Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Algoritmos para realização de operações Booleanas em polígonos

Rafael Vieira de Carvalho

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisora: Profª. Drª. Cristina G. Fernandes

São Paulo
2022

Agradecimentos

To ask the right question is harder than to answer it.
— Georg Cantor

Queria agradecer a todos que me apoiaram nessa tarefa. Amigos, família e aos professores do Instituto de Matemática e Estatística em especial minha orientadora. Queria agradecer também Alexis Sakurai Landgraf Carvalho e Victor Sanches Portella por disponibilizar a plataforma com a qual consegui gerar animações dos algoritmos.

Resumo

Rafael Vieira de Carvalho. **Algoritmos para realização de operações Booleanas em polígonos**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Realizar operações Booleanas em polígonos pode não parecer, mas é extremamente importante para alguns algoritmos e softwares. Elaborar um algoritmo para encontrar a região resultante da operação entre dois polígonos não é uma tarefa trivial. Nesse trabalho de conclusão de curso, estudamos e implementamos alguns algoritmos para esse problema e fizemos alguns experimentos computacionais para compará-los.

Palavras-chave: Polígonos. Operações Booleanas. Clipping. Geometria Computacional.

Abstract

Rafael Vieira de Carvalho. **Title of the document: *a subtitle***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

To perform Boolean operation on polygons is really important for some algorithms and softwares. To elaborate an algorithm that solves this problem is not so easy. In this undergraduate thesis, we studied and implemented some algorithms to solve this problem and we run some computational experiments to compare them.

Keywords: Polygons. Boolean Operations. Clipping. Computational Geometry.

Lista de Figuras

Lista de Tabelas

Lista de Programas

Sumário

Capítulo 1

Introdução

Geralmente operações Boolenas são um tópico estudado por outras áreas como álgebra, teoria dos conjuntos, etc. Embora não seja um tópico corriqueiro para área da geometria computacional, tais operações têm muita aplicabilidade em diversas áreas. Na computação gráfica, por exemplo, para determinar quais objetos são visíveis na cena, são usadas operações de interseção e diferença.

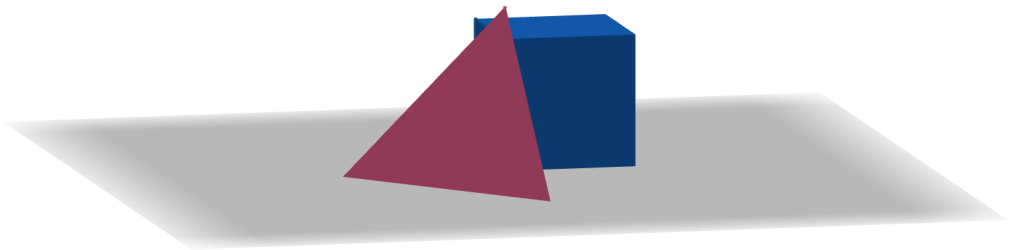


Figura 1.1: *Pirâmide rosa cobrindo um cubo azul. Note que a área oclusa pela pirâmide não precisa ser renderizada.*

Além do uso de operações Booleanas na etapa de renderização de objeto em cena, o uso em softwares como CAD (Computer-aided design) e GIS (Geographic information system) é extremamente importante. No CAD é muito comum querermos construir peças através de outras peças já existentes como mostra a Figura 1.2. Já no GIS, é comum realizarmos consultas para saber alguma localização no mapa. Considere o seguinte exemplo: imagine que temos dois mapas demográficos, um de pessoas que moram em cidades e outro de pessoas com câncer. Veja a Figura 1.3. Por meio de operações Booleanas, podemos visualizar, por exemplo, onde se concentram as pessoas com câncer em determinada cidade.

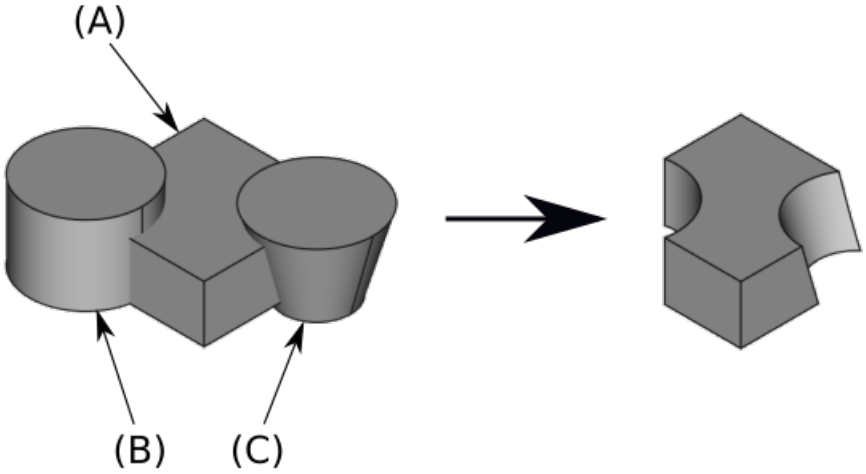


Figura 1.2: *Construção de uma peça no software CAD através das operações de diferença. Nesse caso, a operação é A − B − C. Imagem retirada de https://wiki.freecadweb.org/images/1/16/PartDesign_Boolean_example.png.*

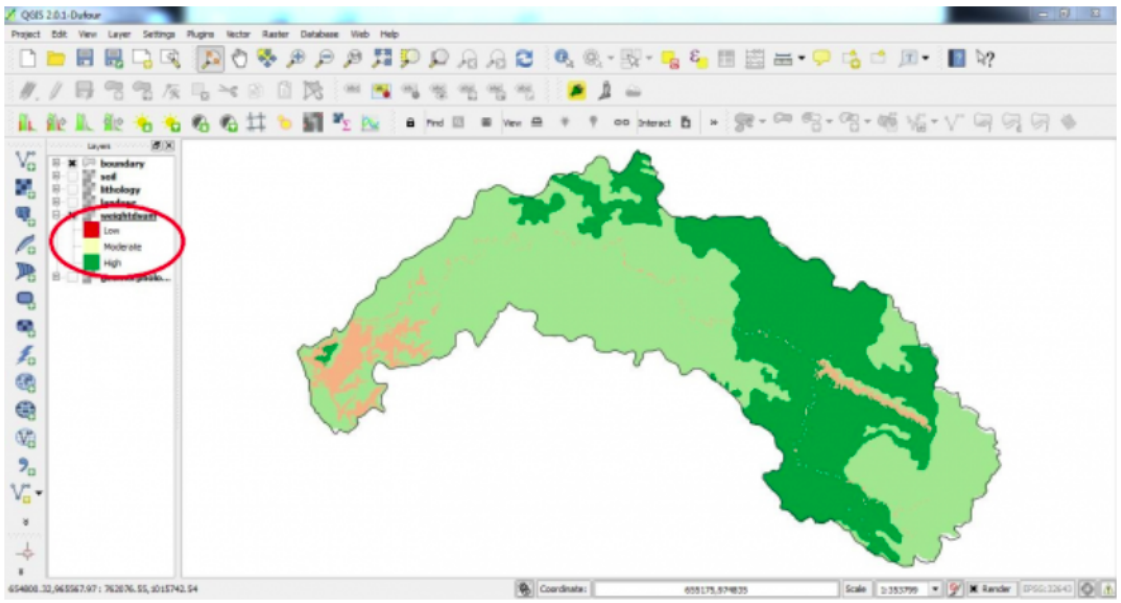


Figura 1.3: *Consulta no software GIS. Figura retirada de https://grindgis.com/software/qgis/raster-overlay-analysis-qgis.*

É com essa motivação que algoritmos para computar operações Booleanas foram surgindo. No começo esses algoritmos suportavam apenas a operação de interseção (comumente chamada de clipping na área de computação gráfica) e os polígonos que eram recebidos como entrada para tais algoritmos eram restritos. Por exemplo, só podiam ser convexos, não podiam se auto-intersectar, não eram permitidos buracos nos polígonos, etc. Hoje em dia, os algoritmos estão mais robustos e permitem polígonos genéricos.

Segue um breve histórico dos algoritmos conhecidos. O algoritmo de [Sutherland e Hodgman, 1974] era restrito a polígonos convexos e realizava apenas clipping. O algoritmo de [Weiler e Atherton, 1977] permitia polígonos côncavos e com buracos, mas não permitia polígonos que se auto-intersectassem e considerava apenas a operação de clipping.

Os algoritmos de [Vatti, 1992], [Greiner e Hormann, 1989] e [Weiler, 1980] foram os primeiros a lidar com polígonos arbitrários.

Nesse trabalho estudaremos os algoritmos de [Greiner e Hormann, 1989], [Foster *et al.*, 2019] e [Martinez *et al.*, 2013] e compararemos as suas performances. Com esse objetivo, cada capítulo será constituído por: explicação do algoritmo, explicação da estrutura de dados utilizada e análise assintótica do algoritmo em questão.

Definições e preliminares

Antes de iniciar o estudo sobre os algoritmos, precisa-se definir e delimitar quais os objetos geométricos com que trabalhar-se-á.

Curva poligonal

Uma *curva poligonal* P é definida por uma sequência $(p_1, p_2, \dots, p_n)$ de pontos no plano chamados de *vértices*. A curva P consiste nos segmentos $\overline{p_i p_{i+1}}$ entre cada par de pontos consecutivos da sequência. Esses segmentos são chamados de *arestas* [Wikipedia, 2021].

A curva poligonal P é *aberta* se $p_n \neq p_1$, é *fechada* se $p_n = p_1$ e é *simples* se não se autointersecta. Assim uma curva poligonal pode ser de quatro tipos: aberta simples, fechada simples, aberta não-simples e fechada não-simples.

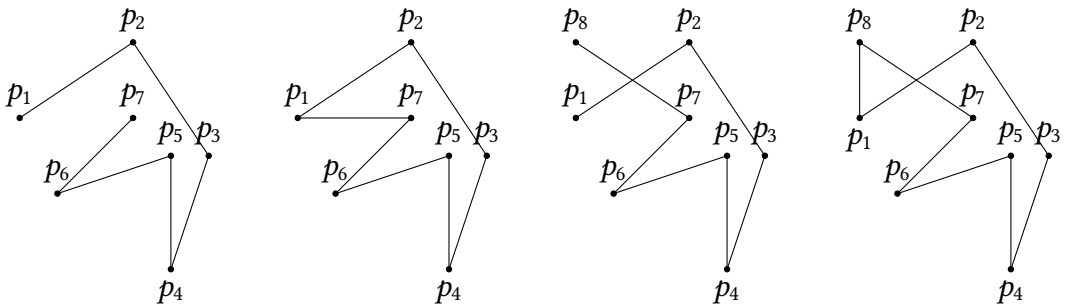


Figura 1.4: *Tipos de curvas poligonais: aberta simples, fechada simples, aberta não-simples e fechada não-simples.*

Polígono simples

Pelo Teorema de Jordan toda curva poligonal fechada simples divide o plano em duas regiões. A região limitada pela curva será chamada de *interna* e a outra, que é ilimitada, será chamada de *externa*. Um *polígono* é definido como a região interna de uma curva poligonal fechada simples. A curva poligonal fechada que determina P é chamada de *fronteira* de P. Veja a Figura 1.5.

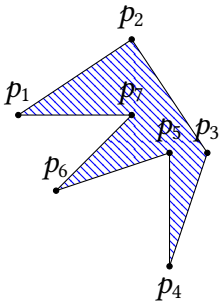
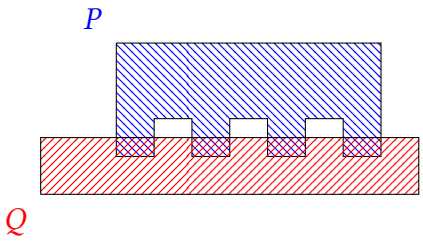


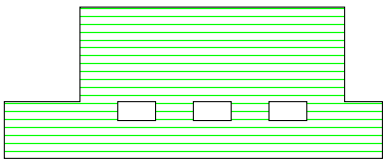
Figura 1.5: *A região hachurada de azul é o polígono definido pela curva $P = (p_1, p_2, \ldots, p_7)$ que é a fronteira de P.*

Operações Booleanas

O estudo de polígonos por vezes envolve operações Booleanas, como a união e a interseção de polígonos. Posto isso, é importante ressaltar que tanto a união quanto a interseção de dois polígonos distintos podem não resultar em um polígono. Uma outra operação de interesse, a diferença de polígonos, também pode não resultar em um polígono.

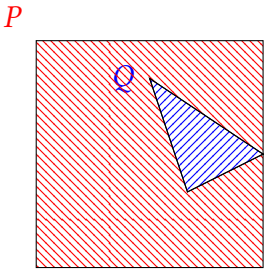


(a) *Dois polígonos distintos P e Q*

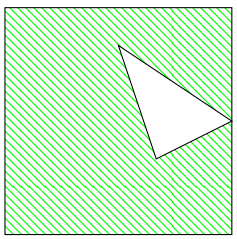


(b) *União $P \cup Q$*

(c) *interseção $P \cap Q$*



(d) *Dois polígono distintos P e Q*



(e) *Diferença $P \setminus Q$*

Figura 1.6: *Exemplos de operações Booleanas entre dois polígonos.*

Nesse contexto, é natural utilizar uma definição mais ampla de polígono, e polígonos que se enquadram na definição anterior serão chamados de *polígonos simples*.

Essa nova definição mais abrangente permite que o polígono seja definido por uma curva poligonal não necessariamente simples. Para identificar o que é o interior de uma curva fechada não-simples, e consequentemente o polígono determinado por ela, utiliza-se o conceito apresentado a seguir.

Winding number

Dada uma curva poligonal P e um ponto q pertencente a $\mathbb{R}^2 \setminus P$, o *winding number* $\omega(q, P)$ é o número de vezes que a curva P gira em torno de q. Mais formalmente, o winding number é definido pela seguinte expressão:

$$\omega(q, P) = \frac{1}{2\pi} \sum_{i=0}^{n} \theta_i$$

onde o valor absoluto de θ_i é o menor ângulo em torno de q formado pelos pontos p_i, q, p_{i+1}. Se os pontos p_i, q, p_{i+1} estiverem em sentido anti-horário então θ_i é positivo, caso contrário θ_i é negativo. Refere-se a esse número como o ângulo (sinalizado) do segmento $\overline{p_i p_{i+1}}$ em relação ao ponto q.

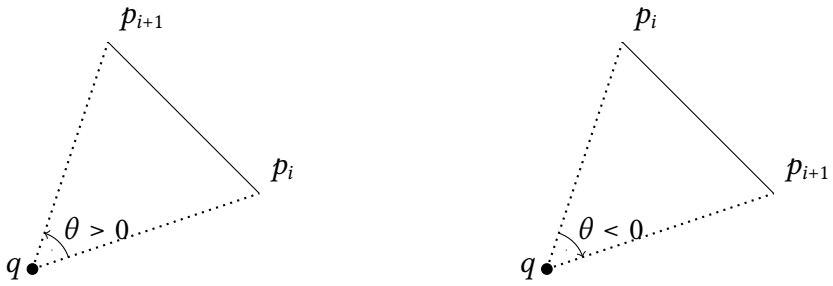


Figura 1.7: *Ângulo do segmento $\overline{P_i P_{i+1}}$ em relação a q.*

Uma propriedade importante do winding number é que cada uma das regiões delimitadas pela curva P possui winding number constante. Olhe os exemplos na Figura 1.8.

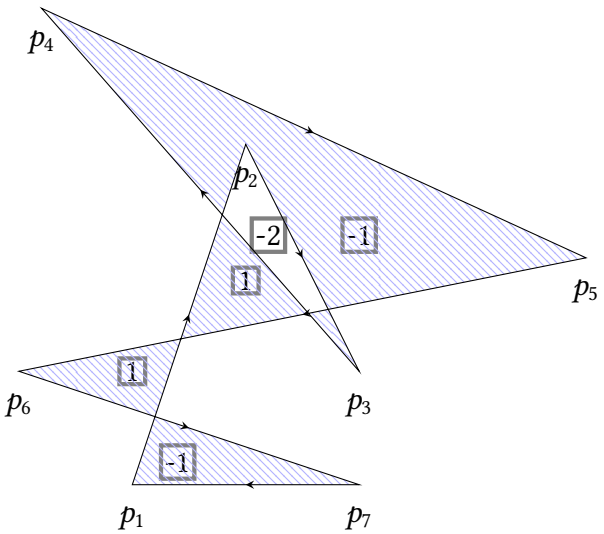


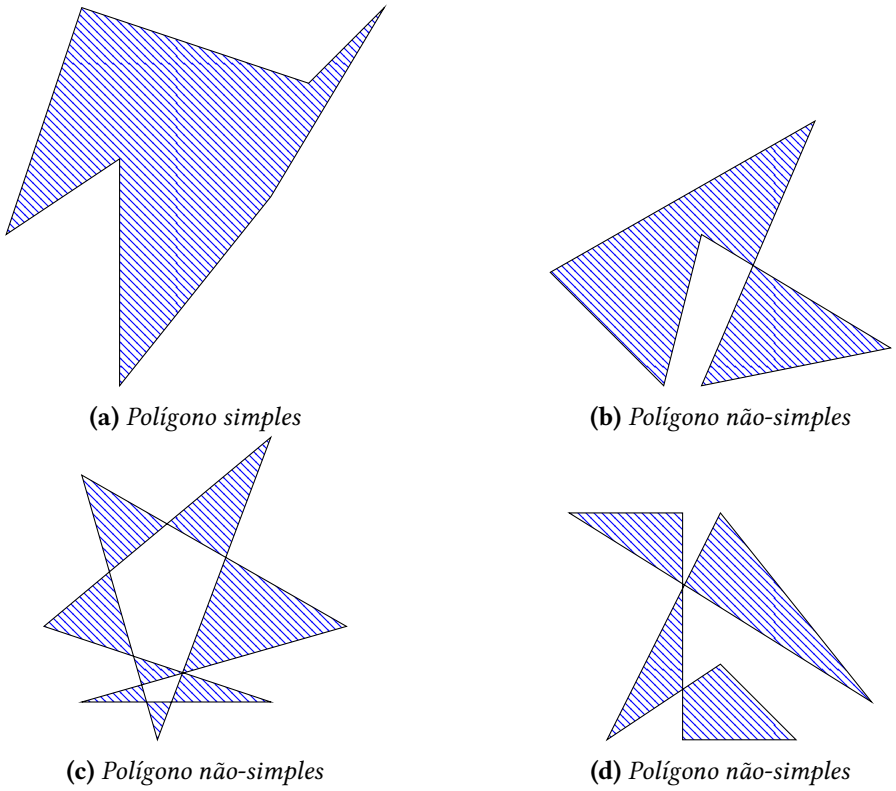
Figura 1.8: *Winding number de cada região delimitada por uma curva poligonal fechada.*

Definição mais abrangente de polígono

Dada uma curva poligonal fechada P, defini-se um polígono Q como

$$Q = \{q \in \mathbb{R}^2 \setminus P \,:\, \omega(q, P) = 2n + 1,\ n \in \mathbb{Z}\} \cup P.$$

Ou seja, um polígono é a união de todas as regiões cujo winding number é ímpar em relação à curva P bem como a curva poligonal P. Muitas vezes, abusa-se da notação e refere-se à curva poligonal P como se fosse o polígono definido por P.



(a) *Polígono simples*

(b) *Polígono não-simples*

(c) *Polígono não-simples*

(d) *Polígono não-simples*

Figura 1.9: *Exemplos de polígonos.*

Cálculo do winding number

A primeira ideia de um algoritmo para calcular o winding number de um ponto q em relação a uma curva poligonal $P = (p_1, \ldots, p_n)$ é: percorrer os vértices de P; calcular os ângulos formados entre p_i, q, p_{i+1} com alguma função trigonométrica; somar esses ângulos e dividir por 2π. Embora esse algoritmo consuma tempo linear no número de vértices de P, realizar cálculo de funções trigonométricas é computacionalmente caro. Para contornar isso, uma outra maneira de se calcular o winding number é sugerida a seguir.

Ray casting ou even-odd rule

Assumindo posição geral, para se decidir a pertinência de um ponto q a um polígono P, se lança um raio para a direita a partir do ponto q e o número de vezes que esse raio se intersecta com a fronteira de P indica se o ponto pertence ou não pertence ao polígono P. Desconsiderando os casos degenerados, se o número de interseções for ímpar, então o ponto pertence ao polígono P, caso contrário não pertence.

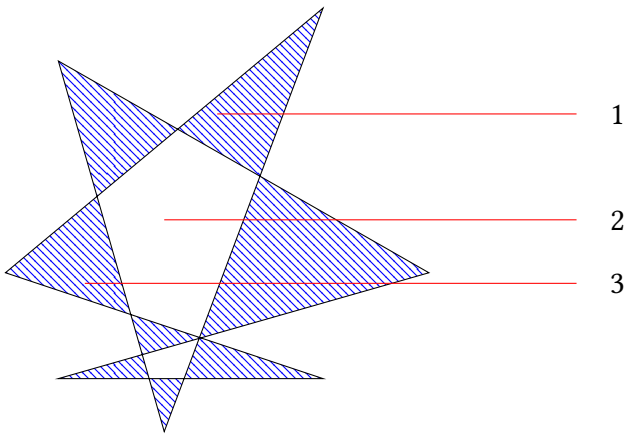


Figura 1.10: *Ray casting das regiões do polígono com as respectivas vezes que o correspondente raio intersecta com a curva poligonal.*

Em geral, o número de interseções do raio com a fronteira de P é a soma do número de interseções do raio com as arestas de P. No entanto, há casos degenerados onde isso não é verdade, e há casos também em que o raio intersecta a fronteira em um número infinito de pontos.

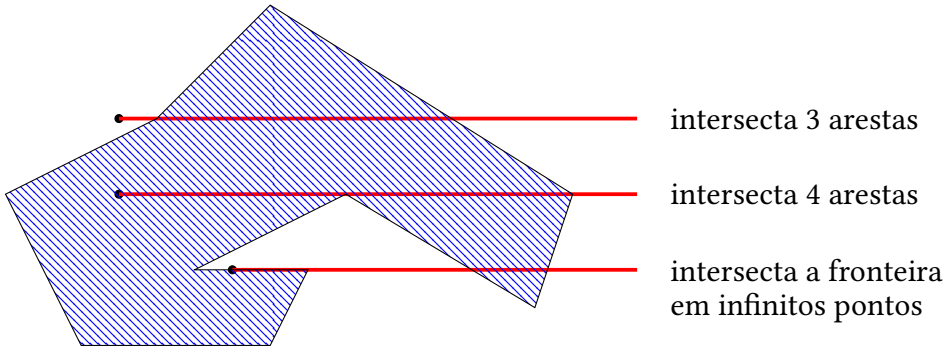


Figura 1.11: *Casos degenerados.*

Para resolver esses casos, realizaremos as seguintes considerações. A primeira é considerar as arestas semi-abertas, isto é, a aresta é fechada em um extremo e aberta em outro. Para saber qual extremo será aberto e qual será fechado segue a seguinte definição. Um extremo é *inferior* se está estritamente abaixo do outro extremo do segmento. Um extremo é *superior* se está estritamente acima do outro extremo do segmento. O extremo que consideraremos fechado é o inferior, e o extremo superior será aberto; essa é chamada a chamada *interpretação 1*. Caso a aresta seja paralela ao eixo X, consideraremos como extremo inferior o extremo com menor X-coordenada, e nenhuma interseção é contabilizada.

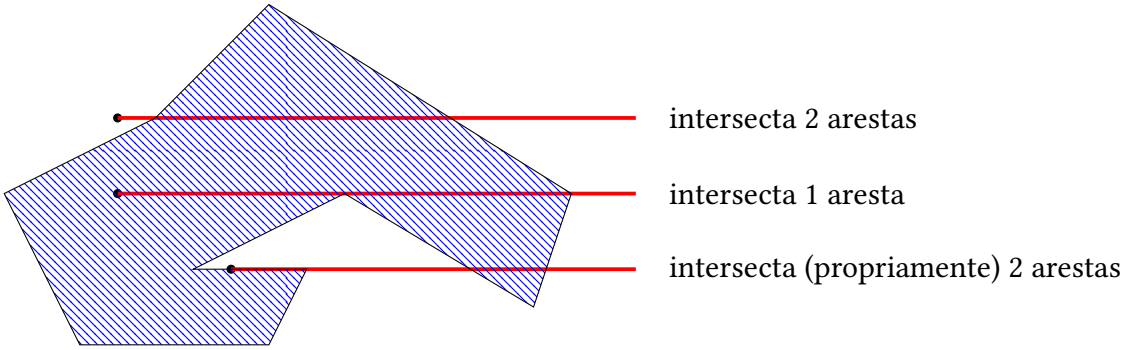


Figura 1.12: *Os dois primeiros casos degenerados são resolvidos considerando as arestas fechadas no extremo inferior e abertas no superior. O terceiro caso, em que o ponto está na fronteira, ainda leva a uma resposta errada.*

A consideração feita anteriormente ainda não resolve todos os casos degenerados. Quando q está em alguns pontos da fronteira de P, a estratégia acima ainda erra. Para resolver esse problema, faremos duas interpretações. A primeira é a que já estávamos realizando, ou seja, lançar um raio na direção $x = +\infty$ e considerar as arestas semi-abertas, com o extremo inferior fechado e o superior aberto. A segunda interpretação trata-se de lançar um raio para $x = -\infty$ e considerar as arestas semi-abertas, porém fechadas no extremo superior e abertas no extremo inferior; essa é a chamada *interpretação 2*. Unindo essas duas interpretações, os casos degenerados são resolvidos exceto quando q é alguns dos vértices do polígono.

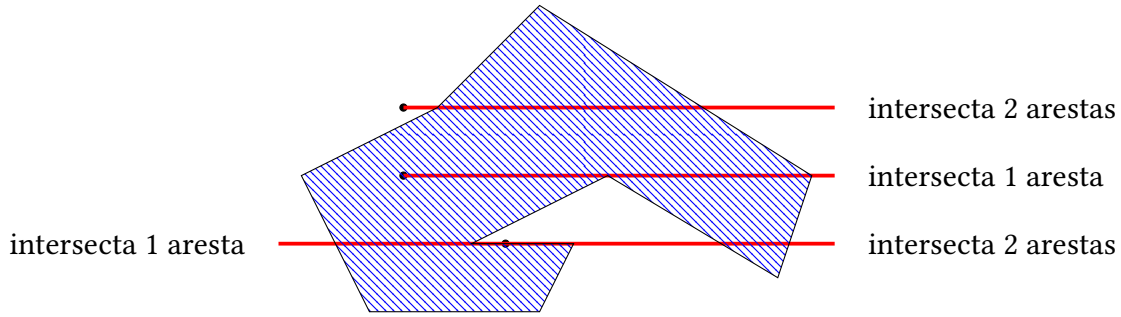
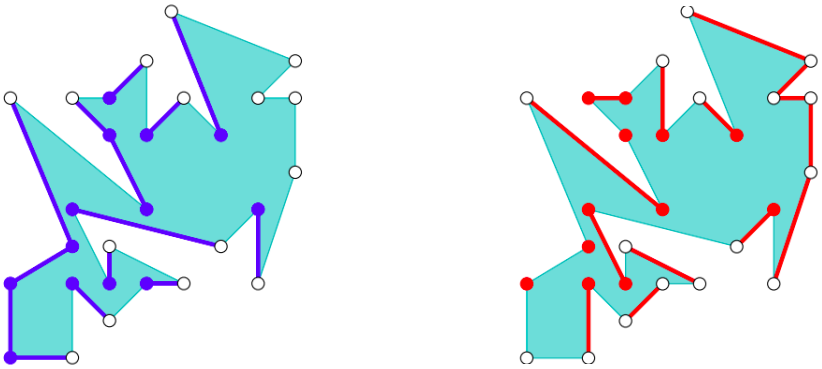


Figura 1.13: *Resultado final após as duas considerações.*

As duas interpretações identificam corretamente pontos do interior do polígono. A interpretação 1 acerta também nos pontos nas arestas à esquerda ou a baixo do polígono, enquanto a interpretação 2 acerta nos pontos nas arestas à direita ou acima do polígono. Observe que há vértices do polígono que não são identificados como pertencentes ao polígono pelas duas interpretações. Veja a Figura 1.14.

Dessa forma podemos determinar se um ponto pertence ou não a um polígono sem nenhuma função computacionalmente cara. No Programa 1.2 está o pseudocódigo para calcular a pertinência do ponto q em relação ao polígono P, que usa a rotina auxiliar mostrada no Programa 1.1. O parâmetro n é o número de vértices de P. Perceba que as variáveis c e d desempenham o papel das duas interpretações do winding number mencionadas anteriormente.



(a) *Interpretação 1* **(b)** *Interpretação 2*

Figura 1.14: *Região hachurada indica os pontos do polígono que são identificados como do polígono segundo a interpretação usada.*

Programa 1.1 Cálculo da interseção entre o raio jogado e a aresta.

```
1  FUNCTION Intersects(q, p, p⁻)    ▷ pré condição p.Y ≠ p⁻.Y
2    ▷ retorna a coordenada X do ponto de interseção entre o raio que parte de q.X e a aresta p⁻p.
3  return ((q.Y − p⁻.Y) * (p.X − p⁻.X))/(p.Y − p⁻.Y) + p⁻.X
```

Programa 1.2 Ray casting.

```
1  FUNCTION rayCasting(q, P)   ▷ retorna verdadeiro se q pertence P, falso caso contrário.
2  c ⟵ 0
3  d ⟵ 0
4  p ⟵ 0
5  for each vertex p_i ∈ polygon P do
6  if p_i.X = q.X and p_i.Y = q.Y
7  return True   ▷ q é vértice de P
8  testC ⟵ (p_i.Y > q.Y) ≠ (p_{i−1}.Y > q.Y)   ▷ se i = 1, então p_{i−1} equivale p_n
9  testD ⟵ (p_i.Y < q.Y) ≠ (p_{i−1}.Y < q.Y)
10 if testC or testD
11 x ⟵ Intersects(q, p_i, p_{i−1})
12 if testC and x > q.X   ▷ intersecta o lado positivo do raio
13 c ⟵ c + 1
14 if testD and x < q.X   ▷ intersecta o lado negativo do raio
15 d ⟵ d + 1
16 if c mod 2 ≠ d mod 2
17 return True   ▷ q está numa aresta de P
18 if c mod 2 = 1
19 return True   ▷ q está no interior de P
20 return False
```

A rotina `Intersects` consome tempo $O(1)$. Do pseudocódigo, é fácil notar que a função `rayCasting` consome tempo $O(n)$, onde n é o número de vértices do polígono.

Polígonos estudados

Começamos essa discussão de winding number para delimitar o escopo dos polígonos que os algoritmos receberão como entrada assim como delimitar os polígonos resultantes, encontrados pelos algoritmos. Posto isso, os algoritmos que serão apresentados trataram de polígonos simples e polígonos que se auto-intersectam.

Capítulo 2

Algoritmo de Greiner-Hormann

Como explicado anteriormente na introdução, os algoritmos de [Vatti, 1992], [Weiler, 1980], [Greiner e Hormann, 1989] e [Martinez *et al.*, 2013]conseguem lidar com operações Booleanas em polígonos genéricos em tempo razoável. Dentre os três algoritmos, o de Greiner-Hormann é o que se destaca por conta de sua simplicidade e, de acordo com o estudo experimental dos autores, de sua performance mais rápida comparada aos outros. O estudo comentado anteriormente compara apenas o algoritmo de Greiner-Hormann com o de Vatti.

Para simplificar o estudo do algoritmo, começa-se explicando o funcionamento para a operação de interseção (ou clipping) e depois para as demais operações.

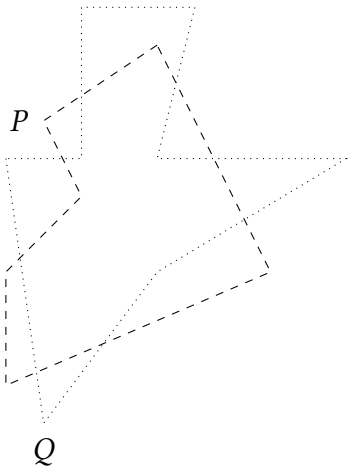
2.1 Ideia geral

Sejam P e Q dois polígonos. O problema de encontrar a interseção entre P e Q pode ser reduzido ao problema de encontrar as arestas (ou porções das arestas) de P que jazem em Q e vice-versa. Depois de localizadas tais arestas, basta conectá-las e então teremos o polígono resultante de $P \cap Q$.

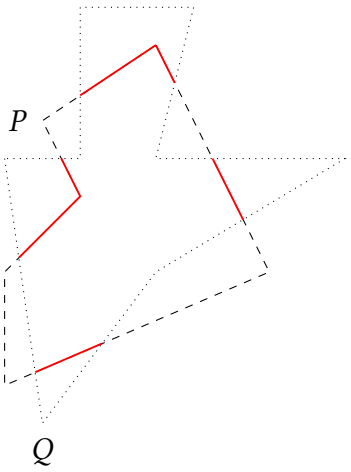
Para deixar claro como essas arestas serão encontradas, tome a seguinte adaptação da analogia presente em [Greiner e Hormann, 1989]. Imagine uma criança com um carrinho de areia que possui uma escotilha. Toda vez que a escotilha é aberta, o carrinho derruba areia. Se a escotilha estiver fechada, a areia permanece intacta no carrinho. Agora, imagine que essa criança está em um dos vértices do polígono P e começa a andar sobre a fronteira de P no sentido anti-horário. Se o vértice em que a criança começou está dentro de Q, a escotilha está aberta, caso contrário a escotilha está fechada. A criança vai andando sobre a fronteira de P e, toda vez que cruzar uma aresta de Q, a criança fecha a escotilha se ela estiver aberta e abre se estiver fechada. A criança para de andar sobre P assim que chegar ao vértice em que começou. Dessa forma, todas as arestas ou porções das arestas de P que estão dentro de Q foram marcadas com areia.

A criança também andará sobre a fronteira de Q e repetirá o mesmo processo marcando as arestas (ou porções delas) de Q que estão dentro de P.

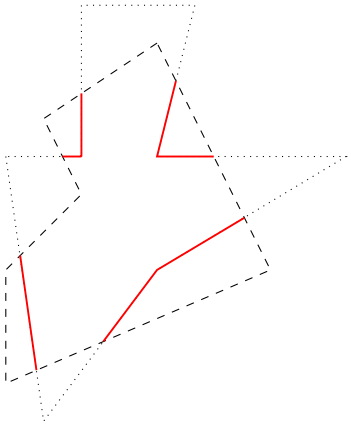
Após marcar as arestas de P e de Q, juntam-se tais arestas para obter a fronteira da interseção dos dois polígonos, que pode consistir de várias curvas poligonais fechadas. Observe a seguinte figura:



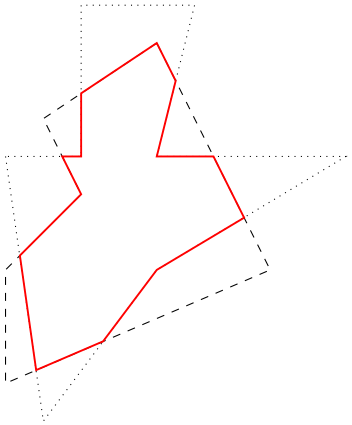
(a) *Exemplos de polígonos P e Q*



(b) *Polígono P com as arestas marcadas em linhas contínuas vermelhas*



(c) *Polígono Q com as arestas marcadas em linhas contínuas vermelhas*



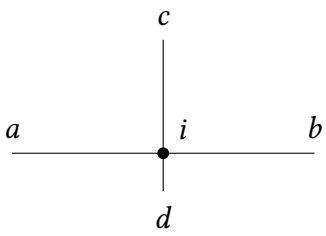
(d) *Junção das marcações de **(b)** e **(c)***

Figura 2.1: *Exemplo de como obter a interseção $P \cap Q$.*

2.2 O algoritmo

Sejam $P = (p_1, p_2, \ldots, p_n)$ e $Q = (q_1, q_2, \ldots, q_m)$ dois polígonos dados em posição geral. O algoritmo de Greiner-Hormann opera em três fases:

Na primeira fase, procura-se todas as interseções entre as arestas de P e as arestas de Q e, para cada aresta, retorna-se um valor, que será denotado por α. Tal valor corresponde à posição do ponto de interseção relativo à aresta que está sendo avaliada. Observe a seguinte figura: Seja $d(\overline{ab})$ a distância do vértice a ao vértice b, pode-se identificar que $d(\overline{ai}) = 0.5 \cdot d(\overline{ab})$, ou seja, a posição relativa de i à aresta $\overline{ab}$ é 0.5 ($\alpha = 0.5$). Para a aresta $\overline{cd}$ a posição relativa do vértice i é 0.75 ($\alpha = 0.75$).



(a) $a = (0, 8)$, $b = (8, 8)$, $c = (4, 11)$, $d = (4, 7)$, $i = (4, 8)$

Figura 2.2: *Exemplo de vértice de interseção*

Após determinar todos os pontos de interseção e suas respectivas posições relativas, adicionam-se esses pontos encontrados às listas de pontos de P e Q nas posições apropriadas. Tais pontos serão chamados de *vértices de interseção*.

Na segunda fase, faz-se algo parecido com o que foi explicado na analogia presente na seção anterior. Ou seja, percorre-se a fronteira de cada um dos polígonos e marca-se os vértices de interseção de P e Q como pontos de "entrada" ou de "saída". Para isso, usa-se o ray-casting no primeiro vértice do polígono que está sendo analisado (por exemplo P) em relação ao outro polígono (por exemplo Q). Por causa da hipótese simplificadora de que P e Q estão em posição geral, sabemos que tal vértice não está sobre a fronteira do outro polígono. Agora que sabemos a pertinência desse primeiro vértice em relação ao outro polígono, a salvamos em alguma variável e percorremos os vértices restantes. Toda vez que encontrarmos um vértice de interseção invertemos o valor salvo na variável e atribuímos esse novo valor para a interseção, isto é, se o valor da variável correspondia a "dentro" quando nos depararmos com um vértice de interseção, mudamos o valor da variável para "fora" e salvamos esse valor na interseção. O mesmo se aplica se o valor salvo na variável for "fora".

Na terceira fase, constrói-se a fronteira da interseção entre P e Q com as informações coletadas nas fases anteriores através de uma filtragem na estrutura de dados. A explicação detalhada de como essa fase funciona será mostrada na Seção 2.4 em que estarão os pseudocódigos de cada uma das fases e suas respectivas análises de complexidade de tempo. Além disso, nas seções seguintes, explicaremos como o autor se livra da hipótese simplificadora, de que os polígonos estão em posição geral, e explicaremos como realizar as outras operações.

2.3 Estrutura de dados

A estrutura de dados utilizada para representar um polígono é uma lista circular duplamente ligada com informações sobre cada vértice do polígono. Cada célula tem os seguinte atributos:

- vertex - corresponde ao par ordenado (x, y) que são as coordenadas do vértice;
- next - apontador para a próxima célula;
- prev - apontador para a célula anterior;

- nextPoly - apontador para um outro polígono;
- interset - *flag* que indica se o vértice é de interseção;
- entry_exit - *flag* que indica se o vértice é de entrada ou saída, caso seja de interseção;
- neighbor - apontador para a célula correspondente do outro polígono, caso seja de interseção;
- alpha - número que indica a localização do vértice em relação à aresta, caso seja de interseção.

Normalmente, para representar um polígono seriam necessários somente os atributos vertex, next e prev. Como a interseção entre dois polígonos pode resultar em vários polígonos disjuntos, o atributo nextPoly serve para navegarmos entre esses polígonos resultantes. Os atributos interset, entry_exit, neighbor e alpha são utilizados internamente pelo algoritmo.

Durante a execução do algoritmo precisa-se determinar todos os pontos de interseção entre P e Q. Após determinar tais pontos, duas células idênticas são criadas. Cada uma é inserida na estrutura de dados que representa um dos polígonos. Essas células têm a flag interset marcada e estão ligadas através do atributo neighbor. O atributo alpha guarda a posição relativa da interseção em relação à aresta. A flag entry_exit armazena a informação de se estamos entrando ou saindo do interior do outro polígono. Segue abaixo uma figura exemplificando a estrutura.

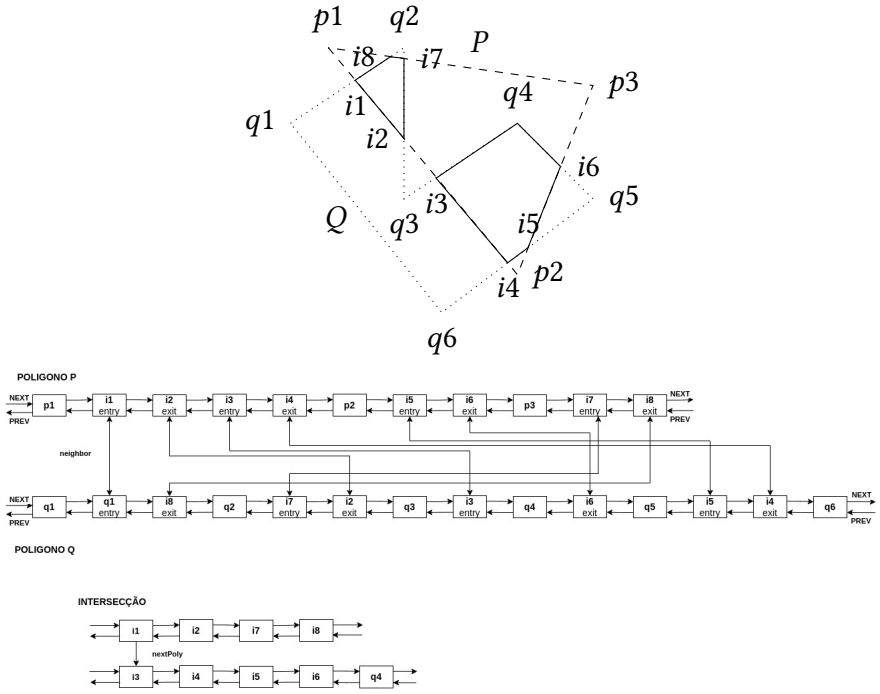


Figura 2.3: *Estrutura de dados de dois polígonos P (linhas tracejadas) e Q (linhas pontilhadas) e a interseção de P e Q (linhas contínuas).*

2.4 Pseudocódigo

2.4.1 Todas as interseções

Rememorando, a primeira etapa do algoritmo de Greiner-Hormann envolve encontrar todas as interseções e inserir esses vértices de interseção na estrutura de dados de ambos os polígonos. No Programa 2.1 encontra-se um algoritmo de força bruta que busca por todas as interseções e insere tais interseções na estrutura de dados.

Programa 2.1 Cálculo e inserção de todas as interseções.

```
1   FUNCTION IntersectionInsertion(P, Q)
2   for each vertex p_l ∈ polygon P do
3       for each vertex q_j ∈ polygon Q do
4           if Intersect(p_l, p_{l+1}, q_j, q_{j+1})
5               i_1 ⟵ CreateVertex1(p_l, p_{l+1}, q_j, q_{j+1})
6               Insert1(i_1, p_l)
7               i_2 ⟵ CreateVertex1(q_j, q_{j+1}, p_l, p_{l+1})
8               Insert1(i_2, q_j)
9               i_1.neighbor ⟵ i_2
10              i_2.neighbor ⟵ i_1
```

A rotina `Intersect`(a, b, c, d) recebe quatro pontos e verifica se a aresta $\overline{ab}$ interseta com a aresta $\overline{cd}$. `CreateVertex1`(a, b, c, d) recebe como entrada quatro pontos e cria uma célula com as coordenadas onde a aresta $\overline{ab}$ cruza com $\overline{cd}$. A função calcula internamente o valor α da aresta $\overline{ab}$ e atribui à célula. `Insert1`(i, p) insere a célula i entre p e o próximo vértice original do polígono, de modo que as interseções apareçam entre esses vértices ordenadas pelo valor α.

Seja n o número de vértices do polígono P e m o número de vértices do polígono Q. As rotinas `Intersect` e `CreateVertex1` consomem tempo $O(1)$. A rotina `Insert1` tem o consumo de tempo dependente do número de interseções presente na aresta que está sendo analisada, ou seja, o consumo dessa função é $O(k)$, onde k é o número de vértices de interseções. Perceba que o consumo de tempo da rotina `IntersectionInsertion` dessa forma será de $O(nmk)$. Propõem-se duas outras soluções para melhorar a complexidade de tempo dessa primeira fase.

A primeira solução é: primeiro encontrar todas as interseções utilizando força bruta, depois inserir as interseções encontradas na estrutura de dados. Para realizar isso deve-se manter duas estruturas de dados: a lista ligada que já foi mencionada anteriormente e um vetor com apontadores para as células da lista. Esse vetor funciona para termos acesso em $O(1)$ às células originais da lista.

A rotina `CalculateAlpha`(a, b, c, d) calcula o valor α da aresta $\overline{ab}$ baseado na interseção de $\overline{ab}$ com $\overline{cd}$. A função `CreateVertex2`(a, b, α) cria uma célula com posição relativa α entre os vértices a e b em tempo $O(1)$. A rotina `Insert2`(i, p) insere a célula i logo após p em tempo $O(1)$. Dessa forma, o consumo de tempo da rotina `Intersections` é $O(nm)$ e o da rotina `Insertions` é $O(k \log(k))$. Com isso, a complexidade da primeira solução em $O(nm + k \log(k))$.

Programa 2.2 Calculo de todas as interseções.

```
1   FUNCTION Intersections(P, Q)
2       S ← ∅
3       for each vertex p_l ∈ polygon P do
4           for each vertex q_j ∈ polygon Q do
5               if Intersect(p_l, p_{l+1}, q_j, q_{j+1})
6                   α ← CalculateAlpha(p_l, p_{l+1}, q_j, q_{j+1})
7                   β ← CalculateAlpha(q_j, q_{j+1}, p_l, p_{l+1})
8                   S ← S ∪ (α, β, l, j)
9       return S
```

Programa 2.3 Insere as interseções.

```
1   FUNCTION Insertions(P, Q)
2       S ← Intersections(P, Q)
3       MergeSort(S, 1)  ▷ ordena pela primeira coordenada α das quádruplas
4       for each (α, β, l, j) ∈ S do
5           i_1 ← CreateVertex2(V_P[l], V_P[l + 1], α)  ▷ V_P é o vetor para o polígono P
6           Insert2(i_1, V_P[l])
7           replace (α, β, l, j) by (i_1, β, l, j) in S
8       MergeSort(S, 2)  ▷ ordena pela segunda coordenada β das quádruplas
9       for each (i_1, β, l, j) ∈ S do
10          i_2 ← CreateVertex2(V_Q[j], V_Q[j + 1], β)  ▷ V_Q é o vetor para o polígono Q
11          Insert2(i_2, V_Q[j])
12          i_2.neighbor ← i_1
13          i_1.neighbor ← i_2
```

A segunda solução é muito parecida com a primeira. Em vez de utilizar um algoritmo de força bruta para encontrar todas as interseções, utiliza-se o algoritmo de [Bentley e Ottmann, 1979]. A etapa da inserção continua a mesma. Portanto, o consumo fica $O\big((n+m+k)\log(n+m)+k\log(k)\big)$, que é melhor que o anterior quando k não é muito grande. Mais precisamente, quando $k = o(\frac{nm}{\log(n+m)})$. A seguinte tabela resume a análise que foi feita:

algoritmo	consumo de tempo
Greiner-Hormann	$O(nmk)$
Primeira solução	$O(nm + k\log(k))$
Segunda solução	$O((n+k+m)\log(n+m) + k\log(k))$

Tabela 2.1: *Consumo de tempo para cada solução onde n é o número de vértices do polígono P, m é o número de vértices do polígono Q e k é o número de vértices de interseção entre as fronteiras de P e Q.*

2.4.2 Carrinho de areia

Nessa segunda etapa determina-se se os vértices de interseção indicam se a fronteira de um dos polígonos está entrando ou saindo do outro polígono. A função do Programa 2.4 utiliza a rotina *rayCasting*, apresentada no Programa 1.2 do Capítulo 1.

Programa 2.4 Marca vértice de interseção.

```
1  FUNCTION MarkIntersections(P, Q)
2      insideOutside ← rayCasting(p_0, Q, m)   ▷ m é o número de vértices do polígono Q
3      for each vertex p_l ∈ polygon P do
4          if p_l.intersect
5              p_l.entryExit ← not insideOutside
6              insideOutside ← not insideOutside
```

Executaremos essa rotina tanto para o polígono P quanto para Q. O consumo de tempo dessa fase é, portanto $\theta(n + m)$.

2.4.3 Marcando os segmentos

Agora que sabemos quais são os vértices de P que indicam entrada e saída de Q e vice-versa, basta percorrer a lista de vértices de P e Q e selecionar os vértices da fronteira do polígono resultante da operação desejada.

Para construir o polígono desejado, usa-se a rotina `newVertex`. Tal função cria uma célula que representa o vértice do polígono que está em construção pelo algoritmo.

Para calcular os segmentos que fazem parte da interseção $P \cap Q$, primeiro precisamos encontrar um vértice de interseção que ainda não foi processado. Esse será chamado de "start". Após encontrar tal vértice, prontamente verificamos se tal vértice é de entrada ou saída. Se for de entrada, percorremos a lista ligada do polígono corrente usando o atributo next, caso contrário percorremos a lista usando o atributo prev. Andaremos sobre a lista até encontrar o próximo vértice de interseção. Quando tal vértice for encontrado, utilizaremos o atributo neighbor para "pular" para lista do outro polígono. Na lista do outro polígono, repetimos o mesmo processo. A cada visita de uma nova célula, a rotina `newVertex` será invocada para criar os vértices do polígono em construção. A construção do novo polígono cessará quando encontrarmos novamente o vértice "start".

O processo descrito acima continuará até que todos os vértices de interseção sejam processados.

Usando a analogia da Seção 2.1, a criança abre a escotilha quando encontra um vértice de interseção não computado (equivale à chamada rotina *novoPolígono*). A criança vai derrubando areia nas arestas enquanto não encontrar o vértice em que começou. A criança repetirá esse processo até todos os vértices de interseção serem processados. O seguinte pseudocódigo pode ser encontrado em [Greiner e Hormann, 1989]. O programa acima precisa percorrer todas as arestas de P e todas as interseções. Portanto, a complexidade do tempo fica sendo $O(n + k)$.

Programa 2.5 Marca Segmentos.

```
1   FUNCTION MarkSegments(P)
2       newPolygon ← ∅
3       q ← P
4       repeat
5           start ← first unprocessed intersection vertex of P
6           oldPolygon ← newPolygon
7           newPolygon ← newVertex(start)
8           ans ← newPolygon
9           p ← start
10          repeat
11              p.intersect = False
12              if p is entry vertex
13                  repeat
14                      p ← p.next
15                      ans.next ← newVertex(p)
16                      ans ← ans.next
17                  until p.intersect
18              else
19                  repeat
20                      p ← p.prev
21                      ans.next ← newVertex(p)
22                      ans ← ans.next
23                  until p.intersect
24              p.intersect ← False
25              newPolygon.nextPoly ← oldPolygon
26              p ← p.neighbor
27          until p = start
28          q ← p
29      until q = P
30      return newPolygon
```

2.5 Outras operações

Até esse momento falamos apenas como o algoritmo funciona para calcular a operação de interseção. As operações de união e diferença de polígonos não estavam sendo consideradas. No entanto, isso não é nenhum problema: o algoritmo pode ser facilmente adaptado para comportar tais operações.

Considerando os polígonos P e Q já aumentados com os pontos de interseção, sejam P_{int} as arestas do polígono P que estão *internas* ao polígono Q e P_{ext} as arestas que estão *externas* a Q. Na interseção percorremos as arestas que entravam no outro polígono, ou seja, o resultado era a combinação das arestas P_{int} com Q_{int}. Para realizar a união, basta percorrer as arestas que estão fora do outro polígono. Isto equivale a combinar as arestas de P_{ext} com Q_{ext}. Para a diferença, se quisermos $P - Q$, basta combinar as arestas de P_{ext} com Q_{int}. A imagem abaixo exemplifica isso:

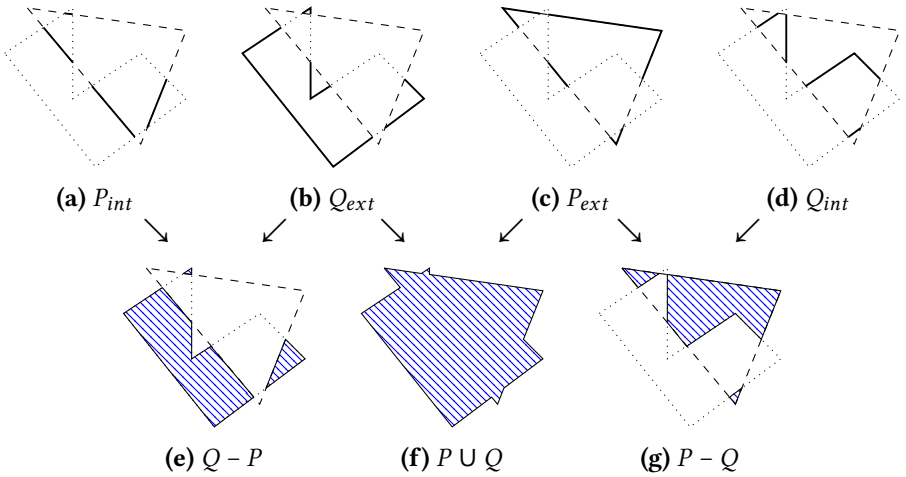


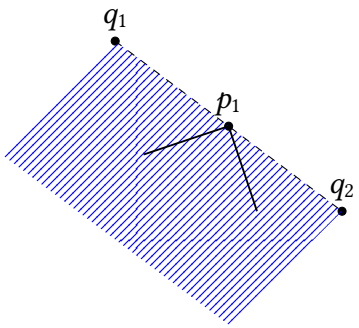
Figura 2.4: *Como realizar outras operações Booleanas.*

No algoritmo, para conseguirmos computar essas outras operações, basta alterar a segunda etapa. Para obter as arestas externas pode-se, na etapa 2 (Programa 2.4), alterar a classificação dos vértices para o inverso, ou seja, todo vértice de interseção que era de "entrada" passa a ser de "saída" e vice-versa.

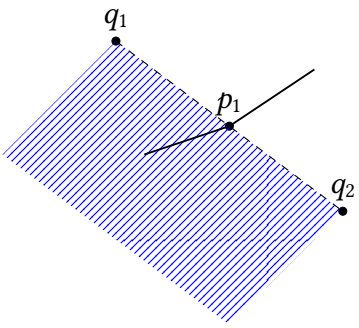
2.6 Hipótese simplificadora

Até agora não estávamos considerando os casos degenerados, isto é, um vértice de um polígono não pode estar contido em uma aresta de outro polígono. Os casos degenerados ocorrem quando $\alpha = 0$ ou $\alpha = 1$ para ambos os polígonos.

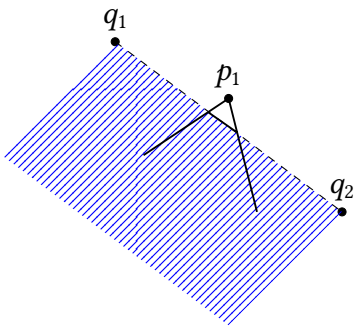
O artigo de [Greiner e Hormann, 1989] sugere que se faça uma pequena perturbação nos vértices de P e Q para evitar esses casos, o que é plausível para fins de computação gráfica, onde a escala da perturbação só precisa ser menor que 1 pixel, a Figura 2.5 mostra as possíveis configurações de perturbação. Em programas em que a precisão é necessária, não convém utilizar tal método, pois ele gera erros. É nessa situação que o aprimoramento do algoritmo descrito no próximo capítulo se faz necessário.



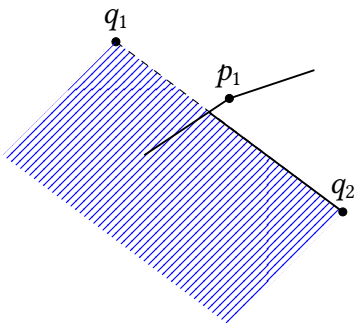
(a) *Configuração inicial com ambas as arestas de P dentro de Q*



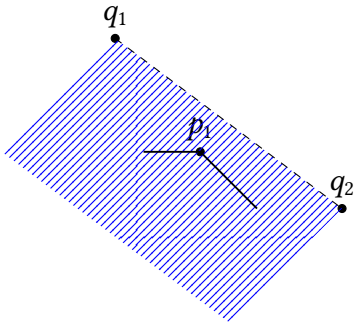
(b) *Configuração inicial com uma das arestas de P dentro de Q*



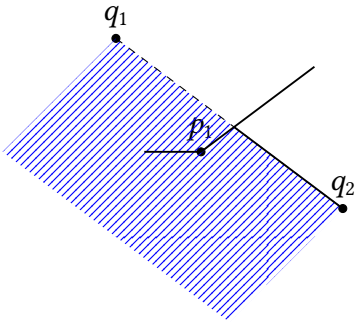
(c) *Configuração **(a)** com perturbação deixando p_1 fora de Q*



(d) *Configuração **(a)** com perturbação deixando p_1 fora de Q*



(e) *Configuração **(a)** com perturbação deixando p_1 dentro de Q*



(f) *Configuração **(b)** com perturbação deixando p_1 dentro de Q*

Figura 2.5: *Possíveis configurações da pertubação.*

Capítulo 3

Algoritmo de Greiner-Hormann com tratamento dos casos degenerados

O algoritmo original de [Greiner e Hormann, 1989], apresentado no Capítulo 2, não faz o tratamento de casos degenerados e depende de se fazer uma perturbação dos pontos dados para funcionar corretamente.

Como comentado no Capítulo 2, dependendo da aplicação, perturbar os pontos não é uma ação desejável, pois a precisão nessas aplicações se faz muito necessária. Exemplos desse tipo de aplicações são: GIS (geographic information system), design de circuitos VLSI (very large scale integration) e simulações numéricas, nas quais tipicamente é necessário realizar as operações Booleanas inúmeras vezes.

Nos artigos de [D. H. Kim e M.-J. Kim, 2006] e [Foster *et al.*, 2019], extensões do algoritmo de [Greiner e Hormann, 1989] são propostas. Tais extensões têm por objetivo resolver o problema dos casos degenerados sem utilizar a perturbação de pontos. Estudaremos o segundo artigo, pois, de acordo com os autores, o algoritmo de [D. H. Kim e M.-J. Kim, 2006] requer métodos computacionalmente mais caros nos cálculos internos. Os autores do segundo artigo propõem uma abordagem diferente para evitar essas computações caras.

3.1 Alterações propostas

Como visto anteriormente, o algoritmo de [Greiner e Hormann, 1989] pode ser dividido em três fases. Na primeira, calculamos todos os vértices de interseção entre as fronteiras dos polígonos P e Q e inserimos esses novos vértices nas listas de P e Q. Na segunda, rotulamos os vértices de interseção como de "entrada" ou de "saída". Na terceira, utilizando a rotulação feita na etapa anterior, conseguimos encontrar as arestas que estão dentro ou fora do outro polígono e desta forma selecionamos as arestas que devem aparecer na resposta final dependendo da operação.

A extensão de [Foster *et al.*, 2019] altera bastante a primeira e a segunda fase. A terceira fase é pouco alterada. Devido a essas poucas alterações, a estrutura de dados continua a mesma. Apenas acrescentaremos o atributo crossing, cuja funcionalidade será explicada posteriormente.

3.2 O algoritmo

Antes de começarmos a falar sobre o algoritmo em si, precisa-se falar sobre a forma como o algoritmo interpreta as arestas do polígono. Seja $P = (p_1,\ p_2,\ \ldots,\ p_n)$ um polígono com os vértices no sentido anti-horário. Para o algoritmo, as arestas de P agora serão semi-abertas da seguinte forma: $(p_1, p_2]$, $(p_2, p_3]$, ..., $(p_n, p_1]$. Posto isso, começaremos as explicações das alterações feitas.

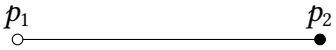


Figura 3.1: *Representação de uma aresta semi-aberta.*

3.2.1 Primeira fase

Essa fase, em essência, realiza a mesma tarefa do algoritmo original. Apenas uma classificação dos vértices de interseção é criada para podermos inseri-los da maneira correta nas listas de P e de Q. Cada vértice de interseção será classificado como um dos quatro tipos seguintes: *interseção-X*, *interseção-T1*, *interseção-T2* e *interseção-V* conforme explicado adiante. Procuramos todas as interseções entre as arestas semi-abertas de P e Q e inserimos nas listas dos polígonos dependendo da classificação que o vértice de interseção possui. É importante considerarmos as arestas semi-abertas, pois nos casos degenerados, o mesmo ponto de interseção pode ser calculado múltiplas vezes, caso não fizermos essa consideração.

Se duas arestas não são paralelas e se intersectam, então existe um único ponto de interseção que pode ser expresso da seguinte forma:

$$i = (1 - \alpha)p_a + \alpha p_b = (1 - \beta)q_c + \beta q_d$$

onde $(p_a, p_b]$ e $(q_c, q_d]$ são as arestas semi-abertas de P e Q. Os números α e β, como explicado no Capítulo 2, pertencem aos reais e indicam a posição do vértice i relativa à aresta $\overline{p_a p_b}$ e $\overline{q_c q_d}$ respectivamente.

No Capítulo 2, deixamos o cálculo do valor de α implícito, mas uma maneira de fazê-lo é utilizando a função Area(a, b, c), que calcula duas vezes a área sinalizada do triângulo formado pelos pontos a, b e c, e é dada pelo seguinte determinante.

$$\text{Area}(a, b, c) = \begin{vmatrix} a.x & a.y & 1 \\ b.x & b.y & 1 \\ c.x & c.y & 1 \end{vmatrix}.$$

Com ela, podemos determinar α e β da seguinte forma:

$$\alpha = \frac{\text{Area}(q_c, q_d, p_a)}{\text{Area}(q_c, q_d, p_a) - \text{Area}(q_c, q_d, p_b)} \quad \text{e} \quad \beta = \frac{\text{Area}(p_a, p_b, q_c)}{\text{Area}(p_a, p_b, q_c) - \text{Area}(p_a, p_b, q_d)}.$$

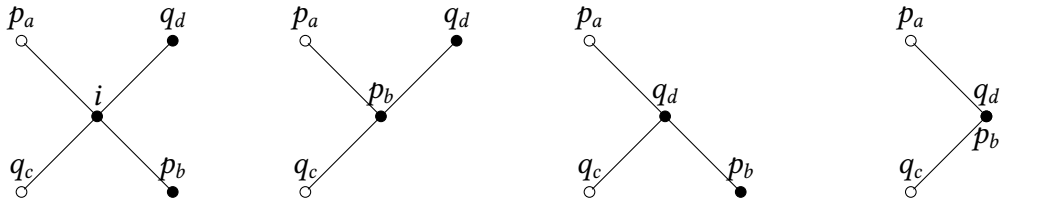
Sabendo os valores de α e β, podemos classificar o ponto de interseção i da seguinte maneira.

Um ponto i é *interseção-X* quando $0 < \alpha < 1$ e $0 < \beta < 1$. Interseções desse tipo corresponde aos casos não degenerados. Portanto, a inserção de i neste caso ocorre exatamente como no algoritmo original, isto é, criamos duas cópias de i ligadas pelo atributo neighbor e adicionamos essas cópias nas listas de P e de Q.

Um ponto i é *interseção-T1* quando $\alpha = 1$ e $0 < \beta < 1$. Nesse tipo de interseção, o vértice p_b está contido na aresta $\overline{q_c q_d}$, mas não coincide com os pontos q_c e q_d. Nesse caso, adicionamos uma cópia de p_b na lista de Q e ligamos o atributo de p_b com a sua cópia através do atributo neighbor.

Um ponto i é *interseção-T2* quando $\beta = 1$ e $0 < \alpha < 1$. Esse tipo de interseção é semelhante à anterior, mas a perspectiva está sobre a aresta $\overline{p_a p_b}$. Dessa forma, a inserção também é semelhante à anterior.

Um ponto i é *interseção-V* quando $\alpha = \beta = 1$. Nesse caso $p_b = q_d$ e não ocorre nenhuma inserção; apenas ligamos pelo atributo neighbor esses vértices.



(a) *Interseção-X* **(b)** *Interseção-T1* **(c)** *Interseção-T2* **(d)** *Interseção-V*

Até agora não estávamos considerando os casos onde $\overline{p_a p_b}$ e $\overline{q_c q_d}$ são paralelos. Neste caso, se a interseção entre tais arestas não for própria, ou seja, se a interseção for um segmento e não um ponto, classificamos de certa forma o segmento. Para isso, o segmento será classificado, como antes, como uma interseção-X, interseção-T1, interseção-T2 ou interseção-V. Serão calculados valores α e β para o segmento de interseção i e a classificação continua a mesma. A maneira como se calculam os valores de α e β é, portanto alterada, e as definições de interseção-X, interseção-T1, interseção-T2 e interseção-V são ajustadas. Fora isso, a inserção permanece inalterada exceto para a interseção-X.

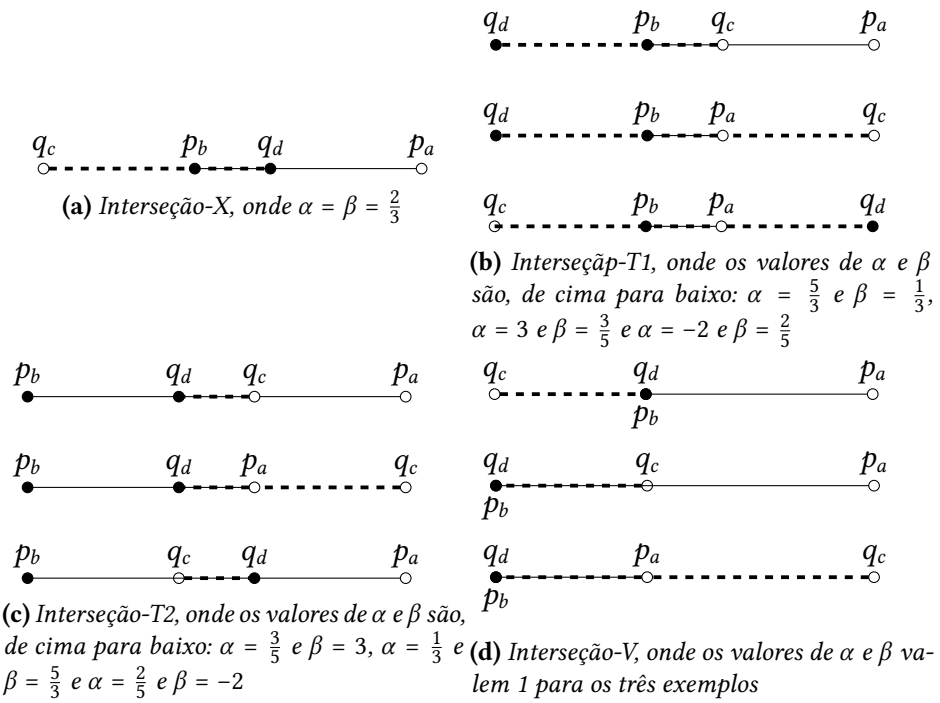
Se as arestas $\overline{p_a p_b}$ e $\overline{q_c q_d}$ forem paralelas e sua interseção for um segmento, os valores de α e β são calculados para cada extremo do segmento de interseção da seguinte forma:

$$\alpha = \frac{\langle q_d - p_a, p_b - p_a \rangle}{\langle p_b - p_a, p_b - p_a \rangle} \quad \text{e} \quad \beta = \frac{\langle p_b - q_c, q_d - q_c \rangle}{\langle q_d - q_c, q_d - q_c \rangle}.$$

Note que α diz respeito a um dos extremos do segmento de interseção e β diz respeito ao outro extremo.

Posto isso, a classificação do segmento para esse caso fica:

- Uma interseção-X ocorre quando $0 < \alpha$ e $\beta < 1$. Nesse caso adicionamos uma cópia de p_b na lista de Q e adicionamos uma cópia de q_d em P;
- Uma interseção-T1 ocorre quando $\alpha < 0$ ou $\alpha \geq 1$ e $0 < \beta < 1$;
- Uma interseção-T2 ocorre quando $\beta < 0$ ou $\beta \geq 1$ e $0 < \alpha < 1$;
- Uma interseção-V ocorre quando $\alpha = \beta = 1$, ou seja, $p_b = q_d$.



(a) *Interseção-X, onde* $\alpha = \beta = \frac{2}{3}$

(b) *Interseçãp-T1, onde os valores de* α *e* β *são, de cima para baixo:* $\alpha = \frac{5}{3}$ *e* $\beta = \frac{1}{3}$, $\alpha = 3$ *e* $\beta = \frac{3}{5}$ *e* $\alpha = -2$ *e* $\beta = \frac{2}{5}$

(c) *Interseção-T2, onde os valores de* α *e* β *são, de cima para baixo:* $\alpha = \frac{3}{5}$ *e* $\beta = 3$, $\alpha = \frac{1}{3}$ *e* $\beta = \frac{5}{3}$ *e* $\alpha = \frac{2}{5}$ *e* $\beta = -2$

(d) *Interseção-V, onde os valores de* α *e* β *valem 1 para os três exemplos*

Figura 3.3: *As coordenadas dos pontos são, em ordem da esquerda para a direita,* $(0, 0)$, $(2, 0)$, $(3, 0)$, $(5, 0)$.

Dessa forma, o pseudocódigo para essa etapa corresponde ao Programa 3.1.

Programa 3.1 Cálculo e inserção de todas as interseções.

```
1  FUNCTION Phase1(P, Q)
2  for each vertex p_l ∈ polygon P do
3      for each vertex q_j ∈ polygon Q do
4          if Intersect(p_l, p_{l+1}, q_j, q_{j+1})
5              α, β ← CalculateRelative(p_l, p_{l+1}, q_j, q_{j+1})
6              if (0 < α < 1 and 0 < β < 1) or (α > 0 and β < 1)   ▷ interseção-X
7                  i_1 ← CreateVertex1(α, p_l, p_{l+1})
8                  i_2 ← CreateVertex1(β, q_j, q_{j+1})
9                  Insert1(i_1, p_l)
10                 Insert1(i_2, q_j)
11                 i_1.neighbor ← i_2
12                 i_2.neighbor ← i_1
13             if (α = 1 and 0 < β < 1) or ((α < 0 or α ≥ 1) and 0 < β < 1)   ▷ interseção-T1
14                 i_2 ← CreateVertex1(β, q_j, q_{j+1})
15                 Insert1(i_2, q_j)
16                 p_l.neighbor ← i_2
17                 i_2.neighbor ← p_l
18                 p_l.intersect ← True
19             if (β = 1 and 0 < α < 1) or ((β < 0 or β ≥ 1) and 0 < α < 1)   ▷ interseção-T2
20                 i_1 ← CreateVertex1(α, p_l, q_{l+1})
21                 Insert1(i_1, p_l)
22                 q_j.neighbor ← i_1
23                 i_1.neighbor ← q_j
24                 q_j.intersect ← True
25             if α = β = 1   ▷ interseção-V
26                 p_l.neighbor ← q_j
27                 q_j.neighbor ← p_l
28                 p_l.neighbor ← q_j.neighbor ← True
```

Lembre-se que o Programa 3.1 não é a maneira mais eficiente de se encontrar e inserir os vértices de interseções (ver Seção 2.4.1). Como a etapa de classificação apenas olha para os valores de α e β para cada interseção, a complexidade continua sendo $O(nmk)$, onde k é o número de pares de arestas que se intersectam.

3.2.2 Segunda fase

No algoritmo original, essa segunda fase é responsável por rotular os vértices de interseção como de "entrada" ou de "saída". Como havia a perturbação de pontos, todos os vértices de interseção eram do tipo interseção-X. Portanto, para todo vértice de "entrada", existia um de "saída", consequentemente o número de vértices de interseção era sempre par para o algoritmo original. Como não estamos mais realizando essa perturbação, precisamos olhar cuidadosamente para os pontos próximos de um vértice de interseção para sabermos se ele é de "entrada" ou de "saída" em relação ao outro polígono. Para isso, vamos definir alguns conceitos que ajudarão a fazer essa análise da vizinhança de um vértice de interseção i.

Seja q um ponto no plano. Falamos que q está à *esquerda* da aresta $\overline{p_a p_b}$

se Area($p_a,\ p_b,\ q$) $>$ 0, está à *direita* se Area($p_a,\ p_b,\ q$) $<$ 0, e é *colinear* se Area($p_a,\ p_b,\ q$) = 0.

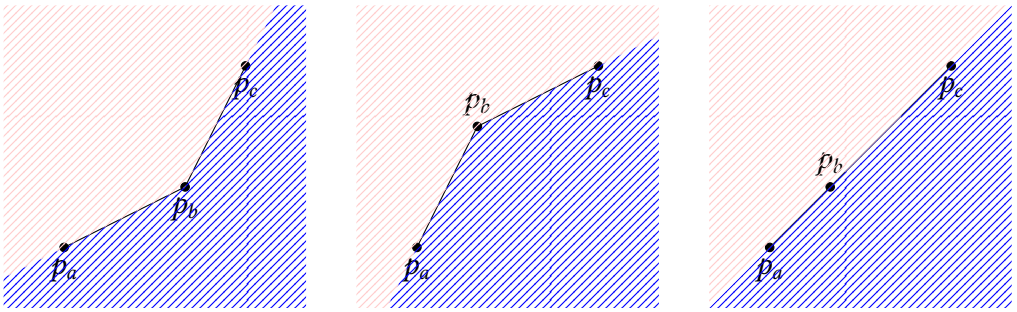
Agora definiremos se o ponto q está à esquerda, à direita ou é colinear a uma curva poligonal aberta. Seja ($p_a,\ p_b,\ p_c$) uma curva poligonal aberta formada por três pontos consecutivos de P, ou seja, $b = a + 1$ e $c = a + 2$. Considere os números:

$$s_1 = \text{Area}(p_a,\ p_b,\ q),\ s_2 = \text{Area}(p_b,\ p_c,\ q),\ \text{e}\ s_3 = \text{Area}(p_a,\ p_b,\ p_c).$$

Se $s_3 > 0$ então a cadeia vira à esquerda no ponto p_b, como na Figura 3.4a, e q está à esquerda se $s_1 > 0$ e $s_2 > 0$, e está à direita se $s_1 < 0$ ou $s_2 < 0$.

Se $s_3 < 0$ então a cadeia vira à direita no ponto p_b, como na Figura 3.4b, e q está à direita se $s_1 < 0$ e $s_2 < 0$, e está à esquerda se $s_1 > 0$ ou $s_2 > 0$.

Se $s_3 = 0$, como na Figura 3.4c, então o sinal de s_1 é igual ao sinal de s_2 e q está à esquerda se $s_1 > 0$, e à direita se $s_1 < 0$.



(a) *Cadeia virando à esquerda, em azul está representada a região à direita de* ($p_a\ p_b,\ p_c$) *e em vermelho a região à esquerda*

(b) *Cadeia virando à direita, em azul está representada a região à direita de* ($p_a\ p_b,\ p_c$) *e em vermelho a região à esquerda*

(c) *Cadeia reta, em azul está representada a região à direita de* ($p_a\ p_b,\ p_c$) *e em vermelho a região à esquerda*

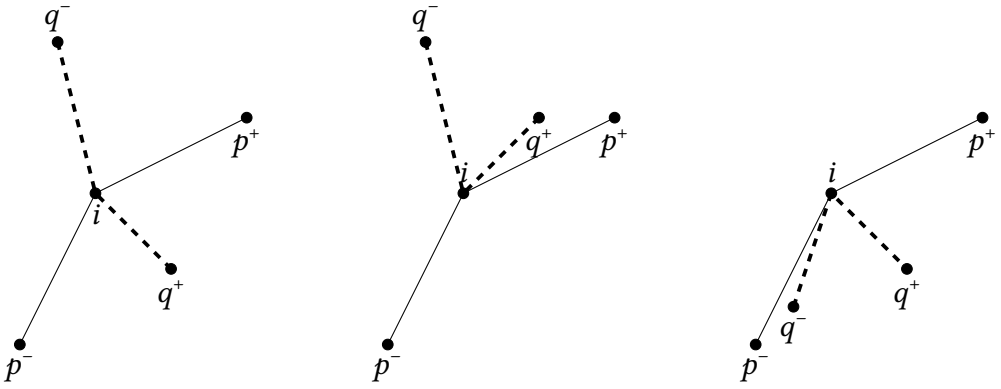
Figura 3.4: *Exemplos de classificação de ponto em relação a uma curva poligonal.*

Sejam p^- e p^+ os vértices predecessor e sucessor do vértice de interseção i no polígono P (lembre-se que estamos no sentido anti-horário). Por consequência da primeira fase, i também está na lista do polígono Q e, existem vértices q^- e q^+.

Se os quatro pontos adjacentes a i não forem vértices de interseção, podemos estabelecer a seguinte classificação:

Um vértice é *crossing* se q^- e q^+ estão em diferentes lados da curva ($p^-,\ i,\ p^+$). Então localmente a fronteira de P atravessa a fronteira de Q e, consequentemente, marcamos como verdadeiro o atributo crossing do vértice i.

Um vértice é *bouncing* se q^- e q^+ estão no mesmo lado da curva ($p^-,\ i,\ p^+$). Logo localmente a fronteira de P não atravessa a fronteira de Q. Consequentemente, marcamos como falso o atributo crossing do vértice i.



(a) *Exemplo de vértice de interseção crossing* **(b)** *Exemplo de vértice de interseção bouncing* **(c)** *Exemplo de vértice de interseção bouncing*

Estávamos considerando que p^+, p^-, q^+ e q^- não eram vértices de interseção. Isso nem sempre é verdade. Se pelo menos um desses pontos for vértice de interseção, uma curva poligonal $I = (i_1, i_2, \dots, i_k)$, com $k > 1$, de vértices de interseção será formada e uma análise mais cautelosa se faz necessária. Nesse sentido, as seguintes classificações são imprescindíveis:

Um vértice é *left/on* se p^+ está ligado com q^+ (ou q^-) e q^- (ou q^+) está à direita de (p^+, i, p^-). Dessa forma, o comportamento local da fronteira de P é passar do lado esquerdo de Q para se justapor à fronteira de Q no vértice i.

Um vértice é *right/on* se p^+ está ligado com q^+ (ou q^-) e q^- (ou q^+) está à esquerda de (p^+, i, p^-). Dessa forma, o comportamento local da fronteira de P é passar do lado direito de Q para se justapor à fronteira de Q no vértice i.

Um vértice é *on/on* se p^+ está ligado com q^+ (ou q^-) e p^- está ligado com q^- (ou q^+). Dessa forma, o comportamento local da fronteira de P é ficar sobreposto à fronteira de Q.

Um vértice é *on/left* se p^- está ligado com q^- (ou q^+) e q^+ (ou q^-) está à direita de (p^+, i, p^-). Dessa forma, o comportamento local da fronteira de P é deixar de se sobrepor à fronteira de Q e passar para à esquerda da fronteira de Q no vértice i.

Um vértice é *on/right* se p^- está ligado com q^- (ou q^+) e q^+ (ou q^-) está à esquerda de (p^+, i, p^-). Dessa forma, o comportamento local da fronteira de P é deixar de se sobrepor à fronteira de Q e passar para à direita da fronteira de Q no vértice i.

Posto isso, agora podemos classificar a curva $I = (i_1, i_2, \dots, i_k)$ olhando para o comportamento local dos seus pontos. O vértice i_1 é do tipo x/on, enquanto $i_2, \dots, i_{k-1}$ são do tipo on/on e i_k é do tipo on/y, com $x, y \in \{\text{left}, \text{right}\}$. A curva I será classificada a partir de x e y e poderá ser dos seguintes tipos:

Delayed crossing se $x \neq y$: localmente as fronteiras de P e de Q se cruzam em I. Nesse caso, o vértice i_1 será do tipo crossing e os restantes serão bouncing.

Delayed bouncing se $x = y$: localmente as fronteiras de P e de Q não se cruzam em I, apenas se sobrepõem. Nesse caso, todos os vértices de I serão do tipo bouncing.

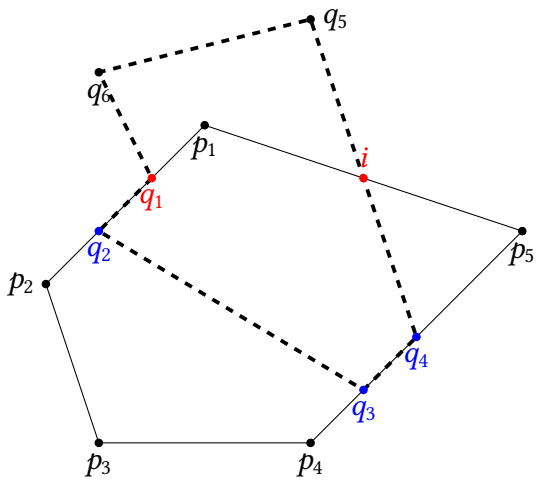


Figura 3.6: *Dois polígonos se intersectando onde existem quatro vértices de interseção que são vértices originais de um dos polígonos. Também existe um vértice de interseção que não é original a nenhum dos polígonos. Em azul estão os vértices do tipo bouncing e em vermelho os do tipo crossing.*

O Programa 3.2 mostra como fazer a classificação dos vértices de interseção em bouncing e crossing.

A rotina `chain`(q) é responsável por determinar se um vértice q está em uma curva I olhando para a vizinhança de q. A rotina `crossing`(q) determina se o vértice q é bouncing ou crossing. A rotina `localPosition`(i) verifica se o vértice i é left/on, right/on, etc. Todas essas funções consomem tempo $\Theta(1)$. Note que, devido à classificação em bouncing e em crossing dos vértices de P, não precisamos fazer a classificação para Q, pois se a fronteira de P entra (ou sai) de Q em i (ou I) então a fronteira de Q entra (ou sai) de P em i (ou I). Portanto, o consumo de tempo do Programa 3.2 é $\Theta(n)$, ou seja, não aumentamos a complexidade da segunda fase do algoritmo. Em suma, a segunda fase continua consumindo tempo $\Theta(n + m)$, onde n é o tamanho do polígono P e m é o tamanho de Q.

Programa 3.2 Classificação em bouncing e crossing.

```
1   FUNCTION BouncingCrossing(P, Q)
2       p ← primeiro vértice que não seja de interseção
3       p_0 ← p
4       do
5           if p.intersect:
6               q ← p
7               I ← ∅
8               while chain(q)
9                   q.crossing ← False   ▷ todo vértice da curva será bouncing; depois
                        determinaremos se o primeiro vértice da curva é crossing
10                  I ← I ∪ {q}
11              if |I| < 1
12                  q.crossing ← q.neighbor.crossing ← crossing(q)
13                  p ← q.next
14              else
15                  x ← localPosition(I[0])   ▷ primeiro vértice da curva I
16                  y ← localPosition(I[−1])   ▷ ultimo vértice da curva I
17                  if x ≠ y
18                      I[0].crossing ← I[0].neighbor.crossing ← True
19                  p ← q
20          else
21              p ← p.next
22      while p ≠ p_0
```

Agora que sabemos como classificar os vértices como crossing e bouncing, só resta descobrir se eles são de "entrada" ou de "saída". Para isso basta fazer uma pequena alteração no Programa 2.4. Alteraremos o estado da variável *insideOutside* apenas quanto ela estiver em um vértice do tipo crossing.

Perceba que o Programa 3.2 requer que o polígono tenha pelo menos um vértice que não seja de interseção. Caso um tal vértice não exista, duas possibilidades para tal acontecimento são possíveis. Na primeira, $P = Q$, ou seja, todos os vértices de interseção são on/on. Nesse caso, o polígono resultante é P (ou Q) e não se precisa executar a terceira etapa. Na segunda, $P \neq Q$, ou seja, existe pelo menos um vértice de interseção que não é on/on. Nesse caso, seja p_i um dos vértices de interseção que não é on/on. Adicionamos temporariamente o vértice $p_{new} = (p_i + p_{i+1})/2$ na lista de P e começamos a classificação a partir dele.

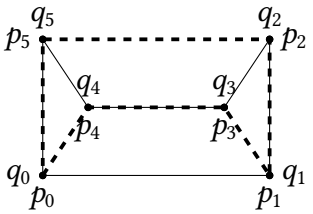


Figura 3.7: *Dois polígonos que se intersectam em todos os vértices.*

3.2.3 Terceira fase

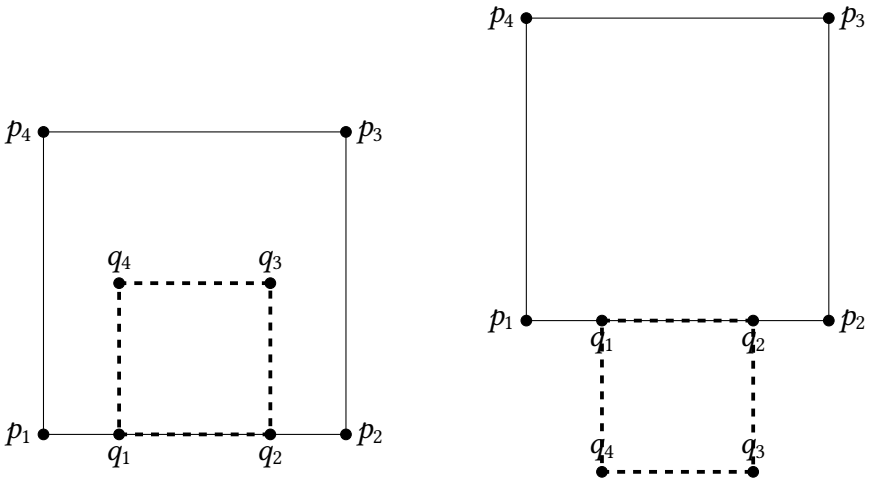
Como dito na Seção 3.1, essa terceira etapa do algoritmo sofre poucas alterações. Duas mudanças ocorrerão. A primeira mudança é que não começaremos a construção do polígono resultante por um vértice de interseção, começaremos por um vértice que além de interseção também é crossing. A segunda é que trocaremos de lista apenas quando encontrarmos um outro vértice de interseção do tipo crossing. Fora isso, ainda continuamos andando pela lista da mesma maneira, isto é, se o vértice de interseção for de entrada, então percorremos a lista com o atributo next, caso contrário percorremos com o atributo prev.

Programa 3.3 Marca Segmentos.

```
1   FUNCTION MarkSegments(P)
2     newPolygon ← ∅
3     q ← P
4     repeat
5       start ← first unprocessed crossing vertex of P
6       oldPolygon ← newPolygon
7       newPolygon ← newVertex(start)
8       ans ← newPolygon
9       p ← start
10      repeat
11        p.intersect ← False
12        if p is entry vertex
13          repeat
14            p ← p.next
15            ans.next ← newVertex(p)
16            ans ← ans.next
17          until p.intersect or p = start
18        else
19          repeat
20            p ← p.prev
21            ans.next ← newVertex(p)
22            ans ← ans.next
23          until p.intersect or p = start
24        p.intersect ← False
25        newPolygon.nextPoly ← oldPolygon
26        p ← p.neighbor
27      until p = start
28      q ← p
29    until q = P
30  return newPolygon
```

É notório que o Programa 3.3 precisa de pelo menos um vértice do tipo crossing para conseguir encontrar o polígono resultante. O caso em que não há nenhum vértice do tipo crossing é degenerado e precisa ser tratado de maneira diferente. Quando estamos em um caso degenerado, há apenas duas possibilidades para P e Q. Ou P está dentro de Q e ambos compartilham a mesma aresta (ou o mesmo ponto) e vice-versa, ou P está fora de Q e ambos compartilham a mesma aresta (ou o mesmo ponto) e vice-versa. A solução para resolver esse problema é: encontrar algum vértice de P que não seja de interseção e determinar a pertinência dele em relação a Q. Após determinada a pertinência desse

ponto, sabemos se o polígono está dentro ou fora do outro e assim conseguimos calcular quais os pontos deverão ser selecionados para a resposta final.



(a) *Dois polígonos onde não existe nenhum vértice crossing e o polígono Q está contido em P*

(b) *Dois polígonos onde não existe nenhum vértice crossing e o polígono apenas a aresta $\overline{q_1 q_2}$ está contida em P*

Figura 3.8: *Casos degenerados.*

3.3 Consumo de tempo

Como esse algoritmo não acrescentou nenhuma computação custosa em nenhuma das três etapas do algoritmo de [Greiner e Hormann, 1989], apenas acrescentou uma rotulação a mais nos vértices, a complexidade continua igual ao algoritmo original, isto é, $O(nmk)$, em que n é o número de vértices do polígono P, m é o número de vértices do polígono Q e k é o número de vértices de interseção.

No próximo capítulo veremos um algoritmo que utiliza uma técnica diferente para realizar operações Booleanas em polígonos. O algoritmo do próximo capítulo utilizará uma técnica chamada de *linha de varredura*, a qual será detalhada posteriormente.

Capítulo 4

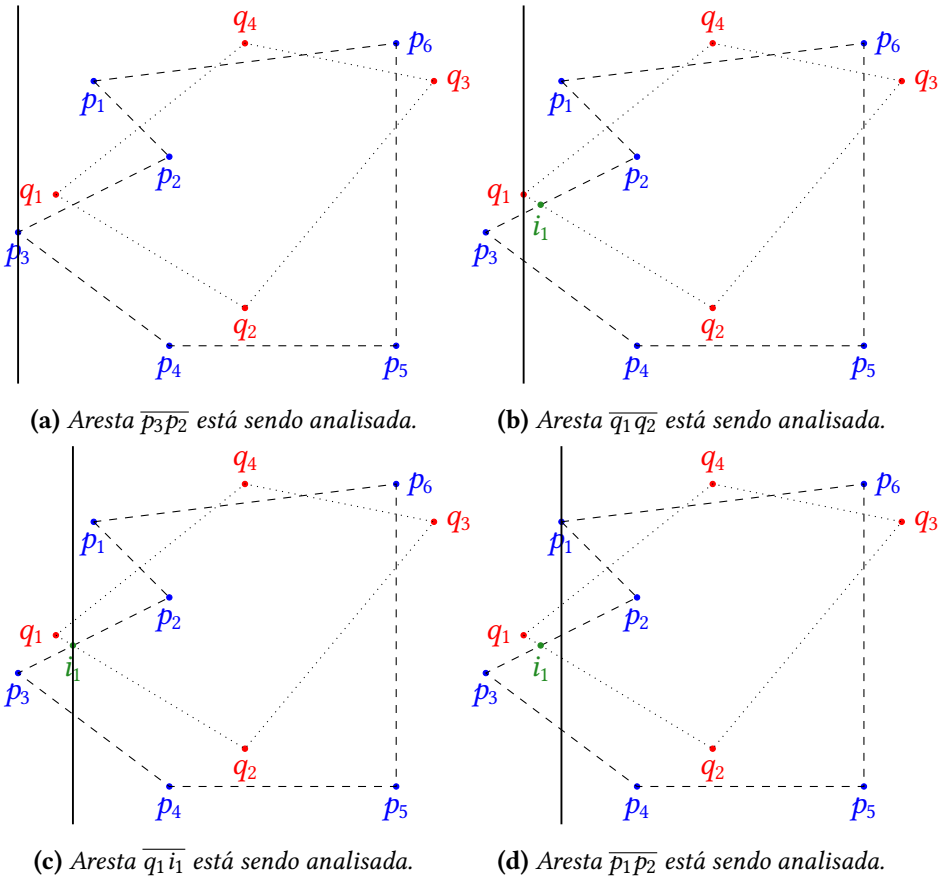
Algoritmo de linha de varredura

Até agora vimos dois algoritmos que resolvem o problema de operações Booleanas, mas ambos utilizam a mesma técnica para abordar o problema. O algoritmo que será visto nesse capítulo utiliza uma técnica diferente e é fortemente inspirado no artigo de [Bentley e Ottmann, 1979].

O algoritmo em questão está proposto em [Martinez *et al.*, 2013] e utiliza uma técnica amplamente conhecida na área da geometria computacional. Tal técnica é denominada de *linha de varredura* e consiste em traçar uma linha imaginária que vai percorrendo o plano numa direção, por exemplo, da esquerda para a direita, de cima para baixo, etc. A medida que tal linha vai percorrendo o plano, o problema restrito aos pontos (ou aresta, ou qualquer ente geométrico) pelos quais essa linha passou já foi resolvido. No caso de operações Booleanas, à medida que a linha passa pelas arestas dos polígonos, determinamos os trechos destas que estão dentro ou fora do outro polígono e, consequentemente, conseguimos determinar a fronteira do polígono resultante.

4.1 Ideia geral

Assumindo posição geral nos vértices dos polígonos, vamos percorrer com a linha de varredura o plano da esquerda para a direita e vamos guardar em uma estrutura de dados S as arestas que foram analisadas. À medida que a linha encontra uma aresta, determinamos se essa aresta se intersecta com alguma aresta do outro polígono. Se existe interseção, dividimos a aresta. Além de dividir as arestas, determinamos se as arestas obtidas da divisão estão dentro ou fora do outro polígono. Dessa forma, à medida que a linha de varredura vai percorrendo o polígono, determinamos as arestas (ou porções de arestas) que estão dentro ou fora do outro polígono. Portanto, ao passo que a linha de varredura percorre o plano, o polígono resultante é construído.



(a) *Aresta $\overline{p_3p_2}$ está sendo analisada.*

(b) *Aresta $\overline{q_1q_2}$ está sendo analisada.*

(c) *Aresta $\overline{q_1i_1}$ está sendo analisada.*

(d) *Aresta $\overline{p_1p_2}$ está sendo analisada.*

Figura 4.1: *Funcionamento da linha de varredura.*

Como pode ser percebido, esse algoritmo em uma etapa só já determina o polígono resultante da operação Booleana entre os polígonos P e Q. No artigo [Martinez *et al.*, 2013] há uma segunda etapa que serve para lidar com polígonos com buraco, mas não falaremos sobre essa etapa nesse trabalho.

4.2 Algoritmo

Como dito anteriormente, em uma etapa só varreremos o plano tentando encontrar os pontos de interseção e determinando as arestas que serão resultantes da operação. Para realizar isso, o algoritmo proposto por [Martinez *et al.*, 2013] é fortemente inspirado no algoritmo de [Bentley e Ottmann, 1979]. Esse algoritmo realiza o seguinte: com uma reta, na vertical, ele varre o plano guardando, ordenadamente, as arestas que intersectam essa reta em uma estrutura de dados S. O que [Martinez *et al.*, 2013] faz é, além de detectar a interseção, rotular as arestas. Posto isso, podemos afirmar duas coisas. O status de uma aresta em S só muda quando ela intersecta uma outra aresta, isto é, se uma aresta estava fora do outro polígono e intersecta com a fronteira do outro polígono, então a partir deste ponto ela está dentro ou vice-versa. Se duas arestas se intersectam então elas são adjacentes em S em algum momento da varredura do plano. A Figura 4.2 mostra um exemplo da estrutura de dados S e de como armazenamos as arestas.

Os vértices dos polígonos e os vértices de interseção serão os nossos *pontos eventos*, isto é, guardaremos em F (uma fila de prioridade) os vértices dos polígonos. Perceba que qualquer ponto p de um polígono P está contido em duas arestas. Dessa forma, em nossa fila F teremos duas cópias de p (uma para cada aresta). Dessa forma, os eventos estarão ordenados na nossa fila de prioridade pela:

1. Coordenada X;
2. Coordenada Y (caso haja um empate no item anterior);
3. Se o ponto é extremo esquerdo da aresta (caso os dois itens anteriores empatem);
4. Se os pontos forem coincidentes e ainda ambos forem o extremo esquerdo, então o critério de desempate fica sendo o outro extremo da aresta a qual p pertence.

Posto isso, os nossos pontos eventos p conterão as seguintes informações:

- vertex - par ordenado (x, y) que indica as coordenadas de um vértice;
- left - flag que indica se o ponto é extremo esquerdo da aresta;
- otherEvent - apontador para um outro evento o qual pertence à mesma aresta de p;
- pol - inteiro que indica a qual polígono essa aresta pertence.

Ainda falta listar dois campos que nossos eventos contêm. Iremos lista-lós posteriormente.

Antes de mostrar o pseudocódigo, vale salientar que guardaremos os pontos eventos numa fila de prioridade F. Estamos usando uma fila de prioridade por dois motivos: queremos que os pontos estejam ordenados da esquerda para a direita e a fila F sofre inserção, pois à medida que a linha varre o plano, pontos de interseção são encontrados e eles necessitam ser inseridos em F. A fila de prioridade é a estrutura de dados ideal para isso, porque insere em tempo $O(\log n)$, em que n é a quantidade de pontos na fila. A seguir iremos explicar algumas rotinas utilizada pelo algoritmo para construir o polígono resultante.

Seja e um evento que está em processamento e a seja e_{other} o outro extremo da aresta a qual e e e_{other} pertencem. As rotinas `insert(`S`, `e`)` e `delete(`S`, `e`)` são responsáveis por inserir e remover e de S, respectivamente. A rotina `possibleInter(`e`, `v`, `F`)` recebe um ponto e e um ponto v e verifica se $\overline{ee_{other}}$ e $\overline{vv_{other}}$ se intersectam. É importante passar F como argumento nessa rotina, pois se $\overline{ee_{other}}$ e $\overline{vv_{other}}$ se intersectarem, precisamos inserir a interseção i duas vezes na fila F (uma para cada polígono). O método `pred(`e`)` recebe um ponto e e retorna o ponto do extremo esquerdo pertencente à aresta predecessora de $\overline{ee_{other}}$ em S , isto é, retorna o ponto do extremo esquerdo da aresta que está logo abaixo de $\overline{ee_{other}}$. O método `succ(`e`)` faz o mesmo que o método `pred`, mas, ao invés procurar pelo predecessor, o sucessor que é procurado. A rotina `setInformation(`e`, `$pred$`)` recebe dois pontos e e $pred$ e determina se e, que está na fronteira de um dos polígonos, está dentro ou fora do outro polígono.

O Programa 4.1 mostra o funcionamento do algoritmo.

Programa 4.1 Interseção entre P e Q.

```
1   FUNCTION Intersection(P, Q)
2    ▷ Pré-processamento da estrutura de dados S e da fila F
3       result = []
4       while not F.empty ()
5           event ⟵ F.get ()
6           if (event.left)   ▷ se for extremo esquerdo da aresta
7               S ⟵ insert(S, event)
8               inside ⟵ setInformation(event, S.pred(event))
9               possibleInter(event, S.pred(event), F)
10              possibleInter(event, S.succ(event), F)
11              if (inside)   ▷ se event esta dentro do outro polígono coloca na lista
12                  result.push(event)
13          else   ▷ caso contrário o extremo é o direito da aresta
14              pred ⟵ S.pred(event.otherEvent)
15              succ ⟵ S.succ(event.otherEvent)
16              S ⟵ delete(S, event.otherEvent)
17              possibleInter(pred, succ)
```

Agora que sabemos como o algoritmo funciona, duas perguntas surgem. Como a rotina `setInformation` determina se uma aresta está dentro ou fora do outro polígono? Qual é a complexidade desse algoritmo? Como S deve ser armazenado? Essas perguntas serão respondidas na próxima seção.

4.3 Estrutura de dados

Seja n o número de vértices do polígono P e m o número de vértices do polígono Q, sabemos que o loop da linha 4 do Programa 4.1 será executado $n + m$ vezes. Para o nosso algoritmo performar bem, queremos que as rotinas consumam a menor quantidade de tempo possível. Essa tarefa está intimamente conectada com a estrutura de dados S que utilizamos para a linha de varredura. Queremos que S sofra inserções, deleções e ainda que seja ordenada. Uma árvore de busca balanceada atende a esses três requisitos.

Cada nó da árvore armazenará um evento com os campos listados acima e mais dois campos que são estes:

- crossOther - inteiro que indica quantos eventos de polígono diferente estão presentes na subárvore à esquerda;
- crossSelf - inteiro que indica quantos eventos do mesmo polígono estão presentes na subárvore à esquerda;
- inside - flag que indica se uma aresta está dentro ou fora do outro polígono.

Além disso, a ordem com que os nós da árvore serão armazenados é de baixo para cima a medida que se intersectam com a linha de varredura.

Com esses campos listados acima, conseguimos saber se uma aresta está dentro ou fora do outro polígono. Na hora em que estamos determinando o predecessor e o sucessor de uma aresta precisamos percorrer árvore. Nessa percorrida aproveita-se para contar quantas arestas estão abaixo do evento que está em análise. Isso equivale lançar um raio do extremo esquerdo da aresta e olhar quantas vezes esse raio se intersecta com outras arestas. Esse conceito foi explicado na Seção 1. Em suma, se o número de interseções for par então a aresta está fora, caso contrário está dentro.

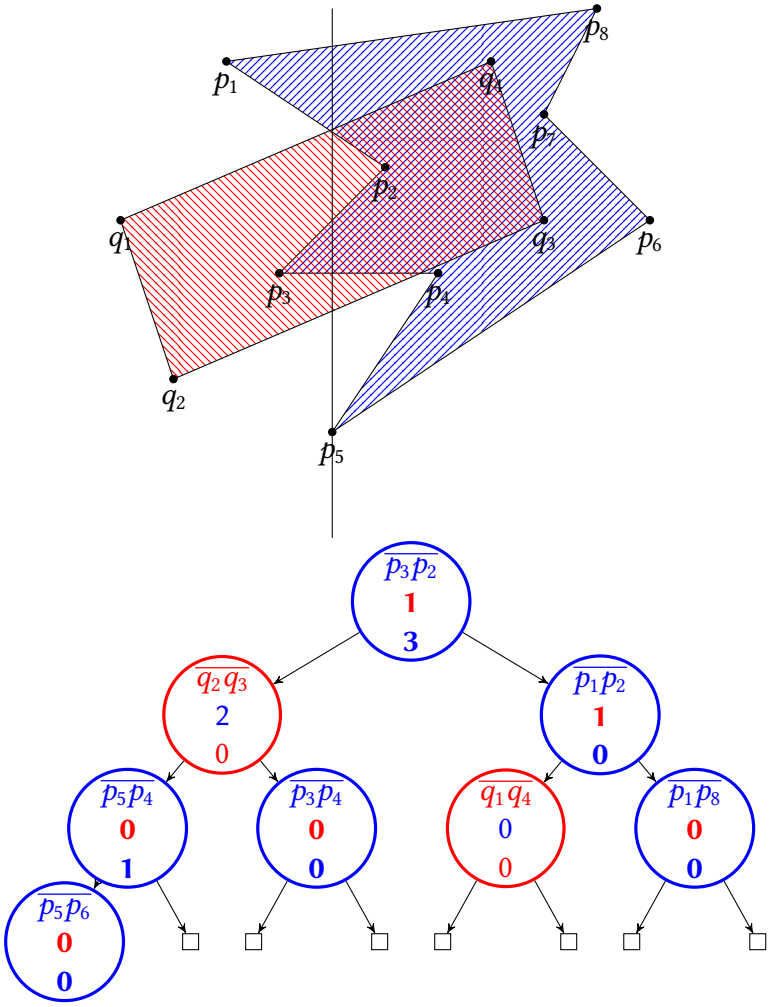


Figura 4.2: *Estrutura de dados S quando a linha está na posição do vértice p_5. Lembre-se que nessa estrutura de dados guardamos apenas os extremos esquerdos das arestas, pois os extremos direitos indicam o fim de uma aresta e, indicam que essa aresta não está mais se intersectando com a linha de varredura. Nessa representação estamos apresentando 3 campos dos nossos eventos (otherEvent, crossOther e crossSelf), nesta sequência de cima para baixo.*

4.4 Complexidade do algoritmo

Usando uma árvore as operações de inserção, deleção, busca por predecessor e sucessor podem ser feitas em $O(\log l)$, em que l é número de nós na árvore. As rotinas `possibleInter` e `setInformation` consomem tempo $O(1)$.

Portanto, sejam n, m e k, respectivamente, o número de vértices do polígono P, o número de vértices do polígono Q e o número de interseção entre os polígonos. O loop da quarta linha do Programa 4.1 será chamado $2(n+m)+k$ vezes, e as operações na estrutura S consomem tempo $O(\log(n+k+m))$, logo o algoritmo terá consumo de $O((n+k+m)\log(n+m+k))$.

4.5 Casos degenerados

Há dois casos degenerados para esse algoritmo. O primeiro é quando a aresta do polígono é vertical, logo não existe um extremo esquerdo e um extremo direito. Quando um polígono tem uma aresta na vertical, fica definido que a aresta à esquerda é a que tem menor y-coordenada. O segundo caso é quando duas arestas se sobrepõem. Nesse caso, o polígono resultante depende se um polígono está dentro ou fora do outro. Olhe a Figura 4.3

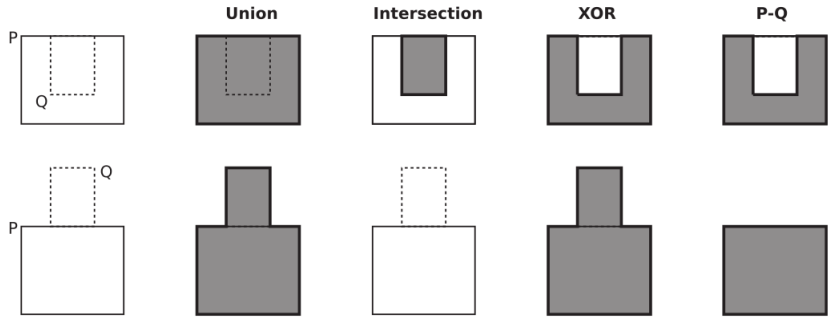


Figura 4.3: *Imagem retirada de [Martinez et al., 2013].*

Quando ocorre essa sobreposição, o algoritmo cria duas cópias dessas arestas sobrepostas (uma para cada polígono). Devemos escolher no máximo uma dessas duas arestas para compor o polígono resultante. Para selecionar a aresta é necessário verificar a flag crossSelf e a operação Booleana que está sendo efetuada. No caso da interseção e união, escolheremos uma das arestas que tiver o mesmo valor na flag crossSelf. No caso da diferença, veremos se o valor da flag crossSelf diferir, se diferir então escolhemos uma das arestas.

Capítulo 5

Conclusão

Foram feitos alguns testes para calcular o tempo médio de execução dos algoritmos. Os algoritmos foram feitos e executados em python e o código-fonte para as implementações pode ser visto em https://github.com/rafarillo/TCC. A máquina que rodou tais algoritmos tem a seguinte especificação:

- Processador - Intel® Core™ i7-8565U CPU @ 1.80GHz × 8
- Memória RAM - 16GB
- Sistema Operacional - Ubuntu 20.04 LTS

Número de vértices ($P \times Q$)	Greiner-Hormann	Foster et al.	Linha de varredura
100 × 100	0.18	0.18	0.07
1000 × 1000	16.6	17.9	0.7
2500 × 2500	114.9	113.4	2.3

Tabela 5.1: *Tempo de execução da operação de interseção, em segundos.*

Algoritmos	Complexidade
Greiner-Hormann	$O(nmk)$
Foster et al.	$O(nmk)$
Linha de varredura	$O((n+m+k)\log(n+m+k))$

Tabela 5.2: *Complexidade de cada algoritmo onde n, m, e k são, respectivamente, o número de vértices do polígono P, o número de vértices do polígono Q e o número de vértices de interseção entre P e Q.*

Sobre os testes, cada algoritmo foi executado 10 vezes para cada linha da Tabela 5.1. O tempo mostrado na Tabela 5.1 é o tempo médio dessas 10 execuções. Só foi executado a operação de interseção, mas como as outras operações não precisam de computação a mais, é esperado que o tempo médio permaneça próximo. Sobre os polígonos utilizados para testes, eles foram gerados aleatoriamente por uma biblioteca em python chamada polygenerator (https://pypi.org/project/polygenerator/). Esses polígonos gerados aleatoriamente eram todos não convexos e sem buracos.

Como dito anteriormente, a etapa computacionalmente mais cara é encontrar os vértices de interseção entre os polígonos. Os algoritmos de [Greiner e Hormann, 1989] e [Foster *et al.*, 2019] demoram tempo quadrático para encontrar as interseções enquanto que o [Martinez *et al.*, 2013] pode demorar tempo linearítmico, se o número de interseções não for quadrático, caso contrário pode demorar tempo quadrático também. Olhando para a Tabela 5.1 vemos que, com o aumento do número de vértices, o algoritmo que continua desempenhando bem é o de linha de varredura. Posto isso, a análise teórica do consumo de tempo se vê presente nos resultados práticos.

Agora, olhemos sobre o ponto de vista de implementação e usabilidade desses três algoritmos. O algoritmo de Greiner é o mais simples de ser implementado entre esses três. No entanto, ele é lento (para instâncias grandes) e realiza uma perturbação nos vértices que pode ser indesejada dependendo da aplicação. O algoritmo de Foster resolve o problema da perturbação, mas é um pouco mais complicado de ser implementado e ainda continua lento para instâncias grandes. O algoritmo de linha de varredura sem dúvidas é o mais complicado de implementar entre os três, mas ele é o mais veloz e ainda não precisa de perturbação dos vértices.

Em suma, a implementação desses três algoritmos estudados depende fortemente da aplicação. Cada algoritmo tem sua especificidade e um tradeoff entre simplicidade e rapidez.

Referências

[Bentley e Ottmann 1979] Jon L. Bentley e Thomas A. Ottmann. "Algorithm for reporting and counting geometric intersections". Em: *IEEE Transaction on Computers* 28 (1979), pgs. 643–647 (citado nas pgs. 16, 33, 34).

[Foster *et al.* 2019] Erich L. Foster, Kai Hormann e Romeo Traian Popa. "Clipping simple polygons with degenerate intersections". Em: *Computers & Graphics: X* 2 (2019), pg. 100007 (citado nas pgs. 3, 21, 22, 40).

[Greiner e Hormann 1989] Gunther Greiner e Kai Hormann. "Efficient clipping of arbitrary polygons". Em: *ACM Transactions on Graphics* 17 (1989), pgs. 71–83 (citado nas pgs. 3, 11, 17, 19, 21, 31, 40).

[D. H. Kim e M.-J. Kim 2006] Dae Hyun Kim e Myoung-Jun Kim. "An extension of polygon clipping to resolve degenerate cases". Em: *Computer-Aided Design and Applications* 3.1-4 (2006), pgs. 447–456 (citado na pg. 21).

[Martinez *et al.* 2013] Francisco Martinez, Carlos Ogayar, Juan R. Jiménez e Antonio J. Rueda. "A simple algorithm for boolean operations on polygons". Em: *Advances in Engineering Software* 64 (2013), pgs. 11–19 (citado nas pgs. 3, 11, 33, 34, 38, 40).

[Sutherland e Hodgman 1974] Ivan E. Sutherland e Gary W. Hodgman. "Reentrant polygon clipping". Em: *Communications of the ACM* 17.1 (1974), pgs. 32–42 (citado na pg. 2).

[Vatti 1992] Bala R. Vatti. "A generic solution to polygon clipping". Em: *Communications of the ACM* 35 (1992), pgs. 56–63 (citado nas pgs. 3, 11).

[Weiler 1980] Kevin Weiler. "Polygon comparision using a graph representation". Em: *SIGGRAPH '80* (1980), pgs. 10–18 (citado nas pgs. 3, 11).

[Weiler e Atherton 1977] Kevin Weiler e Peter Atherton. "Hidden surface removal using polygon area sorting". Em: *ACM SIGGRAPH Computer Graphics* 11.2 (1977), pgs. 214–222 (citado na pg. 2).

[Wikipedia 2021] Wikipedia. *Polygonal chain — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Polygonal%20chain&oldid=1058902113. [Online; accessed 23-May-2022]. 2021 (citado na pg. 3).