Experiment- 6

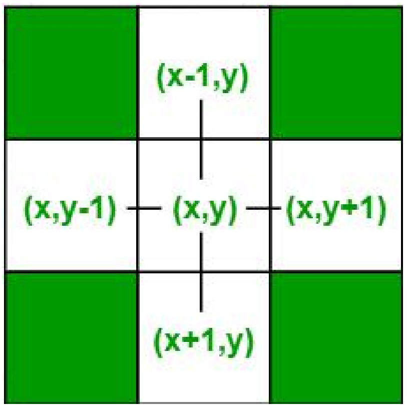
Aim

To implement Boundary-Fill Colouring Algorithm.

Description/Theory

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 colour with which the region has to be filled and the colour of the boundary of the region are different. If the boundary is of one single colour, this approach proceeds outwards pixel by pixel until it hits the boundary of the region.

Boundary Fill Algorithm is recursive in nature. It takes an interior point(x, y), a fill colour, and a boundary colour as the input. The algorithm starts by checking the colour of (x, y). If it’s colour is not equal to the fill colour and the boundary colour, then it is painted with the fill colour and the function is called for all the neighbours of (x, y). If a point is found to be of fill colour or of boundary colour, the function does not call its neighbours and returns. This process continues until all points up to the boundary colour for the region have been tested.
The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.



Algorithm

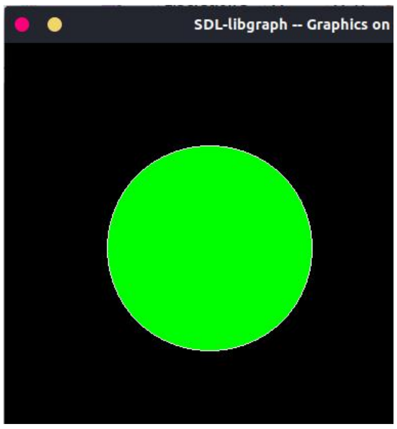
The algorithm for Boundary Fill Algorithm is given below:

```
void boundaryFill4(int x, int y, int fill_colour,int boundary_colour)
{
   if(getpixel(x, y) != boundary_colour &&
     getpixel(x, y) != fill_colour)
   {
      putpixel(x, y, fill_colour);
      boundaryFill4(x + 1, y, fill_colour, boundary_colour);
      boundaryFill4(x, y + 1, fill_colour, boundary_colour);
      boundaryFill4(x - 1, y, fill_colour, boundary_colour);
      boundaryFill4(x, y - 1, fill_colour, boundary_colour);
   }
}
```

Code

```
#include <graphics.h>
void boundaryFill4(int x, int y, int fill_colour,int boundary_colour)
{
   if(getpixel(x, y) != boundary_colour &&
     getpixel(x, y) != fill_colour)
   {
      putpixel(x, y, fill_colour);
                  boundaryFill4(x + 1, y, fill_colour, boundary_colour);
      boundaryFill4(x, y + 1, fill_colour, boundary_colour);
      boundaryFill4(x - 1, y, fill_colour, boundary_colour);
      boundaryFill4(x, y - 1, fill_colour, boundary_colour);
   }
}

int main()
{
   int gd = DETECT, gm;
   initgraph(&gd, &gm, "");
   int x = 250, y = 200, radius = 100;
   circle(x, y, radius);
   boundaryFill4(x, y, 4, 15);
   delay(10000);
   getch();
   closegraph();
   return 0;
}
```

Result



Discussion

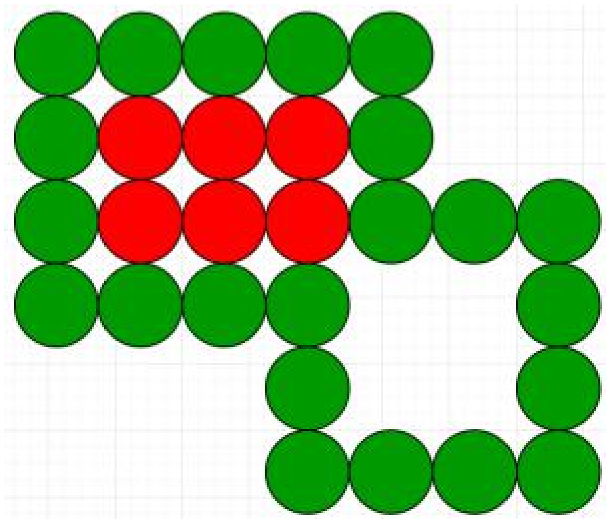
The boundary fill algorithm works as its name. This algorithm picks a point inside an object and starts to fill until it hits the boundary of the object. The colour of the boundary and the colour that we fill should be different for this algorithm to work.
In this algorithm, we assume that colour of the boundary is same for the entire object. The boundary fill algorithm can be implemented by 4-connected pixels or 8-connected pixels.

For, 4-connected structures:-
4-connected pixels are used. We are putting the pixels above, below, to the right, and to the left side of the current pixels and this process will continue until we find a boundary with different colour.
For 8-connected structures:-

8-connected pixels are used. We are putting pixels above, below, right and left side of the current pixels as we were doing in 4-connected technique.
In addition to this, we are also putting pixels in diagonals so that entire area of the current pixel is covered. This process will continue until we find a boundary with different colour.

Finding and Learning

4-connected pixels Vs 8-connected pixels :
Let us take a figure with the boundary colour as GREEN and the fill colour as RED. The 4-connected method fails to fill this figure completely. This figure will be efficiently filled using the 8-connected technique.



Flood-Fill Vs Boundary-Fill :
Though both Flood-Fill and Boundary-Fill algorithms colour a given figure with a chosen colour, they differ in one aspect. In Flood fill, all the connected pixels of a selected colour get replaced by a fill colour. On the other hand, in Boundary fill, the program stops when a given colour boundary is found.

Experiment- 7

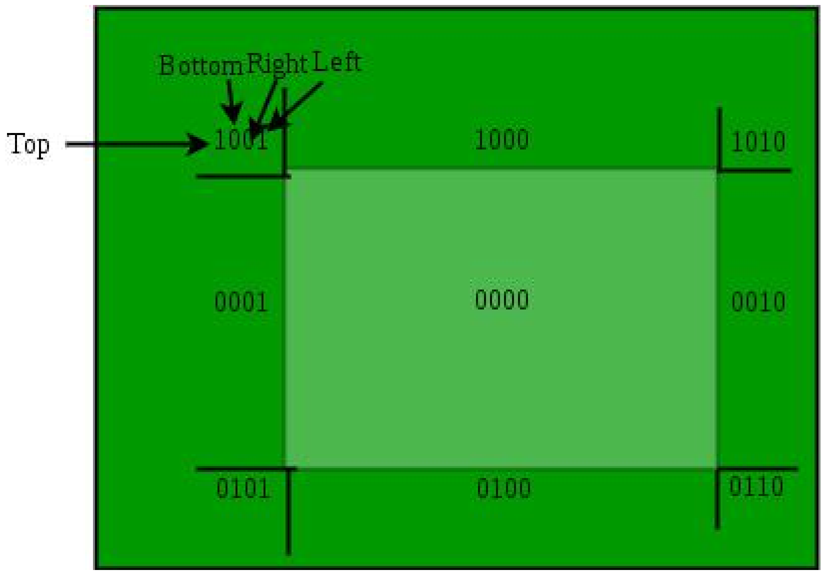
Aim

To implement Cohen-Sutherland Line Clipping Algorithm on a given line.

Description/Theory

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.
There are three possible cases for any given line.

1. **Completely inside the given rectangle :** Bitwise OR of region of two end points 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).
3. **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

The algorithm Cohen-Sutherland Line Clipping is given below:

```
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.
```

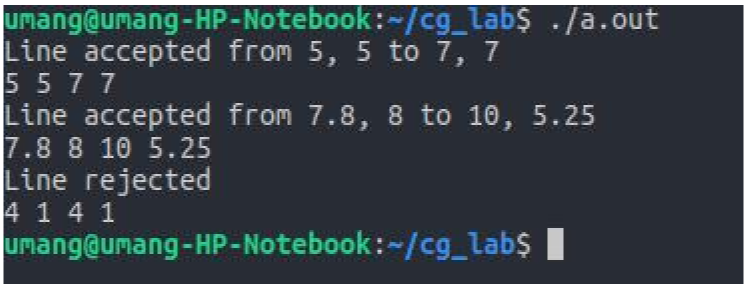
Step 4 : Repeat step 1 for other lines

Code

```
#include <iostream>
using namespace std;
const int INSIDE = 0; //  0000
const int LEFT = 1;   // 0001
const int RIGHT = 2;  // 0010
const int BOTTOM = 4; // 0100
const int TOP = 8;    // 1000
const int x_max = 10;
const int y_max = 8;
const int x_min = 4;
const int y_min = 4;

int computeCode(double x, double y)
{
   int code = INSIDE;
   if (x < x_min)
      code |= LEFT;
   else if (x > x_max)
      code |= RIGHT;
   if (y < y_min)
      code |= BOTTOM;
   else if (y > y_max)
      code |= TOP;
   return code;
}
void cohenSutherlandClip(double x1, double y1, double x2, double y2)
{
   int code1 = computeCode(x1, y1);
   int code2 = computeCode(x2, y2);

   bool accept = false;

   while (true)
   {
      if ((code1 == 0) && (code2 == 0))
      {
         accept = true;
         break;
      }
      else if (code1 & code2)
      {
         break;
      }
      else
      {
         int code_out;
         double x, y;
               if (code1 != 0)
            code_out = code1;
         else
            code_out = code2;
         if (code_out & TOP)
         {
            x = x1 + (x2 - x1) * (y_max - y1) / (y2 - y1);
            y = y_max;
         }
         else if (code_out & BOTTOM)
         {
            x = x1 + (x2 - x1) * (y_min - y1) / (y2 - y1);
            y = y_min;
         }
         else if (code_out & RIGHT)
         {
            y = y1 + (y2 - y1) * (x_max - x1) / (x2 - x1);
            x = x_max;
         }
         else if (code_out & LEFT)
         {
            y = y1 + (y2 - y1) * (x_min - x1) / (x2 - x1);
            x = x_min;
         }
         if (code_out == code1)
         {
            x1 = x;
            y1 = y;
            code1 = computeCode(x1, y1);
         }
         else
         {
            x2 = x;
            y2 = y;
            code2 = computeCode(x2, y2);
         }
      }
   }
   if (accept)
   {
     cout <<"Line accepted from (" << x1 << ", "
          << y1 << ") to ("<< x2 << ", " << y2 << ")" << endl;     }
   else
      cout << "Line rejected" << endl;
}

int main()
{
   cout<<"Window parametres are:"<<endl;
   cout<<"x_max= "<<x_max<<", y_max= "<<y_max<<", x_min= "<<x_min<<", y_min=
"<<y_min<<endl;
   cout<<"End-point coordinates of the line are (5,5) and (7,7)"<<endl;
   cohenSutherlandClip(5, 5, 7, 7);
   cout<<"End-point coordinates of the line are (7,9) and (11,4)"<<endl;
   cohenSutherlandClip(7, 9, 11, 4);
   cout<<"End-point coordinates of the line are (1,5) and (4,1)"<<endl;
   cohenSutherlandClip(1, 5, 4, 1);
   return 0;
```

}

Result



```
umang@umang-HP-Notebook:~/cg_lab$ ./a.out
Line accepted from 5, 5 to 7, 7
5 5 7 7
Line accepted from 7.8, 8 to 10, 5.25
7.8 8 10 5.25
Line rejected
4 1 4 1
umang@umang-HP-Notebook:~/cg_lab$
```

Discussion, Finding and Learning

Once the codes for each endpoint of a line are determined, the logical **AND** operation of the codes determines if the line is completely outside of the window. If the logical AND of the endpoint codes is **not zero**, the line can be trivially rejected. For example, if an endpoint had a code of 1001 while the other endpoint had a code of 1010, the logical AND would be 1000 which indicates the line segment lies outside of the window. On the other hand, if the endpoints had codes of 1001 and 0110, the logical AND would be 0000, and the line could not be trivially rejected.

The logical **OR** of the endpoint codes determines if the line is completely inside the window. If the logical OR is **zero**, the line can be trivially accepted. For example, if the endpoint codes are 0000 and 0000, the logical OR is 0000 - the line can be trivially accepted. If the endpoint codes are 0000 and 0110, the logical OR is 0110 and the line can’t be trivially accepted.