**DEERWALK INSTITUTE OF TECHNOLOGY**

**Tribhuvan University**

**Faculties of Computer Science**

**A logo of a sea creature

Description automatically generated**

**Bachelors of Science in Computer Science and Information Technology (BSc. CSIT)**

**Course: Computer Graphics (CSC214)**

**Year/Semester: II/III**

**A Lab report on:**

**Implementation of Filling Algorithms in C++**

Submitted by:

Name: Parth Poudyal

Roll: 1317

Submitted to:

Binod Sitaula

Department of Computer Science

**Theory**

Boundary Fill Algorithm:

The Boundary Fill algorithm works by starting from a point inside the polygon and then filling outward until a specified boundary color is encountered. This algorithm works for regions that are enclosed by a single color. The algorithm starts at a seed point (x, y) inside the region and colors all the connected pixels that are not of the boundary color.

There are two approaches to implementing the boundary fill algorithm:

1. **4-connected approach**: Each pixel is connected to 4 neighbors (left, right, top, bottom).
2. **8-connected approach**: Each pixel is connected to 8 neighbors (4 adjacent + 4 diagonal).

The steps for the boundary filling algorithm are as follows:

1. Start from the seed point (x, y).
2. Check if the current pixel is neither colored with the boundary color nor the fill color.
3. If the condition is true, color the current pixel with the fill color.
4. Recursively call the boundary fill function for the neighboring pixels.

The recursive implementation of the boundary fill algorithm can cause stack overflow for large regions, so iterative versions using a stack or queue are often preferred in practice.

Flood Fill Algorithm:

The Flood Fill algorithm is like the boundary fill algorithm, but instead of looking for a boundary color, it replaces a specific old color with a new fill color. This algorithm is suitable for filling a region consisting of similar colors.

The algorithm starts from a seed point (x, y) and colors all the connected pixels that have the same color as the seed pixel. Like Boundary Fill, Flood Fill can also be implemented using 4-connected or 8-connected approaches.

The steps for the flood fill algorithm are as follows:

1. Start from the seed point (x, y).
2. Check if the current pixel color is the same as the old color.
3. If the condition is true, color the current pixel with the new fill color.
4. Recursively call the flood fill function for the neighboring pixels.

Advantages and Disadvantages:

Boundary Fill:

* Advantages:
* Simple to implement • Works well for simple shapes
* Disadvantages:
* Stack overflow for large regions
* Only works for regions with a single boundary color Flood Fill:
* Advantages:
* Flexible for regions with similar colors
* Simple to implement
* Disadvantages:
* Stack overflow for large regions
* Not efficient for complex shapes

**Algorithm**

Boundary Fill Algorithm:

1. Start with a seed point (x, y) inside the boundary.
2. Check the color of the current pixel:
   1. If the pixel is not the boundary color and not the fill color, proceed.
   2. Otherwise, return (base case for recursion).
3. Set the pixel color to the fill color.
4. Recursively apply the algorithm to its neighboring pixels (4-connected or 8-connected).
   1. 4-connected: Move to (x+1, y), (x-1, y), (x, y+1), (x, y-1).
   2. 8-connected: Move to (x+1, y), (x-1, y), (x, y+1), (x, y-1), (x+1, y+1), (x-1, y+1), (x+1, y-1), (x-1, y-1).
5. Repeat the process until all pixels inside the boundary are filled.

Flood Fill Algorithm:

1. Start with a seed point (x, y).
2. Get the current color of the pixel.
3. If the current color is already the fill color or is different from the original color, return.
4. Set the pixel color to the fill color.
5. Recursively apply the algorithm to its neighboring pixels (4-connected or 8-connected):
   1. 4-connected: (x+1, y), (x-1, y), (x, y+1), (x, y-1)
   2. 8-connected: (x+1, y+1), (x-1, y+1), (x+1, y-1), (x-1, y-1)
6. Repeat the process until the entire region is filled.

**Program**

#include <graphics.h>

#include <iostream>

using namespace std;

void boundaryFill(int x, int y, int fill\_color, int boundary\_color)

{

    int current\_color = getpixel(x, y);

    if (current\_color != boundary\_color && current\_color != fill\_color)

    {

        putpixel(x, y, fill\_color);

        delay(1); //this adds delay to fill next pixel to look animated

        boundaryFill(x + 1, y, fill\_color, boundary\_color);

        boundaryFill(x - 1, y, fill\_color, boundary\_color);

        boundaryFill(x, y + 1, fill\_color, boundary\_color);

        boundaryFill(x, y - 1, fill\_color, boundary\_color);

    }

}

int main()

{

    int gd = DETECT, gm;

    initgraph(&gd, &gm, "");

        rectangle(100, 100, 300, 300);

    int boundary\_color = WHITE;

    int fill\_color = RED;

    boundaryFill(100, 100, fill\_color, boundary\_color);

    getch();

    closegraph();

    return 0;

}

A red square with black border

AI-generated content may be incorrect.**A black square with red line

AI-generated content may be incorrect.**

Figure After Filling

Figure Boundary Filling