# Resolução do Problema da Galeria de Arte

Diogo Tuler, João Marcos Tomaz e Rafael Martins

Universidade Federal de Minas Gerais

## 1 Introdução

O problema da Galeria de Arte é um dos desafios mais conhecidos na área de geometria computacional. Neste relatório, apresentamos a implementação de um algoritmo de triangulação de polígonos e discutimos como ele pode ser aplicado para resolver o problema em questão. Para uma visualização dinâmica de todos os processos descritos, recomendamos acessar o GitHub Pages do projeto, onde estão disponíveis demonstrações interativas. As imagens incluídas neste relatório servem apenas para ilustrar as explicações.

## 2 Descrição do problema

Imagine a seguinte situação: você é dono de uma galeria em que vários quadros artísticos e grandes antiguidades são expostos. Sabendo dos perigos existentes, você precisa ter alguma forma de segurança dentro do espaço, como câmeras, por exemplo. A ideia do algoritmo é encontrar o menor número de câmeras que vigiam toda a sua galeria.

Para resolver esse problema, consideramos uma entrada entrada 2D conforme ilustrado na Figura 1 que representaria uma vista por cima da galeria.

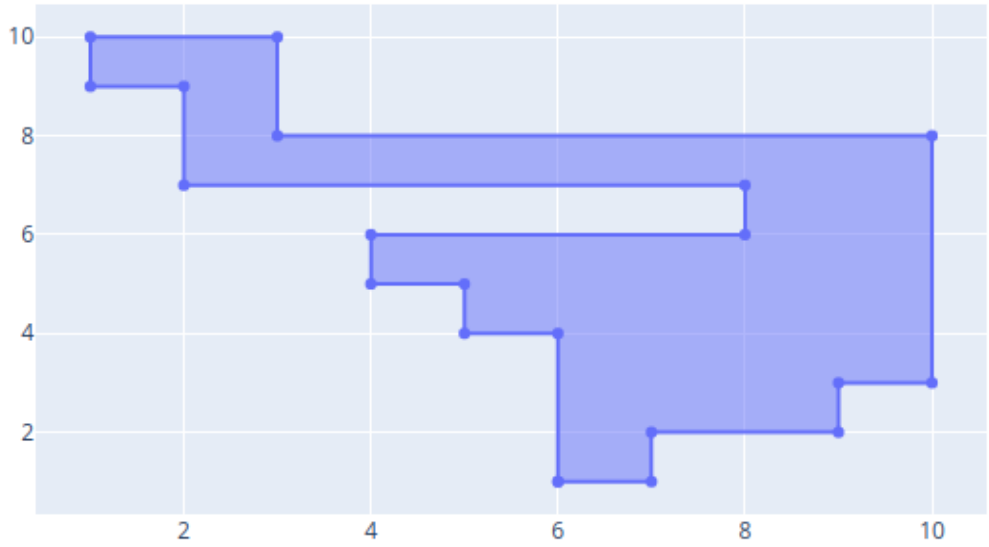


Figura 1: Exemplo de entrada para o problema

## 3 Solução

Começamos com a ideia de que um triângulo precisa apenas de uma câmera para ser completamente vigiado. Assim, podemos utilizar um algoritmo que decompõe o polígono em triângulos, facilitando a solução do problema. A técnica de divisão em triângulos envolve encontrar um conjunto de diagonais máximas que não se interceptam.

### 3.1 Triangulação

O algoritmo de triangulação utilizado é o "ear-clipping", um método geométrico usado para dividir um polígono simples (um polígono que não se intersecta) em um conjunto de triângulos.

O algoritmo recebe como entrada um polígono simples $P$. Primeiro, é necessário usar a função abaixo para saber se o polígono passado é convexo.

```
def is_convex(prev, curr, next):
    return (curr[0] - prev[0]) * (next[1] - curr[1]) - (curr[1] - prev[1])
        * (next[0] - curr[0]) > 0
```

Listing 1: Código em Python para verificar se o polígono é convexo

Para verificar se um vértice $v$ é uma "orelha", você precisa garantir que o triângulo formado por $v$, o vértice anterior e o sucessor esteja completamente dentro do polígono e que nenhum outro vértice esteja dentro deste triângulo. Isso pode ser feito verificando a orientação dos pontos e usando a função de ponto em triângulo. A função "is_point_in_triangle"abaixo cumpre essa verificação.

```
def is_point_in_triangle(p, a, b, c):
    # Coordenadas baricentricas
    d = (b[1] - c[1]) * (a[0] - c[0]) + (c[0] - b[0]) * (a[1] - c[1])
    w1 = ((b[1] - c[1]) * (p[0] - c[0]) + (c[0] - b[0]) * (p[1] - c[1])) /
        d
    w2 = ((c[1] - a[1]) * (p[0] - c[0]) + (a[0] - c[0]) * (p[1] - c[1])) /
        d
    w3 = 1 - w1 - w2
    return 0 <= w1 <= 1 and 0 <= w2 <= 1 and 0 <= w3 <= 1
```

Listing 2: Código em Python para verificar se um ponto está no triângulo

Com essas funções auxiliares, é possível verificar se existe uma orelha e fazer o corte. Uma "orelha"é um triângulo formado por três vértices consecutivos, tal que:

- O triângulo formado pelos vértices está completamente dentro do polígono.
- Nenhum outro vértice está dentro do triângulo.

A função abaixo verifica se um triângulo é uma "orelha":

```
def is_ear(polygon, i):
    n = len(polygon)
    prev = polygon[(i-1) % n]
    curr = polygon[i]
    next = polygon[(i+1) % n]
    if not is_convex(prev, curr, next):
        return False
    for j in range(n):
        if j in [(i-1) % n, i, (i+1) % n]:
            continue
        if is_point_in_triangle(polygon[j], prev, curr, next):
            return False
```

```
    return True
```

Listing 3: Código em Python para verificar se o triângulo é uma "orelha"

Com essas funções, é possível receber o polígono e fazer os cortes. A função abaixo visa pegar cada triângulo possível, verificar se é uma orelha e remover as orelhas. O algoritmo segue a seguinte ideia:

- Iterativamente, selecione e remova uma orelha do polígono, adicionando o triângulo formado aos resultados finais.
- Atualize a lista de vértices do polígono. Continue até que reste apenas o último triângulo.

```
def ear_clipping_triangulation(polygon):
    n = len(polygon)
    if n < 3:
        return []
    remaining_vertices = polygon[:]
    triangles = []
    while len(remaining_vertices) > 3:
        for i in range(len(remaining_vertices)):
            if is_ear(remaining_vertices, i):
                prev = remaining_vertices[i-1]
                curr = remaining_vertices[i]
                next = remaining_vertices[(i+1) % len(remaining_vertices)]
                triangles.append([prev, curr, next])
                del remaining_vertices[i]
                break
    triangles.append(remaining_vertices)
    return triangles
```

Listing 4: Código em Python para fazer a triangulação

A seguir, é apresentado um exemplo que ilustra o funcionamento do algoritmo:

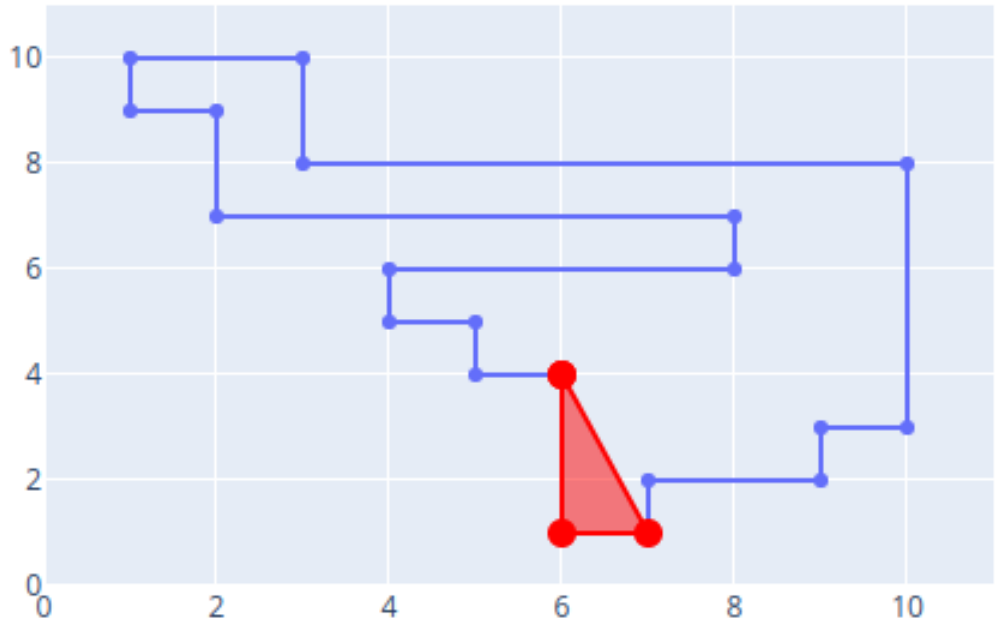


Figura 2: Uma possível "orelha"é identificada pelo algoritmo, e os testes necessários são realizados para verificar sua validade.

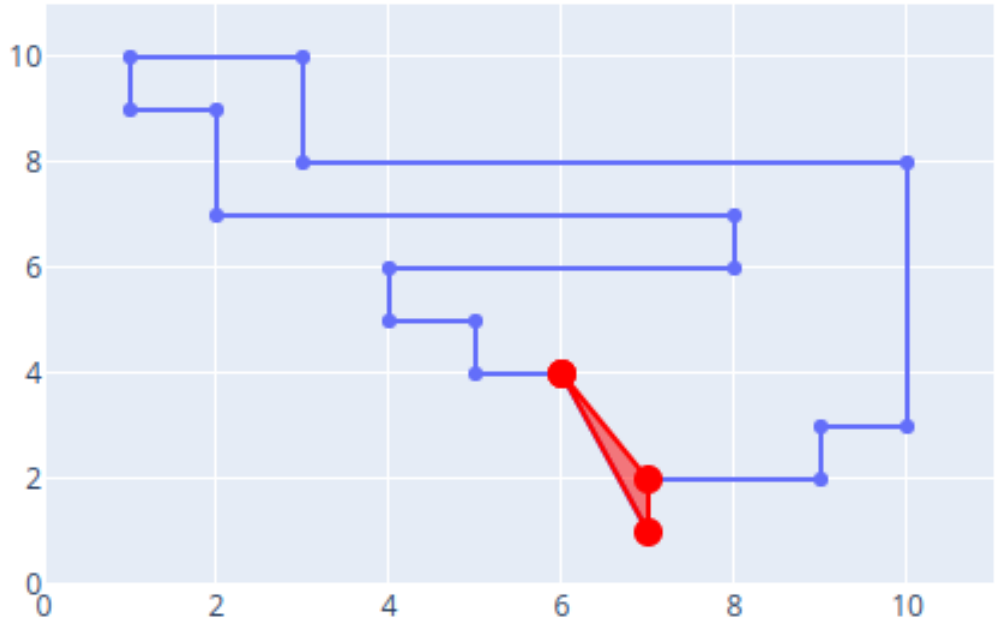


Figura 3: A possível orelha passou nos testes e foi confirmada como uma orelha válida. Assim, ela foi identificada como um dos triângulos e removida do polígono para a continuação do processo.

O algoritmo tem, no pior caso, complexidade quadrática, uma vez que para as verificações das orelhas pode ser necessário passar por todos os vértices do polígono. A saída do algoritmo é o polígono triangularizado conforme ilustrado na Figura 1.

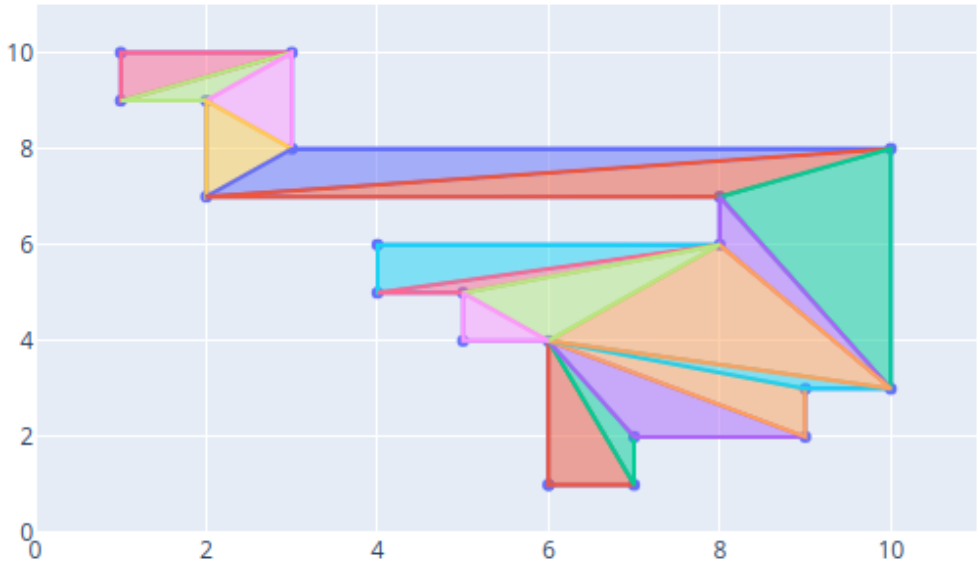


Figura 4: Exemplo de triangularização final

Após realizar a triangulação do polígono, temos uma solução simples para o problema: podemos colocar uma câmera no centro de cada triângulo. De acordo com o teorema que afirma: "Todo polígono simples possui pelo menos uma triangulação, e toda triangulação de um polígono simples com n vértices consiste em exatamente n-2 triângulos", precisaríamos de n-2 câmeras com essa abordagem.

Embora esse número pareça elevado, podemos reduzi-lo utilizando uma lógica simples. Se posicionarmos as câmeras nas diagonais, podemos cobrir mais de um triângulo com a mesma câmera, necessitando assim de apenas n/2 câmeras. Além disso, se colocarmos as câmeras nos vértices, cada câmera poderia proteger até 3 triângulos, reduzindo o total para n/3 câmeras. A questão agora é como escolher esses vértices, e para isso, utilizaremos um algoritmo de coloração.

### 3.2 Coloração

A ideia agora é encontrar uma forma de selecionar os vértices que serão utilizados como posições para as câmeras, garantindo que cada triângulo tenha pelo menos um de seus vértices escolhido. Podemos transformar esse problema em um de coloração. Basicamente, se conseguirmos colorir o grafo formado pelos vértices, arestas e diagonais do polígono usando exatamente 3 cores, garantimos que os vértices de cada triângulo não possuam a mesma cor (pois estão conectados) e que os triângulos terão as 3 cores em seus vértices (já que temos apenas 3 cores e elas não podem se repetir). Dessa forma, precisamos apenas escolher uma das cores e posicionar as câmeras nos vértices dessa cor, garantindo que todos os triângulos sejam cobertos, pois todos os triângulos terão um vértice dessa cor.

O problema de 3-Coloração é NP-Completo, o que torna necessário encontrar uma abordagem alternativa para resolvê-lo. Nesse sentido surge a seguinte ideia de utilizar a triangularização para fazer essa coloração. Para isso primeiro precisamos de um grafo dual ao formado pelos vértices, arestas e diagonais do polígono montado da seguinte forma:

- Cada triangulo do polígono passa a ser um vértice.
- Aresta passa a ser se dois triângulos (vértices) compartilham ou não uma diagonal.

```python
def create_dual_graph(triangles):
    G = nx.Graph()

    trianglesIndexMap = {i: tuple(tuple(vertex) for vertex in triangle)
       for i, triangle in enumerate(triangles)}

    for i, triangle in trianglesIndexMap.items():
        for j in range(i + 1, len(triangles)):
            diagionalVertices = set(triangle).intersection(set(
               trianglesIndexMap[j]))
            if len(diagionalVertices) == 2:
                G.add_edge(i, j)

    return G, trianglesIndexMap
```

Listing 5: Código em Python para montar grafo dual

Com o grafo dual em mãos, a ideia é realizar uma busca em profundidade nesse novo grafo. Para cada vértice visitado, escolhemos exatamente uma cor para cada vértice do triângulo correspondente no polígono original. Essa escolha é mantida conforme avançamos na busca, garantindo que os vértices do polígono original não recebam mais de uma cor e que suas cores não sejam alteradas. Dessa forma, se um vértice do grafo dual pertence a dois triângulos, sua cor é determinada quando visitamos o primeiro triângulo, e ao visitar o segundo triângulo, os outros vértices devem ser coloridos de modo a não usar essa mesma cor.

```python
def color_vertices(triangles):
    dual_graph, trianglesIndexMap = create_dual_graph(triangles)
    vertexColors = {}
    colorNames = ['purple', 'green', 'blue']
    visited = set()

    def dfs(index):
        visited.add(index)
        triangle = trianglesIndexMap[index]
        colorSet = set(colorNames)

        for vertex in triangle:
            if vertex in vertexColors:
                colorSet.discard(vertexColors[vertex])

        for vertex in triangle:
            if vertex not in vertexColors:
                vertexColors[vertex] = colorSet.pop()

        for neighbor in dual_graph.neighbors(index):
            if neighbor not in visited:
```

```
                dfs(neighbor)

    first = list(dual_graph.nodes)[0]
    dfs(first)

    return vertexColors
```

Listing 6: Código em Python para a coloração

A seguir, é apresentado um exemplo que ilustra o funcionamento do algoritmo:

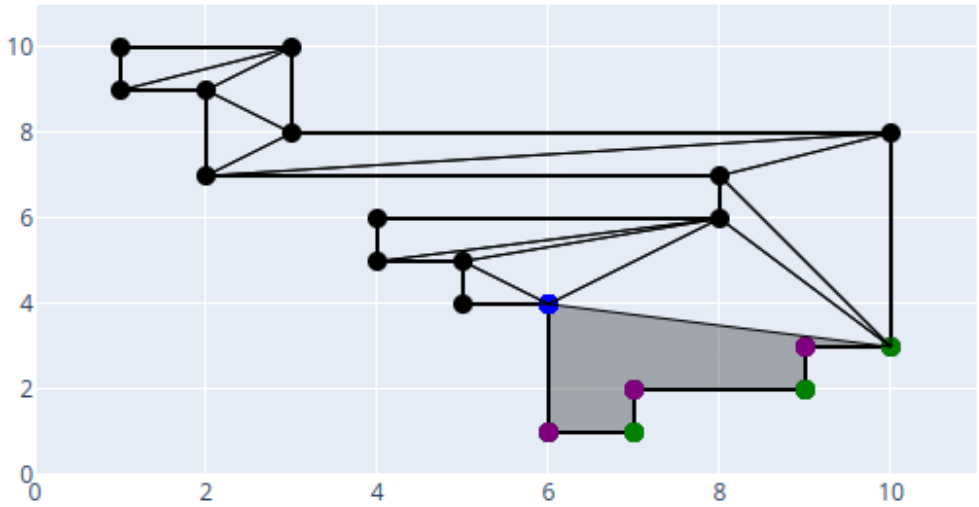


Figura 5: Situação antes de ir para o próximo vértice na busca em profundidade do grafo dual no qual os triângulos são vértices

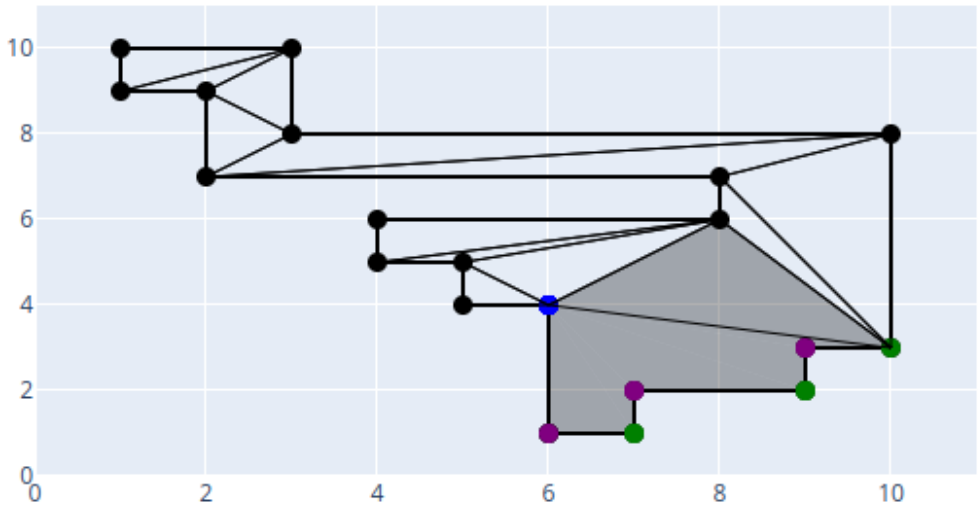


Figura 6: A busca avança para mais um vértice no grafo dual, onde os triângulos são representados como vértices.

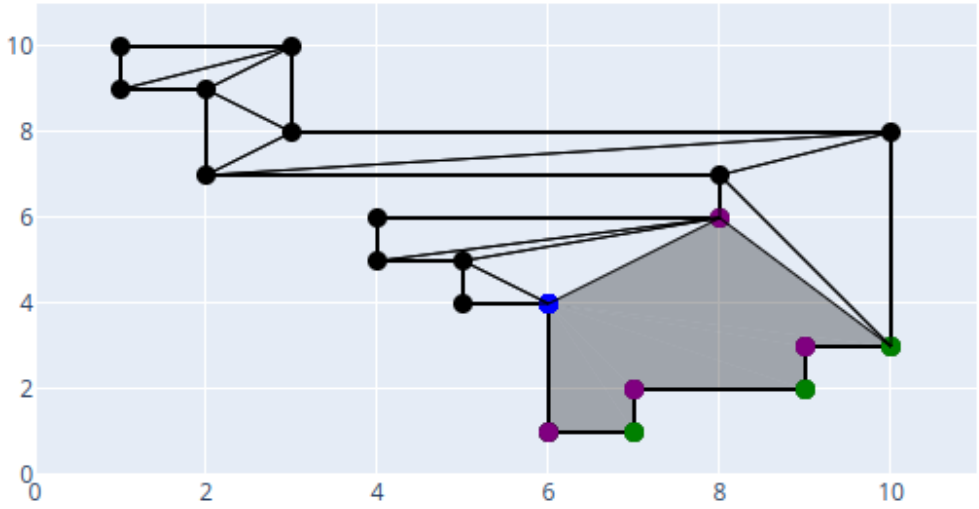


Figura 7: Após encontrar um vértice no grafo dual, os vértices do triângulo correspondente no polígono original que ainda não foram coloridos recebem as cores que ainda estão disponíveis, e o processo se repete

O algoritmo tem, no pior caso, complexidade (n sei). A saída do algoritmo é o polígono com seu vértices 3-coloridos, conforme ilustrado na Figura 8.

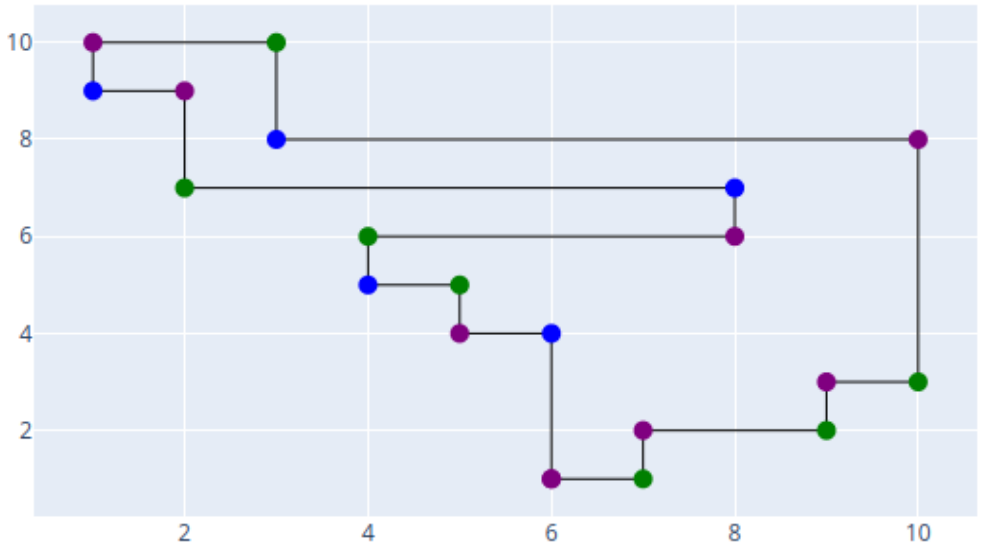


Figura 8: Exemplo de coloração final

### 3.3 Escolha das câmeras

Com essa coloração temos uma forma fácil de escolher os vértices a serem escolhidos para o posicionamento das câmeras. Para isso escolhemos a cor menos frequente entre os vértices e selecionamos aqueles que a possuem como câmera.

```
def minimum_camera_positions(triangles):
    vertexColors = color_vertices(triangles)

    colorPartitions = {'purple': [], 'green': [], 'blue': []}
    for vertex, color in vertexColors.items():
        colorPartitions[color].append(vertex)

    smallestPartition = min(colorPartitions.values(), key=len)
    cameraPositions = list(set(smallestPartition))

    return cameraPositions
```

Listing 7: Código em Python para a escolha das câmeras

Como usamos um vértice por triangulo, usaremos no máximo n/3 câmeras no total. A saída do algoritmo é o polígono com suas câmeras posicionadas conforme ilustrado na Figura 9.

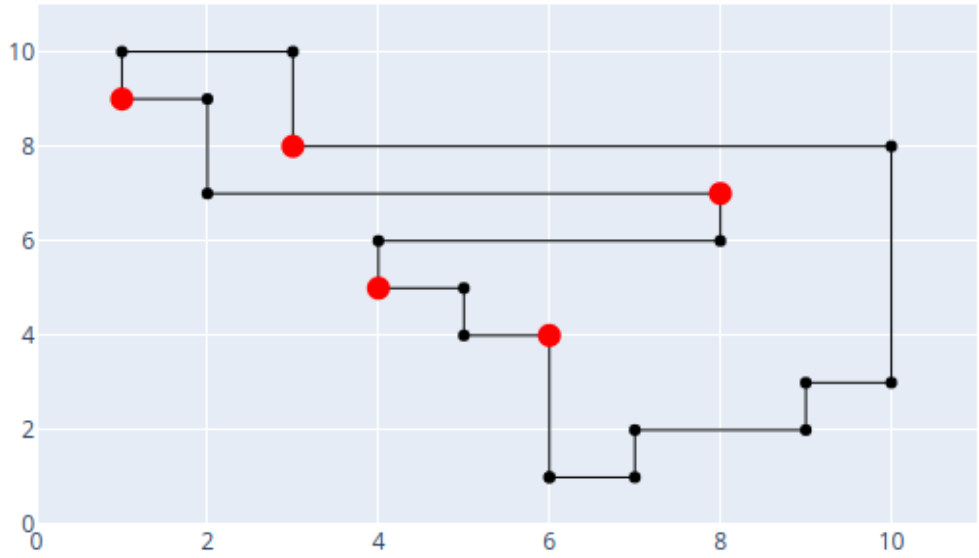


Figura 9: Exemplo de posicionamento de câmeras final

## 4 Forma de uso da nossa ferramenta

Para utilizar nossa ferramenta no GitHub Pages, basta selecionar o tamanho do polígono que deseja visualizar o processo e apertar os botões correspondentes para ver cada etapa sendo executada de forma dinâmica.

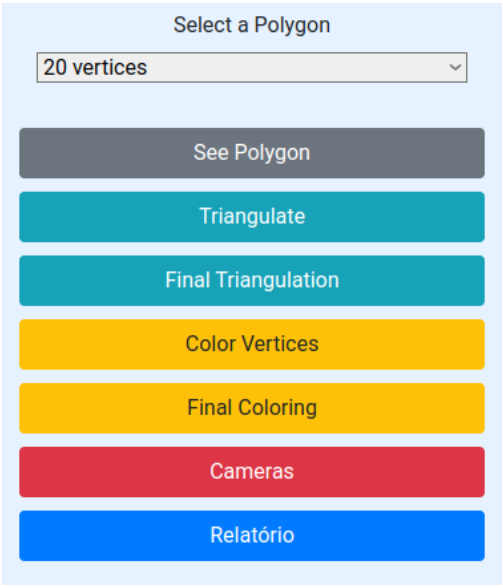


Figura 10: Interface do sistema

# 5 Conclusão

Dada a implementação da ferramenta que resolve o problema da Galeria de Arte, vê-se que a compreensão e resolução do problema requerem um entendimento considerável de geometria computacional. A partir desse pré-requisito, fez-se possível a construção de uma ferramenta que, dado um polígono como instância do problema, utilize o algoritmo de "ear-clipping"para a triangulação do polígono, colora os vértices a fim de encontrar um posicionamento ideal das câmeras e, assim, dê uma solução que garanta a cobertura de todos os pontos internos.

Além disso, foi desenvolvida uma página web que exemplifica o funcionamento passo-a-passo do algoritmo para polígonos predefinidos, que podem ser escolhidos pelo usuário de acordo com seu número de vértices. Tal página oferece a estudantes e outros interessados pelo problema um meio de entender visualmente como o algoritmo que o resolve funciona.

Com isso, é possível concluir que a construção de tal ferramenta contribuiu não apenas para a fixação dos conceitos de geometria computacional e algoritmos apresentados em sala, mas também ofereceu uma compreensão prática do problema da Galeria de Arte. Não obstante, a criação de uma página que permite a interação com essa resolução viabiliza um aprendizado considerável acerca dos conceitos explorados pelo problema.

# Referências


[1] Plotly. (n.d.). Animations. Disponível em: `https://plotly.com/python/animations/`

[2] Art Gallery Problem Library. Disponível em: `https://www.ic.unicamp.br/~cid/Problem-instances/Art-Gallery/AGPVG/index.html`

[3] Desafio da Galeria de Arte: Quantos Guardas Precisa. Disponível em: `https://www.youtube.com/watch?v=_i4o6DihoCw`