UNIVERSIDADE FEDERAL DE RORAIMA
CENTRO DE CIÊNCIA E TECNOLOGIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
DCC703 - COMPUTAÇÃO GRÁFICA

GEORGE LUCAS MONCAO ZAMBONIN

RELATÓRIO TRABALHO PRIMEIRO, SEGUNDO E TERCEIRO TRABALHO

BOA VISTA, RR
2023

GEORGE LUCAS MONCAO ZAMBONIN

RELATÓRIO TRABALHO PRIMEIRO, SEGUNDO E TERCEIRO TRABALHO

Relatório da disciplina de Computação Gráfica. Neste documento, são apresentadas informações sobre preenchimento e rasterização de linhas e circunferências.

Professor: Luciano Ferreiro Silva

BOA VISTA, RR

2023

INTRODUÇÃO

Este relatório apresenta as principais funções desenvolvidas durante os trabalhos da disciplina de Computação Gráfica, sendo estas funções úteis para o processamento de gráficos em 2D. Sendo estas funções:

1. Rasterização de linhas:
    a. Metodo analitico;
    b. DDA;
    c. Bresenham.
2. Rasterização de circunferências:
    a. Equação paramétrica;
    b. Simetria incremental;
    c. Bresenham

RASTERIZAÇÃO DE LINHAS

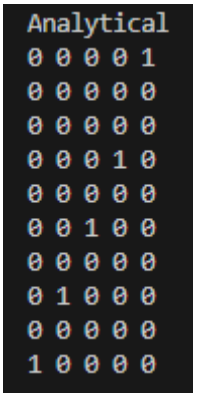
A rasterização de linhas é um processo fundamental na computação gráfica que envolve a conversão de representações vetoriais de linhas em imagens de grade ou matriz de pixels. Isso é feito determinando quais pixels na grade devem ser ativados para representar com precisão a linha desejada, levando em consideração sua posição, orientação e espessura. Esse método é amplamente utilizado em displays de computadores, monitores e impressoras, permitindo a renderização de elementos gráficos, como texto, gráficos e imagens, em dispositivos digitais. Para aplicação da técnica, neste trabalho foram utilizados três algoritmos distintos: Método analítico; DDA; Bresenham.

Método analítico

O método analítico consiste na rasterização de uma linha de forma intuitiva, utilizando a equação da reta. Veja o algoritmo:

```
void rtz::Line::analytical(int x1, int y1, int x2, int y2, arr::Array2d frame_buffer){
    int** array = frame_buffer.data;
    int rows = frame_buffer.rows;
    int cols = frame_buffer.cols;

    if (x1 == x2){
        if (y2 < y1)
            swap(&y1, &y2);

        for(int i = y1; i <= y2; i++)
            array[i][x1] = 1;
        return;
    }
    // find line equation
    float m = (float)(y2 - y1) / (x2 - x1);
    int b = (int)(y1 - m * x1);

    // x1 to x2 variation
    int y;
    for(int x = x1; x <= x2; x++){
        y = (int) (x * m + b);
        array[y][x] = 1;
        // cout << "(" << x << ", " << y << ") ";
    }
    return;
}
```

A primeira etapa do código consiste em detectar a existência de uma reta vertical e, caso haja, preencher de y1 até y2. Caso contrário, não haja uma reta vertical a equação da reta **y = f(x)** é descoberta com o auxílio de fórmulas. Os pontos y então são plotados para cada valor de x.



O método analítico é um algoritmo lento, quando comparado aos demais, utiliza números flutuando e possui problemas com a descontinuação das linhas, como apresentado na figura acima.

DDA: Analisador Diferencial Digital

O algoritmo Analisador Diferencial Digital, ou DDA, é mais rápido por utilizar menos operações de multiplicação e divisão, também resolve o problema de descontinuação do método analítico fazendo uma análise das variações de X e Y. Veja o algoritmo:

```
void rtz::Line::dda(int x1, int y1, int x2, int y2, arr::Array2d frame_buffer){
    int** array = frame_buffer.data;
    int rows = frame_buffer.rows;
    int cols = frame_buffer.cols;

    int d_x, d_y;
    d_x = x2 - x1;
    d_y = y2 - y1;

    float increment;

    if (abs(d_x) > abs(d_y)){
        float y;

        increment = (float) d_y / d_x;
        y = (float)y1;

        for(int x = x1; x <= x2; x++){
            array[(int)y][x] = 1;
            y += increment;
        }

        return;
    }

    increment = (float) d_x / d_y;
    float x = (float) x1;

    for(int y = y1; y <= y2; y++){
        array[y][(int)x] = 1;
        x += increment;
```

O algoritmo DDA verifica qual das variáveis (x e y) possuem a maior variação e, a partir desta análise, decide qual variável utilizará para fazer os incrementos e passos presentes na reta. Aquela variável que possuir a menor variação sofrerá incrementos decimais (0.0 à 1.0), enquanto a outra é incrementada em 1, isto resolve o problema de descontinuação presente no método analítico.

```
DDA
0 0 0 0 1
0 0 0 1 0
0 0 0 1 0
0 0 1 0 0
0 0 1 0 0
0 1 0 0 0
0 1 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
```

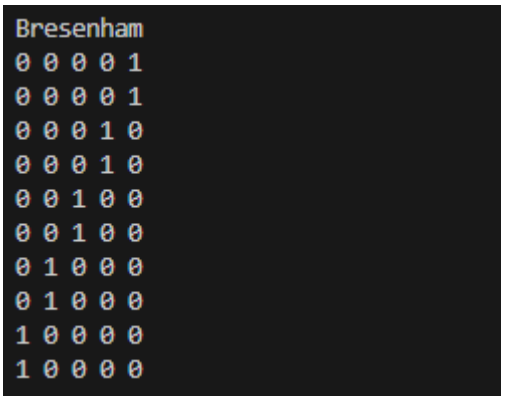
Este algoritmo resolve o problema de descontinuação, mas continua sendo um algoritmo que precisa trabalhar com a conversão de números flutuantes para inteiros.

Linhas de Bresenham

O Algoritmo de Bresenham consegue resolver os maiores problemas dos algoritmos anteriores, como multiplicações, números flutuantes e descontinuações. Veja o algoritmo:

```
void rtz::Line::bresenham(int x1, int y1, int x2, int y2, arr::Array2d frame_buffer) {
    int** array = frame_buffer.data;
    int rows = frame_buffer.rows;
    int cols = frame_buffer.cols;
    if (x2 < x1){
        swap(&x1, &x2);
        swap(&y1, &y2);
    }
    bool y1_bigger = false;
    if (y2 < y1) {
        y1 = -y1;
        y2 = -y2;
        y1_bigger = true;
    }
    int d_x, d_y, y, x;
    d_x = x2 - x1;
    d_y = y2 - y1;
    y = y1;
    x = x1;
    if (d_y > d_x) {
        int p = 2 * d_x - d_y;
        for (y = y1; y <= y2; y++) {
            y1_bigger ? array[-y][x] = 1 : array[y][x] = 1;
            if (p >= 0) {
                x += 1;
                p = p + 2 * (d_x - d_y);
                continue;
            }
            p = p + 2 * d_x;

        }
        return;
    }
    int p = 2 * d_y - d_x;
    for (int x = x1; x <= x2; x++) {
        y1_bigger ? array[-y][x] = 1 : array[y][x] = 1;
        if (p >= 0) {
            y += 1;
            p = p + 2 * (d_y - d_x);
            continue;
        }
        p = p + 2 * d_y;
```

Este algoritmo utiliza multiplicações por 2, por serem simples de serem calculadas na forma binária, acelerando significativamente sua execução. Ele calcula quais pixels devem ser ativados para representar uma linha entre dois pontos, levando em consideração a inclinação da linha e tomando decisões discretas para escolher os pixels mais próximos da linha ideal. O algoritmo utiliza cálculos de diferenças entre os valores dos eixos x e y para determinar o próximo pixel a ser ativado, minimizando erros de arredondamento e otimizando o desempenho de renderização. Isso o torna uma escolha popular para desenhar linhas de forma eficiente em ambientes gráficos.



Em contraste com os demais algoritmos, este algoritmo, por não trabalhar com números flutuantes e arredondamentos, possui uma linha mais uniforme, além de resolver os problemas apresentados pelos algoritmos anteriores.

RASTERIZAÇÃO DE CIRCUNFERÊNCIAS

A rasterização de circunferências é o processo de converter representações geométricas de circunferências, definidas por equações matemáticas, em imagens compostas por uma grade de pixels. Para realizar essa conversão, diversos algoritmos são empregados, sendo o algoritmo de Bresenham um dos mais conhecidos e utilizados. O objetivo é determinar quais pixels da matriz de imagem devem ser ativados para representar a circunferência de forma precisa, levando em consideração sua posição, raio e resolução da imagem. Esse processo é fundamental na renderização gráfica e é amplamente utilizado em aplicações que envolvem gráficos computacionais, jogos, design gráfico e visualização de dados. Para aplicação da técnica, neste trabalho foram utilizados três algoritmos distintos: Equação paramétrica; Algoritmo Incremental com Simetria; Circunferência de Bresenham.

Equação paramétrica

O algoritmo equação paramétrica permite os cálculos das curvas de formas complexas, reduzindo falhas. O algoritmo define funções paramétricas que descrevem a posição da curva em termos de um parâmetro variável, como o tempo ou um ângulo, e, em seguida, mostra essa função em intervalos discretos para determinar quais pixels devem ser ativados para representar a curva na grade de pixels.

```cpp
void rtz::Circumference::parametric_equation(int x_c, int y_c, int radius, arr::Array2d frame_buffer) {
    int rows, cols;
    rows = frame_buffer.rows;
    cols = frame_buffer.cols;

    if (x_c + radius >= cols || x_c - radius < 0 || y_c + radius >= rows || y_c - radius < 0){ // enchan
        throw std::invalid_argument("Circumference out of bound.");
        return;
    }

    int t;
    float x, y, t_radians;
    float pi_180 = (float) PI / 180.f;

    for(t = 1; t <= 360; t++) {
        t_radians = (float) t * pi_180;
        x = x_c + radius * cosf(t_radians);
        y = y_c + radius * sinf(t_radians);
        frame_buffer.data[(int)round(y)][(int)round(x)] = 1;
```

Este algoritmo faz o cálculo dos pontos em um intervalo de 1º até 360º. Com isto, os valores de x e y são calculados, então são somados com a posição do centro da circunferência e plotar eles.

```
Parametric
0 0 0 0 1 1 1 1 1 1 0 0 0 0
0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 0 1 1 0 0 0 0 0 0 0 1 1 0 0
0 1 1 0 0 0 0 0 0 0 0 0 1 1 0
1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 1 1 0 0 0 0 0 0 0 0 0 1 1 0
0 0 1 1 0 0 0 0 0 0 0 1 1 0 0
0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
```

Algoritmo Incremental com Simetria

O algoritmo incremental com simetria se baseia em deslocamentos angulares constantes e pequenos, com a rotação a partir de um ponto inicial. Além disso, ele divide a circunferência em 8 partes e as preenches utilizando a simetria da circunferência. Veja o algoritmo:

```
void rtz::Circumference::simmetric_incremental(int x_center,
int y_center, int radius, arr::Array2d frame_buffer, int value, int teta){
    int rows, cols;
    rows = frame_buffer.rows;
    cols = frame_buffer.cols;

    int** array = frame_buffer.data;

    float pi_180 = (float) PI / 180.f;

    float teta_radians, cos_teta, sin_teta;

    teta_radians = teta * pi_180;
    cos_teta = cosf(teta_radians);
    sin_teta = sinf(teta_radians);


    float vector_x, vector_y;

    vector_x = radius * cosf(teta_radians);
    vector_y = radius * sinf(teta_radians);

    int x, y, x_k, y_k;
    // int_limit = (360 / 8) * (45 / teta)

    for (int t = 2; t <= 45; t++) {
        x = x_center + (int)vector_x;
        y = y_center + (int)vector_y;

        x_k = (int)roundf(vector_x);
        y_k = (int)roundf(vector_y);

        this->simmetric_dot(array, x_center, y_center, x_k, y_k);

        vector_x = vector_x * cos_teta - vector_y * sin_teta;
        vector_y = vector_y * cos_teta + vector_x * sin_teta;
    }
}
```

Para o cálculo de uma circunferencia dividida em 360º, após dividido por 8 partes, serão necessarios calcular apenas 45º, após isto a simetria é aplicada aos 315º restantes. Para calcular os pontos, primeiro se calcula o cosseno e seno de 1º, depois calculasse o ponto inicial, depois disso são aplicados os deslocamentos, dados por:

$$x_{k+1} = x_k \cdot \cos\theta - y_k \cdot sen\,\theta$$

$$y_{k+1} = y_k \cdot \cos\theta + x_k \cdot sen\,\theta$$

Segue o algoritmo para aplicação da simetria:

```
void rtz::Circumference::simmetric_dot(int** array, int x_center, int y_center, int x_k, int y_k, int value){
    array[y_center + (int)y_k][x_center + (int)x_k] = value;
    array[y_center + (int)y_k][x_center - (int)x_k] = value;
    array[y_center - (int)y_k][x_center + (int)x_k] = value;
    array[y_center - (int)y_k][x_center - (int)x_k] = value;

    array[x_center + (int)x_k][y_center + (int)y_k] = value;
    array[x_center + (int)x_k][y_center - (int)y_k] = value;
    array[x_center - (int)x_k][y_center + (int)y_k] = value;
    array[x_center - (int)x_k][y_center - (int)y_k] = value;
}
```

Resultado da execução:

```
Simmetric incremental
0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 0 1 1 0 0 0 0 0 0 0 1 1 0 0
0 1 1 0 0 0 0 0 0 0 0 0 1 1 0
1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 1 1 0 0 0 0 0 0 0 0 0 1 1 0
0 0 1 1 0 0 0 0 0 0 0 1 1 0 0
0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
```

Este algoritmo apresenta erros acumulativos, em função do uso de x e y nas iterações seguintes e, assim como o algoritmo anterior, trabalha com números reais e arredondamentos.

Circunferência de Bresenham

O algoritmo da circunferência de Bresenham também utiliza a simetria da circunferência. Evita utilizar raízes, potências e funções trigonométricas. Ele utiliza cálculos matemáticos para determinar quais pixels devem ser ativados para representar uma circunferência com um raio dado. A escolha recai sobre três pixels, na tentativa de Selecionar o que está mais próximo da curva ideal. O critério de seleção entre tais pontos leva em conta a distância relativa entre os mesmos e a circunferência ideal.
Para decisão do pixel, levamos em conta o valor de:

$$\Delta i = (x_i+1)^2 + (y_i-1)^2 - R^2$$

O valor de *delta_i* é o fator determinante na escolha dos pixels.

```
void rtz::Circumference::bresenham(int x_center, int y_center,
 int radius, arr::Array2d frame_buffer, int value) {
    int rows, cols;
    rows = frame_buffer.rows;
    cols = frame_buffer.cols;
    int** array = frame_buffer.data;

    int x, y;

    x = 0;
    y = radius;

    int p = 1 - radius;

    while (x != y) {
        simmetric_dot(array, x_center, y_center, x, y);

        if (p >= 0) {
            y = y - 1;
            p = p + 2*(x - y) + 5;
            x = x + 1;
            continue;
        }

        // não altera y;
        p = p + 2*x +3;
        x = x + 1;
    }
    simmetric_dot(array, x_center, y_center, x, y);
```

O algoritmo mostrado acima se baseia no mesmo princípio, mas utiliza uma série de equivalências para simplificação do algoritmo.

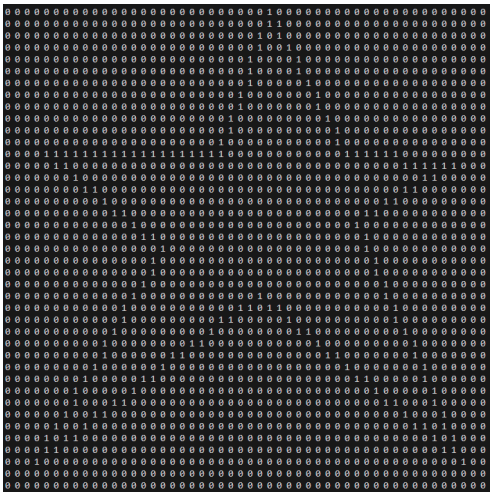
```
Bresenham
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 1 1 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
```

PREENCHIMENTO

Os algoritmos de preenchimentos são necessários em computação gráfica, pois permitem diversas aplicações, como o colorimento de formas geométricas. Estes algoritmos se baseiam nas bordas destas formas geométricas. Para aplicação da técnica, neste trabalho foram utilizados dois algoritmos distintos: Flood Fill e análise geométrica.

Ambos algoritmos utilizaram a figura a seguir para preenchimento:

```
polygon.add_vertice(3,47);
polygon.add_vertice(16,29);
polygon.add_vertice(4,21);
polygon.add_vertice(22,21);
polygon.add_vertice(27,9);
polygon.add_vertice(35,21);
polygon.add_vertice(46,22);
polygon.add_vertice(36,27);
polygon.add_vertice(47,47);
polygon.add_vertice(26,33);
```

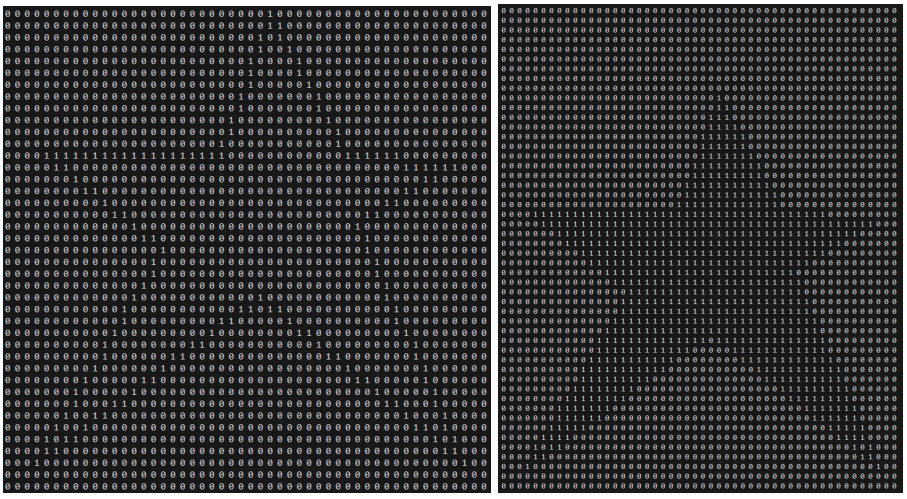


Flood Fill

O Flood Fill é um algoritmo recursivo, ou seja, em sua execução ele se repete diversas vezes. É um algoritmo bem simples, que consiste em preencher determinada superfície baseada na cor inicial (semente) escolhida e seus vizinhos. Veja o algoritmo:

```
void Fill::flood_fill_recursive(int x, int y, arr::Array2d frame_buffer, int seed_color, int new_color){
    int rows, cols, **array;
    rows = frame_buffer.rows;
    cols = frame_buffer.cols;
    array = frame_buffer.data;

    if (x > cols || x < 0 || y > rows || y < 0)
        return;

    if (array[y][x] == seed_color) {
        // array[y][x] == -1; // view effect
        array[y][x] = new_color;
        this->flood_fill_recursive(x+1, y, frame_buffer, seed_color, new_color);
        this->flood_fill_recursive(x-1, y, frame_buffer, seed_color, new_color);
        this->flood_fill_recursive(x, y+1, frame_buffer, seed_color, new_color);
        this->flood_fill_recursive(x, y-1, frame_buffer, seed_color, new_color);
    }
}

void Fill::flood_fill(int x, int y, arr::Array2d frame_buffer, int new_color){

    int rows, cols, **array;
    rows = frame_buffer.rows;
    cols = frame_buffer.cols;
    array = frame_buffer.data;

    if (x > cols || x < 0 || y > rows || y < 0) {
        throw std::invalid_argument("Flood_fill coordinates out of bound.");
        return;
    }

    int seed_color = array[y][x];

    this->flood_fill_recursive(x, y, frame_buffer, seed_color, new_color);
```

O resultado do algoritmo aplicado na figura dada anteriormente é:



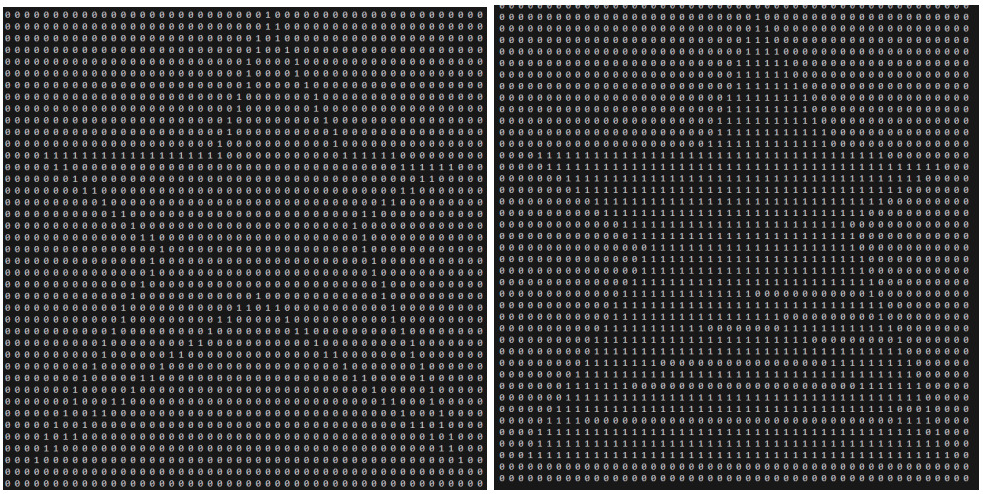
Na figura acima um problema causado pelo algoritmo flood fill, onde nas pontas inferiores da figura, uns dos pixels não são preenchidos pois não há vizinhança com a cor semelhante a semente.

Varredura com Análise Geométrica

O algoritmo de varredura com análise geométrica consiste na busca de pares de interseções presentes em linhas horizontais de uma figura. Com o encontro desses pares de interseções, o que há entre eles é preenchido com o valor escolhido. Veja o algoritmo abaixo:

```
void Fill::geometric(int x_min, int y_min, int x_max, int y_max, arr::Array2d frame_buffer, int color) {
    // fiz cansado, refatorar depois !!!
    // fiz cansado, refatorar depois !!!
    int** array = frame_buffer.data;
    int rows = frame_buffer.rows;
    int cols = frame_buffer.cols;

    int x, y, x_min_pair;
    bool pair;
    for (y = y_min; y <= y_max; y++) {
        x = x_min;
        pair = false;
        while (x <= x_max) {
            if (array[y][x] == color) {
                if (pair) {
                    if (x_min_pair == x - 1){
                        x_min_pair = x;
                    } else {
                        pair = false;
                        for (; x_min_pair < x; x_min_pair++)
                            array[y][x_min_pair] = color;
                    }
                } else {
                    pair = true;
                    x_min_pair = x;
                }
            }
            x++;
        }
```

Segue o resultado da execução do algoritmo na figura dada anteriormente:



Na figura acima, podemos perceber as lacunas e falhas causadas pelo algoritmo de varredura com análise geométrica. Onde algumas interseções acabam confundindo a ideia do algoritmo fazendo com o que haja preenchimentos indevidos em alguns lugares e ausência em outros.

CONCLUSÃO

Em conclusão, os algoritmos de rasterização desempenham um papel fundamental na computação gráfica, permitindo a representação precisa de formas geométricas em uma matriz de pixels. O algoritmo de Bresenham é uma abordagem notável para rasterização de linhas e circunferências, destacando-se por sua eficiência e capacidade de minimizar erros de arredondamento. Esses algoritmos são essenciais em uma ampla variedade de aplicações, desde jogos e design gráfico até visualização de dados, desempenhando um papel crítico na criação de elementos gráficos em dispositivos digitais. Portanto, o entendimento desses algoritmos é essencial para profissionais de computação gráfica e desenvolvedores que buscam criar experiências visuais impressionantes e eficientes.