Classificação de Polígonos e Teste de Contenção de Pontos

Autor: Ludwig Aumann

Introdução

Este artigo apresenta um conjunto de algoritmos para resolver dois problemas fundamentais em geometria computacional:

1. **Classificação de polígonos**: determinar se um polígono é simples ou não, e se um polígono simples é convexo ou não.
2. **Teste de contenção de pontos**: determinar quais polígonos contêm cada ponto de um conjunto dado.

Definições

Antes de descrever os algoritmos, definimos alguns conceitos fundamentais:

- **Polígono**: sequência de pontos no plano conectados por segmentos de reta (arestas) formando um circuito fechado.
- **Polígono simples**: polígono sem auto-interseções, ou seja, nenhuma aresta cruza outra aresta.
- **Polígono convexo**: polígono simples onde todos os ângulos internos são menores ou iguais a 180 graus.
- **Orientação**: três pontos no plano podem formar uma curva horária, anti-horária ou colinear.

Algoritmos

1. Cálculo de Orientação

Este algoritmo fundamental determina a orientação formada por três pontos (***p, q, r***).

```
enum class Orientation {
    COLINEAR,    // pontos são colineares
    HORARIO,     // sentido horário (clockwise)
    ANTIHORARIO  // sentido anti-horário (counter-clockwise)
};

Orientation orientation(Point p, Point q, Point r) {
    // calcula o produto vetorial 2D
    long long val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y

    if (val == 0) return Orientation::COLINEAR;
    return (val > 0) ? Orientation::ANTIHORARIO : Orientation::HORARIO;
}
```

Se interpretarmos (***p, q, r***) como vértices de um triângulo, o valor absoluto do **produto vetorial 2D** (também chamado de determinante em duas dimensões) corresponde a **duas vezes a área** desse triângulo. Formalmente, se temos os pontos $p=(p_x, p_y)$, $q=(q_x, q_y)$ e $r=(r_x, r_y)$, então o determinante:

`(q_x - p_x)*(r_y - p_y) - (q_y - p_y)*(r_x - p_x)`

é igual ao dobro da área do triângulo ***p q r***. Assim o cálculo se relaciona ao **ângulo** formado pelos pontos, pois a maneira como eles se dispõem no plano (em sentido horário, anti-horário ou colinear) se reflete no sinal do determinante. O **sinal** desse determinante (positivo, negativo ou zero) determina a **orientação** do triângulo no plano:

- Positivo: anti-horário
- Negativo: horário
- Zero: colinear

Este algoritmo tem complexidade temporal O(1) e é utilizado como base para os demais algoritmos.

2. Interseção de Segmentos

Para verificar se dois segmentos de reta se interceptam, utiliza-se o seguinte algoritmo:

```
bool do_intersect(const Point& p1, const Point& q1, const Point& p2, con
```

O algoritmo opera em duas etapas:

1. **Teste Geral de Interseção:**

   São calculadas as orientações de quatro grupos de três pontos usando a função `orientation`:

- `o1 = orientation(p1, q1, p2)`
- `o2 = orientation(p1, q1, q2)`
- `o3 = orientation(p2, q2, p1)`
- `o4 = orientation(p2, q2, q1)`

A ideia é que, para dois segmentos `p1q1` e `p2q2` se cruzarem, os pontos `p2` e `q2` devem estar em lados opostos da reta definida por `p1q1` — isto é, `o1` e `o2` devem ter sinais opostos. De forma análoga, `p1` e `q1` devem estar em lados opostos da reta definida por `p2q2` (logo, `o3` e `o4` diferem). Se essas duas condições forem satisfeitas, há interseção.

2. **Casos Especiais (Colinearidade):**

   Quando alguma das orientações é colinear (ou seja, igual a zero), os pontos não formam uma "curva" definida em relação à reta e é preciso checar se o ponto colinear está realmente contido no segmento. Essa verificação é feita pela função `on_segment`, avaliando os seguintes casos:

   - Se `o1` é colinear e `p2` está entre `p1` e `q1`.
   - Se `o2` é colinear e `q2` está entre `p1` e `q1`.
   - Se `o3` é colinear e `p1` está entre `p2` e `q2`.
   - Se `o4` é colinear e `q1` está entre `p2` e `q2`.

Se qualquer uma dessas condições for verdadeira, conclui-se que os segmentos se intersectam.

Complexidade

Como o algoritmo realiza um número fixo de operações (cálculo de orientações e comparações), sua complexidade temporal é **O(1)**.

3. Verificação de Polígono Simples

Um polígono é simples se não tiver auto-interseções:

```
bool is_simple(const Polygon& poly)
```

O algoritmo verifica se existem interseções entre quaisquer pares de arestas não adjacentes no polígono. Para cada aresta, comparamos com todas as outras arestas não adjacentes e verificamos se há interseção utilizando a função `do_intersect()`.

A complexidade é $O(n^2)$, onde n é o número de vértices do polígono.

4. Verificação de Convexidade

Um polígono simples é convexo se todos os seus ângulos internos forem menores ou iguais a 180 graus:

```
bool is_convex(const Polygon& poly)
```

A implementação encontra a primeira orientação não-colinear e verifica se todas as demais orientações entre três pontos consecutivos são consistentes (iguais à primeira ou colineares). Se encontrarmos alguma orientação diferente, o polígono não é convexo.

A complexidade é O(n), onde n é o número de vértices do polígono.

5. Teste de Contenção de Ponto em Polígono

Para verificar se um ponto está dentro de um polígono, implementamos o algoritmo de ray casting:

```
bool is_inside(const Point& point, const Polygon& polygon)
```

Este algoritmo utiliza o método de *ray casting* (método da paridade) para determinar se um ponto está contido em um polígono simples. A ideia central é a seguinte:

1. **Verificação Inicial:**

    - Primeiramente, o algoritmo checa se o ponto coincide com algum vértice ou se está sobre uma das arestas do polígono. Nestes casos, o ponto é considerado **dentro**.

2. **Lançamento do Raio Horizontal:**

    - Caso o ponto não esteja diretamente sobre um vértice ou aresta, lança-se um raio horizontal (para a direita) a partir do ponto.
    - Ao longo desse raio, o algoritmo conta quantas vezes o mesmo cruza as arestas do polígono.

3. **Princípio da Paridade:**

    - Se o número de interseções for **ímpar**, o ponto está **dentro** do polígono.
    - Se o número de interseções for **par**, o ponto está **fora**.

Essa abordagem baseia-se no fato de que, para um ponto interno, o raio saindo dele deverá cruzar as fronteiras do polígono um número ímpar de vezes antes de se estender indefinidamente. Como o algoritmo avalia cada uma das n arestas, sua complexidade temporal é **O(n)**.

A complexidade temporal é $O(n)$, onde n é o número de vértices do polígono.

Análise de Complexidade

- Leitura dos dados: $O(m + n)$, onde m é o número total de vértices de todos os polígonos e n é o número de pontos
- Classificação dos polígonos: $O(m^2)$, dominada pela verificação de polígono simples
- Teste de contenção: $O(m \times n)$, pois testamos cada um dos n pontos em cada um dos m polígonos
- Complexidade total: $O(m^2 + m \times n)$

Considerações sobre Precisão Numérica

Os cálculos geométricos são sensíveis a problemas de precisão numérica. Para minimizar erros de arredondamento, utilizamos o tipo `long long` para as coordenadas e para os cálculos intermediários, evitando assim imprecisões em operações com números de ponto flutuante.

Conclusão

Os algoritmos implementados resolvem os problemas de classificação de polígonos e teste de contenção de pontos. As técnicas utilizadas são fundamentais em geometria computacional e podem ser aplicadas em diversos contextos práticos.

Referências

1. de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer-Verlag.
2. O'Rourke, J. (1998). *Computational Geometry in C*. Cambridge University Press.
3. Preparata, F. P., & Shamos, M. I. (1985). *Computational Geometry: An Introduction*. Springer-Verlag.