



UNIVERSIDADE FEDERAL DE RORAIMA CENTRO DE CIÊNCIA E
TECNOLOGIA BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
DCC703 – COMPUTAÇÃO GRÁFICA (2024.2)

Data de entrega: 07/02/2025

DISCENTES:

FELIPE RUBENS DE SOUSA BORGES (2020020120)

COMPUTAÇÃO GRÁFICA

Relatório Preenchimento de Polígonos

Relatório técnico de acordo com a proposta apresentada pelo docente Prof. Luciano Ferreira relacionado ao projeto de Preenchimento de Polígonos da disciplina de Computação Gráfica, que deve desenvolver um programa que permita preencher polígonos por meio dos algoritmos: Flood Fill e Varredura com Análise Geométrica. Além de construir um relatório que descreva a construção e os resultados de maneira comparativa.

**Boa Vista-RR
2024.2**

Relatório de Preenchimento de Polígonos

Relatório apresentado para o projeto de Preenchimento de Polígonos da disciplina de Computação Gráfica, ofertada pelo curso de Ciência da Computação da Universidade Federal de Roraima.

Prof. Luciano Ferreira

Resumo

Este relatório descreve a implementação de uma interface gráfica para preencher um retângulo, uma circunferência e dois polígonos utilizando a biblioteca Pygame, Matplotlib e Numpy, em Python. O programa tem como objetivo preencher as formas na tela ao selecionar um dos dois algoritmos disponíveis: Flood Fill e Varredura.

Algoritmos

- **Flood Fill**

O Flood Fill é um algoritmo de preenchimento de áreas. Ele começa a partir de um ponto-semente e expande para os pixels vizinhos que possuem a mesma cor que o ponto inicial, preenchendo a região até encontrar uma borda de cor diferente. Esse algoritmo é muito utilizado para preenchimento de regiões fechadas, como na ferramenta de "balde de tinta" em programas de desenho.

Tipos de Flood Fill:

- **Flood Fill Recursivo:**

- Usa chamadas recursivas para verificar os pixels vizinhos (cima, baixo, esquerda, direita).
- Problema: pode causar **estouro de pilha** em regiões muito grandes.

- **Flood Fill com Fila (Iterativo):**

- Usa uma estrutura de dados como uma **fila** ou **pilha** para armazenar os pontos a serem processados.
- Mais eficiente e evita o problema de estouro de pilha.

Funcionamento:

- Se um pixel (x, y) deve ser preenchido:
 1. Verifica a cor do pixel.
 2. Se for da mesma cor que o ponto-semente, preenche com a nova cor.
 3. É necessário um pixel interno do corpo (semente) para dar início ao processo.
 4. Regiões são definidas por critérios de vizinhança ao pixel semente.
 5. Pixels com cor semelhante à semente: Borda tem cor diferente
 6. Pixel com cor diferente de uma cor dada: Borda tem cor igual à cor dada
 7. Repete o processo para os pixels vizinhos.
- **Varredura com Análise Geométrica**

A Varredura com Análise Geométrica, também chamada de Scanline Fill, é um método de preenchimento de polígonos e outras formas geométricas linha por linha, analisando as interseções das bordas com cada linha horizontal (scanline).

Como funciona:

- **Identificar as interseções**
 - Percorre a imagem linha por linha (varredura horizontal).
 - Identifica os pontos de interseção das bordas do polígono com a linha de varredura.
- **Ordenar os pontos de interseção**
 - Os pontos de interseção são ordenados pelo eixo x.
- **Preencher os pares de interseção**
 - Cada par de interseção define um intervalo a ser preenchido.
 - O preenchimento acontece somente entre pares de interseção.
- **Algoritmo clássico usa técnica de varredura**
 - Arestas são ordenadas
 1. Chave primária: y mínimo
 2. Chave secundária: x mín.
 3. Exemplo: (e,d,a,b,c)
 - Linha de varredura perpendicular ao eixo y percorre o polígono (desde ymin até ymax).
 - Intervalos horizontais entre pares de arestas são preenchidos.
- **Intervalos de preenchimento**
 - Definidos sobre a linha de varredura.
 - Cada intervalo começa e termina sobre um pixel interceptado por uma aresta.
 - Arestas horizontais não são consideradas.
 - Um vértice de uma aresta horizontal é considerado apenas se for o vértice com menor y.

Varredura para retângulo:

```
for (y=ymin; y<=ymax; y++)  
    for (x=xmin, x<=xmax; x++)  
  
writepixel(x,y,value);
```

Varredura para circunferência:

- Calcula-se facilmente a interseção dela com a linha de varredura.
 - $x1 = xc - \sqrt{R^2 - (y - yc)^2}$
 - $x2 = xc + \sqrt{R^2 - (y - yc)^2}$
 - Preenche-se de (x1, y) até (x2, y)
 - Com $yc - R < y < yc + R$

Implementação

Inicialmente, foi feita uma tela, de resolução 800 x 600, que indica qual número o usuário deve seleccionar para escolher qual forma deve ser preenchida no método de Varredura:

```

93 # tela
94 WIDTH, HEIGHT = 800, 600
95 tam_celula = 10
96 screen = pygame.display.set_mode((WIDTH, HEIGHT))
97 pygame.display.set_caption("Preenchimento com Varredura com Análise Geométrica")
98 screen.fill((255, 255, 255))
99

```

▪ Varredura com Análise Geométrica

Para implementação da Varredura com Análise Geométrica para polígonos foi usada como base a fórmula dada nos slides da disciplina:

2º passo: identificar as diversas interseções com a linha de varredura: usamos a fórmula de cálculo - eq. da reta:

$$Y - Y_0 = m (X - X_0)$$

$$Y_{\text{varredura}} - Y_{\text{min}} = m (X - X_{\text{min}})$$

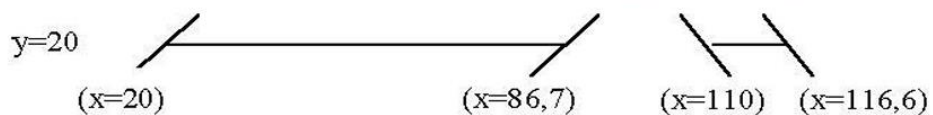
$$X - X_{\text{min}} = \frac{1}{m} (Y_{\text{varredura}} - Y_{\text{min}})$$

$$X = \frac{1}{m} \cdot (Y_{\text{varredura}} - Y_{\text{min}}) + X_{\text{min}}$$

Com:

- $Y_{\text{varredura}} < Y_{\text{max}}$
- $Y_{\text{varredura}} > Y_{\text{min}}$

3º passo: ordenam-se os pontos e traçam-se linhas, tomando-as de duas em duas, a partir de x de valores crescentes:



- **Regra de paridade:** iniciar contador com um número par, acrescentá-lo quando encontra uma intersecção e pintar quando ele for impar;


Função no código:

```
11 # poligonos
12 def varredura(surface, color, points):
13     points = ord_pontos_polig(points)
14     edges = sorted(points, key=lambda p: p[1])
15     y_min, y_max = edges[0][1], edges[-1][1]
16
17     for y in range(y_min, y_max):
18         intersections = []
19         for i in range(len(points)):
20             p1, p2 = points[i], points[(i + 1) % len(points)]
21             if (p1[1] <= y < p2[1]) or (p2[1] <= y < p1[1]):
22                 x = p1[0] + (y - p1[1]) * (p2[0] - p1[0]) / (p2[1] - p1[1])
23                 intersections.append(int(x))
24
25         intersections.sort()
26         if len(intersections) % 2 == 0:
27             for i in range(0, len(intersections), 2):
28                 pygame.draw.line(surface, color, (intersections[i], y), (intersections[i + 1], y))
29                 pygame.display.flip()
30                 pygame.time.delay(10)
31
```

Para implementação da Varredura com Análise Geométrica para o retângulo foi usada como base a fórmula dada nos slides da disciplina:

UFRR – Departamento de Ciência da Computação
Computação Gráfica – Prof. Dr. Luciano F. Silva

- **Preenchimento da Retângulos: um dos mais simples;**



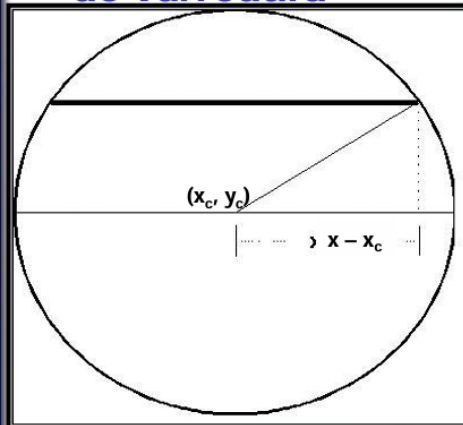
For (y=y_{min}; y<=y_{max}; y++)
 for (x=x_{min}, x<=x_{max};
 x++)
 writepixel(x,y,value);

Função no código:

```
34 # retângulo
35 def varredura_retangulo(surface, color, rect):
36     x_min, y_min, width, height = rect
37     for y in range(y_min, y_min + height):
38         pygame.draw.line(surface, color, (x_min, y), (x_min + width, y))
39         pygame.display.flip()
40         pygame.time.delay(10)
41     pygame.display.flip()
```

Para implementação da Varredura com Análise Geométrica para a circunferência foi usada como base a fórmula dada nos slides da disciplina:

■ **Preenchimento da Circunferência: calcula-se facilmente a interseção da mesma com a linha de varredura**



- $x_1 = x_c - \sqrt{R^2 - (y - y_c)^2}$
- $x_2 = x_c + \sqrt{R^2 - (y - y_c)^2}$
- Preenche-se de (x_1, y) até (x_2, y)
- Com $y_c - R < y < y_c + R$

Função no código:

```
43 # circunferência
44 def varredura_circunferencia(surface, color, center, radius):
45     cx, cy = center
46     for y in range(cy - radius, cy + radius):
47         dx = math.sqrt(radius**2 - (y - cy)**2)
48         x1, x2 = int(cx - dx), int(cx + dx)
49         pygame.draw.line(surface, color, (x1, y), (x2, y))
50         pygame.display.flip()
51         pygame.time.delay(10)
52     pygame.display.flip()
53
```

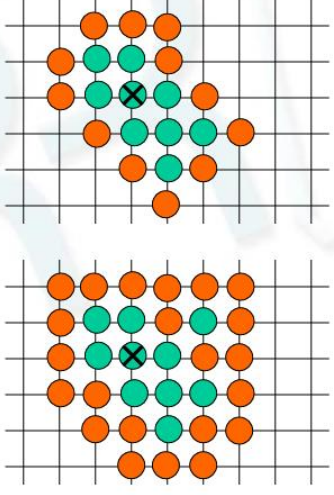
▪ Flood Fill

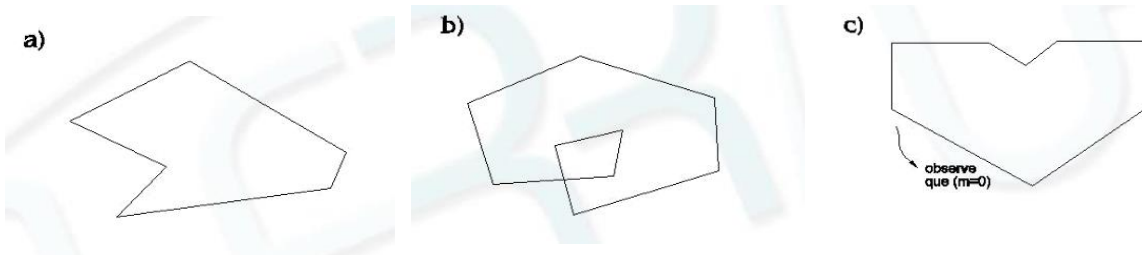
Assim como no método anterior, também foi utilizada como base a explicação presente nos slides da disciplina, mas o método utilizado não foi o recursivo e sim o iterativo para preencher o Retângulo, a Circunferência, a forma A, B e C:

UFRR - Departamento de Ciência da Computação
Computação Gráfica - Prof. Dr. Luciano F. Silva

▪ Princípios:

- ✓ O polígono (área) já está desenhado na tela com uma dada cor;
- ✓ É necessário um pixel interno do corpo (semente) para dar início ao processo;
- ✓ Regiões são definidas por critérios de vizinhança ao pixel semente;
- ✓ Exemplo:
 - Pixels com cor semelhante à semente
 - Borda tem cor diferente
 - Pixels com cor diferente de uma cor dada
 - Borda tem cor igual à cor dada





Função no código:

(Retângulo)

```

4 def flood_fill(x_inicial, y_inicial, matriz, dim):
5     pilha = [(x_inicial, y_inicial)]
6
7     while pilha:
8         x_atual, y_atual = pilha.pop()
9
10        if 0 <= x_atual < matriz.shape[1] and 0 <= y_atual < matriz.shape[0] and np.array_equal(matriz[dim - y_atual - 1, x_atual], [1, 1, 1]):
11            matriz[dim - y_atual - 1, x_atual] = [0, 0, 0]
12
13            # adição dos pontos vizinhos na pilha
14            pilha.append((x_atual + 1, y_atual))
15            pilha.append((x_atual - 1, y_atual))
16            pilha.append((x_atual, y_atual + 1))
17            pilha.append((x_atual, y_atual - 1))
18
19    return matriz

```


(Circunferência)

```
4 def flood_fill(px, py, matriz, dim):
5     pilha = [(px, py)] # pilha de pontos
6
7     while pilha:
8         px, py = pilha.pop()
9
10        if 0 <= px < matriz.shape[1] and 0 <= py < matriz.shape[0] and np.array_equal(matriz[dim - py - 1, px], [1, 1, 1]):
11            matriz[dim - py - 1, px] = [0, 0, 0]
12
13            # adição dos vizinhos na pilha
14            pilha.append((px + 1, py))
15            pilha.append((px - 1, py))
16            pilha.append((px, py + 1))
17            pilha.append((px, py - 1))
18
19    return matriz
```

(Forma A)

```
4 def flood_fill(px, py, matriz, dim):
5     pilha = [(px, py)] # pilha de pontos
6
7     while pilha:
8         px, py = pilha.pop()
9
10        if 0 <= px < matriz.shape[1] and 0 <= py < matriz.shape[0] and np.array_equal(matriz[dim - py - 1, px], [1, 1, 1]):
11            matriz[dim - py - 1, px] = [0, 0, 0]
12
13            # adição dos vizinhos na pilha
14            pilha.append((px + 1, py))
15            pilha.append((px - 1, py))
16            pilha.append((px, py + 1))
17            pilha.append((px, py - 1))
18
19    return matriz
```

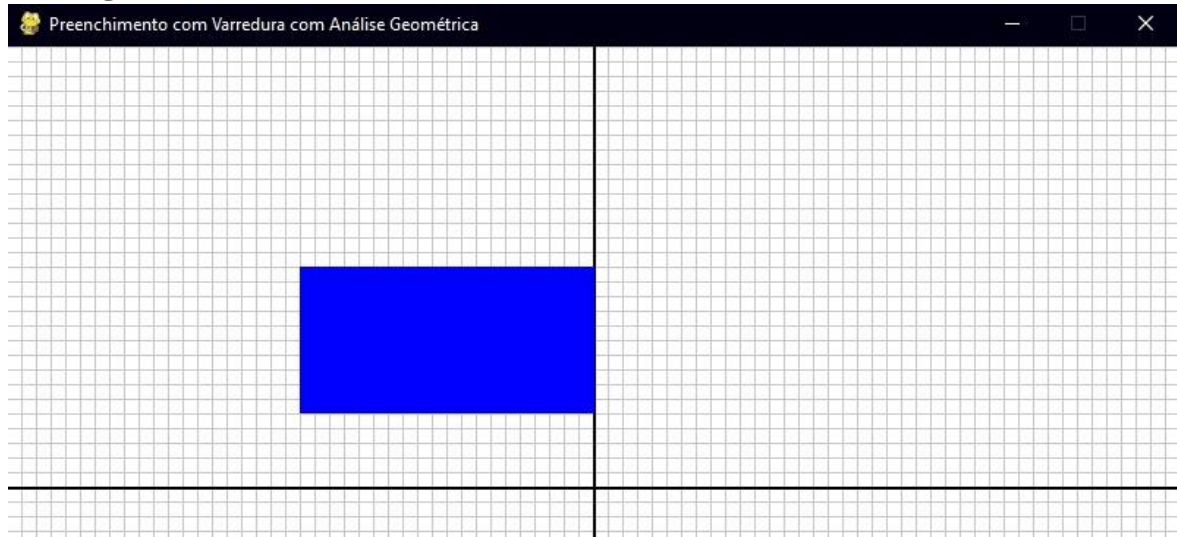
(Forma B)

```
4 def flood_fill(px, py, matriz, dimensao):
5     pilha = [(px, py)]
6
7     while pilha:
8         px, py = pilha.pop()
9
10        if 0 <= px < matriz.shape[1] and 0 <= py < matriz.shape[0] and np.array_equal(matriz[dimensao - py - 1, px], [1, 1, 1]):
11            matriz[dimensao - py - 1, px] = [0, 0, 0]
12
13            # adição dos pontos vizinhos na pilha
14            pilha.append((px + 1, py))
15            pilha.append((px - 1, py))
16            pilha.append((px, py + 1))
17            pilha.append((px, py - 1))
18
19    return matriz
```

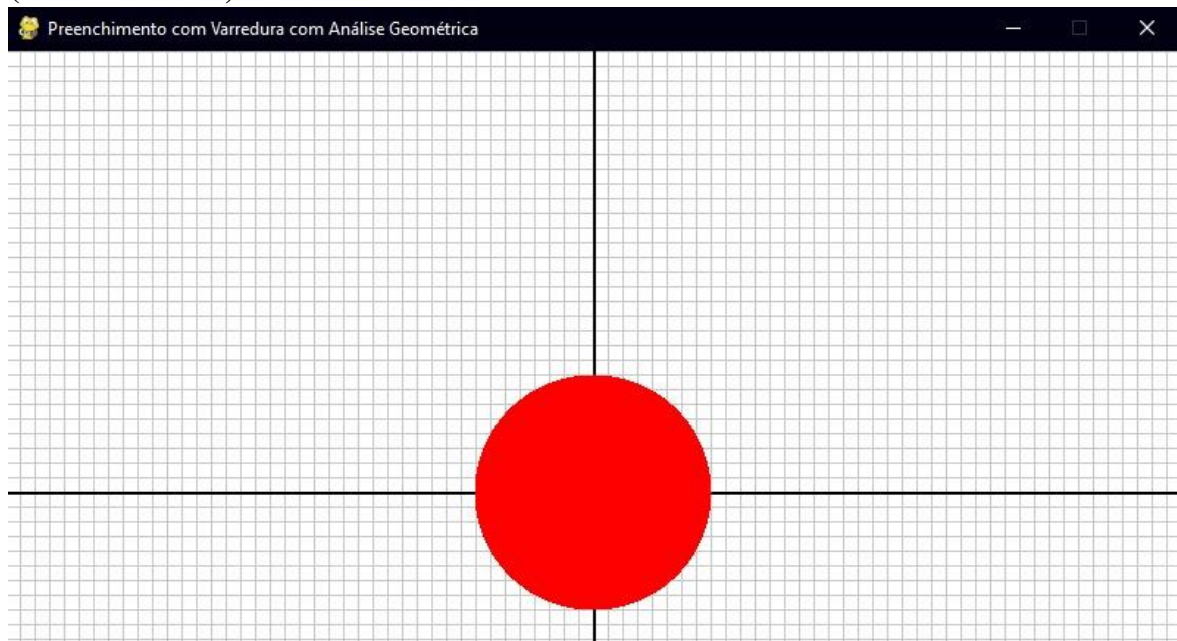
Resultados

- Varredura com Análise Geométrica

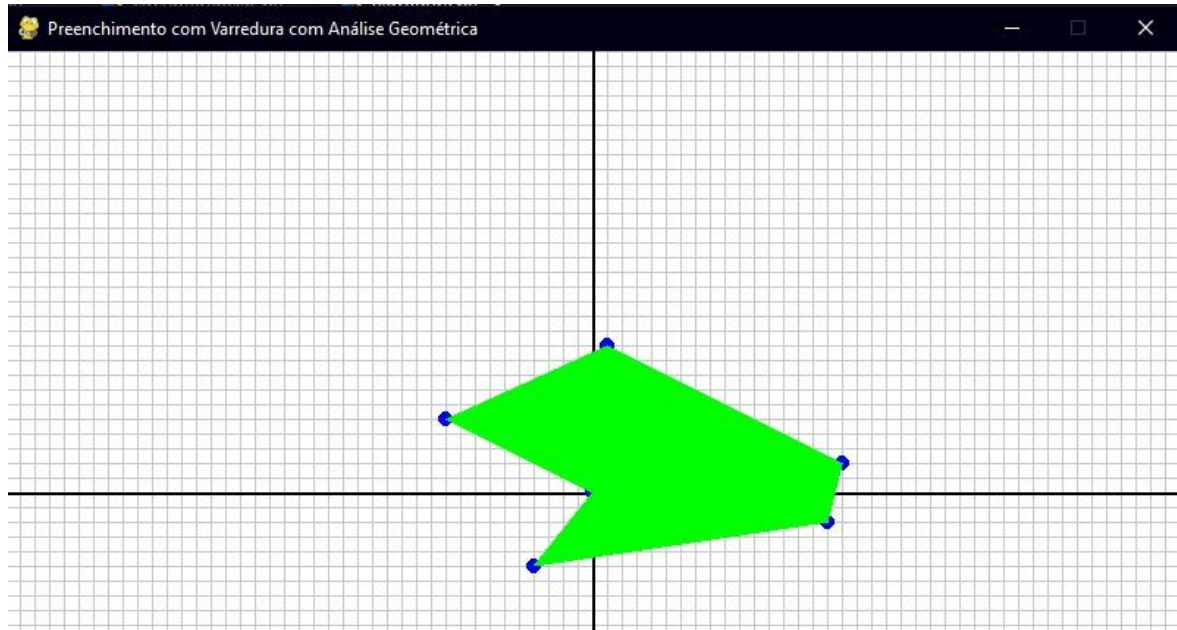
(Retângulo)



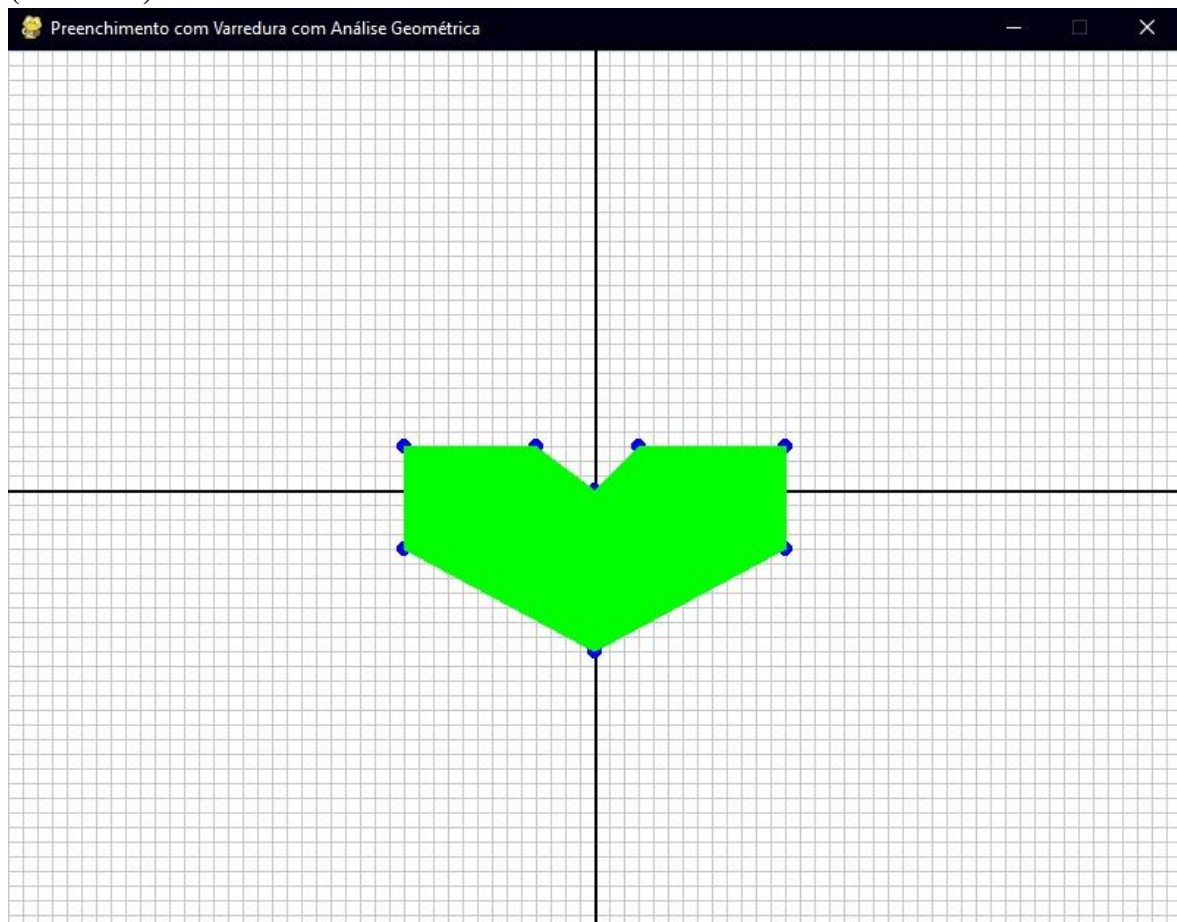
(Circunferência)



(Forma A)

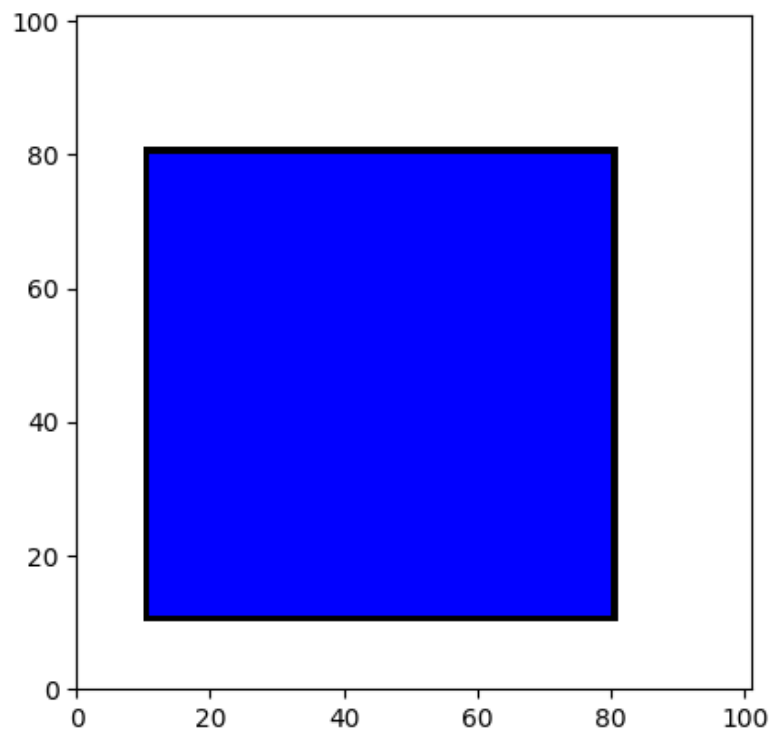


(Forma C)



- **Flood Fill**

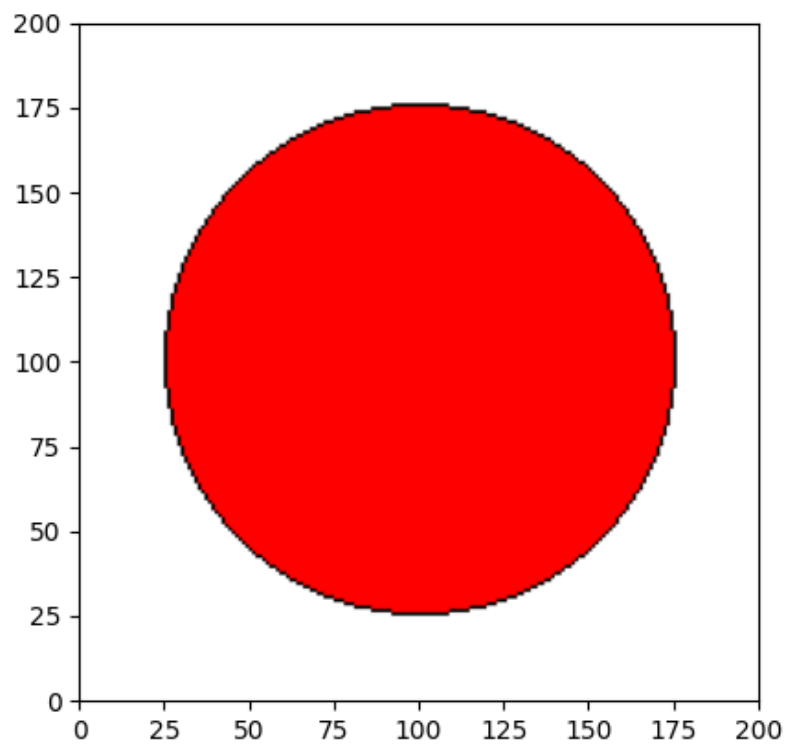
(Retângulo)



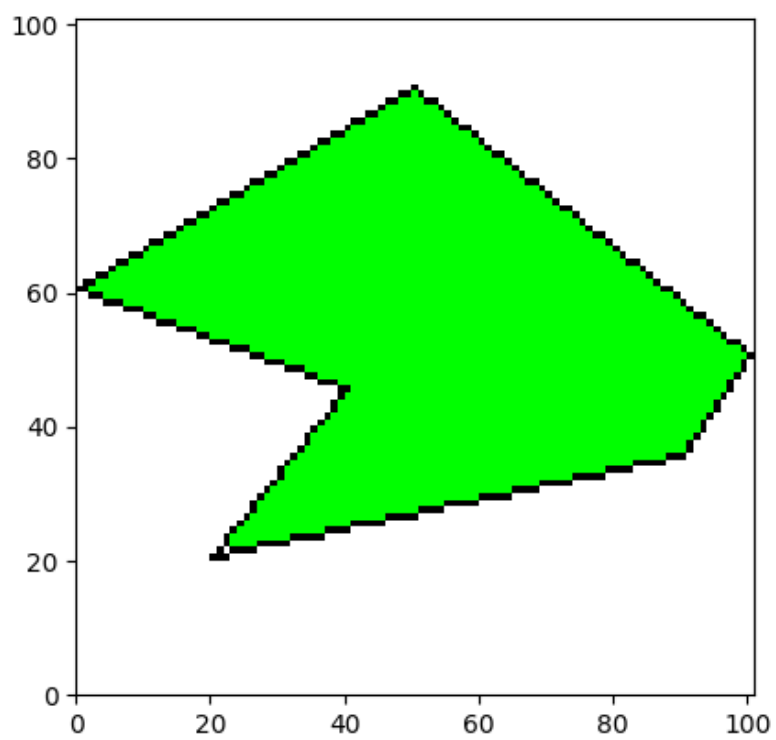
(Circunferência)

Figure 1

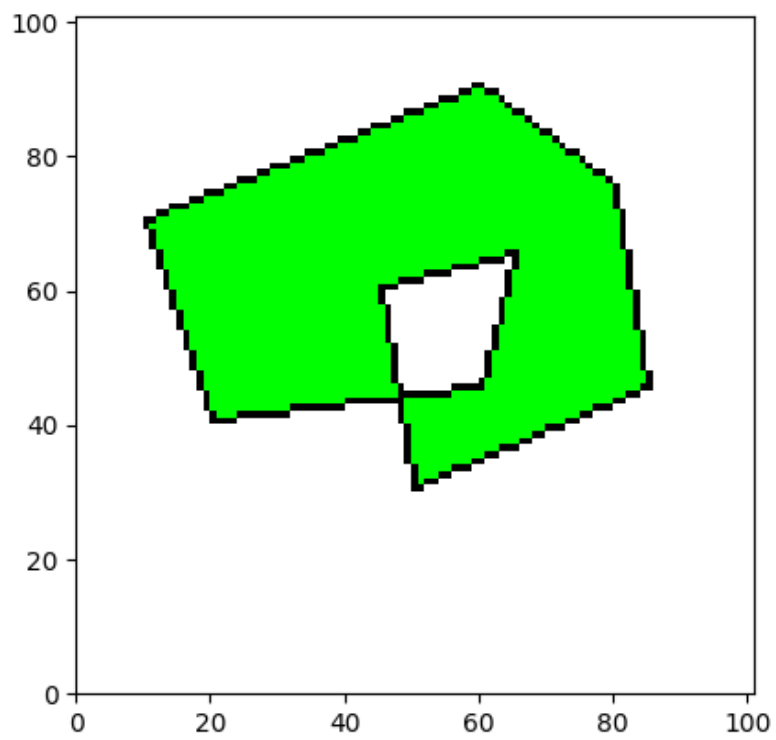
— □ ×



(Forma A)



(Forma B)



Comparações

A partir dos resultados, pode-se fazer comparações entre os preenchimentos feitos por cada algoritmo:

- **Vantagens da Varredura com Análise Geométrica**
 - Mais eficiente que o Flood Fill para preencher polígonos e formas fechadas.
 - Evita vazamentos de preenchimento (o Flood Fill pode vazar em regiões mal delimitadas).
 - Funciona bem para polígonos complexos sem necessidade de verificar cada pixel.
- **Desvantagens**
 - Pode falhar em polígonos que não têm bordas bem definidas.
 - Precisa de cálculo preciso das interseções para evitar falhas visuais.
- **Vantagens do Flood Fill**
 - Simplicidade na Implementação.
 - O algoritmo é relativamente fácil de entender e implementar, especialmente na versão recursiva.
 - Requer apenas uma verificação de vizinhança e uma substituição de cor.
 - Muito utilizado em edição de imagens, como a ferramenta de balde de tinta do Photoshop, GIMP ou Paint.
 - Funciona para preenchimento de texturas e sprites em gráficos 2D.
 - Diferente do Scanline Fill, que precisa calcular interseções exatas, Flood Fill não precisa de um contorno matemático bem definido.
 - Ele simplesmente verifica se o pixel tem a cor original e preenche.
- **Desvantagens:**
 - A maior limitação do Flood Fill é que ele pode ser lento para áreas muito grandes e consumir muita memória se não for otimizado corretamente (versão recursiva pode causar estouro de pilha).

Conclusão

Os algoritmos de preenchimento, como Flood Fill e Varredura, desempenham papéis fundamentais em computação gráfica, cada um com suas vantagens e aplicações específicas. O Flood Fill se destaca pela simplicidade e versatilidade, sendo ideal para preenchimento de regiões baseadas em cor, como em editores de imagem e mapas de jogos. No entanto, pode ser ineficiente para áreas grandes devido ao alto consumo de memória. Já o de Varredura é mais eficiente para preencher polígonos bem definidos, pois processa a imagem linha por linha, tornando-se uma escolha superior para gráficos vetoriais e formas geométricas complexas. Em resumo, a escolha do algoritmo depende do contexto: enquanto o Flood Fill é útil para preenchimentos dinâmicos e baseados em cor, o de Varredura oferece melhor desempenho para polígonos e superfícies bem delimitadas.