**Trabalho Prático 1**

**Geometria computacional**

**Vinicius Trindade Dias Abel**

**Matrícula: 2020007112**

**Gustavo Ferreira Dias**

**Matrícula: 2017068170**

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

viniciustda@ufmg.br

ggustavof@ufmg.br

# **1. Introdução**

O problema proposto foi implementar um algoritmo para triangular um polígono, usando o método do corte de orelhas, computar o limite inferior de câmeras para vigiá-lo e, além disso, realizar a coloração dos vértices do polígono. A ferramenta deve permitir que o usuário possa interagir e executar passo-a-passo o algoritmo, destacando as faces que já foram exploradas, ou a face que está sendo avaliada em cada iteração.

# **2. Método**

## **2.1. Estrutura de Dados**

A implementação do programa teve como base a estrutura de dados de um grafo, representado por uma matriz de adjacência. Os vértices do grafo correspondem aos pontos do polígono e as arestas representam as conexões entre esses pontos. A escolha da matriz de adjacência foi motivada pela facilidade de manipulação e pela clareza na representação das conexões.

Para a resolução do problema de triangulação e coloração dos vértices do polígono, foram utilizados os seguintes algoritmos e estruturas:

1. Triangulação por Corte de Orelhas:
   1. Descrição: Este método consiste em iterativamente remover "orelhas" do polígono até que reste um único triângulo. Uma orelha é definida como um vértice do polígono que, junto com seus dois vizinhos, forma um triângulo convexo.
   2. Estrutura: A lista de vértices do polígono é manipulada diretamente para remover os vértices das orelhas encontradas, formando assim os triângulos.
2. Coloração de Vértices:
   1. Descrição: Utilizamos um algoritmo de coloração de vértices para garantir que dois vértices adjacentes não tenham a mesma cor. No caso do polígono, foi utilizado um algoritmo de coloração com no máximo 3 cores.
   2. Estrutura: A coloração foi realizada em um grafo gerado a partir dos vértices do polígono e dos triângulos resultantes da triangulação. Cada vértice do polígono foi colorido de forma que não compartilhe a mesma cor com seus vértices adjacentes.
3. Visualização:
   1. Descrição: A visualização da triangulação e da coloração dos vértices é feita utilizando a biblioteca Plotly. O processo é realizado em etapas, permitindo uma visualização clara de cada passo da triangulação e da coloração.
   2. Estrutura: A visualização é feita por meio de gráficos interativos, onde cada etapa da triangulação e da coloração é apresentada em um gráfico, permitindo ao usuário visualizar o progresso do algoritmo.

## **2.2. Funções**

## **addEdge**

A função addEdge recebe como parâmetros dois vértices u e v que representam os pontos a serem conectados. A função cria uma aresta bidirecional entre u e v na matriz de adjacência do grafo e adiciona essa aresta à lista de arestas.

A função funciona da seguinte forma:

Função addEdge(u, v):

matriz\_adj[u][v] ← 1

matriz\_adj[v][u] ← 1

lista\_arestas.adicionar((u, v))

## **isSafe**

A função isSafe verifica se é seguro colorir um vértice v com uma cor c. Ela retorna False se qualquer vértice adjacente a v já estiver colorido com a cor c, caso contrário, retorna True.

A função funciona da seguinte forma:

Função isSafe(v, cor, c):

Para cada i de 0 até n:

Se matriz\_adj[i][v] == 1 e cor[i] == c:

Retorna Falso

Retorna Verdadeiro

## **vertexColor\_util**

A função vertexColor\_util é uma função recursiva que tenta colorir os vértices do grafo usando até mmm cores. Se todos os vértices são coloridos com sucesso, retorna True; caso contrário, retorna False.

A função funciona da seguinte forma:

Função vertexColor\_util(m, cor, v):

Se v == total\_vertices:

Retorna Verdadeiro

Para cada c de 1 até m + 1:

Se isSafe(v, cor, c):

cor[v] ← c

Se vertexColor\_util(m, cor, v + 1):

Retorna Verdadeiro

cor[v] ← 0

Retorna Falso

## vertexColor

A função vertexColor tenta colorir os vértices do grafo usando até m cores chamando a função vertexColor\_util. Se bem-sucedida, armazena a solução globalmente e retorna True; caso contrário, retorna False.

A função funciona da seguinte forma:

Função vertexColor(m):

cor ← lista de zeros do tamanho de total\_vertices

Se não vertexColor\_util(m, cor, 0):

Retorna Falso

solução\_global ← cor

Retorna Verdadeiro

## tuplasIntDict

A função tuplasIntDict transforma uma lista de tuplas em um dicionário, onde cada tupla é mapeada para um índice inteiro.

A função funciona da seguinte forma:

Função tuplasIntDict(lista\_de\_tuplas):

dicionario ← {}

Para cada i, tupla em enumerar(lista\_de\_tuplas):

dicionario[tupla] ← i

Retorna dicionario

## **readPoligono**

A função readPoligono lê as coordenadas dos vértices de um polígono a partir de um arquivo e retorna uma lista de tuplas representando as coordenadas.

A função funciona da seguinte forma:

Função readPoligono(nome\_arquivo):

Abrir arquivo em modo leitura

Ler dados do arquivo e separar por espaços

n\_vertices ← converter dados[0] para inteiro

coordenadas ← []

Para i de 1 até n\_vertices\*2+1 com passo 2:

x ← converter dados[i] para ponto flutuante

y ← converter dados[i+1] para ponto flutuante

coordenadas.adicionar((x, y))

Retorna coordenadas

## isConvex

A função isConvex verifica se três pontos formam um triângulo convexo.

A função funciona da seguinte forma:

Função isConvex(p1, p2, p3):

Retorna (p3.y - p1.y) \* (p2.x - p1.x) - (p3.x - p1.x) \* (p2.y - p1.y) > 0

## **inTriangle**

A função inTriangle verifica se um ponto está dentro de um triângulo definido por três vértices.

A função funciona da seguinte forma:

Função inTriangle(pt, v1, v2, v3):

b1 ← não isConvex(pt, v1, v2)

b2 ← não isConvex(pt, v2, v3)

b3 ← não isConvex(pt, v3, v1)

Retorna (b1 == b2) e (b2 == b3)

## **earClippingTriangulation**

A função earClippingTriangulation realiza a triangulação de um polígono usando o método de corte de orelhas e retorna uma lista de triângulos. A função só funciona com uma entrada de polígono fechado e em ordem anti-horária; polígonos de entrada diferentes dessas condições não funcionam.

A função funciona da seguinte forma:

Função earClippingTriangulation(polígono):

triângulos ← []

vértices ← cópia do polígono

Enquanto tamanho de vértices > 3:

Para cada i de 0 até tamanho de vértices:

p1 ← vértices[i]

p2 ← vértices[i + 1]

p3 ← vértices[i + 2]

Se isConvex(p1, p2, p3):

orelha ← Verdadeiro

Para cada j em vértices:

Se j não está em (p1, p2, p3) e inTriangle(j, p1, p2, p3):

orelha ← Falso

Adicionar (p1, p2, p3) em triangulos\_plot

Pausar loop

Se orelha:

Adicionar (p1, p2, p3) em triângulos

Adicionar (p1, p2, p3) em triangulos\_plot

Remover vértices[i + 1]

Pausar loop

Adicionar (vértices[0], vértices[1], vértices[2]) em triângulos

Adicionar (vértices[0], vértices[1], vértices[2]) em triangulos\_plot

Retorna triângulos

## coloracaoVertices

A função coloracaoVertices recebe um polígono e sua triangulação, cria um grafo representando as conexões e realiza a coloração dos vértices utilizando no máximo 3 cores. Retorna um dicionário onde cada vértice está associado a uma cor.

A função funciona da seguinte forma:

Função coloracaoVertices(polígono, triângulos):

g ← novo Grafo(tamanho de polígono)

Para i de 0 até tamanho de polígono - 1:

g.addEdge(i, i + 1)

g.addEdge(tamanho de polígono - 1, 0)

dicionario\_indice\_vertices ← tuplasIntDict(polígono)

conexoes ← []

Para cada triângulo em triângulos:

Adicionar (triângulo[0], triângulo[1]) em conexoes

Adicionar (triângulo[1], triângulo[2]) em conexoes

Adicionar (triângulo[2], triângulo[0]) em conexoes

Para cada (a, b) em conexoes:

g.addEdge(dicionario\_indice\_vertices[a], dicionario\_indice\_vertices[b])

g.vertexColor(3)

sol\_coloracao ← última solução de cor no tamanho de polígono

vertex\_colors ← {}

cores ← ['black', 'green', 'blue', 'red', 'purple']

Para cada i, vértice em enumerar(polígono):

vertex\_colors[vértice] ← cores[sol\_coloracao[i]]

Retorna vertex\_colors

## plotCompleto

A função plotCompleto realiza a visualização da triangulação e da coloração dos vértices do polígono. Utiliza a biblioteca Plotly para gerar gráficos interativos que mostram cada passo do processo.

A função funciona da seguinte forma:

Função plotCompleto(polígono, triângulos, vertex\_colors):

fig ← nova Figura()

(x, y) ← desempacotar coordenadas do polígono

Adicionar Scatter na fig com x e y, modo linhas, nome 'Polígono'

step\_index ← 1

x\_cumulative ← ()

y\_cumulative ← ()

Para cada triângulo em triangulos\_plot:

(tx, ty) ← desempacotar coordenadas do triângulo

Se triângulo está em triângulos:

x\_cumulative ← x\_cumulative + tx

y\_cumulative ← y\_cumulative + ty

Adicionar Scatter na fig com x e y acumulados e triangulação, modo linhas, nome 'Triangulacao' + step\_index, legenda falsa

Caso contrário:

Adicionar Scatter na fig com x e y da orelha, modo linhas, linha pontilhada, cor vermelha, nome 'TriangulacaoOrelha' + step\_index

step\_index ← step\_index + 1

triangulacao\_completa ← step\_index - 1

Para cada vértice, cor em vertex\_colors:

Adicionar Scatter na fig com coordenadas do vértice, modo marcadores, cor, tamanho 20, nome cor + " passo " + step\_index, legenda falsa

step\_index ← step\_index + 1

steps ← []

Para cada i de 0 até tamanho de fig.data:

step ← {

método: "update",

argumentos: [{"visible": [Falso] \* tamanho de fig.data}, {"title": "Passo: " + i}]

}

step["argumentos"][0]["visible"][i] ← Verdadeiro

Se i >= triangulacao\_completa:

step["argumentos"][0]["visible"][triangulacao\_completa] ← Verdadeiro

Para cada j de triangulacao\_completa + 1 até i:

step["argumentos"][0]["visible"][j] ← Verdadeiro

steps.adicionar(step)

sliders ← [{ ativo: 0, valor\_corrente: {"prefixo": ""}, passos: steps }]

Atualizar layout da fig com título, títulos dos eixos, sliders

Mostrar fig

## 

## execAll

A função execAll executa todas as etapas do processo: leitura do polígono, triangulação, coloração dos vértices e visualização.

A função funciona da seguinte forma:

Função execAll(arquivo):

polígono ← readPoligono(arquivo)

triângulos ← earClippingTriangulation(polígono)

Imprimir "Triangulação ok"

vertex\_colors ← coloracaoVertices(polígono, triângulos)

Imprimir "Coloração ok"

plotCompleto(polígono, triângulos, vertex\_colors)

Imprimir "Plot ok"

**OBS.:** Para executar o programa, chame a função execAll com o caminho do arquivo do polígono:

execAll('caminho/do/arquivo.pol')

## **2.**3**. Configuração para teste**

– Sistema Operacional do computador: Windows 11;

– Linguagem de programação implementada: Python3;

– Dados do processador: i7;

– Quantidade de memória RAM: 16GB.

# 

# **3. Análise de Complexidade**

## addEdge

A função addEdge tem complexidade O(1). Ela realiza uma operação constante ao adicionar uma aresta na matriz de adjacência e na lista de arestas, independente do tamanho do grafo.

## isSafe

A função isSafe tem complexidade O(n). Ela verifica os vértices adjacentes em uma matriz de adjacência de tamanho n.

## vertexColor\_util

A função vertexColor\_util tem complexidade O(mn) no pior caso, onde m é o número de cores e n é o número de vértices. Isso ocorre porque a função tenta todas as combinações possíveis de coloração.

## vertexColor

A função vertexColor tem complexidade O(mn) no pior caso, devido à chamada de vertexColor\_util.

## tuplasIntDict

A função tuplasIntDict tem complexidade O(n). Ela percorre uma lista de n tuplas uma vez para criar o dicionário.

## readPoligono

A função readPoligono tem complexidade O(n), onde n é o número de vértices no polígono. Ela lê os dados do arquivo e os converte para coordenadas.

## isConvex

A função isConvex tem complexidade O(1). Ela realiza uma operação matemática constante para determinar se três pontos formam um triângulo convexo.

## inTriangle

A função inTriangle tem complexidade O(1). Ela realiza três verificações constantes de convexidade.

## earClippingTriangulation

A função earClippingTriangulation tem complexidade O(n²). O laço principal itera sobre os vértices do polígono, e para cada vértice, verifica os triângulos formados, resultando em uma complexidade quadrática.

## coloracaoVertices

A função coloracaoVertices tem complexidade O(mn) no pior caso, devido à chamada da função vertexColor para colorir os vértices do grafo.

## plotCompleto

A função plotCompleto tem complexidade O(n) para adicionar e visualizar os vértices e arestas do polígono, onde n é o número de vértices. No entanto, a complexidade real pode variar dependendo da implementação da biblioteca Plotly.

## execAll

A função execAll tem uma complexidade geral que depende das funções que ela chama:

execAll:

readPoligono: O(n)

earClippingTriangulation: O(n²)

coloracaoVertices: O(mn)

plotCompleto: O(n)

## Geral

O TP tem complexidade O(n²) para a triangulação e O(mn) para a coloração dos vértices. No entanto, a triangulação O(n²) é a operação dominante, especialmente para valores de m pequenos (por exemplo, m=3) na coloração.

A função principal chama readPoligono uma vez, que é O(n), seguida por earClippingTriangulation que é O(n²), e coloracaoVertices que é O(mn). A visualização final com plotCompleto é O(n).

Assim, a complexidade geral do TP é O(n²), assumindo m como uma constante pequena.

# 

# **4. Conclusões**

Este trabalho lidou com a triangulação e coloração de um polígono simples utilizando métodos de grafos. A abordagem utilizada para a resolução foi a Triangulação por Corte de Orelhas e a coloração dos vértices.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados à geometria computacional e algoritmos de grafos, especificamente técnicas de triangulação e coloração de grafos.

Durante a implementação da solução para o problema, surgiram desafios importantes a serem superados, como a correta triangulação de polígonos e a coloração eficiente dos vértices. A função de triangulação exige que o polígono de entrada seja fechado e esteja em ordem anti-horária, o que impôs uma restrição adicional na preparação dos dados de entrada. E a inexperiência com a biblioteca Plotly.

Inicialmente, a triangulação por corte de orelhas foi implementada para dividir o polígono em triângulos. Este método foi escolhido por sua simplicidade e eficácia em polígonos simples. No entanto, garantir que o polígono de entrada esteja em ordem anti-horária foi um desafio, e pode requerer pré-processamento dos dados.

Para a coloração dos vértices, utilizamos um algoritmo baseado na coloração de grafos, que, apesar de sua simplicidade, apresenta alta complexidade para polígonos com muitos vértices. Esse algoritmo garante que nenhuma cor se repete em vértices adjacentes, utilizando um máximo de três cores devido às propriedades dos grafos planares.

A visualização dos resultados foi um passo essencial para verificar a correta implementação dos algoritmos. Utilizamos a biblioteca Plotly para criar uma animação que mostra o processo de triangulação e coloração passo a passo. Isso não apenas facilitou a verificação de erros, mas também proporcionou uma maneira intuitiva de entender o funcionamento dos algoritmos.

Em resumo, a implementação deste trabalho permitiu uma compreensão prática e aprofundada dos conceitos de triangulação e coloração em grafos.

# **5. Bibliografia**

- Standard C++ Library reference. Disponível em: https://cplusplus.com/reference/

- Algoritmo de caminho de custo mínimo de Dijkstra - uma introdução detalhada e visual.

Disponível em: https://www.freecodecamp.org/portuguese/news/algoritmo-de-caminho-de-custo-minimo-de-dijkstra-uma-introducao-detalhada-e-visual/

- Grafos e algoritmos. Disponível em: https://medium.com/programadores-ajudando-programadores/os-grafos-e-os-algoritmos-697c1fd4a416

# **6. Instruções para compilação e execução**

• Requisito Python3

• Requisito Instalar Plotly - pip3 install plotly

• Executar main.py - python3 main.py