Trabalho Prático I
SCC0250 - Computação Gráfica

Preenchimento de Polígonos

Ian de Holanda Cavalcanti Bezerra - 13835412
Julia Graziosi Ortiz - 11797810

Setembro de 2025

1 Objetivo

Este trabalho tem como objetivo implementar o *algoritmo de preenchimento de polígonos* baseado na coerência de arestas, utilizando as estruturas *ET* (Edge Table) e *AET* (Active Edge Table), em uma aplicação gráfica interativa desenvolvida com a biblioteca Pygame. Tal implementação visa preencher corretamente qualquer polígono 2D (simples ou complexo) escolhido pelo usuário, com vértices inseridos por cliques do mouse, através do método de varredura por linhas (*scanline*).

1.1 Participação dos integrantes

O trabalho foi divido em duas partes principais: implementação do algoritmo de preenchimento e construção da interface gráfica. A base da interface foi reaproveitada de trabalhos anteriores realizados pela dupla.

- Algortimo do preenchimento : o principal contribuínte foi Ian, encarregado de estruturar a base do código e de implementar as estruturas ET e AET utilizadas no método de varredura de linhas (*scanline*), incluindo a criação da classe *EdgeEntry* para representar as arestas.
- Interface gráfica : a maior participação foi de Julia, responsável pela organização visual da interface, cuidando da disposição e conteúdo das mensagens informativas, e por implementar as interações entre usuário e a janela da aplicação como tratamento dos cliques no tabuleiro e nos botões.

Embora cada integrante tenha se concentrado em áreas específicas do desenvolvimento, ambos contribuiram com sugestões e ajustes em cada parte. Além disso, os testes para validar a implementação foram realizados em conjunto.

2 Implementação

As bibliotecas utilizadas foram: `math` para arredondar os valores com ponto flutuante; `sys` para interagir com o sistema; e `pygame` para criar a interface gráfica do sistema.

2.1 Interface gráfica

A interface do sistema é composta por um tabuleiro, utilizado como área de desenho, e um menu lateral que contém botões de controle e mensagens informativas. A exibição das mensagens é dinâmica e depende do estado atual do polígono e da interação do usuário. A Figura **??** exibe a interface gráfica no estado inicial, antes de qualquer comando.

O menu lateral possui os botões:

- ***Fechar Polígono*** : insere uma aresta entre o último vértice clicado e o primeiro;
- ***Validar*** : verifica se os vértices escolhidos formam um polígono válido para preenchimento;
- ***Preencher*** : desenha o resultado final do preenchimento do polígono através do método scanline;
- ***Scanline*** : cada vez que é acionado, exibe a linha de varredura atual, os pontos de intersecção entre as arestas e os pixels desenhados;
- ***Remover Últimо*** : apaga do tabuleiro e remove da lista o último vértice selecionado;
- ***Limpar Tela*** : apaga todos os elementos desenhados no tabuleiro;
- ***Cores Disponíveis para Preenchimento*** : troca a cor de preenchimento do polígono desenhado, identificando a cor atual pelo contorno do botão.

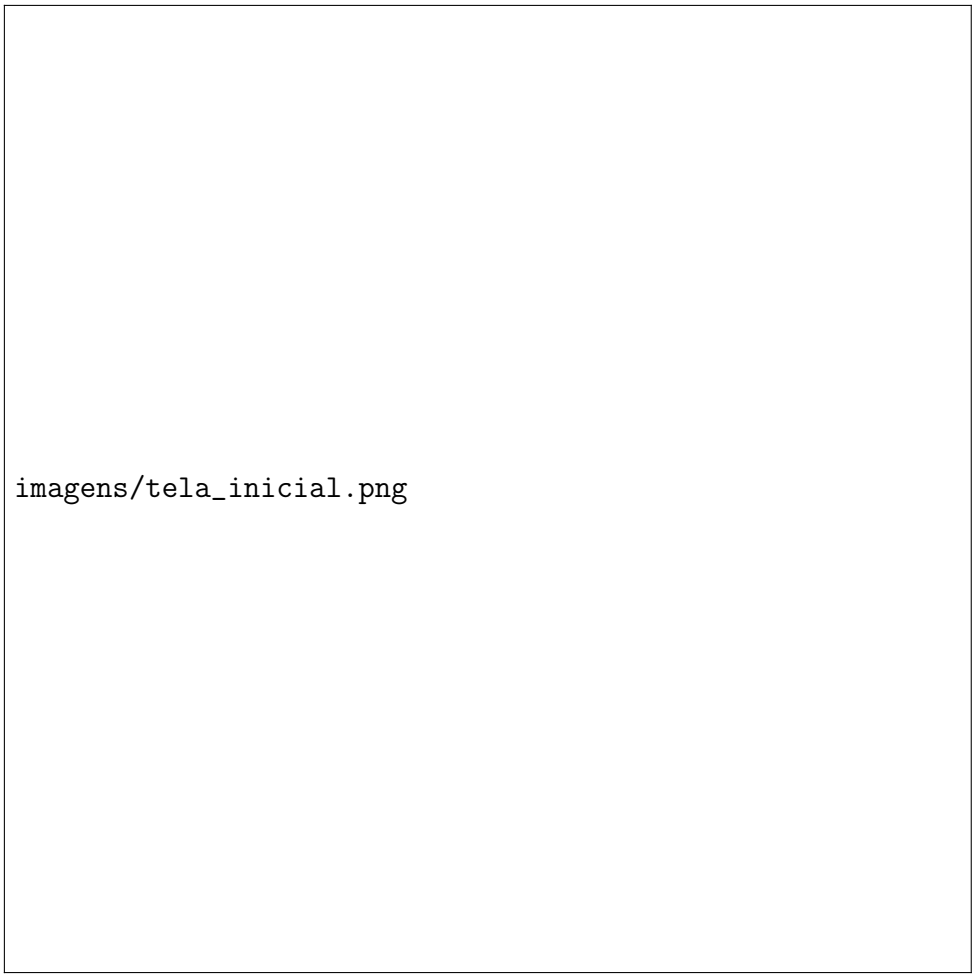

Figura 1: Interface inicial da aplicação.

O campo *Status* informa a situação atual do polígono e é atualizado conforme os vértices são inseridos para guiar as próximas ações do usuário. A partir do primeiro vértice selecionado, mais mensagens são exibidas no menu: resultado da validação, progresso do preenchimento (quantidade de pontos desenhados) e lista de vértices.

Cada vértice da lista é rotulado com um índice e uma cor: vermelho (primeiro), azul (último) ou branco (intermediários). Para evitar excesso de informações na tela, a lista é truncada caso o número de vértices ultrapasse o limite de exibição.

2.2 Construção e validação do polígono

A construção do polígono é iniciada com um clique do usuário em uma célula do tabuleiro, definindo o vértice inicial. A partir daí, o sistema mostra dinamicamente uma linha entre o último vértice selecionado e a posição atual do cursor, desde que permaneça dentro dos limites do tabuleiro. Ao escolher outro vértice, as coordenadas são armazenadas e a aresta correspondente é fixada.

Finalizada a escolha dos vértices, o polígono passa pela etapa de validação, que consiste em verificar se são satisfeitas as seguintes condições:

i) o polígono possui pelo menos três vértices distintos;

ii) os vértices do polígono não podem ser colineares (uma reta);

```python
def checar_vertices_alinhados(vertices: List[CelulaNaGrade]):

# Caso horizontal
if all(v[1] == vertices[0][1] for v in vertices):
    return True

# Caso vertical
if all(v[0] == vertices[0][0] for v in vertices):
    return True

# Caso obliqua
x1, y1 = vertices[0]
x2, y2 = vertices[1]

# Para cada vertice subsequente, verificar se esta na mesma linha
for i in range(2, len(vertices)):
    x3, y3 = vertices[i]

    # Usar produto cruzado para verificar colinearidade
    produto_cruzado = (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1)

    if produto_cruzado != 0:
        return False  # Nao sao colineares

return True  # Todos os vertices sao colineares
```

iii) o primeiro e o último vértice do polígono são iguais (fechado).

Se todas as condições forem atendidas, então o polígono poderá ser preenchido. A função `poligono_e_valido` é acionada somente quando há garantia de (i), sendo responsável por verificar as condicões (ii) e (iii) e fornecer um feedback.

```python
def poligono_e_valido(vertices: List[CelulaNaGrade]):

   # Remove vertices duplicados consecutivos
    vertices_limpos = []
    for i, vertice in enumerate(vertices):
        if i == 0 or vertice != vertices[i-1]:
            vertices_limpos.append(vertice)

    # Verifica se o poligono esta fechado
    if vertices_limpos[0] != vertices_limpos[-1]:
        return False, "Poligono nao esta fechado"

    # Verifica se os vertices nao estao todos alinhados
    if checar_vertices_alinhados(vertices_limpos):
        return False, "Poligono degenerado (reta)"

    # Qualquer poligono fechado e valido para preenchimento
    return True, "Poligono valido"
```

São válidos polígonos simples e complexos, com auto-intersecção e/ou com buracos. No caso de polígonos com buracos, basta dividí-lo em polígonos independentes e definir os vértices de cada parte separadamente. Validado(s) o(s) polígono(s), o usuário pode

escolher entre visualizar o preenchimento final ou acompanhar o passo a passo via *scanline* (varredura por linha).

2.3 Algoritmo de preenchimento

Cada aresta do polígono, definida pelo par de vértices (x_{min}, y_{min}) e (x_{max}, y_{max}), será representada pela classe `EdgeEntry`, a qual armazena:

- `ymax` : coordenada y_{max} da aresta
- `x` : coordenada x atual (iniciada em x_{min})
- `inv_slope` : inverso da inclinação $\left(\frac{x_{max} - x_{min}}{y_{max} - y_{min}}\right)$ para atualização de x

Com essa representação, definimos a estrutura *Edge Table* (ET), construída pela função `construir_tabela_arestas` ilustrada abaixo. Resumidamente, a ET funciona como um dicionário de arestas, em que cada linha y está associada a uma lista de arestas, com coordenadas x mantidas em ordem crescente.

O algoritmo de preenchimento por varredura (*scanline fill*) explora duas propriedades fundamentais de coerência para otimizar o processo de rasterização: a coerência de arestas e a coerência de linhas de varredura. A ideia central consiste em processar o polígono iterando sobre ele linha a linha e identificando os pontos de interseção entre o poligono e a linha de varredura, assim saberiamos o que esta dentro e o que esta fora do poligono. Para cada linha y, os pontos de interseção determinam os intervalos que devem ser preenchidos. O algoritmo utiliza duas estruturas de dados principais: a *Edge Table* (ET) e a *Active Edge Table* (AET). A ET armazena todas as arestas do polígono organizadas por suas coordenadas y mínimas, funcionando como um dicionário onde cada entrada y_{min} está associada a uma lista de arestas que começam naquela linha. Arestas horizontais são descartadas, pois não contribuem para os pontos de interseção em varreduras horizontais. Cada aresta na ET é representada pela classe `EdgeEntry`, que mantém o y_{max} (coordenada vertical máxima), o x atual (inicializado em x_{min}) e o inverso da inclinação $(1/m)$, permitindo calcular incrementalmente a posição x conforme a varredura avança.

A *Active Edge Table* (AET) é a estrutura dinâmica que armazena, para a linha de varredura atual, apenas as arestas ativas — aquelas que efetivamente cruzam a linha. À medida que a varredura itera sobre o poligono de linha em linha, a AET é atualizada seguindo três operações principais: (1) transferência de novas arestas da ET para a AET quando a linha de varredura alcança seus y_{min}; (2) remoção de arestas cujo y_{max} foi atingido, pois não cruzarão mais linhas subsequentes; e (3) atualização incremental das coordenadas x de cada aresta ativa usando $x_{novo} = x_{antigo} + \frac{1}{m}$, evitando cálculos trigonométricos custosos. Após cada atualização, a AET é reordenada por x crescente para facilitar o emparelhamento de interceptos.

Os valores x coletados são agrupados em pares consecutivos, definindo os intervalos $[x_i, x_{i+1}]$ que devem ser preenchidos. Como os interceptos são valores em ponto flutuante, aplica-se arredondamento para cima no início do intervalo (`ceil`) e para baixo no final (`floor`), garantindo que apenas pixels completamente dentro do polígono sejam rasterizados. Este processo se repete até que a ET e a AET estejam vazias, completando o preenchimento do polígono de forma eficiente e precisa.

```python
def construir_tabela_arestas(vertices: List[CelulaNaGrade]):

    ET = {}  # Edge Table
    n = len(vertices)

    # Inicializa limites y
    ymin_global = vertices[0][1]
    ymax_global = vertices[0][1]

    # Processa cada aresta do poligono
    for i in range(n):
        x1, y1 = vertices[i]
        x2, y2 = vertices[(i + 1) % n]

        # Ignora arestas horizontais
        if y1 == y2:
            continue

        # Determina ymin, ymax e x inicial
        if y1 < y2:
            ymin, ymax = y1, y2
            x_inicial = x1
        else:
            ymin, ymax = y2, y1
            x_inicial = x2

        # Calcula inverso da inclinacao (1/m)
        if y2 != y1:
            inv_slope = (x2 - x1) / (y2 - y1)
        else:
            inv_slope = 0

        # Adiciona aresta a ET
        if ymin not in ET:
            ET[ymin] = []
        ET[ymin].append(EdgeEntry(ymax, x_inicial, inv_slope))

        # Atualiza limites globais
        ymin_global = min(ymin_global, ymin)
        ymax_global = max(ymax_global, ymax)

    # Ordena arestas por x em cada bucket da ET
    for y in ET:
        ET[y].sort(key=lambda e: e.x)

    return ET, ymin_global, ymax_global
```

Visto que a origem do tabuleiro está localizada no canto superior esquerdo, ou seja, as coordenadas y aumentam de cima para baixo, foi necessário adaptar o Algoritmo 4.3 da apostila para implementar o preenchimento de polígonos. A modificação consiste em reordenar a ET, iniciando em $ymax$, para que visualmente a varredura ocorra de baixo para cima.

Baseada na coerência de arestas e na coerência de linhas de varredura, a função `algortimo_preenchimento_scanline` utiliza a estrutura *Active Edge Table* (AET) para armazenar, a cada linha de varredura, todas as arestas ativas, isto é, que cruzam a linha

atual.

```python
def algoritmo_preenchimento_scanline(ET, ymin, ymax):

    # Reconstroi ET: indexa por ymax em vez de ymin
    ET_bottomup = {}
    for y_entry in ET:
        for edge in ET[y_entry]:
            ymax_edge = edge.ymax
            ymin_edge = y_entry

            # Para bottom-up, comecamos em ymax com x_final
            if edge.inv_slope != 0:
                x_final = edge.x + edge.inv_slope * (ymax_edge -
                    ymin_edge)
            else:
                x_final = edge.x

            # Cria nova entrada invertida
            edge_invertida = EdgeEntry(ymin_edge, x_final, -edge.
                inv_slope)

            if ymax_edge not in ET_bottomup:
                ET_bottomup[ymax_edge] = []
            ET_bottomup[ymax_edge].append(edge_invertida)

    # Ordena arestas por x em cada bucket da ET invertida
    for y in ET_bottomup:
        ET_bottomup[y].sort(key=lambda e: e.x)

    AET = []  # Active Edge Table (inicialmente vazia)
    y = ymax  # Linha de varredura atual (comecando de baixo)
    resultados_scanline = []

    # Repita ate que ET e AET estejam vazias
    while y >= ymin or AET:
        # Transfere do cesto y na ET para AET as arestas cujo ymax = y
        if y in ET_bottomup:
            AET.extend(ET_bottomup[y])
            # Remove as arestas transferidas da ET (opcional)
            del ET_bottomup[y]

        # Retira os lados que nao mais envolvidos nesta linha
        AET = [aresta for aresta in AET if aresta.ymax != y]

        # 3.3. Coleta coordenadas x para desenho (pares de interceptos)
        interceptos_x = [aresta.x for aresta in AET]
        resultados_scanline.append((y, interceptos_x.copy()))

        # Decrementa y de 1 (proxima linha de varredura - bottom-up)
        y -= 1

        # Atualiza x para o novo y
        for aresta in AET:
            aresta.x += aresta.inv_slope

        # Reordena AET por coordenada x
```

```
54          AET.sort(key=lambda aresta: aresta.x)
55
56      return resultados_scanline
```

Esta função retorna uma lista em que cada elemento é uma tupla contendo uma linha y e os respectivos valores de x interceptados pelas arestas ativas naquela linha. Os interceptos determinarão os blocos de pixels que serão desenhados.

Enfim, para preencher os pixels, os interceptos de cada linha de varredura são separados dois a dois para formar blocos (intervalos). Como os interceptos são pontos flutuantes e não queremos preencher pixels que não fazem parte do polígono, o inicio do intervalo é arredondado para cima enquanto o final, para baixo.

3 Resultados

Aqui vamos colocare alguns exemplos de diferentes polígonos:

3.1 Côncavo

Polígonos côncavos possuem pelo menos um ângulo interno maior que 180°, criando uma "reentrância"na forma. O algoritmo scanline preenche corretamente essas regiões ao identificar todos os pares de interceptos em cada linha de varredura, respeitando a regra de paridade ímpar-par para determinar quais pixels estão dentro do polígono.

Figura 2: Polígono côncavo

3.2 Convexo

Este é o caso mais simples para o algoritmo, pois cada linha de varredura gera exatamente um par de interceptos, resultando em um único intervalo contínuo de pixels a serem preenchidos por linha.

Figura 3: Polígono convexo

3.3 Auto-Intersecção

Polígonos com auto-interseção possuem arestas que se cruzam, criando regiões sobrepostas. O algoritmo scanline trata naturalmente esses casos através da regra de paridade: cada interseção alterna entre interior e exterior, preenchendo corretamente as regiões de acordo com o número ímpar ou par de cruzamentos até cada ponto.

Figura 4: Polígono com auto-interseção

3.4 Buracos

Polígonos com buracos são tratados definindo múltiplos polígonos independentes: um externo e um ou mais internos para as cavidades. O algoritmo processa todas as arestas simultaneamente na ET global, permitindo que a regra de paridade exclua automaticamente as regiões internas, criando os buracos desejados.

Figura 5: Polígono com buraco

4 Execução do código

Conseguimos executar o código com esses comandos.

```
$ python3 -m venv Engine

$ pip3 install -r Requirements.txt

$ python3 Main.py
```