DDA Line

Aim

Write a program to draw a line program is to draw a line between two specified points using the DDA Line algorithm.

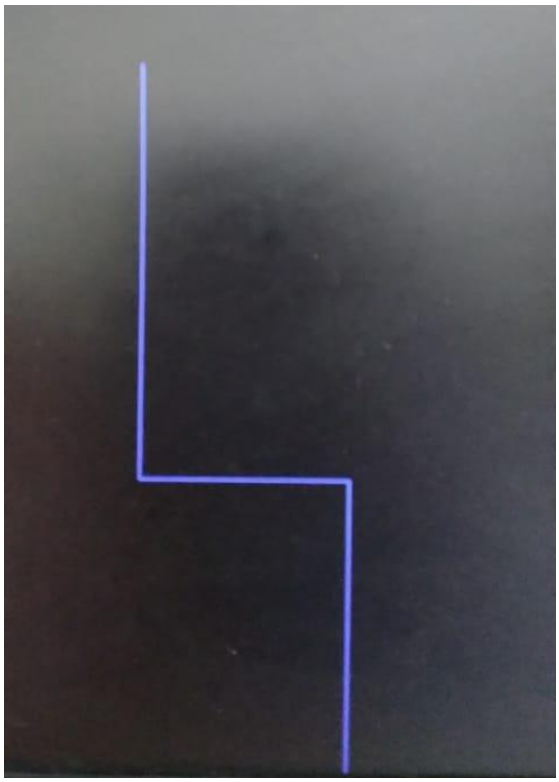
Algorithm

1. Accept the starting and ending coordinates of the line: (x1, y1) and (x2, y2).
2. Calculate the differences in x and y coordinates:
    - dx = x2 - x1
    - dy = y2 - y1
3. Determine the number of steps required to reach from (x1, y1) to (x2, y2). You can take the maximum absolute difference between dx and dy as the number of steps:
    - steps = max(abs(dx), abs(dy))
4. Calculate the increment values for each step:
    - xIncrement = dx / steps
    - yIncrement = dy / steps
5. Initialize the current coordinates as (x1, y1):
    - x = x1
    - y = y1
6. Perform the following steps steps number of times:
    - Plot the current pixel at (x, y).
    - Update the current coordinates:
        - x = x + xIncrement
        - y = y + yIncrement
7. End.

Source Code

```
#include <graphics.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
 void main()
 {
 int x1,y1,x2,y2,i,step,xn,yn,dx,dy;
 int gdriver=DETECT,gmode;
  clrscr();
  initgraph(&gdriver,&gmode,"c:\\turboc3\\bgi");
  printf("enter the value of x1 and y1 : ");
  scanf("%d%d",&x1,&y1);
  printf("enter the value of x2 and y2 : ");
  scanf("%d%d",&x2,&y2);
  dx=abs(x2-x1);
  dy=abs(y2-y1);
  if(dx>=dy)
  {
  step=dx;
  }
  else
  {
  step=dy;
  } xn=dx/step;
  yn=dy/step;
  for(i=1;i<=step;i++)
  {
  putpixel(x1,y1,BLUE);
  delay(100);
  x1=x1+xn;
  y1=y1+yn;
  }
  getch();
  closegraph();
```

Output



Conclusion

The DDA line program provides a simple and efficient method for approximating straight lines on a raster display.

Bresenham Line

Aim

Write a program to draw a line program is to draw a line between two specified points using the DDA Line algorithm.

Algorithm

1. Accept the starting and ending coordinates of the line: (x1, y1) and (x2, y2).
2. Calculate the differences in x and y coordinates:
    - dx = abs(x2 - x1)
    - dy = abs(y2 - y1)
3. Determine the signs for the increments in x and y coordinates:
    - signX = sign(x2 - x1)
    - signY = sign(y2 - y1)
4. Initialize the current coordinates as (x1, y1):
    - x = x1
    - y = y1
5. Determine the primary axis (whether the line is more steep in x or y direction) by comparing dx and dy:
    - If dx > dy, set primary = "x" and secondary = "y".
    - Else, set primary = "y" and secondary = "x".
6. Calculate the decision parameter p based on the primary axis:
    - If primary == "x", then p = 2 * dy - dx.
    - If primary == "y", then p = 2 * dx - dy.
7. Perform the following steps max(dx, dy) number of times:
    - Plot the current pixel at (x, y).
    - Update the current coordinates based on the primary axis:
        - x = x + signX
        - y = y + signY
    - If primary == "x", update the decision parameter p:

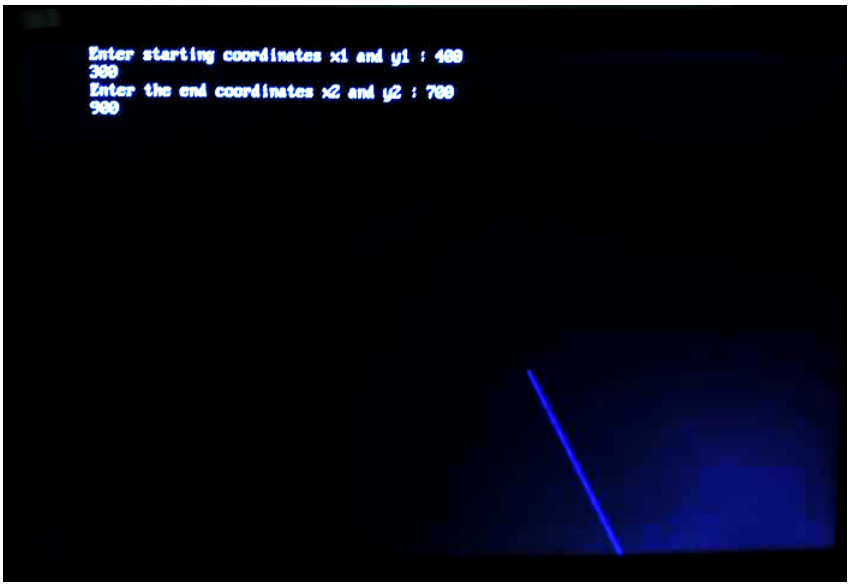
        - If p < 0, then p = p + 2 * dy.
        - If p >= 0, then p = p + 2 * dy - 2 * dx.
    - If primary == "y", update the decision parameter p:
        - If p < 0, then p = p + 2 * dx.
        - If p >= 0, then p = p + 2 * dx - 2 * dy.
8. End.

Source Code

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
#include<stdlib.h>
void main()
{
int gdriver = DETECT,gmode;
int x1,x2,y1,y2,dx,dy,pc,pn,xk,yk;
float m;
initgraph(&gdriver,&gmode,"c:\\turboc3\\bgi");
printf("Enter starting coordinates x1 and y1 : ");
scanf("%d%d",&x1,&y1);
printf("Enter the end coordinates x2 and y2 : ");
scanf("%d%d",&x2,&y2);
xk=x1;
yk=y1;
dx=x2-x1;
dy=y2-y1;
m=(dy/dx);
if(m<1)
{
pc=(2*dy)-dx;
while(xk<=x2){
putpixel(xk,yk,BLUE);
if(pc<0)
{
xk+=1;
```

```
pn=pc+(2*dy);
}
else
{
xk+=1;
yk+=1;
pn=pc+(2*dy)-(2*dx);
}
pc=pn;
}
}
else
{
pc=(2*dx)-dy;
while(yk<=y2)
{
putpixel(xk,yk,BLUE);
if(pc<0)
{
yk+=1;
pn=pc+(2*dx);
}
else
{
xk+=1;
yk+=1;
pn=pc+(2*dx)-(2*dy);
}
pc=pn;
}
}
getch();
closegraph();
}
```

Output



Conclusion

The Bresenham Circle algorithm effectively approximates circles on a computer screen, providing accurate and visually pleasing results with reduced computational requirements.

Mid point Circle

Aim

Write a program in C to draw a circle on the display using the Mid point Circle Algorithm.

Algorithm

1. Accept the center coordinates of the circle: (centerX, centerY), and the radius.
2. Set the initial point on the circle as (0, radius).
3. Initialize the decision parameter as 1 - radius.
4. Repeat the following steps until x becomes greater than or equal to y:
    - Plot the eight symmetric points of the circle centered at (centerX, centerY):
        - Plot (centerX + x, centerY + y)
        - Plot (centerX - x, centerY + y)
        - Plot (centerX + x, centerY - y)
        - Plot (centerX - x, centerY - y)
        - Plot (centerX + y, centerY + x)
        - Plot (centerX - y, centerY + x)
        - Plot (centerX + y, centerY - x)
        - Plot (centerX - y, centerY - x)
    - Increment x by 1.
    - If the decision parameter is less than 0, update it as decision += 2 * x + 1.
    - If the decision parameter is greater than or equal to 0, update it as decision += 2 * (x - y) + 1 and decrement y by 1.
5. End.

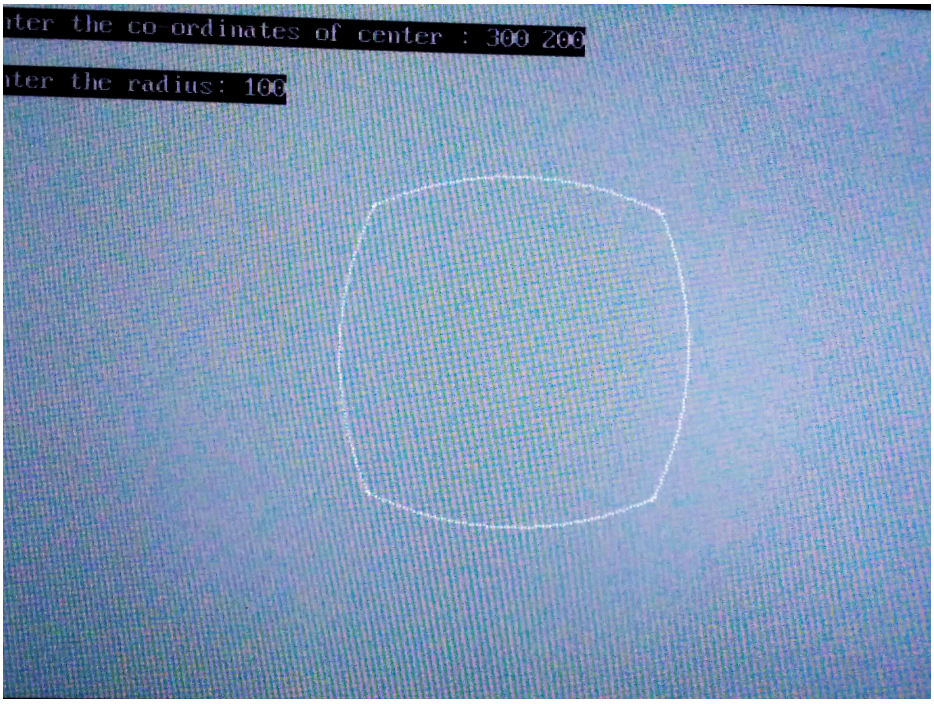
Source Code

```
#include <stdio.h>
```

```
#include <conio.h>
#include <graphics.h>
void main()
{
int xc,yc,r,p,x,y;
int gd,gm;
detectgraph(&gd,&gm);
initgraph(&gdriver,&gmode,"c:\\turboc3\\bgi");
clrscr();
printf("\nEnter the co-ordinates of center : ");
scanf("%d %d",&xc,&yc);
printf("\nEnter the radius: ");
scanf("%d",&r);

x = 0;
y = r;
p=(1-r);

for(x=0;x<=y;x++)
{
if(p < 0)
{
x=x+1;
y=y;
p=p+2*(x+1)+1;
}
else
{
x=x+1;
y=y-1;
p = p -2*(y+1)+2*(x+1)+1;
}

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

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

getch();
closegraph();
}
```

Output



Conclusion

The Midpoint Circle algorithm successfully generates accurate circles on a computer screen with minimal computational overhead, allowing for smooth and precise circular shapes.

Bresenham Circle

Aim

Write a program in C to draw a circle on the display using the Bresenham Circle Algorithm.

Algorithm

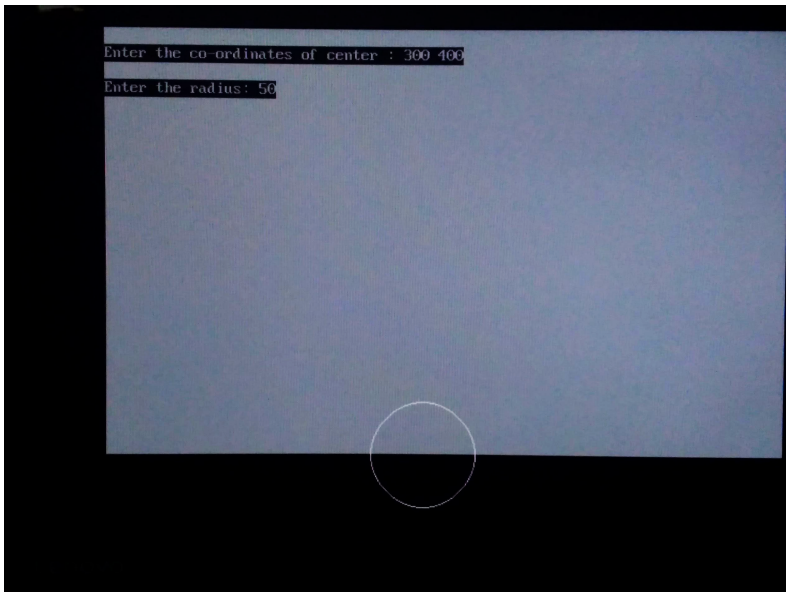
- Accept the center coordinates of the circle: (centerX, centerY), and the radius.
- Set the initial point on the circle as (0, radius).
- Initialize the decision parameter as 3 - 2 * radius.
- Repeat the following steps until x becomes greater than or equal to y:
    - Plot the eight symmetric points of the circle centered at (centerX, centerY):
        - Plot (centerX + x, centerY + y)
        - Plot (centerX - x, centerY + y)
        - Plot (centerX + x, centerY - y)
        - Plot (centerX - x, centerY - y)
        - Plot (centerX + y, centerY + x)
        - Plot (centerX - y, centerY + x)
        - Plot (centerX + y, centerY - x)
        - Plot (centerX - y, centerY - x)
    - Increment x by 1.
    - If the decision parameter is less than 0, update it as decision += 4 * x + 6.
    - If the decision parameter is greater than or equal to 0, update it as decision += 4 * (x - y) + 10 and decrement y by 1.
- End.

Source Code

```
# include <stdio.h>
# include <conio.h>
# include <graphics.h>

void main()
{
int xc,yc,r,p,x,y;
int gd,gm;
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"C:\\TurboC3\\BGI");
clrscr();
printf("\nEnter the co-ordinates of center : ");
scanf("%d %d",&xc,&yc);
printf("\nEnter the radius: ");whar
scanf("%d",&r);

x = 0;
y = r;
p=3-(2*r);

for(x=0;x<=y;x++)
{
if(p < 0)
{
p = p + (4 * x)+6;
}
else
{
y=y-1;
p = p +4 *(x-y)+10;
}

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

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

getch();
closegraph();
}
```

Output



Conclusion

The Bresenham Circle algorithm effectively approximates circles on a computer screen, providing accurate and visually pleasing results with reduced computational requirements

Boundary Fill

Aim

Write a program in C to fill a closed region with a specific color on a digital display by accurately determining the boundary pixels and filling the region using the Boundary Fill algorithm.

Algorithm

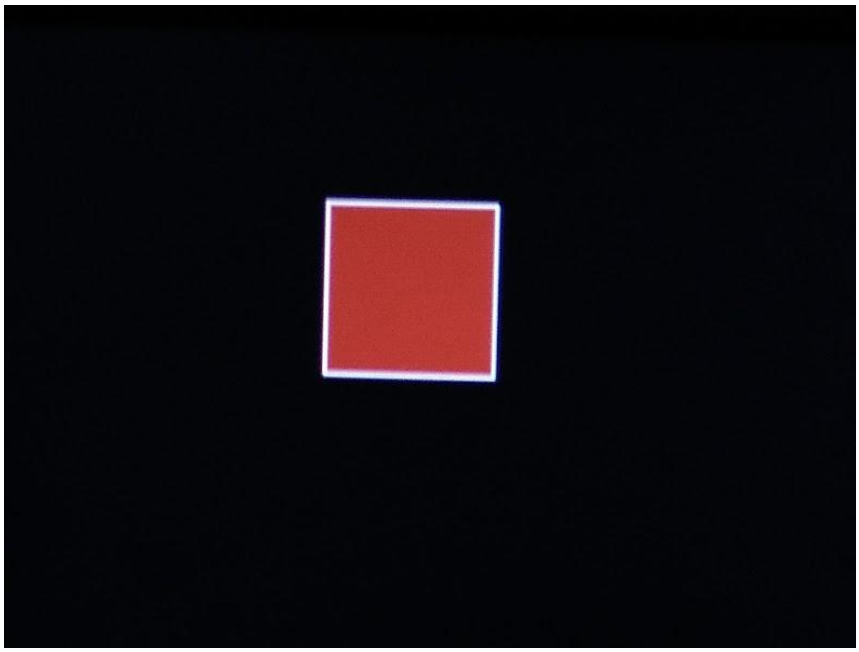
1. Accept the starting seed point (x, y) and the boundary color.
2. Check the color of the current pixel (x, y).
3. If the color of the current pixel is not the boundary color and is not the fill color:
    - Set the color of the current pixel to the fill color.
4. Recursively call the Boundary Fill algorithm on the neighboring pixels:
    - (x + 1, y)
    - (x - 1, y)
    - (x, y + 1)
    - (x, y - 1)
5. Note: Make sure to add suitable conditions to prevent infinite recursion or going out of the boundary of the shape.
6. End.

Source Code

```
#include<graphics.h>
#include<dos.h>
#include<conio.h>
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);
        boundaryFill(x - 1, y - 1, fill_color, boundary_color);
        boundaryFill(x - 1, y + 1, fill_color, boundary_color);
        boundaryFill(x + 1, y - 1, fill_color, boundary_color);
        boundaryFill(x + 1, y + 1, fill_color, boundary_color);
        }
}
void main()
{
        int gd = DETECT, gm;
        initgraph(&gd, &gm, "c:\\Turboc3\\bgi");
        rectangle(50, 50, 100, 100);
        boundaryFill(55, 55, 4, 15);
        delay(10000);
        getch();
        closegraph();
}
```

Output



Conclusion

The Boundary Fill algorithm successfully fills closed regions with the desired color on a computer screen, ensuring accurate boundary identification and efficient filling of the region.

Scan line

Aim

Write a program in C to fill a rectangle with a specific color on a digital display by scanning horizontal lines and identifying intersections with the polygon edges using Scan Line Algorithm.

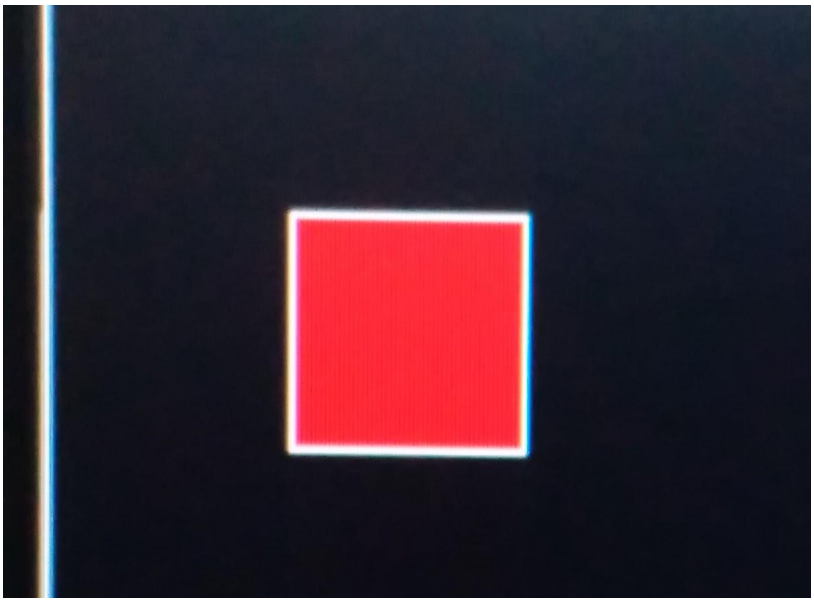
Algorithm

1. Draw the boundary shape using a specific color (boundary color).
2. Take the seed point (x, y) inside the boundary shape.
3. Read the fill color.
4. Call the scanLineFill function with the seed point, fill color, and boundary color.
5. In the scanLineFill function:
    - Check if the current pixel color at (x, y) is not the boundary color or the fill color.
    - If the condition is true, set the current pixel at (x, y) to the fill color.
    - Recursively call the scanLineFill function for the adjacent pixels (x+1, y), (x-1, y), (x, y+1), and (x, y-1).
6. Return from the scanLineFill function.
7. End

Source Code

```
#include <stdio.h>
#include<dos.h>
#include<conio.h>
#include <graphics.h>
void scanLineFill(int x, int y, int fillColor, int boundaryColor) {
   int currentColor;
   currentColor = getpixel(x, y);
   if (currentColor != boundaryColor && currentColor != fillColor) {
      // Set the current pixel to the fill color
      putpixel(x, y, fillColor);

      // Recursive calls for adjacent pixels
      scanLineFill(x + 1, y, fillColor, boundaryColor);
      scanLineFill(x - 1, y, fillColor, boundaryColor);
      scanLineFill(x, y + 1, fillColor, boundaryColor);
      scanLineFill(x, y - 1, fillColor, boundaryColor);
   }
}

void main() {
    int gd = DETECT, gm;
   initgraph(&gd, &gm, "c:\\Turboc3\\bgi");
   rectangle(50, 50, 100, 100);
   scanLineFill  (55, 55, 4, 15);
   delay(10000);
   getch();
   closegraph();
}
```

Output



Conclusion

The Scan Line algorithm effectively fills polygons with the desired color on a computer screen, providing accurate and visually appealing results by efficiently identifying intersections and filling the polygons.

Flood fill

Aim

Write a program in C to fill a closed region with a specific color on a digital display by recursively visiting neighboring pixels and filling the region using Flood Fill Algorithm.

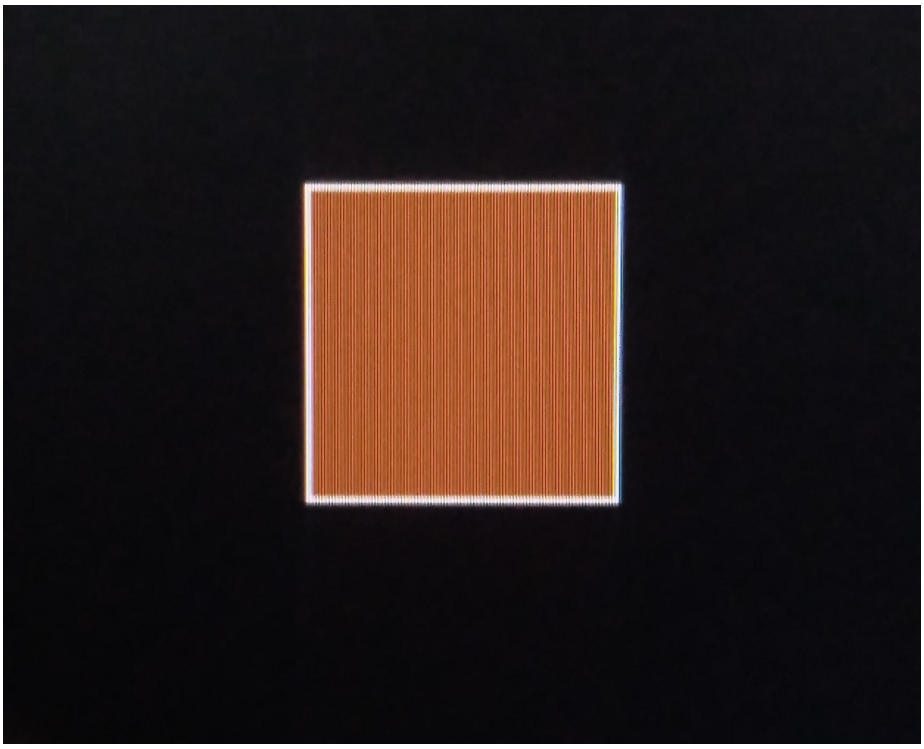
Algorithm

1. Create a function named floodFill() that takes the starting coordinates (x, y), the target color, and the replacement color as parameters.
2. Inside the function, check if the target color at the starting coordinates matches the replacement color. If they are the same, return.
3. Create a stack to store the coordinates of pixels to be processed.
4. Push the starting coordinates (x, y) onto the stack.
5. While the stack is not empty, do the following:
    - Pop the top coordinates (x, y) from the stack.
    - If the current pixel at (x, y) matches the target color, set it to the replacement color.
    - Check the neighboring pixels (up, down, left, right) of (x, y).
    - If a neighboring pixel has the target color, push its coordinates onto the stack.
6. Repeat step 5 until the stack becomes empty.
7. Return from the function.

Source Code

```
#include <graphics.h>
void floodFill(int x, int y, int fillColor, int boundaryColor) {
   int currentColor;
   currentColor = getpixel(x, y);
   if (currentColor != boundaryColor && currentColor != fillColor) {
      putpixel(x, y, fillColor);
      floodFill(x + 1, y, fillColor, boundaryColor);
      floodFill(x - 1, y, fillColor, boundaryColor);
      floodFill(x, y + 1, fillColor, boundaryColor);
      floodFill(x, y - 1, fillColor, boundaryColor);
   }
}

void main() {
    int gd = DETECT, gm;
   initgraph(&gd, &gm, "c:\\Turboc3\\bgi");
   rectangle(50, 50, 100, 100);
   floodFill(55, 55, 6, 15);
   delay(10000);
   getch();
   closegraph();
}
```

Output



Conclusion

The Flood Fill algorithm successfully fills closed regions with the desired color on a computer screen, ensuring accurate and efficient filling by recursively visiting neighboring pixels.

2D Transformation/Translation

Aim

Write a program in C to perform translation operations on 2D objects by shifting their positions on the display using 2D Translation Algorithm.

Algorithm

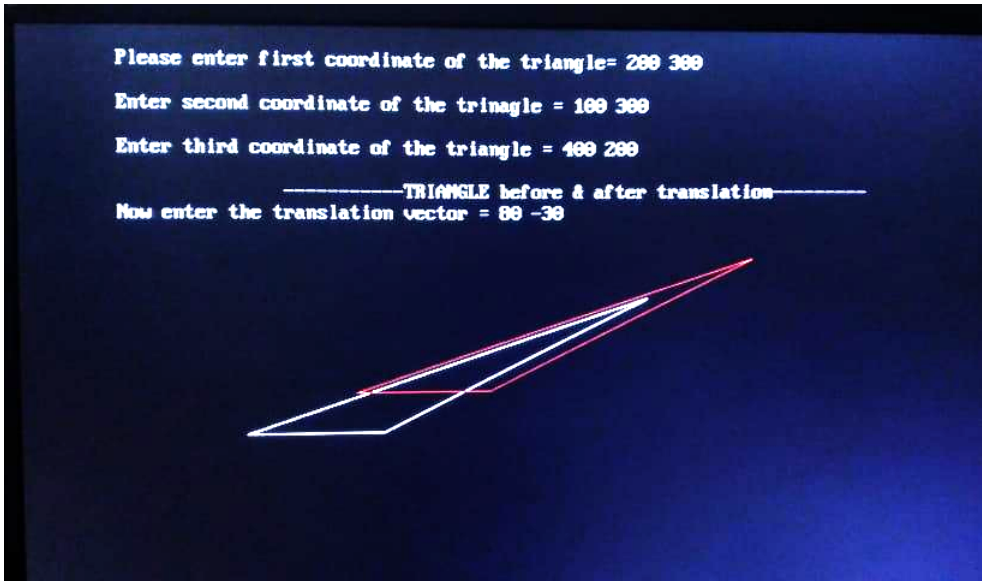
1. Accept the coordinates of the three vertices of the triangle: (x1, y1), (x2, y2), and (x3, y3).
2. Accept the translation vector: (tx, ty).
3. Translate each vertex of the triangle using the translation vector:
    a. Translate vertex 1: (x1 + tx, y1 + ty)
    b. Translate vertex 2: (x2 + tx, y2 + ty)
    c. Translate vertex 3: (x3 + tx, y3 + ty)
4. End.

Source Code

```
#include<conio.h>
#include<graphics.h>
#include<stdio.h>
void main()
{
int gd=DETECT,gm;
int x,y,x1,y1,x2,y2,tx,ty;
clrscr();
initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");
printf("\n Please enter first coordinate of the triangle= ");
scanf("%d %d", &x,&y);
printf("\n Enter second coordinate of the triangle = ");
scanf("%d %d",&x1,&y1);
```

```
printf("\n Enter third coordinate of the triangle = ");
scanf("%d %d",&x2,&y2);
printf("\n\t\t----------TRIANGLE before & after translation---------");
line(x,y,x1,y1);
line(x1,y1,x2,y2);
line(x2,y2,x,y);
printf("\n Now enter the translation vector = ");
scanf("%d %d",&tx,&ty);
setcolor(RED);
line(x+tx,y+ty,x1+tx,y1+ty);
line(x1+tx,y1+ty,x2+tx,y2+ty);
line(x2+tx,y2+ty,x+tx,y+ty);
getch();
closegraph();
}
```

Output



Conclusion

The 2D Translation algorithm effectively translates 2D objects on a computer screen, allowing for accurate repositioning and manipulation of graphical elements.