to largest value of x.
     If dx < 0
        then x = $x_2$
             y = $y_2$
             $x_{end}$= $x_1$
     If dx > 0
        then x = $x_1$
             y = $y_1$
             $x_{end}$= $x_2$
Step 9: Check whether the complete line has been drawn if x=$x_{end}$, stop
Step 10: Plot a point at current (x, y) coordinates
Step 11: Increment value of x, i.e., x = x+1
Step 12: Compute next value of y from equation y = mx + b
Step 13: Go to Step9.
```

Code:

```
#include<bits/stdc++.h>
#include <graphics.h>
using namespace std;
main(){
   int x1,y1,x2,y2,x,y,dy,dx,b;
   float m;
   cout<<"\nEnter the x coordinates(x1,x2)\n";
   cin>>x1>>x2;
   cout<<"\nEnter the y coordinates(y1,y2)\n";
   cin>>y1>>y2;
   dx=x2-x1;
   dy=y2-y1;
```

```
 m=dy/dx;
   b=y1-(m*x1);
   x=x1;y=y1;
   int gd=DETECT,gm;
   initwindow(600,600,"C201242");
   if(m<1)
       while(x<=x2)
      {
           putpixel(x,int(y+0.5),12);
           x++;y=(m*x)+b;
       }
   else
   {
       while(x<=x1)
     {
           putpixel(int(x+0.5),y,8);
           y++;
           x=(y-b)/m;
     }
    }
   getch();
}
```

<u>**Input & Output:**</u>



```
Enter the x coordinates(x1,x2)
220 440

Enter the y coordinates(y1,y2)
100 200
```

<u>**Conclusion:**</u> The direct line equation method for scan converting lines in computer graphics offers a straightforward and efficient approach to rasterize lines, making it a valuable tool for rendering graphics on displays and screens.

Experiment no: 03

Experiment name: Implementation of Digital Difference Analyzer (DDA) algorithm.

Introduction: Scan conversion is the process of converting vector graphics into raster graphics, enabling them to be displayed on digital screens. The Digital Differential Analyzer (DDA) algorithm is a popular technique used for scan converting lines due to its simplicity and accuracy.

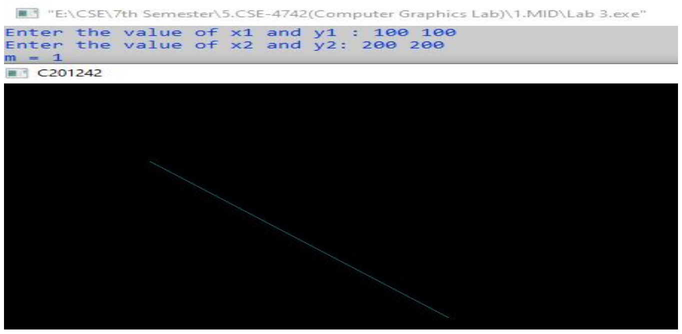
Algorithm:

Step1: Declare x1,y1,x2,y2,dx,dy,x,y as integer variables.
Step2: Enter value of x1,y1,x2,y2.
Step3: Calculate dx = x2-x1
Step4: Calculate dy = y2-y1
Step5: Calculate m = dy/dx, then steps =abs(dx)
Step6: assign x = x1 , assign y = y1
Step 7: Set pixel(x,y)
Step 8: *case-1)* m<1 then x++, y+m , *case-2)* m>1 then x+(1/m),y++ and *case-3)* m=1 then x++,y++
Step 9: Repeat step 9 until x = x2

Code:

```
#include <graphics.h>
#include <bits/stdc++.h>
using namespace std;
int main()
{
    float x, y, x1, y1, x2, y2, dx, dy, m, steps;
    int i, gd = DETECT, gm;
    initwindow(700, 700, "C201242");
    cout << "Enter the value of x1 and y1 : ";
    cin >> x1 >> y1;
    cout << "Enter the value of x2 and y2: ";
    cin >> x2 >> y2;
    dx = (x2 - x1);
    dy = (y2 - y1);
    m = dy/dx;
    cout << "m = " << m << endl;
    x = x1;
    y = y1;
    for (i = 0; i <= x2; i++)
    {
        if (m<1)
        {
            putpixel(x, y, 15);
            x = x + 1;
            y = y + m;
            delay(100);
        }
        else if(m>1)
        {
            putpixel(x, y, 4);
            x = x + (1/m);
            y = y + 1;
            delay(100);
        }
        else
        {
            putpixel(x, y, 3);
            x = x + 1;
            y = y + 1;
            delay(100);
        }
    }
    getch();
}
```

Input & Output:



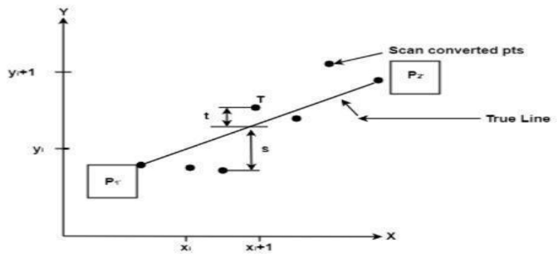
```
"E:\CSE\7th Semester\5.CSE-4742(Computer Graphics Lab)\1.MID\Lab 3.exe"
Enter the value of x1 and y1 : 100 100
Enter the value of x2 and y2: 200 200
m = 1
```

Conclusion: In this lab, we successfully implemented the scan conversion of lines using the Digital Differential Analyzer (DDA) algorithm. The DDA algorithm is an essential tool in computer graphics for rendering lines on digital displays. Despite its limitations in handling steep slopes, the algorithm remains a popular choice due to its simplicity and effectiveness.

Experiment no: 04

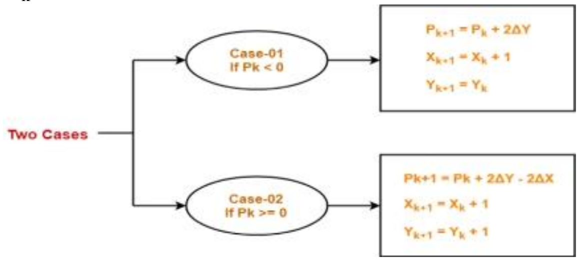
Experiment name: Implementation of Bresenham's Line Algorithm

Introduction: Bresenham's line algorithm is an algorithm that determines the points of an n-dimensional raster that should be selected in order to form a close approximation to a straight line between two points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction, and bit shifting, all of which are very cheap operations in standard computer architecture. These operations can be performed very rapidly so lines can be generated quickly.In this method, the next pixel selected is that one who has the least distance from true line. **The method works as follows:** Assume a pixel P1'(x1',y1'), then select subsequent pixels as we work our way to the night, one-pixel position at a time in the horizontal direction toward P2'(x2',y2'). Once a pixel is in choose at any step. The next pixel is 1. Either the one to its right (lower-bound for the line) 2. One top it's right and up (upper-bound for the line) The line is best approximated by those pixels that fall the least distance from the path between P1', P2'



Algorithm:

Given- Starting coordinates = (X_0, Y_0), Ending coordinates = (X_n, Y_n)
Step-01: $dX = X_n – X_0$, $dY = Y_n – Y_0$
Step-02: $P_k = 2dY – dX$
Step-03: Suppose the current point is (X_k, Y_k) and the next point is (X_{k+1}, Y_{k+1}).Find the next point depending on the value of the decision parameter P_k.



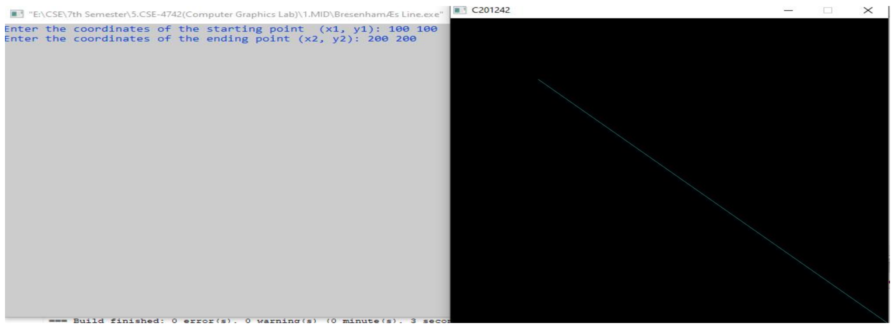
Step-04: Keep repeating Step-03 until the end point is reached or number of iterations equals to (ΔX-1) times.

Code:

```
#include <bits/stdc++.h>
#include <graphics.h>
using namespace std;
main()
{
   int gd=DETECT,gm;
   initwindow(500,500,"C201242");
   int x1, y1, x2, y2, x, y, dx, dy, m, p;
   cout << "Enter the coordinates of the starting point  (x1, y1): ";
   cin >> x1 >> y1;
   cout << "Enter the coordinates of the ending point (x2, y2): ";
   cin >> x2 >> y2;
   x=x1;
   y=y1;
   dx=x2-x1;
   dy=y2-y1;
   p = 2 * (dy) - (dx);
   while(x1 <= x2)
   {
      if(p < 0)
      {
         putpixel(x,y,5);
         x=x+1;
         y=y;
         p = p + 2 * (dy);
      }
      else
      {
         putpixel(x,y,3);
         x=x+1;
         y=y+1;
         p = p + 2 * (dy - dx);
```

```
        }
        delay(10);
    }
    getch();
}
```

Input & Output:



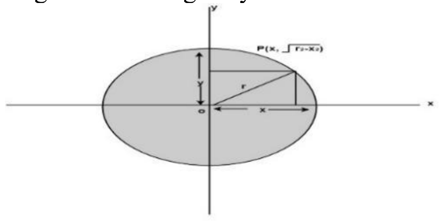
```
"E:\CSE\7th Semester\5.CSE-4742(Computer Graphics Lab)\1.MID\BresenhamÆs Line.exe"
Enter the coordinates of the starting point  (x1, y1): 100 100
Enter the coordinates of the ending point (x2, y2): 200 200

=== Build finished: 0 error(s), 0 warning(s) (0 minute(s), 3 secon
```

Conclusion: The implementation of Bresenham's Line Algorithm offers a highly efficient method for drawing lines in computer graphics, enabling precise and optimized rendering on digital displays.

Experiment no: 04

Experiment name: Implementation of Polynomial Circle Algorithm.

Introduction: The first method defines a circle with the second-order polynomial equation as shown in $y^2=r^2-x^2$ Where x = the x coordinate, y = the y coordinate and r = the circle radius. With the method, each x coordinate in the sector, from 90° to 45°, is found by stepping x from 0 to Defining a circle using Polynomial Method and each y coordinate is found by evaluating Defining a circle using Polynomial Method for each step of x.



Algorithm:

Step 1: Set the initial variables where r = circle radius, (h, k) = coordinates of circle center
x=o, I = step size, $x_{end}= r/\sqrt{2}$
Step 2: Test to determine whether the entire circle has been scan-converted.If $x > x_{end}$ then stop.
Step 3: Compute $y = \sqrt{r^2-x^2}$
Step4: Plot the eight points found by symmetry concerning the center (h, k) at the current (x, y) coordinates.

```
    Plot (x + h, y +k)         Plot (-x + h, -y + k)
    Plot (y + h, x + k)        Plot (-y + h, -x + k)
    Plot (-y + h, x + k)       Plot (y + h, -x + k)
    Plot (-x + h, y + k)       Plot (x + h, -y + k)
```

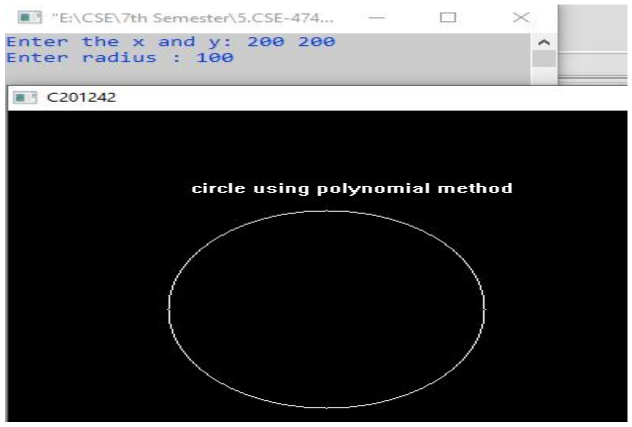
Step5: Increment x = x + i
Step6: Go to step 2

Code:

```
#include<bits/stdc++.h>
#include<graphics.h>
```

```
using namespace std;
void setPixel(int x, int y, int h, int k) {
   putpixel(x+h, y+k, WHITE);
   putpixel(x+h, -y+k, WHITE);
   putpixel(-x+h, -y+k, WHITE);
   putpixel(-x+h, y+k, WHITE);
   putpixel(y+h, x+k, WHITE);
   putpixel(y+h, -x+k, WHITE);
   putpixel(-y+h, -x+k, WHITE);
   putpixel(-y+h, x+k, WHITE);}
main(){
   int gd=0, gm,h,k,r;
   double x,y,x2;
   initwindow(500, 500, "C201242");
   cout<<"Enter the x and y: ";cin>>h>>k;
   cout<<"Enter radius : ";cin>>r;
   x=0,y=r;  x2 = r/sqrt(2);
   while(x<=x2) {
      y = sqrt(r*r - x*x);
      setPixel(floor(x), floor(y), h,k);
      outtextxy(115,70,"circle using polynomial method");
      x += 1;
    }
   getch();
}
```

Input & Output:



```
Enter the x and y: 200 200
Enter radius : 100
```

"E:\CSE\7th Semester\5.CSE-474...
C201242
circle using polynomial method

Conclusion: In conclusion, we successfully implemented the polynomial method to draw circles in computer graphics, highlighting its efficiency and adaptability for various applications.

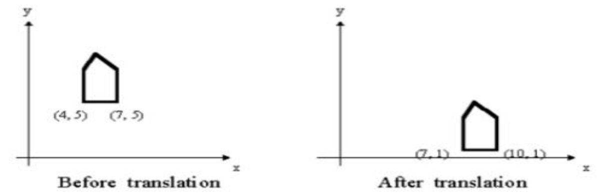
Experiment no: 06

Experiment name: Implementation of 2D transformation (Translation, Scaling, Rotation).

Introduction: We have to perform 2D transformations on 2D objects. Here we perform transformations on a line segment. The 2D transformations are:

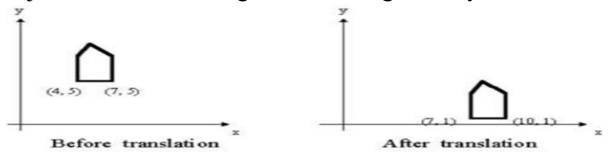
1. Translation
2. Scaling
3. Rotation

1. **Translation:** Translation is defined as moving the object from one position to another position along a straight line path.



   We can move the objects based on translation distances along x and y axis. tx denotes translation distance along x-axis and ty denotes translation distance along y axis.

2. **Scaling:** scaling refers to changing the size of the object either by increasing or decreasing. We will increase or decrease the size of the object based on scaling factors along x and y-axis.

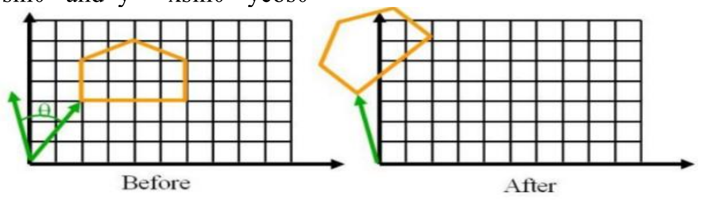


   If (x, y) are old coordinates of object, then new coordinates of object after applying scaling transformation are obtained as:

   - x'=x*sx and y'=y*sy

3. **Rotation:** A rotation repositions all points in an object along a circular path in the plane centered at the pivot point. We rotate an object by an angle theta. New coordinates after rotation depend on both x and y.

   - x' = xcosθ -y sinθ   and y' = xsinθ+ ycosθ



Algorithm:

```
1.  Initialize the graphics mode.
2.  Construct a 2D object  (use Drawpoly()) e.g. (x,y)
3.  A) Translation
      i.   Get the translation value tx, ty
      ii.  Move the 2d object with tx, ty (x'=x+tx,y'=y+ty)
      iii. Plot (x',y')
    B) Scaling
      i.   Get the scaling value Sx,Sy
      ii.  Resize the object with Sx,Sy  (x'=x*Sx,y'=y*Sy)
      iii. Plot (x',y')
    C) Rotation
      i.   Get the Rotation angle
      ii.  Rotate the object by the angle ɸ
        a.  x'=x cos ɸ -  y sin ɸ
        b.  y'=x sin ɸ  - y cosɸ
      iii. Plot (x',y')
```

Code:

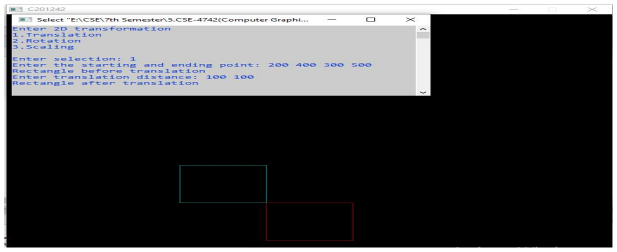
```
#include<bits/stdc++.h>
#include<graphics.h>
using namespace std;
int main()
{
```

```
int gd=DETECT,gm,s;
initwindow(700, 700, "C201242");
cout<<"Enter 2D transformation\n1.Translation\n2.Rotation\n3.Scaling\n"<<endl;
cout<<"Enter selection: ";
cin>>s;
switch(s)
{
case 1:
{
   int x1,y1,x2,y2;
   int tx,ty;
   cout<<"Enter the starting and ending point: ";
   cin>>x1>>y1>>x2>>y2;
   delay(10);
   cout<<"Rectangle before translation"<<endl;
   setcolor(3);
   rectangle(x1,y1,x2,y2);
   cout<<"Enter translation distance: ";
   cin>>tx>>ty;
   setcolor(4);
   cout<<"Rectangle after translation"<<endl;
   rectangle(x1+tx,y1+ty,x2+tx,y2+ty);
   getch();
   break;
}
case 2:
{
   int x1,y1,radius;
   double s,c, angle;
   setcolor(RED);
   cout<<"Enter coordinates of circle: ";
   cin>>x1>>y1;
   cout<<"Enter radius of circle: ";
   cin>>radius;
   cout<<"Circle before rotation"<<endl;
   circle(x1,y1,radius);
   cout<<"Enter rotation angle: ";
   cin>>angle;
   c = cos(angle *3.1416/180);
   s = sin(angle *3.1416/180);
   x1 = x1 * c - y1 *s;
   y1 = -x1 * s + y1 * c;
   setcolor(4);
   circle(x1,y1,radius);
   getch();
   break;
}
case 3:
{
   int x1,y1,x2,y2;//30 30 70 70
   int Sy,Sx;
   cout<<"Enter the starting and ending point: ";
   cin>>x1>>y1>>x2>>y2;
   cout<<"Before scaling"<<endl;
   setcolor(1);
   rectangle(x1,y1,x2,y2);
```

```
        delay(5);
        cout<<"Scaling factor: ";
        cin>>Sx>>Sy;
        cout<<"After scaling"<<endl;
        setcolor(10);
        rectangle(x1*Sx,y1*Sy,x2*Sx,y2*Sy);
        getch();
        break;
    }
    }
    getch();
}
```

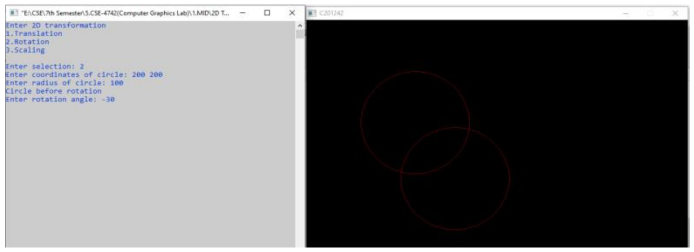
Input & Output:

Translation



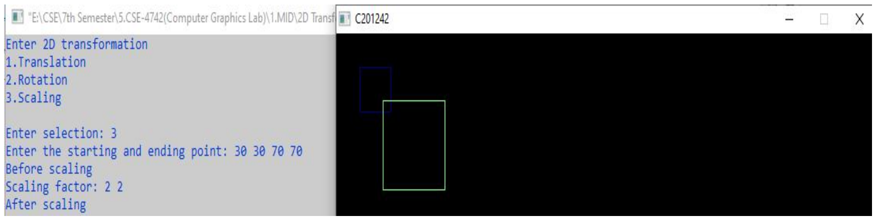
```
Enter 2D transformation
1.Translation
2.Rotation
3.Scaling

Enter selection: 1
Enter the starting and ending point: 200 400 300 500
Rectangle before translation
Enter translation distance: 100 100
Rectangle after translation
```

Rotation



```
Enter 2D transformation
1.Translation
2.Rotation
3.Scaling

Enter selection: 2
Enter coordinates of circle: 200 200
Enter radius of circle: 100
Circle before rotation
Enter rotation angle: -30
```

Scaling



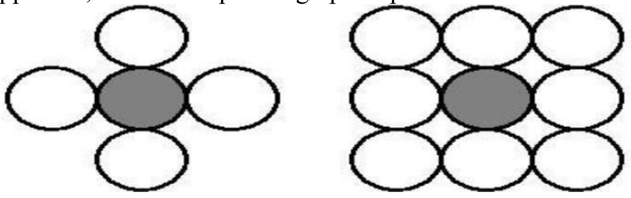
```
Enter 2D transformation
1.Translation
2.Rotation
3.Scaling

Enter selection: 3
Enter the starting and ending point: 30 30 70 70
Before scaling
Scaling factor: 2 2
After scaling
```

Conclusion: Our implementation of 2D transformations (translation, scaling, rotation) enhances the versatility of computer graphics, providing tools for manipulating and positioning objects with precision and creativity.

Experiment no: 07

Experiment name: Implementation of flood fill algorithm.

Introduction: Sometimes we want to fill in (recolor) an area that is not defined within a single color boundary. We paint such areas by replacing a specified interior color instead of searching for a boundary color value. This approach is called a flood-fill algorithm. 1. We start from a specified interior pixel (x, y) and reassign all pixel values that are currently set to a given interior color with the desired fill color. 2. If the area has more than one interior color, we can first reassign pixel values so that all interior pixels have the same color. 3. Using either 4-connected or 8-connected approach, we then step through pixel positions until all interior pixels have been repainted.



Algorithm:

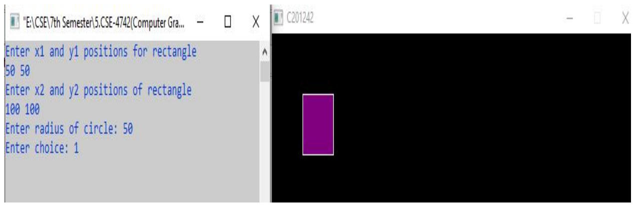
```
Procedure floodfill (x, y,fill_ color, old_color: integer)
If (getpixel (x, y)=old_color) {
setpixel (x, y, fill_color);
fill (x+1, y, fill_color, old_color);
fill (x-1, y, fill_color, old_color);
fill (x, y+1, fill_color, old_color);
fill (x, y-1, fill_color, old_color); }
}
```

Code:

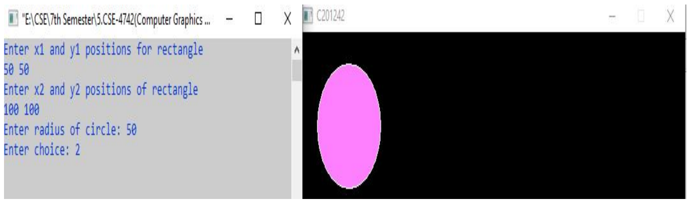
```
#include<bits/stdc++.h>
#include<graphics.h>
using namespace std;
void floodFill(int x,int y,int oldcolor,int newcolor) {
   if(getpixel(x,y) == oldcolor) {
      putpixel(x,y,newcolor);
      floodFill(x+1,y,oldcolor,newcolor);
      floodFill(x,y+1,oldcolor,newcolor);
      floodFill(x-1,y,oldcolor,newcolor);
      floodFill(x,y-1,oldcolor,newcolor); }}
void floodFill1(int x,int y,int oldcolor,int newcolor) {
   if(getpixel(x,y) == oldcolor) {
      putpixel(x,y,newcolor);
      floodFill(x+1,y,oldcolor,newcolor);
      floodFill(x,y+1,oldcolor,newcolor);
      floodFill(x-1,y,oldcolor,newcolor);
      floodFill(x,y-1,oldcolor,newcolor);
      floodFill(x-1,y-1,oldcolor,newcolor);
      floodFill(x+1,y-1,oldcolor,newcolor);
      floodFill(x-1,y+1,oldcolor,newcolor);
      floodFill(x+1,y+1,oldcolor,newcolor); }}
int main(){
   int gm,gd=DETECT,x2,y2,radius,ch;
   initwindow(600, 600, "C201242");
   int x1,y1,x,y;
   cout<<"Enter x1 and y1 positions for rectangle\n";cin>>x1>>y1;
   cout<<"Enter x2 and y2 positions of rectangle\n";cin>>x2>>y2;
   x=(x1+x2)/2;    y=(y1+y2)/2;
   cout<<"Enter radius of circle: ";cin>>radius;
   cout<<"Enter choice: ";cin>>ch;
   switch(ch) {
   case 1:
      rectangle(x1,y1,x2,y2);
      floodFill(x,y,0,5);
      break;
   case 2:
      circle(x,y,radius);
      floodFill1(x,y,0,13);
      break;}
   delay(5000);
   getch();}
```

Input & Output:

4-Connectd **8-Connected**



```
"E:\CSE\7th Semester\5.CSE-4742(Computer Gra...
Enter x1 and y1 positions for rectangle
50 50
Enter x2 and y2 positions of rectangle
100 100
Enter radius of circle: 50
Enter choice: 1
```



```
"E:\CSE\7th Semester\5.CSE-4742(Computer Graphics ...
Enter x1 and y1 positions for rectangle
50 50
Enter x2 and y2 positions of rectangle
100 100
Enter radius of circle: 50
Enter choice: 2
```

Conclusion: The flood fill algorithm is a robust and versatile tool for coloring closed areas in computer graphics, offering applications in image editing, design, and interactive simulations.

Experiment no: 08

Experiment name: Implementation of boundary fill algorithm.

Introduction: The boundary Fill Algorithm starts at a pixel inside the polygon to be filled and paints the interior proceeding outwards towards the boundary. This algorithm works only if the color with which the region has to be filled and the color of the boundary of the region are different. If the boundary is of one single color, this approach proceeds outwards pixel by pixel until it hits the boundary of the region. Introduction: Boundary Fill Algorithm starts at a pixel inside the polygon to be filled and paints the interior proceeding outwards towards the boundary. This algorithm works only if the color with which the region has to be filled and the color of the boundary of the region are different. If the boundary is of one single color, this approach proceeds outwards pixel by pixel until it hits the boundary of the region.

Algorithm:

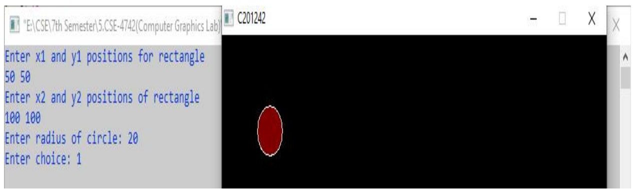
```
void boundaryFill(int x, int y, int fill_color,int boundary_color){
if(getpixel(x, y) != boundary_color &&getpixel(x, y) != fill_color){
putpixel(x, y, fill_color);
boundaryFill(x + 1, y, fill_color, boundary_color);
boundaryFill(x, y + 1, fill_color, boundary_color);
boundaryFill(x - 1, y, fill_color, boundary_color);
boundaryFill(x, y - 1, fill_color, boundary_color);}}
```

Code:

```
#include<bits/stdc++.h>
#include<graphics.h>
using namespace std;
void boundaryfill4(int x,int y,int f_color,int b_color) {
    if(getpixel(x,y)!=b_color && getpixel(x,y)!=f_color) {
        putpixel(x,y,f_color);
        boundaryfill4(x+1,y,f_color,b_color);
        boundaryfill4(x,y+1,f_color,b_color);
        boundaryfill4(x-1,y,f_color,b_color);
        boundaryfill4(x,y-1,f_color,b_color); }}
void boundaryFill8(int x, int y, int fill_color,int boundary_color) {
    if(getpixel(x, y) != boundary_color &&getpixel(x, y) != fill_color)    {
        putpixel(x, y, fill_color);
        boundaryFill8(x + 1, y, fill_color, boundary_color);
        boundaryFill8(x, y + 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y, fill_color, boundary_color);
        boundaryFill8(x, y - 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y - 1, fill_color, boundary_color);
        boundaryFill8(x - 1, y + 1, fill_color, boundary_color);
        boundaryFill8(x + 1, y - 1, fill_color, boundary_color);
        boundaryFill8(x + 1, y + 1, fill_color, boundary_color); }}
//getpixel(x,y) gives the color of specified pixel
int main(){
    int gm,gd=DETECT,x2,y2,radius,ch;
    initwindow(600, 600, "C201242");
    int x1,y1, x,y;
    cout<<"Enter x1 and y1 positions for rectangle\n";cin>>x1>>y1;
    cout<<"Enter x2 and y2 positions of rectangle\n";cin>>x2>>y2;
    x=(x1+x2)/2;
    y=(y1+y2)/2;
    cout<<"Enter radius of circle: ";cin>>radius;
    cout<<"Enter choice: ";cin>>ch;
    switch(ch) {
    case 1:
        circle(x,y,radius);
        boundaryfill4(x,y,4,15);
        break;
    case 2:
        rectangle(x1,y1,x2,y2);
        boundaryFill8(x,y,4,15);
        break;  }
    delay(1000);
    getch();}
```

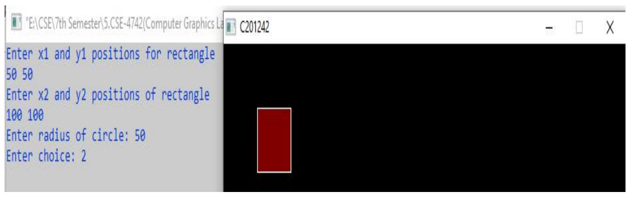
Input & Output:

4-Connected



```
Enter x1 and y1 positions for rectangle
50 50
Enter x2 and y2 positions of rectangle
100 100
Enter radius of circle: 20
Enter choice: 1
```

8-Connected



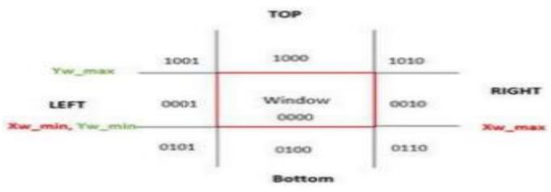
```
Enter x1 and y1 positions for rectangle
50 50
Enter x2 and y2 positions of rectangle
100 100
Enter radius of circle: 50
Enter choice: 2
```

Conclusion: The boundary fill algorithm is a reliable and efficient method for filling closed regions in computer graphics, with the potential for various applications in image processing and design.

Experiment no: 09

Experiment name: Implementation of Cohen Sutherland Line Clipping Algorithm.

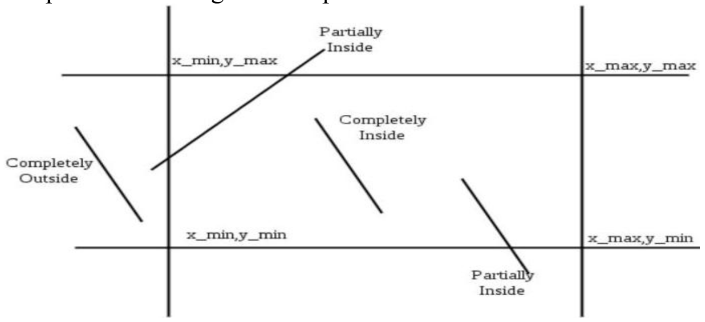
Introduction: Cohen-Sutherland algorithm divides a two-dimensional space into 9 regions and then efficiently determines the lines and portions of lines that are inside the given rectangular area. The central part is the viewing region or window, all the lines that lie within this region are completely visible. A region code is always assigned to the endpoints of the given line. Formula to check binary digits: - TBRL which can be defined as top, bottom, right, and left accordingly.



For finding clipping lines we have to assign the region codes to both endpoints.

1. Perform OR operation on both of these endpoints. if OR = 0000, then it is completely visible (inside the window).
2. Else Perform AND operation on both these endpoints.
3. Then, if AND ≠ 0000, then the line is invisible and not inside the window.
4. Also, it can't be considered for clipping. Otherwise AND = 0000, the line is partially inside the window and considered for clipping.

After confirming that the line is partially inside the window, then we find the intersection with the boundary of the window. By using the following formula :- Slope:- m= (y2-y1)/(x2-x1)
a) If the line passes through top or the line intersects with the top boundary of the window.
x = x + (y_wmax – y)/m , y = y_wmax
b) If the line passes through the bottom or the line intersects with the bottom boundary of the window.
x = x + (y_wmin – y)/m , y = y_wmin
c) If the line passes through the left region or the line intersects with the left boundary of the window.
y = y+ (x_wmin – x)*m , x = x_wmin
d) If the line passes through the right region or the line intersects with the right boundary of the window.
y = y + (x_wmax -x)*m , x = x_wmax
Now, overwrite the endpoints with a new one and update it. Repeat the step till your line doesn't get completely clipped.There are three possible cases for any given line.
1. Completely inside the given rectangle: Bitwise OR of the region of two endpoints of line is 0
(Both points are inside the rectangle)
2. Completely outside the given rectangle: Both endpoints share at least one outside region which implies that the line does not cross the visible region. (bitwise AND of endpoints != 0).
1. Partially inside the window: Both endpoints are in different regions. In this case, the algorithm finds one of the two points that is outside the rectangular region. The intersection of the line from outside point and rectangular window becomes new corner point and the algorithm repeats.



Algorithm:

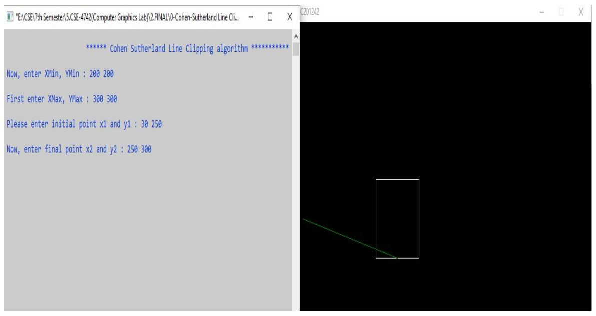
Step 1: Assign a region code for two endpoints of given line.
Step 2: If both endpoints have a region code 0000 then given line is completely inside.
Step 3: Else, perform the logical AND operation for both region codes.
Step 3.1: If the result is not 0000, then given line is completely outside.
Step 3.2: Else line is partially inside.
Step 3.2.1: Choose an endpoint of the line that is outside the given rectangle.
Step 3.2.2: Find the intersection point of the rectangular boundary (based on region code).
Step 3.2.3: Replace endpoint with the intersection point and update the region code.
Step 3.2.4: Repeat step 2 until we find a clipped line either trivially accepted or trivially rejected.
Step 4: Repeat step 1 for other lines.

Code:

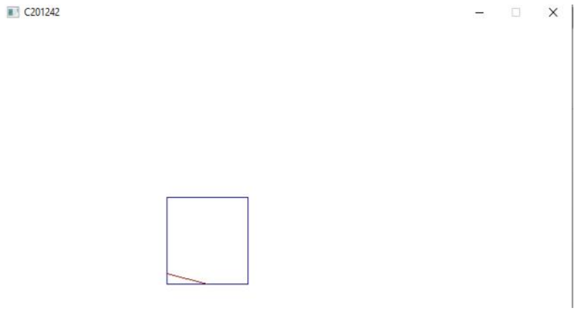
```
#include <bits/stdc++.h>
#include <graphics.h>
using namespace std;
// Define region codes
int INSIDE = 0; // 0000
int LEFT = 1;   // 0001
int RIGHT = 2;  // 0010
int BOTTOM = 4; // 0100
int TOP = 8;    // 1000
int X_MIN, X_MAX, Y_MIN, Y_MAX;
int main(){
   int gd = DETECT, gm;
   initwindow(700, 700, "C201242");
   setbkcolor(WHITE);
   cout<<"\n\t\t\t****** Cohen Sutherland Line Clipping algorithm ***********";
   cout<<"\n\n Now, enter XMin, YMin : ";cin>>X_MIN>>Y_MIN;
   cout<<"\n First enter XMax, YMax : ";cin>>X_MAX>>Y_MAX;
   setcolor(WHITE);
   rectangle(X_MIN, Y_MIN, X_MAX, Y_MAX);    // Draw the window
   int x1, y1, x2, y2;
   cout<<"\n Please enter initial point x1 and y1 : "; cin>>x1>>y1;
   cout<<"\n Now, enter final point x2 and y2 : ";cin>>x2>>y2;
   setcolor(GREEN);
   line(x1, y1, x2, y2);
   int code1 = 0, code2 = 0;
   bool accept = false;
   while (true) {
      code1 = 0; code2 = 0;
      if (x1 < X_MIN) code1 |= LEFT;
      if (x1 > X_MAX) code1 |= RIGHT;
      if (y1 < Y_MIN) code1 |= BOTTOM;
      if (y1 > Y_MAX) code1 |= TOP;
      if (x2 < X_MIN) code2 |= LEFT;
      if (x2 > X_MAX) code2 |= RIGHT;
      if (y2 < Y_MIN) code2 |= BOTTOM;
      if (y2 > Y_MAX) code2 |= TOP;
  if (!(code1 | code2)) // Both endpoints inside the window
{accept = true;
         break; }
      else if (code1 & code2) // Both endpoints outside the window
      { break; }
      else{
         int codeOut;
         if (code1 != 0) {  codeOut = code1; }
         else{ codeOut = code2; }
         int x, y;
         float slope=(float)(y2-y1)/(x2-x1);
         if (codeOut & TOP) {
            x = x1 + (Y_MAX - y1) / slope;
            y = Y_MAX; }
         else if (codeOut & BOTTOM) {
            x = x1 +  (Y_MIN - y1) / slope;
            y = Y_MIN; }
         else if (codeOut & RIGHT) {
            y = y1 + (X_MAX - x1) * slope;
            x = X_MAX; }
         else{
            y = y1 +  (X_MIN - x1) * slope;
            x = X_MIN; }
         if (codeOut == code1) {
            x1 = x;
            y1 = y; }
          else{
            x2 = x;
            y2 = y;
          }}}
    if (accept) {
        delay(1000);
        clearviewport();
        setcolor(BLUE);
        rectangle(X_MIN, Y_MIN, X_MAX, Y_MAX);
        setcolor(RED);
        line(x1,y1,x2,y2); }
    else{
        // Line is outside the window, do not draw it}
    getch();

}
}
```

Input & Output:

Before clipping **After clipping**



```
****** Cohen Sutherland Line Clipping algorithm ***********

Now, enter XMin, YMin : 200 200

First enter XMax, YMax : 300 300

Please enter initial point x1 and y1 : 30 250

Now, enter final point x2 and y2 : 250 300
```

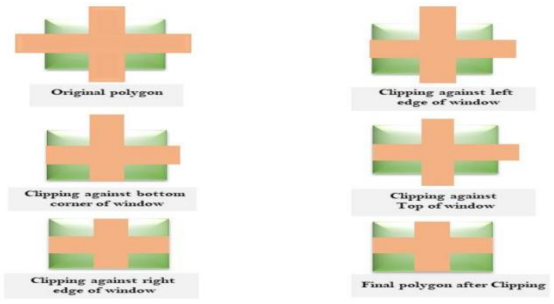


Conclusion: Our implementation of Cohen-Sutherland Line Clipping using graphics.h provides an effective solution for efficiently clipping lines, enhancing the precision and accuracy of computer graphics applications.

Experiment no: 10

Experiment name: Implementation of Sutherland Hodgeman polygon clipping algorithm

Introduction: It is performed by processing the boundary of the polygon against each window corner or edge. First of all entire polygon is clipped against one edge, then the resulting polygon is considered, then the polygon is considered against the second edge, so on for all four edges. **Four possible situations while processing**
1. If the first vertex is outside the window, the second vertex is inside the window. Then second vertex is added to the output list. The point of intersection of the window boundary and polygon side (edge) is also added to the output line.
2. If both vertexes are inside the window boundary. Then only a second vertex is added to the output list.
3. If the first vertex is inside the window and the second is an outside window. The edge that intersects with the window is added to output list.
4. If both vertices are the outside window, then nothing is added to the output list.
The following figures shows original polygon and clipping of polygon against four windows.


Original polygon
Clipping against left edge of window
Clipping against bottom corner of window
Clipping against Top of window
Clipping against right edge of window
Final polygon after Clipping

Algorithm: (Step)

1. Read the coordinates of all vertices of the polygon.
2. Read the coordinates of the clipping window
3. Consider the left edge of the window
4. Compare the vertices of each edge of the polygon, individually with the clipping plane
5. Save the resulting intersections and vertices in the new list of vertices according to four possible relationships between the edge and the clipping boundary discussed earlier.
6. Repeat the steps 4 and 5 for remaining edges of the clipping window. Each time the resultant list of vertices is successively passed to process the next edge of the clipping window.

Code:

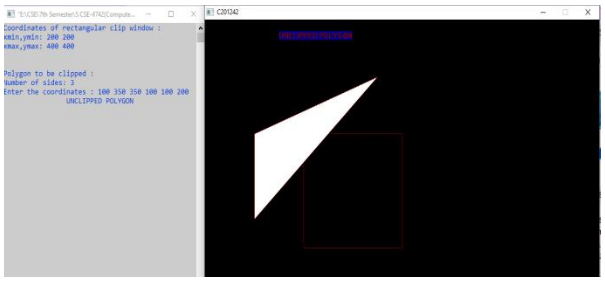
```
#include<bits/stdc++.h>
#include<graphics.h>
using namespace std;
int k;
float xmin,ymin,xmax,ymax,arr[20],m;
void left_clip(float x1,float y1,float x2,float y2){
   if(x2-x1)
      m = (y2-y1)/(x2-x1);
   if(x1 >= xmin && x2 >= xmin)//in-in {
      arr[k]=x2;
      arr[k+1]=y2;
      k+=2;}
   if(x1 < xmin && x2 >= xmin)//Out-in {
      arr[k]=xmin;
      arr[k+1]=y1+m*(xmin-x1);
      arr[k+2]=x2;
      arr[k+3]=y2;
      k+=4; }
   if(x1 >= xmin  && x2 < xmin)//in -out{
      arr[k]=xmin;
      arr[k+1]=y1+m*(xmin-x1);
      k+=2;}}
void top_clip(float x1,float y1,float x2,float y2){
   if(y2-y1)
      m=(x2-x1)/(y2-y1);
   if(y1 <= ymax && y2 <= ymax){
      arr[k]=x2;
      arr[k+1]=y2;
      k+=2; }
   if(y1 > ymax && y2 <= ymax){
      arr[k]=x1+m*(ymax-y1);
```

```
        arr[k+1]=ymax;
        arr[k+2]=x2;
        arr[k+3]=y2;
        k+=4; }
    if(y1 <= ymax  && y2 > ymax) {
        arr[k]=x1+m*(ymax-y1);
        arr[k+1]=ymax;
        k+=2;  }}
void right_clip(float x1,float y1,float x2,float y2){
    if(x2-x1)
        m=(y2-y1)/(x2-x1);
    if(x1 <= xmax && x2 <= xmax){
        arr[k]=x2;
        arr[k+1]=y2;
        k+=2;}
    if(x1 > xmax && x2 <= xmax) {
        arr[k]=xmax;
        arr[k+1]=y1+m*(xmax-x1);
        arr[k+2]=x2;
        arr[k+3]=y2;
        k+=4;}
    if(x1 <= xmax  && x2 > xmax)
    {
        arr[k]=xmax;
        arr[k+1]=y1+m*(xmax-x1);
        k+=2;
    }
}

void bottom_clip(float x1,float y1,float x2,float y2)
{
    if(y2-y1)
        m=(x2-x1)/(y2-y1);
    if(y1 >= ymin && y2 >= ymin)
    {
        arr[k]=x2;
        arr[k+1]=y2;
        k+=2;
    }
    if(y1 < ymin && y2 >= ymin)
    {
        arr[k]=x1+m*(ymin-y1);
        arr[k+1]=ymin;
        arr[k+2]=x2;
        arr[k+3]=y2;
        k+=4;
    }
    if(y1 >= ymin  && y2 < ymin)
    {
        arr[k]=x1+m*(ymin-y1);
        arr[k+1]=ymin;
        k+=2;
    }
}

int main()
```

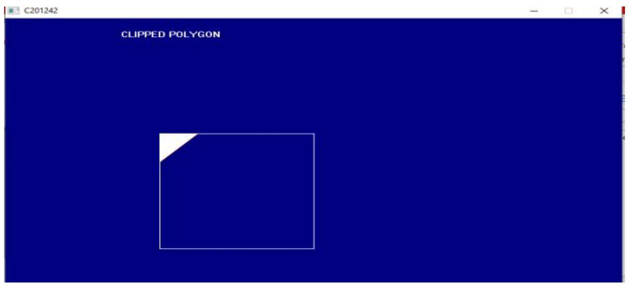
```
{
   initwindow(800,800,"C201242");
   setbkcolor(BLUE);
   int n,poly[20],i;
   float polyy[20];
   cout<<"Coordinates of rectangular clip window :\nxmin,ymin: ";
   cin>>xmin>>ymin;
   cout<<"xmax,ymax: ";
   cin>>xmax>>ymax;
   setcolor(RED);
   rectangle(xmin,ymin,xmax,ymax);
   cout<<"\n\nPolygon to be clipped :\nNumber of sides: ";
   cin>>n;
   cout<<"Enter the coordinates : ";
   for(i=0;i < 2*n;i++)
                   cin>>polyy[i];
   polyy[i]=polyy[0];
   polyy[i+1]=polyy[1];
   for(i=0;i < 2*n+2;i++)
                   poly[i]=round(polyy[i]);
   outtextxy(150,20,"UNCLIPPED POLYGON");
   cout<<"\t\tUNCLIPPED POLYGON";
   fillpoly(n,poly);
           getch();
   cleardevice();
   k=0;
   for(i=0;i < 2*n;i+=2)
                   left_clip(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
   n=k/2;
   for(i=0;i < k;i++)
                   polyy[i]=arr[i];
   polyy[i]=polyy[0];
   polyy[i+1]=polyy[1];
   k=0;
   for(i=0;i < 2*n;i+=2)
                   top_clip(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
   n=k/2;
   for(i=0;i < k;i++)
                   polyy[i]=arr[i];
   polyy[i]=polyy[0];
   polyy[i+1]=polyy[1];
   k=0;
   for(i=0;i < 2*n;i+=2)
                   right_clip(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
   n=k/2;
   for(i=0;i < k;i++)
                   polyy[i]=arr[i];
   polyy[i]=polyy[0];
   polyy[i+1]=polyy[1];
   k=0;
   for(i=0;i < 2*n;i+=2)
                   bottom_clip(polyy[i],polyy[i+1],polyy[i+2],polyy[i+3]);
   for(i=0;i < k;i++)
                   poly[i]=round(arr[i]);
   if(k)
                   fillpoly(k/2,poly);
```

```
    setcolor(WHITE);
    rectangle(xmin,ymin,xmax,ymax);
    outtextxy(150,20,"CLIPPED POLYGON");
    cout<<"\tCLIPPED POLYGON";
    getch();
}
```

<u>Input & Output:</u>



```
Coordinates of rectangular clip window :
xmin,ymin: 200 200
xmax,ymax: 400 400


Polygon to be clipped :
Number of sides: 3
Enter the coordinates : 100 350 350 100 100 200
                UNCLIPPED POLYGON
```



<u>Conclusion:</u> Our Sutherland-Hodgman polygon clipping implementation using graphics.h demonstrates the algorithm's utility in efficiently removing portions of polygons, facilitating complex shape rendering in computer graphics applications.