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

Descrever a participação de cada um.

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 janela do sistema é composta por um tabuleiro, que será utilizado como área de desenho, e um menu lateral. 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 Último*** : apaga do tabuleiro e remove da lista o último vértice selecionado;
- ***Limpar Tela*** : apaga todos os elementos desenhados no tabuleiro.

Além dos botões, o usuário pode escolher a cor de preenchimento do polígono. O menu também contém informações sobre os vértices (quantidade, e coordenadas) e sobre o resultado da validação.

O *Status* é atualizado a conforme os vértices são inseridos e a cada tentativa de validar um polígono, indicando se é possível iniciar o processo de preenchimento ou se é necessário alterar os vértices.

2.2 Vértices e validação do polígono

A construção do polígono é iniciada com um clique do usuário em uma casa do tabuleiro para defini-la como vértice inicial. A partir daí, o sistema mostra dinamicamente uma linha entre o último vértice selecionado e a posição atual do cursor. Quando outro vértice é escolhido, as coordenadas são armazenadas e a aresta 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` só é acionada quando há garantia de (i) e verifica as condições (ii) e (iii).

```
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 $\left(\dfrac{x_{max} - x_{min}}{y_{max} - y_{min}}\right)$ 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.

```
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]
```

```
9
10      # Processa cada aresta do poligono
11      for i in range(n):
12          x1, y1 = vertices[i]
13          x2, y2 = vertices[(i + 1) % n]
14
15          # Ignora arestas horizontais
16          if y1 == y2:
17              continue
18
19          # Determina ymin, ymax e x inicial
20          if y1 < y2:
21              ymin, ymax = y1, y2
22              x_inicial = x1
23          else:
24              ymin, ymax = y2, y1
25              x_inicial = x2
26
27          # Calcula inverso da inclinacao (1/m)
28          if y2 != y1:
29              inv_slope = (x2 - x1) / (y2 - y1)
30          else:
31              inv_slope = 0
32
33          # Adiciona aresta a ET
34          if ymin not in ET:
35              ET[ymin] = []
36          ET[ymin].append(EdgeEntry(ymax, x_inicial, inv_slope))
37
38          # Atualiza limites globais
39          ymin_global = min(ymin_global, ymin)
40          ymax_global = max(ymax_global, ymax)
41
42      # Ordena arestas por x em cada bucket da ET
43      for y in ET:
44          ET[y].sort(key=lambda e: e.x)
45
46      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.

```
1  def algoritmo_preenchimento_scanline(ET, ymin, ymax):
2
3      # Reconstroi ET: indexa por ymax em vez de ymin
4      ET_bottomup = {}
5      for y_entry in ET:
6          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
        AET.sort(key=lambda aresta: aresta.x)

    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

Testar diferentes polígonos:

- Côncavos
- Convexos
- Auto-intersecção
- Buracos

4 Referências