# Relatório de Funcionamento: Preenchimento de Polígonos com Coerência de Arestas

**Curso:** Computação Gráfica - SCC0650

**Data:** 01 de Outubro de 2025

**Autores:**

Pedro Bizon Dania - 11812667;

---

## 1. Introdução

Este relatório detalha a implementação de um sistema gráfico para o preenchimento de polígonos 2D, conforme especificado no trabalho da disciplina de Computação Gráfica. O objetivo principal foi implementar o algoritmo de preenchimento por coerência de arestas, que utiliza as estruturas de dados **Tabela de Arestas (ET)** e **Tabela de Arestas Ativas (AET)**.

O sistema foi desenvolvido em Python 3, utilizando a biblioteca PyOpenGL para renderização e o kit de ferramentas GLUT para a criação da janela e gerenciamento de eventos de interface, como cliques de mouse. A aplicação permite ao usuário desenhar interativamente um polígono e visualizá-lo preenchido, com opções para customização de cores.

## 2. Estrutura do Sistema

O projeto foi modularizado em quatro arquivos Python principais, cada um com uma responsabilidade clara:

- **`Ponto.py`**: Define a estrutura de dados mais fundamental, a classe `Ponto`, que armazena as coordenadas cartesianas (x, y) de um vértice.
- **`Aresta.py`**: Define a classe `Aresta`, que conecta dois objetos `Ponto`. Esta classe é central para o algoritmo, pois pré-calcula e armazena os valores essenciais para o método de scan-line:
  - `y_min`: A menor coordenada y da aresta, determinando em qual "cesto" da ET a aresta será inserida.
  - `y_max`: A maior coordenada y, usada para saber quando remover a aresta da AET.
  - `x_min`: A coordenada x correspondente ao `y_min`.
  - `inverso_m`: O inverso do coeficiente angular (`dx/dy`), usado para atualizar a coordenada x da interseção a cada passo da scan-line. Arestas horizontais (`dy=0`) são tratadas e recebem `inverso_m = 0.0` para serem descartadas.
- **`Edgetable.py`**: Implementa a classe `EdgeTable` (ET). A ET é uma estrutura de dados que organiza todas as arestas não-horizontais do polígono em "cestos"

(buckets), onde cada cesto corresponde a uma linha de varredura (coordenada y). A função `preencher_ET` popula essa estrutura, inserindo cada aresta no cesto correspondente ao seu `y_min`.

- **`main.py`**: É o arquivo principal que orquestra toda a aplicação. Suas responsabilidades incluem:
    - Inicialização da janela e do ambiente OpenGL/GLUT.
    - Gerenciamento de variáveis de estado globais (pontos, arestas, cores, etc.).
    - Registro e implementação das funções de callback para eventos de mouse (`mouse_click`) e de renderização (`display`).
    - A implementação da interface gráfica (botões).
    - A execução do algoritmo `preencher_poligono_scanline`.

**3. O Algoritmo de Preenchimento (Scan-Line)**

O núcleo do sistema é a função `preencher_poligono_scanline`, que implementa o algoritmo de preenchimento da seguinte forma:

1. **Inicialização**: O algoritmo começa localizando a primeira linha de varredura (`y`) que contém o início de pelo menos uma aresta (o primeiro cesto não-vazio da ET). A Tabela de Arestas Ativas (AET) é inicializada como vazia.
2. **Loop Principal**: O sistema entra em um laço que se repete para cada linha de varredura (`y`), de baixo para cima, até que tanto a AET quanto o restante da ET estejam vazios.
3. **Ordem de Operações (dentro do loop)**: A ordem correta das operações a cada passo `y` é crucial para o funcionamento correto em vértices complexos: a. **Remoção da AET**: Primeiramente, são removidas da AET todas as arestas cujo `y_max` é igual à linha de varredura atual `y`. Isso evita que uma aresta que termina e uma que começa no mesmo vértice causem problemas de paridade. b. **Adição na AET**: Em seguida, as arestas armazenadas na ET no cesto correspondente à `scanline y` são movidas para a AET. c. **Ordenação da AET**: A AET é ordenada. O critério principal é a coordenada `current_x` (interseção com a `scanline`). Para lidar com casos de empate (vértices ou cruzamentos), um critério secundário, o `inverso_m`, é utilizado. Isso garante uma ordenação estável e correta. d. **Desenho**: O algoritmo percorre a AET ordenada, pegando as interseções em pares (`1ª` com `2ª`, `3ª` com `4ª`, etc.). Para cada par, uma linha horizontal é desenhada entre as coordenadas `x` de início e fim. e. **Atualização**: A `scanline y` é incrementada. Para cada aresta que permanece na AET, sua coordenada `current_x` é atualizada somando-se o valor de `inverso_m`, preparando-a para a próxima iteração.

**4. Interface e Funcionalidades**

O sistema é controlado por eventos, gerenciados pelo GLUT:

- **`display()`**: É a função de renderização principal, chamada continuamente. Ela limpa a tela, desenha os botões e, se um polígono estiver finalizado

(`poligono_pronto_para_preencher == True`), executa o algoritmo de preenchimento, seguido pelo desenho das bordas.
- **`mouse_click()`**: Trata todos os cliques do mouse.
    - **Botão Esquerdo**: Adiciona um novo vértice à lista de `pontos` ou, se o clique for sobre um botão, executa a ação correspondente (limpar tela, mudar cor de preenchimento ou de fundo).
    - **Botão Direito**: Finaliza o processo de coleta de pontos, fecha o polígono, chama `criar_arestas()` para processar os dados e ativa a flag que inicia o preenchimento no `display loop`.

**5. Testes e Resultados**

A implementação foi validada com diversos tipos de polígonos:

- **Polígonos Simples (convexos e côncavos)**: O preenchimento ocorre conforme o esperado.
- **Polígonos Complexos (auto-intersecionados)**: A regra par-ímpar, implícita na ordenação e no desenho em pares da AET, funciona corretamente, preenchendo as regiões internas de forma consistente.
- **Casos Especiais de Vértices**: Graças à ordenação robusta da AET (usando `x` e `1/m`) e à ordem correta de remoção/adição de arestas, o algoritmo lida corretamente com vértices que são mínimos/máximos locais e com vértices onde uma aresta termina e outra começa na mesma `scanline`, evitando artefatos visuais.

**6. Conclusão**

O projeto implementou com sucesso o algoritmo de preenchimento por coerência de arestas. A estrutura modular do código facilitou o desenvolvimento e a depuração. Os desafios encontrados, principalmente relacionados ao tratamento de casos complexos de vértices, foram superados ajustando-se a lógica de ordenação e a ordem de operações dentro do laço principal do algoritmo, resultando em uma aplicação robusta e funcional.

**7. Participação dos Alunos**

- **Pedro Bizon Dania**: Todas as responsabilidades descritas acima;