Geometria Computacional:

Principais Algoritmos e Predicados

André Maués Brabo Pereira – Depto. de Eng. Civil – UFF

e

Luiz Fernando Martha – Depto. de Eng. Civil – PUC-Rio

Adaptado para a disciplina:

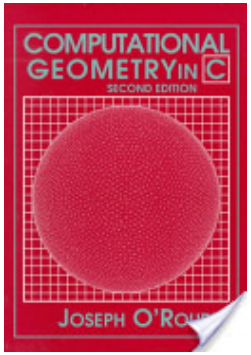
CIV2802 – Sistemas Gráficos para Engenharia

Conteúdo

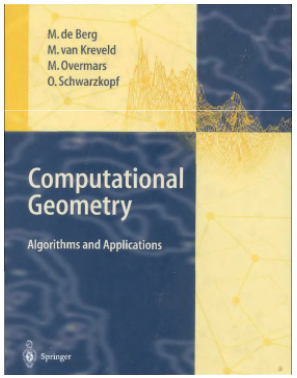
- Referência e fontes
- Introdução e escopo
- Necessidade de estruturas de dados
- Definições e notações gerais
- Ponto mais próximo em um segmento de reta
- Interseção de segmentos de reta
- Verificação de inclusão de ponto em polígono
- Área orientada de polígono
- Tesselagem de polígonos
- Predicados geométricos
- Aritmética exata e adaptativa

Referências e Fontes

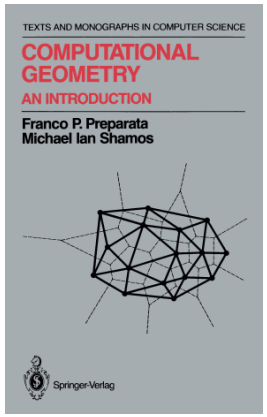
[OROURKE98]
Joseph O'Rourke
Computational Geometry in C
Cambridge University Press, 1998



[BERG97]
M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf
Computational Geometry - Algorithms and Applications
Springer, 1997

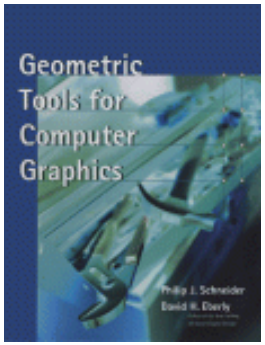


[PREPARATA85]
Franco P. Preparata, Michael Ian Shamos
Computational Geometry – An Introduction
Springer-Verlag, 1985

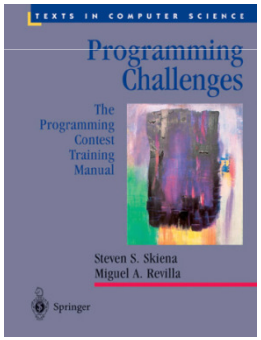


Referências e Fontes

[SCHNEIDER03]
Philip Schneider and David Eberly
Geometric Tools for Computer Graphics
Elsevier, 2003



[SKIENA02]
Steven S. Skiena, Miguel A. Revilla
Programming Challenges
Springer, 2002



[GHALI08]
Sherif Ghali
Introduction to Geometric Computing
Springer, 2008

[VINCE05]
John Vince
Geometry for Computer Graphics
Springer, 2005

Introdução e Escopo

Fonte: [OROURKE98]

Geometria Computacional é o estudo de algoritmos para resolver problemas geométricos em um computador. A ênfase nesse curso é no projeto de tais algoritmos, com de alguma forma menos atenção tomada na análise de desempenho. [OROURKE98]

Existem diversas áreas dentro de Geometria, e a que tem se tornado conhecida como Geometria Computacional, segundo [OROURKE98], é primariamente geometria discreta e combinatória. Portanto, polígonos tem um papel mais importante do que regiões com fronteiras curvas. A maioria dos trabalhos com curvas e superfícies contínuas são mais escopo da Modelagem Geométrica ou Modelagem de Sólidos, um campo com suas próprias conferências e textos, distintos da geometria computacional. Obviamente que existe substancial sobreposição, e não existe razão fundamental para que os campos sejam particionados dessa forma, eles parecem se fundir em alguma extensão. [OROURKE98]

Introdução e Escopo

Fonte: [BERG97]

A Geometria Computacional emergiu do campo de projeto e análise de algoritmos no final da década de 1970. Ela tem crescido como uma disciplina reconhecida com suas próprias revistas, conferências e uma grande comunidade de pesquisadores ativos. O sucesso do campo como uma disciplina de pesquisa pode por um lado ser explicado pela beleza dos problemas estudados e das soluções obtidas, e por outro lado, pelo domínio de diversas aplicações – computação gráfica, sistemas de informações geográficas (GIS), robótica, e outros – em que algoritmos geométricos tem um papel fundamental. [BERG97]

Para diversos problemas geométricos as primeiras soluções algorítmicas foram ou lentas ou difíceis de serem entendidas ou implementadas. Mais recentemente, uma grande quantidade de novas técnicas algorítmicas tem sido desenvolvidas que melhoram ou simplificam diversas das primeiras abordagens. A ideia desse curso é tentar fazer com que essas soluções algorítmicas modernas sejam acessíveis para uma grande audiência. [BERG97]

Introdução e Escopo

Fonte: [PREPARATA85]

Um grande número de áreas de aplicações tem sido a incubation bed da disciplina reconhecida atualmente como Geometria Computacional, já que tais aplicações fornecem inerentemente problemas geométricos que requerem o desenvolvimento de algoritmos eficientes. Estudos algorítmicos desses e outros problemas tem aparecido no século passado na literatura científica, com uma intensidade aumentando nas últimas três décadas. Porém, apenas recentemente, estudos sistemáticos de algoritmos geométricos tem sido assumidos, e um crescente número de pesquisadores tem sido atraído para essa disciplina, batizada como Geometria Computacional em um artigo de M. I. Shamos em 1975. Uma característica fundamental dessa disciplina é a percepção de que caracterizações clássicas de objetos geométricos são frequentemente não amenas para o projeto de algoritmos eficientes. Para tornar obvio essa inadequação, é necessário identificar os conceitos úteis e estabelecer suas propriedades, que irão conduzir para computações eficientes. De certa forma, a geometria computacional deve remodelar – sempre que necessário – a disciplina clássica em sua encarnação computacional.

Introdução e Escopo

Fonte: [SCHNEIDER03]

O campo da Geometria Computacional é muito vasto e é um dos que avançam mais rapidamente nos últimos tempos. Os tópicos gerais abordados neste curso são feixes convexos de conjuntos finitos de pontos, testes de ponto em polígono, tesselagem de polígonos (particionamento de polígonos em pedaços convexos ou triângulos) e interseções de segmentos de retas. [SCHNEIDER03]

A ênfase será nos algoritmos para implementar as várias ideias. Porém, é dada atenção especial para o problema de computação quando realizada por um sistema com números em formato de ponto flutuantes. Esses problemas ocorrem sempre que se precisa determinar quando pontos são colineares, coplanares e cocirculares. Isso é fácil de fazer quando o sistema computacional é baseado em aritmética de inteiros, mas bem problemático quando aritmética de pontos flutuantes é usada. [SCHNEIDER03]

Introdução e Escopo

Fonte: [SKIENA02]

Computação geométrica vai se tornando cada vez mais importante em aplicações como computação gráfica, robótica e projeto auxiliado por computador, isso tudo porque forma é uma propriedade inerente de objetos reais. Porém, objetos do mundo real não são feitos de linhas que vão até o infinito. Ao invés, a maioria dos programas de computador representam geometria como arranjos de segmentos de retas. Curvas ou formas arbitrarias fechadas podem ser representadas por coleções ordenadas de segmentos de retas ou polígonos.

Geometria computacional pode ser definida (para o propósito deste curso) como a geometria de segmentos de retas discretos e polígonos. É um assunto interessante e prazeroso, porém não tipicamente ensinado em cursos que requerem tal conhecimento. Isso dá ao estudante ambicioso que aprende um pouco de geometria computacional um estimulo e uma janela em uma área fascinante de algoritmos ainda hoje com pesquisas ativas. Livros excelentes sobre geometria computacional estão disponíveis na literatura, porém essa seção do curso deve ser suficiente para iniciar o estudande nesse assunto.

Necessidade de Estruturas de Dados

Algoritmos geométricos envolvem a manipulação de objetos que não são manipulados no nível da linguagem de máquina. O usuário tem que portanto organizar esses objetos complexos por meio de tipos de dados mais simples diretamente representáveis pelo computador. Essas organizações são universalmente referidas como **Estruturas de Dados**.

Os objetos complexos mais comuns encontrados no projeto de algoritmos geométricos são os conjuntos e as sequências (conjuntos ordenados). Estruturas de dados particularmente apropriadas para esses objetos combinatórios complexos são bem descritas na literatura padrão sobre algoritmos. Restringe-se aqui a revisar a classificação dessas estruturas de dados, junto com suas capacidades funcionais e desempenho computacional.

Seja S um conjunto representado em uma estrutura de dados e seja u um elemento arbitrário de um conjunto universal em que S é um subconjunto. As operações fundamentais que ocorrem na manipulação de conjuntos são:

1. **MEMBER**(u,S). Tem-se que $u \in S$? (Resposta SIM/NÃO.)
2. **INSERT**(u,S). Adiciona u em S.
3. **DELETE**(u,S). Remove u de S.

Fonte: [PREPARATA85]

Suponha agora que $\{S_1, S_2, \ldots, S_k\}$ é uma coleção de conjuntos (com interseção vazia por pares). Operações úteis nessa coleção são:

4. **FIND**(u). Reporta j, se u $\in$ S.

5. **UNION**(S_i,S_j; S_k). Forma a união de S_i e S_j e chama esse novo conjunto de S_k.

Quando o conjunto universal é totalmente ordenado, as seguintes operações são muito importantes:

6. **MIN**(S). Reporta o elemento mínimo de S.

7. **SPLIT**(u,S). Particiona S em $\{S_1,S_2\}$, tal que $S_1 = \{v: v \in S \text{ e } v \leq u\}$ e $S_2 = S - S_1$.

8. **CONCATENATE**(S_1,S_2). Assumindo que, para arbitrário $u' \in S_1$ e $u'' \in S_2$ tem-se que $u' \leq u''$, forma o conjunto ordenado $S = S_1 \cup S_2$.

As Estruturas de Dados podem ser classificadas com base nas operações que elas supotam (sem se preocupar com eficiência). Assim, para conjuntos ordenados tem-se a seguinte tabela:

Data Structure	Supported Operations
Dictionary	MEMBER, INSERT, DELETE
Priority queue	MIN, INSERT, DELETE
Concatenable queue	INSERT, DELETE, SPLIT, CONCATENATE

Para eficiência, cada uma dessas estruturas de dados é normalmente realizada como uma árvore de busca binária balanceada pela altura. Com essa realização, cada uma das operações acima é executada em tempo proporcional ao logaritmo do número de elementos armazenados nessa estrutura de dados; o armazenamento é proporcional ao tamanho do conjunto.

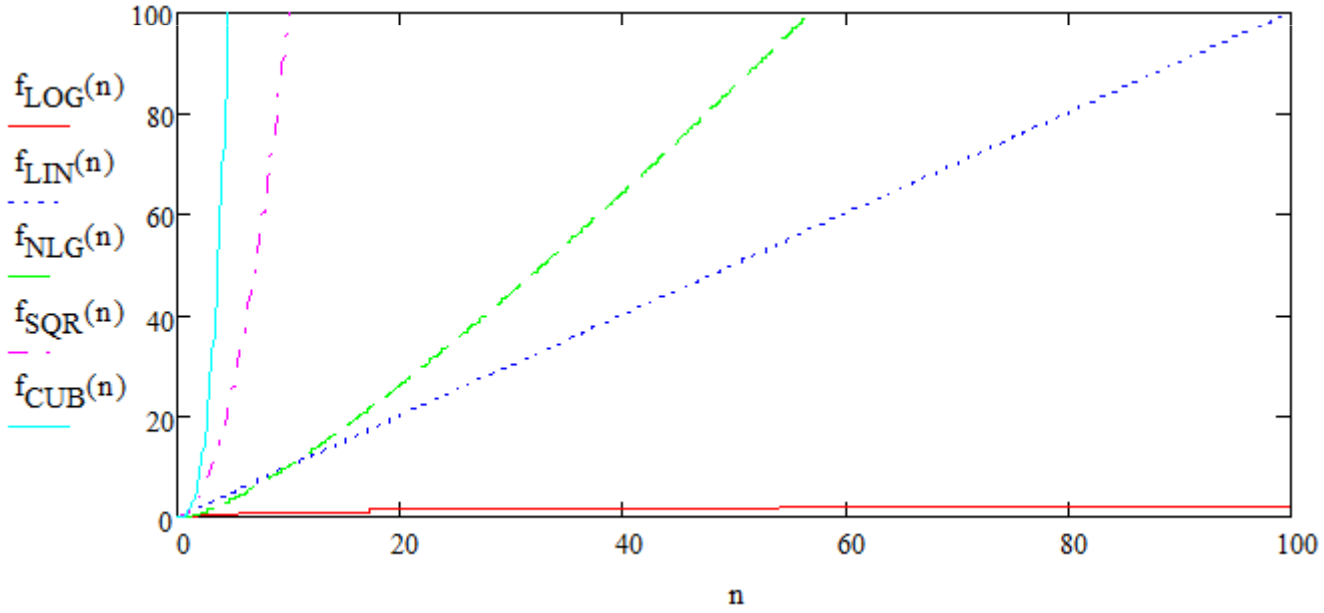
$f_{LOG}(n) := \log(n)$

$f_{LIN}(n) := n$

$f_{NLG}(n) := n \cdot \log(n)$

$f_{SQR}(n) := n^2$

$f_{CUB}(n) := n^3$



As estruturas de dados acima podem ser vistas abstratamente como uma coleção linear de elementos (uma **lista**), tal que inserções e remoções podem ser executadas uma posição arbitrária da coleção. Em alguns casos, alguns modos mais restritivos de acesso são adequados para algumas aplicações, com as subsequentes simplificações. Tais estruturas são: **Filas**, onde inserções ocorrem em um extremo e remoções no outro. **Pilhas**, onde ambas inserções e remoções ocorrem em um extremo (o topo da pilha). Claramente, um e dois ponteiros são tudo o que é necessário para gerenciar uma pilha ou uma fila, respectivamente.

As estruturas de dados padrões revisadas acima são usadas extensivamente em conjunto com os algoritmos da Geometria Computacional. Porém, a natureza dos problemas geométricos tem levado ao desenvolvimento de estruturas de dados não convencionais específicas.

Definições e Notações Gerais

Os objetos considerados normalmente em Geometria Computacional são normalmente conjuntos de pontos no espaço Euclidiano. Um sistema de coordenadas de referência é assumido, tal que cada ponto é representado por um vetor de coordenadas cartesianas da dimensão apropriada. Os objetos geométricos não consistem necessariamente de conjuntos finitos de pontos, mas tem que obdecer a concenvão de ser finitamente especificado (tipicamente, como strings finitas de parâmetros). Então, considera-se além de pontos individuais, a linha reta contendo dois pontos dados, o segmento de linha reta definido pelos seus dois pontos extremos, o plano contendo três pontos dados, o polígono definido por uma (ordenada) sequência ou pontos, etc.

Essa seção não tem pretenção de fornecer definições formais dos conceitos geométricos usados neste curso; ela tem apenas os objetivos de revisar notações que são certamente conhecidas pelo leitor e de introduzir a notação adotada.

Por E^d denota-se o espaço euclidiano *d*-dimensional, isto é, o espaço das *d*-tuplas $(x_1,...,x_d)$ de números reais x_i, $i = 1,..., d$ com métrica $(\Sigma^d_{i=1}\ x_i^2)^{1/2}$.

Revisa-se agora a definição dos principais objetos considerados pela Geometria Computacional.

Fonte: [PREPARATA85]

Point. A d-tuple $(x_1,...,x_d)$ denotes a point p of E^d; this point may be also interpreted as a d-component vector applied to the origin of E^d, whose free terminus is the point p.

Line, **plane**, **linear variety**. Given two distinct points q_1 and q_2 in E^d, the linear combination

$$\alpha\, q_1 + (1 - \alpha)\, q_2 \qquad (\alpha \in \mathrm{R})$$

is a **line** in E^d. More generally, given k linearly independent points $q_1, \ldots, q_k$ in E^d ($k \leq d$), the linear combination

$$\alpha_1\, q_1 + \alpha_2\, q_2 + \ldots + \alpha_{k-1} q_{k-1} + (1 - \alpha_1 - \ldots - \alpha_{k-1})\, q_k$$
$$(\alpha_j \in \mathrm{R}\,,\ j = 1, \ldots, k-1)$$

is a **linear variety** of dimension $(k-1)$ in E^d.

Line segment. Given two distinct points q_1 and q_2 in E^d, if in the expression $\alpha q_1 + (1-\alpha) q_2$ we add the condition $0 \leq \alpha \leq 1$, we obtain the convex combination of q_1 and q_2, i.e.,

$$\alpha\, q_1 + (1 - \alpha)\, q_2 \qquad (\alpha \in \mathrm{R}\,,\ 0 \leq \alpha \leq 1).$$

This convex combination describes the straight line segment joining the two points q_1 and q_2. Normally this segment is denoted as $q_1 q_2$ (unordered pair).

Convex set. A domain D in E^d is *convex* if, for any two points q_1 and q_2 in D, the segment q_1q_2 is entirely contained in D. It can be shown that the intersection of convex domains is a convex domain.

Convex hull. The *convex hull* of a set of points S in E^d is the boundary of the smallest convex domain in Ed containing S.

Polygon. In E^2 a *polygon* is defined by a finite set of segments such that every segment extreme is shared by exactly two edges and no subset of edges has the same property. The segments are the **edges** and their extremes are the **vertices** of the polygon. (Note that the number of vertices and edges are identical.) An *n*-vertex polygon is called an *n-gon*.

A polygon is **simple** if there is no pair of nonconsecutive edges sharing a point. A simple polygon partitions the plane into two disjoint regions, the **interior** (bounded) and the **exterior** (unbounded) that are separated by the polygon (Jordan curve theorem). (Note: in common parlance, the term polygon is frequently used to denote the union of the boundary and of the interior.)

A simple polygon P is **convex** if its interior is a convex set.

A simple polygon P is **star-shaped** if there exists a point z not external to P such that for all points p of P the line segment zp lies entirely within P. (Thus, each convex polygon is also star-shaped.) The locus of the points z having the above property is the kernel of *P.* (Thus, a convex polygon coincides with its own kernel.)

Planar graph. A graph $G = (V, E)$ (vertex set V, edge set E) is *planar* if it can be embedded in the plane without crossings. A straight line planar embedding of a planar graph determines a partition of the plane called *planar subdivision* or *map*. Let v, e, and f denote respectively the numbers of vertices, edges, and regions (including the single unbounded region) of the subdivision. These three parameters are related by the classical *Euler's formula*

$$v - e + f = 2.$$

If we have the additional property that each vertex has degree ≥ 3, then it is a simple exercise to prove the following inequalities

$$v \leq 2/3\, e, \qquad e \leq 3v - 6$$
$$e \leq 3f - 6, \qquad f \leq 2/3\, e$$
$$v \leq 2f - 4, \qquad f \leq 2v - 4$$

which show that v, e and f are pairwise proportional. (Note that the three rightmost inequalities are unconditionally valid.)

Triangulation. A planar subdivision is a *triangulation* if all its bounded regions are triangles. A *triangulation of a finite set S* of points is a planar graph on S with the maximum number of edges (this is equivalent to saying that the triangulation of S is obtained by joining the points of S by nonintersecting straight line segments so that every region internal to the convex hull if S is a triangle).

Polyhedron. In E^3 a polyhedron is defined by a finite set of plane polygons such that every edge of a polygon is shared by exactly one other polygon (adjacent polygons) and no subset of polygons has the same property. The vertices and the edges of the polygons are the vertices and the edges of the polyhedron; the polygons are the *facets* of the polyhedron.

A polyhedron is *simple* if there is no pair of nonadjacent facets sharing a point. A simple polyhedron partitions the space into two disjoint domains, the *interior* (bounded) and the *exterior* (unbounded). (Again, in common parlance the term polyhedron is frequently used to denote the union of the boundary and of the interior.)

The surface of a polyhedron (of genus zero) is isomorphic to a planar subdivision. Thus the numbers v, e, and f of its vertices, edges, and facets obey Euler's formula.

A simple polyhedron is *convex* if its interior is a convex set.

Ponto mais Próximo em um Segmento de Reta

PONTO MAIS PRÓXIMO EM UM SEGMENTO DE RETA UTILIZANDO PRODUTO INTERNO

Projeção do ponto C na reta AB:

$$C' = A + t_{C'}(B - A)$$

Ponto mais próximo de C no segmento AB:

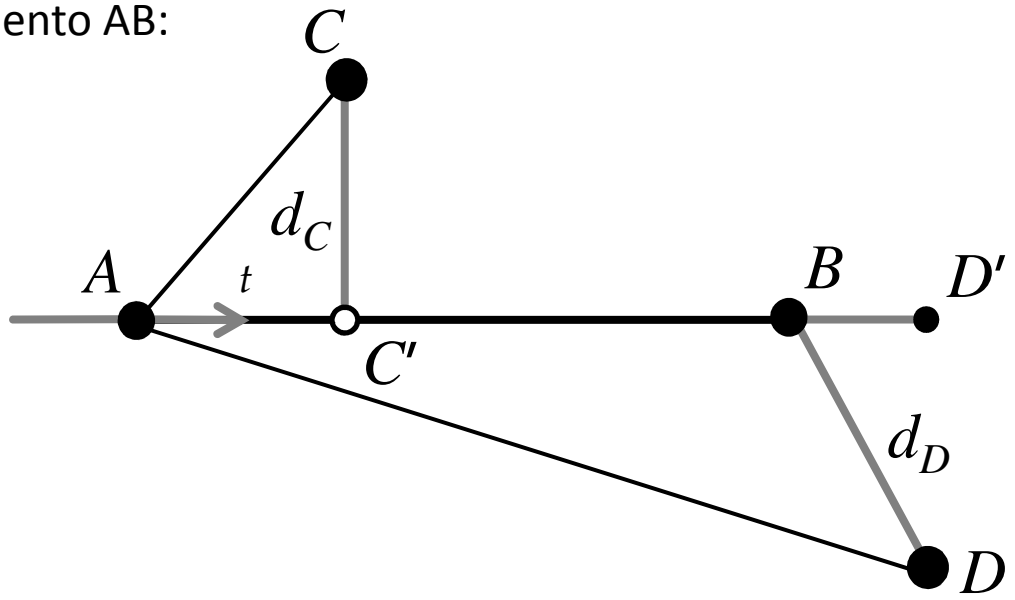
$$P = C'$$

Valor paramétrico do ponto C' no segmento AB:

$$t_{C'} = \frac{\overrightarrow{AB} \circ \overrightarrow{AC}}{\left|\overrightarrow{AB}\right|^2}$$

(produto interno)

$$0 < t_{C'} < 1$$



Projeção do ponto D na reta AB:

$$D' = A + t_{D'}(D - C)$$

Valor paramétrico do ponto D' no segmento AB:

$$t_{D'} = \frac{\overrightarrow{AB} \circ \overrightarrow{AD}}{\left|\overrightarrow{AB}\right|^2}$$

$$t_{D'} > 1$$

Ponto mais próximo de D no segmento AB:

$$P = B$$

Algoritmos para Interseção de Segmentos de Reta

Line Segments and Intersection

Fonte: [SKIENA02]

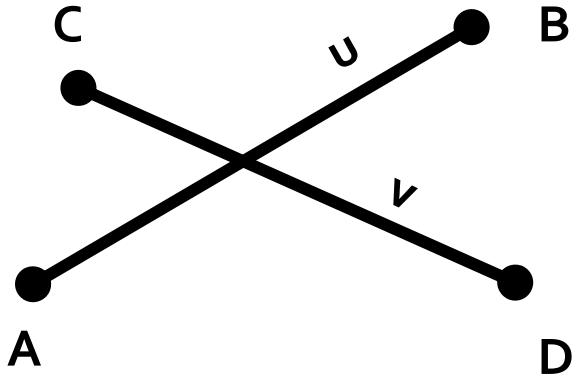
A *line segment s is the portion of a line l which lies between two given points inclusive.* Thus line segments are most naturally represented by pairs of endpoints.

The most important geometric primitive on segments, testing whether a given pair of them intersect, proves surprisingly complicated because of tricky special cases that arise. Two segments may lie on parallel lines, meaning they do not intersect at all. One segment may intersect at another's endpoint, or the two segments may lie on top of each other so they intersect in a segment instead of a single point.

This problem of geometric special cases, or *degeneracy,* seriously complicates the problem of building robust implementations of computational geometry algorithms. Degeneracy can be a real pain in the neck to deal with. Read any problem specification carefully to see if it promises no parallel lines or overlapping segments. Without such guarantees, however, you had better program defensively and deal with them.

The right way to deal with degeneracy is to base all computation on a small number of carefully crafted geometric primitives.

COMO TRATAR A INTERSEÇÃO DE SEGMENTOS DE RETA DE FORMA ROBUSTA E EFICIENTE?



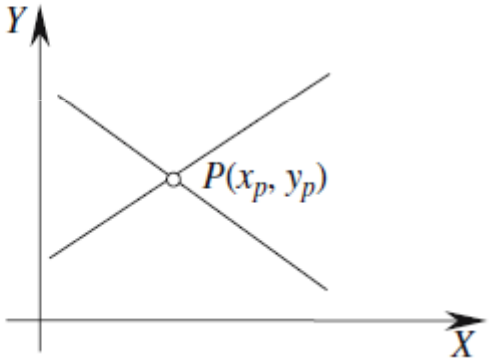
Fonte: Ricardo Marques

Fonte [VINCE05]

1.11.7 Point of intersection of two straight lines

General form of the line equation

Given
$$a_1x + b_1y + c_1 = 0$$
$$a_2x + b_2y + c_2 = 0$$



then
$$\frac{x_P}{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}} = \frac{y_P}{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}} = \frac{-1}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

Intersect at
$$x_P = \frac{c_2b_1 - c_1b_2}{a_1b_2 - a_2b_1} \qquad y_P = \frac{a_2c_1 - a_1c_2}{a_1b_2 - a_2b_1}$$

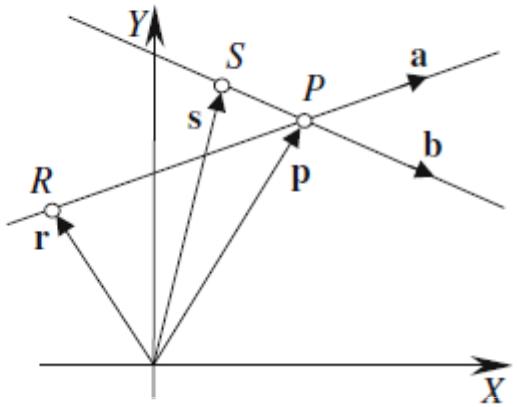
The lines are parallel if $a_1b_2 - a_2b_1 = 0$

Parametric form of the line equation

Given $\mathbf{p} = \mathbf{r} + \lambda\mathbf{a}$ $\mathbf{q} = \mathbf{s} + \varepsilon\mathbf{b}$

where $\mathbf{r} = x_R\mathbf{i} + y_R\mathbf{j}$ $\mathbf{s} = x_S\mathbf{i} + y_S\mathbf{j}$

and $\mathbf{a} = x_a\mathbf{i} + y_a\mathbf{j}$ $\mathbf{b} = x_b\mathbf{i} + y_b\mathbf{j}$



then $$\lambda = \frac{x_b(y_S - y_R) - y_b(x_S - x_R)}{x_b y_a - x_a y_b}$$

and $$\varepsilon = \frac{x_a(y_S - y_R) - y_a(x_S - x_R)}{x_b y_a - x_a y_b}$$

Point of intersection $x_P = x_R + \lambda x_a$ $y_p = y_R + \lambda y_a$

or $x_P = x_S + \varepsilon x_b$ $y_P = y_S + \varepsilon y_b$

The lines are parallel if $x_b y_a - x_a y_b = 0$

Finding Line and Segment Intersections

Line Intersection

In symmetry to the construction of a line from two points (§ 1.5), the intersection of two lines in the plane Line_E2d L1, L2 represented by the equations

$$a_1 x + b_1 y + c_1 = 0,$$
$$a_2 x + b_2 y + c_2 = 0$$

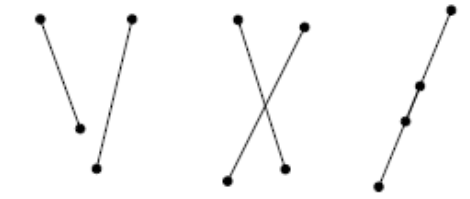
is given by

$$x = +\frac{\begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}, \qquad y = -\frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}.$$

Sherif Ghali
Introduction to Geometric Computing
Springer
2008

```
class Intersection_E2d {
  static returnObject intersect(
                    const Segment_E2d& s1,
                    const Segment_E2d& s2);
  ...
};
```

The two line segments s1 and s2 can be in one of three configurations. There may be no intersection; the two segments may intersect at a single point (possibly coinciding with an endpoint of one or both segments); or the two segments may lie on the same carrying line and have an overlap, in which case they would intersect at an infinite number of points: a segment.



The segment intersection routine is likely to be called often and so we have the following constraints:

1. The code should handle special cases (e.g., one of the two segments is vertical) as easily as possible.
2. The number of floating point operations should be minimized.

The two constraints are satisfied by determining the intersection using the following four turns:

```
TURN s2s = s1.turn(s2.source());
TURN s2t = s1.turn(s2.target());
TURN s1s = s2.turn(s1.source());
TURN s1t = s2.turn(s1.target());
```

If the four predicates suggest that the two segments intersect in their interior, the problem reduces to finding the point of intersection of two lines.

Sherif Ghali
Introduction to Geometric Computing
Springer
2008

Line definition

Storing the coefficients of the parametric form of the line $ax + by + c = 0$ is superior to the single-valued form $y = mx + b$ since lines parallel to the y-axis would be captured easily by setting $a = 0$.

The $ax + by + c = 0$ could be normalized to $Ax + By + 1 = 0$, but, as discussed, doing so would make it impossible to capture lines passing by the origin, and so we choose the following representation for a line:

```
class Line_E2d {
private:
  // The source and target of the line are not stored
  double a, b, c;
public:
  Line_E2d(const Point_E2d& source, const Point_E2d& target);
  ...
};
```

To find the coefficients given two points defining the line, we solve for A and B in $Ax + By + 1 = 0$. If S and T are the two points, we solve the two equations

$$\begin{aligned} AS_x + BS_y + 1 &= 0, \\ AT_x + BT_y + 1 &= 0 \end{aligned}$$

as

$$A = +\frac{\begin{vmatrix} S_y & 1 \\ T_y & 1 \end{vmatrix}}{\begin{vmatrix} S_x & S_y \\ T_x & T_y \end{vmatrix}}, \qquad B = -\frac{\begin{vmatrix} S_x & 1 \\ T_x & 1 \end{vmatrix}}{\begin{vmatrix} S_x & S_y \\ T_x & T_y \end{vmatrix}}.$$

We then avoid the division by storing instead the coefficients of $ax + by + c = 0$, where

$$a = +\begin{vmatrix} S_y & 1 \\ T_y & 1 \end{vmatrix}, \qquad b = -\begin{vmatrix} S_x & 1 \\ T_x & 1 \end{vmatrix}, \qquad c = \begin{vmatrix} S_x & S_y \\ T_x & T_y \end{vmatrix}, \qquad S \neq T.$$

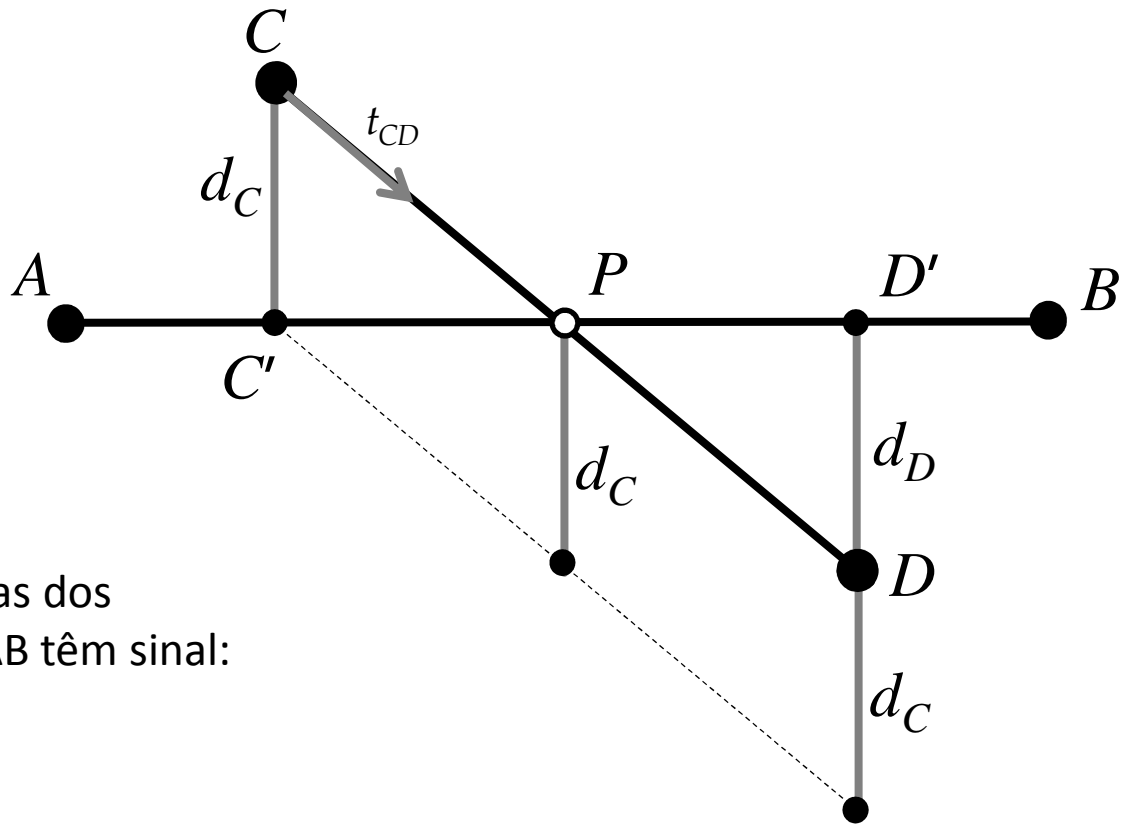
We would define a line by saying that the two points used for its construction are distinct. The word *distinct* appears as a constraint to the solution: The determinant of the matrix defining c above vanishes exactly when the two points S and T coincide. It may be wise to guard against that case by using an assert statement or by throwing an exception (§ 12.2), while ensuring that no run-time penalty is paid by, for example, turning assertions off for production code.

Sherif Ghali
Introduction to Geometric Computing
Springer
2008

INTERSEÇÃO BASEADA NA REPRESENTAÇÃO PARAMÉTRICA DOS SEGMENTOS

$$P = A + t_{AB}(B - A)$$

$$P = C + t_{CD}(D - C)$$



Considere que as distâncias dos pontos C e D para a reta AB têm sinal:

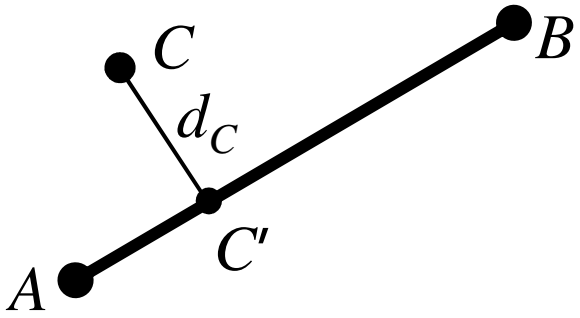
$$d_C > 0 \qquad d_D < 0$$

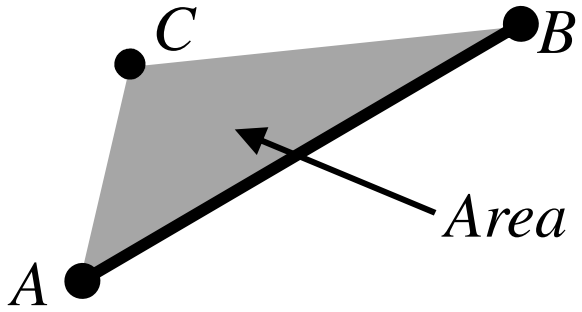
Valor paramétrico do ponto P na reta CD:

$$t_{CD} = \frac{|d_C|}{|d_C| + |d_D|} \qquad t_{CD} = \frac{d_C}{d_C - d_D} \qquad 0 \le t_{CD} \le 1$$

INTERSEÇÃO BASEADA NA REPRESENTAÇÃO PARAMÉTRICA DOS SEGMENTOS

Distância com sinal pode ser substituída por produto vetorial





$$Area = \frac{\left|\overrightarrow{AB}\right| \cdot d_C}{2}$$

$$Area = \frac{\overrightarrow{AB} \times \overrightarrow{AC}}{2}$$

$$Area = \frac{(B-A)\times(C-A)}{2}$$

Definição: (dobro da) área orientada do triângulo

$$orient2d(A,B,C) = (B-A)\times(C-A)$$

Portanto:

$$d_C = \frac{orient2d(A,B,C)}{\left|\overrightarrow{AB}\right|}$$

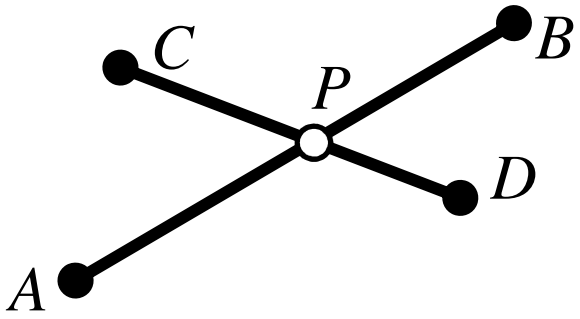
Analogamente:

$$d_D = \frac{orient2d(A,B,D)}{\left|\overrightarrow{AB}\right|}$$

Observe que os sinais das distâncias são resolvidos naturalmente.

INTERSEÇÃO BASEADA NA REPRESENTAÇÃO PARAMÉTRICA DOS SEGMENTOS

Valor paramétrico do ponto P na reta CD:



$$t_{CD} = \frac{d_C}{d_C - d_D}$$

$$d_C = \frac{orient2d(A,B,C)}{\left|\overrightarrow{AB}\right|} \qquad d_D = \frac{orient2d(A,B,D)}{\left|\overrightarrow{AB}\right|}$$

Portanto:

$$t_{CD} = \frac{orient2d(A,B,C)}{orient2d(A,B,C) - orient2d(A,B,D)} \qquad P = C + t_{CD}\left(D - C\right)$$

Analogamente:

$$t_{AB} = \frac{orient2d(C,D,A)}{orient2d(C,D,A) - orient2d(C,D,B)} \qquad P = A + t_{AB}\left(B - A\right)$$

Segment_Segment_Intersection(u, v):

if both endpoints of **u** are over **v** **then**
  return false
end if
$orient2d(C,D,A) > 0$ $orient2d(C,D,B) > 0$

if both endpoints of **u** are under **v** **then**
  return false
end if
$orient2d(C,D,A) < 0$ $orient2d(C,D,B) < 0$

if both endpoints of **v** are over **u** **then**
  return false
end if
$orient2d(A,B,C) > 0$ $orient2d(A,B,D) > 0$

if both endpoints of **v** are under **u** **then**
  return false
end if
$orient2d(A,B,C) < 0$ $orient2d(A,B,D) < 0$

if **u** and **v** are collinear **then**
  return false
end if
$orient2d(C,D,A) = 0$ $orient2d(C,D,B) = 0$
ou
$orient2d(A,B,C) = 0$ $orient2d(A,B,C) = 0$

~~**if** **u** and **v** are parallel **then**~~
  return false
~~**end if**~~
$orient2d(C,D,A) = orient2d(C,D,B)$
ou
$orient2d(A,B,C) = orient2d(A,B,D)$

if **u** and **v** touch at an end point **then** **// there are many situations to check**
  Compute parametric value of touch point on the other line
  if parametric value is between **0** and **1** **then**
    Intersection point is one of the segments' points
    return true
  else
    return false
  end if
end if

$P = B$

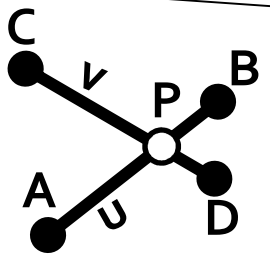
$orient2d(C,D,B) = 0$ $orient2d(C,D,A) < 0$

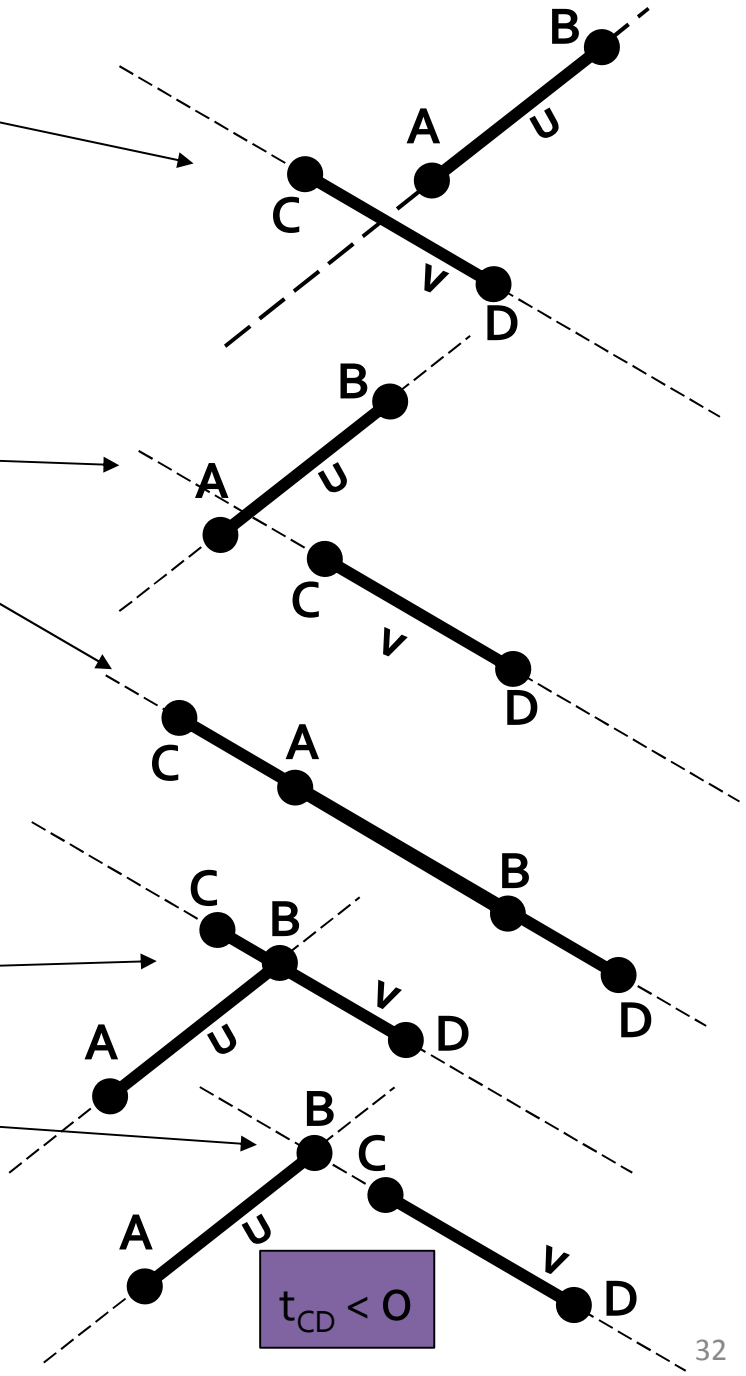
// When get to this point, there is an intersection point

$$t_{CD} = \frac{orient2d(A,B,C)}{orient2d(A,B,C) - orient2d(A,B,D)}$$

$P = C + t_{CD}(D - C)$

return true

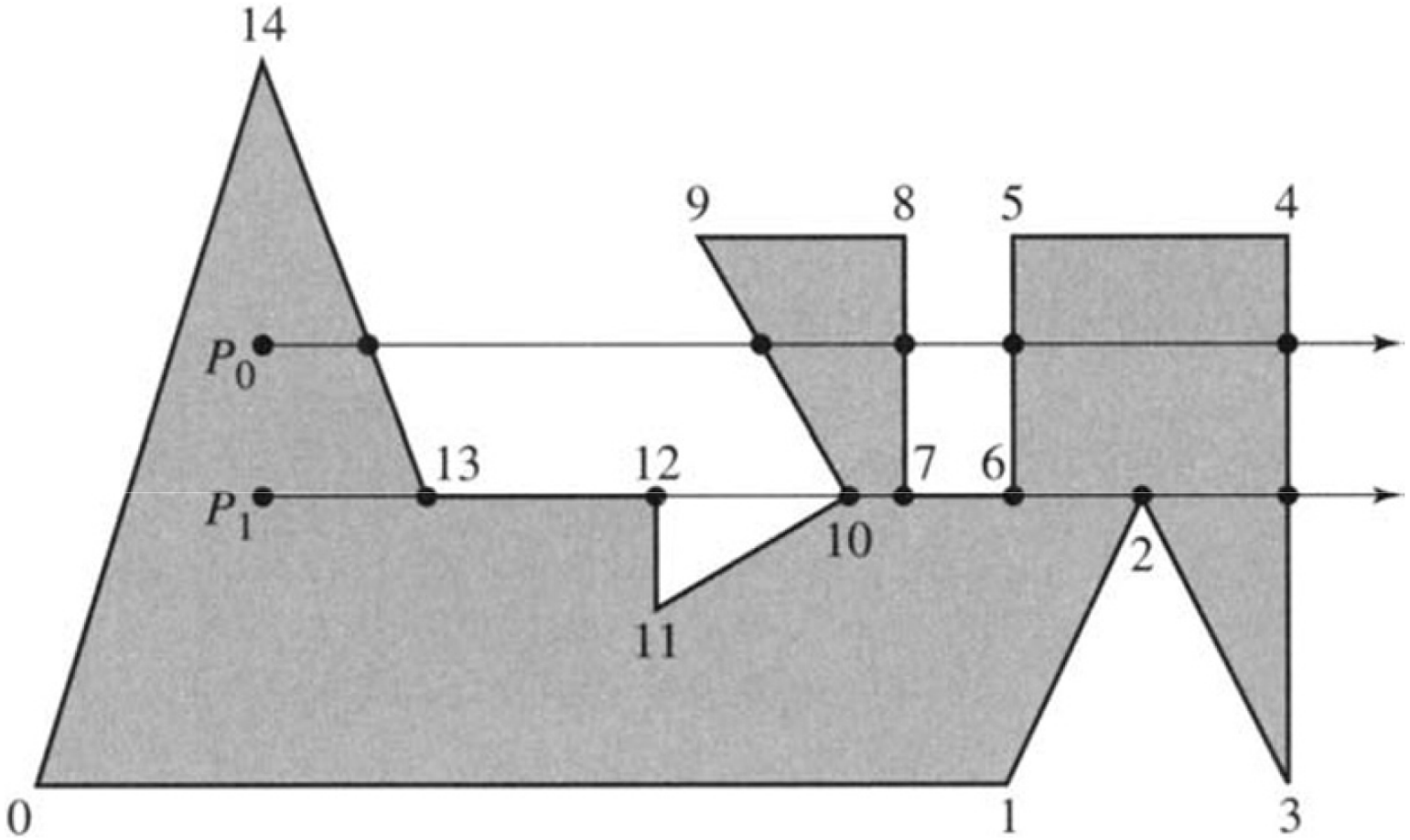




Algoritmo para verificação de ponto dentro de polígono

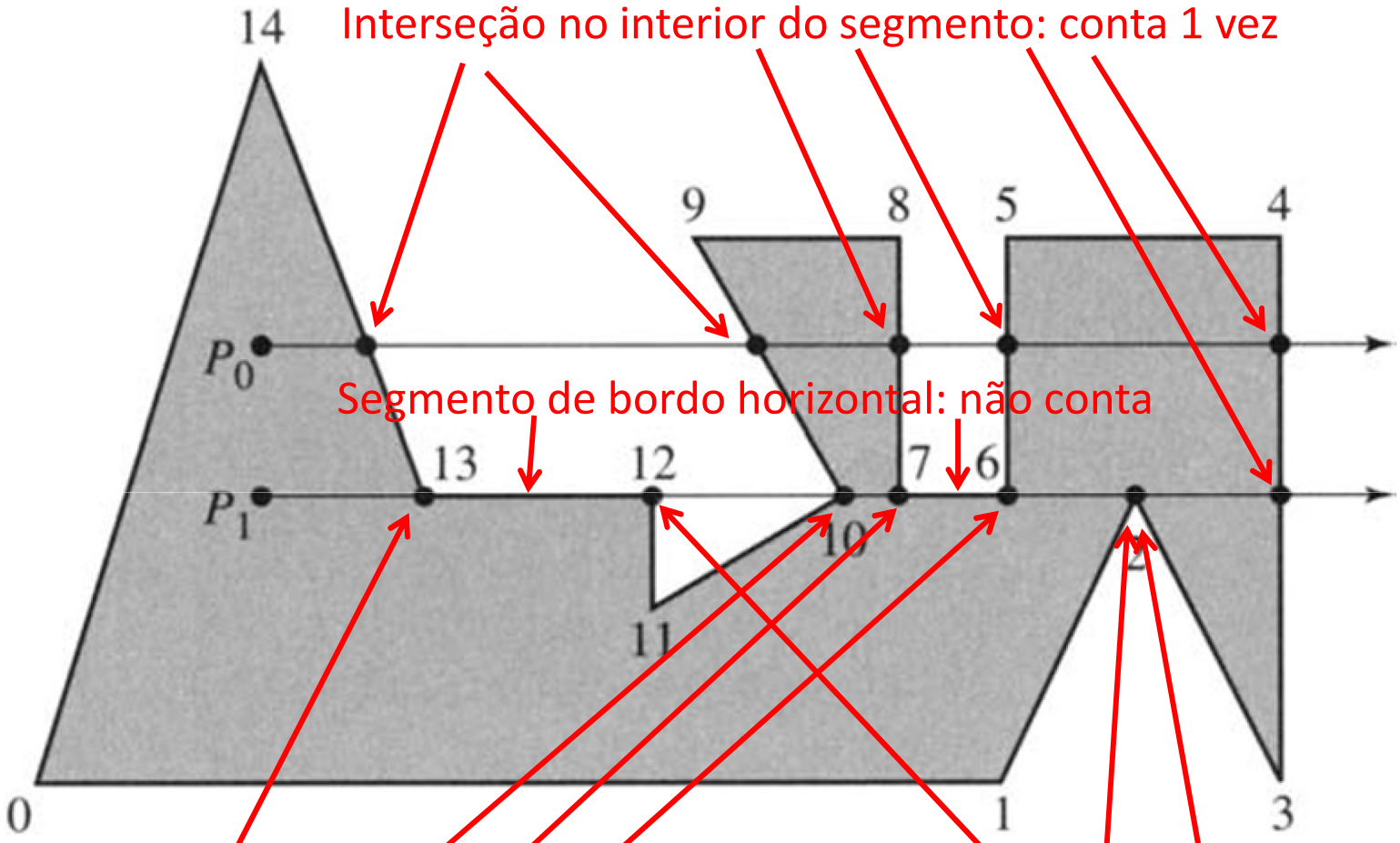
Algoritmo do raio (ou tiro)

Philip Schneider and David Eberly *Geometric Tools for Computer Graphics*, 2003, p.70



Uma semirreta (raio) que parte de qualquer ponto dentro de polígono em uma direção qualquer cortará as curvas no bordo do polígono um número impar de vezes. Se a semirreta cortar a fronteira do polígono um número par de vezes, o ponto está fora do polígono.

Critérios para contar interseções do raio com um segmento de bordo



Interseção no interior do segmento: conta 1 vez

Segmento de bordo horizontal: não conta

Interseção no ponto inferior do segmento: conta 1 vez

Interseção no ponto superior do segmento: não conta

Área Orientada de Polígono

Area Computations Fonte: [SKIENA02]

We can compute the area of any triangulated polygon by summing the area of all triangles. This is easy to implement using the routines we have already developed.

However, there is an even slicker algorithm based on the notion of signed areas for triangles, which we used as the basis for our `ccw` routine. By properly summing the signed areas of the triangles defined by an arbitrary point *p* with each segment of polygon *P* we get the area of *P*, because the negatively signed triangles cancel the area outside the polygon. This computation simplifies to the equation

$$A(P) = \frac{1}{2}\sum_{i=0}^{n-1}(x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

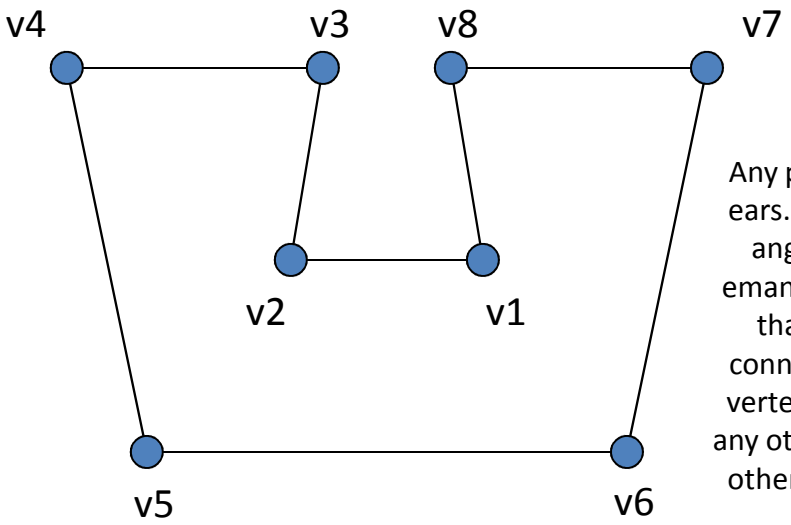
where all indices are taken modulo the number of vertices. See [O'R00] for an exposition of why this works, but it certainly leads to a simple solution:

```
double area(polygon *p)
{
  double total = 0.0; /* total area so far */
  int i, j;           /* counters */
  for (i=0; i<p->n; i++) {
    j = (i+1) % p->n;
    total += (p->p[i][X]*p->p[j][Y]) - (p->p[j][X]*p->p[i][Y]);
  }
  return(total / 2.0);
}
```

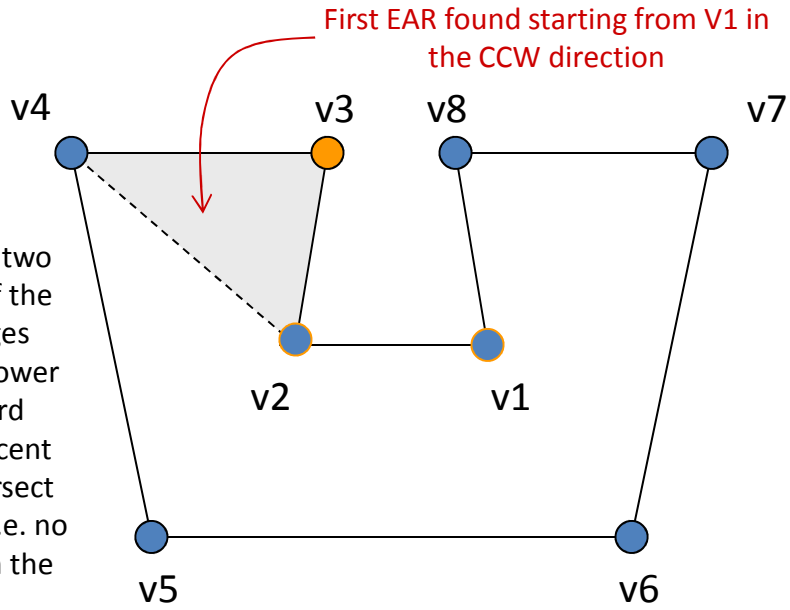
Algoritmos para Tesselagem de Polígonos

How to tessellate a face which is not convex?

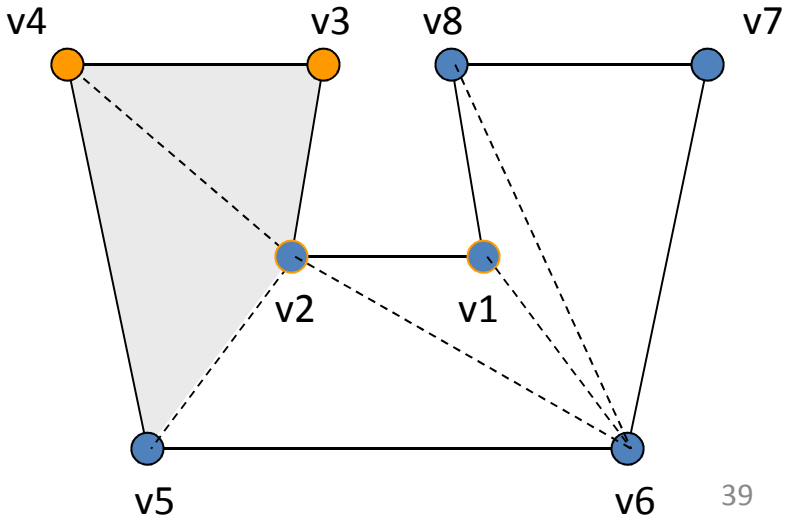
Solution from SKIENA & REVILLA, 2002, Programming Challenges, p.319



Any polygon has at least two ears. An EAR is defined if the angle between the edges emanating the vertex is lower than 180º and the chord connecting the two adjacent vertexes should not intersect any other polygon edge (i.e. no other vertex should lie in the triangle/ear).



Finds EAR of the polygon until remains a single triangle.
POL = 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8
Set two lists with the previous and next vertexes
L = 8 / 1 / 2 / 3 / 4 / 5 / 6 / 7
R = 2 / 3 / 4 / 5 / 6 / 7 / 8 / 1
Updates the list after the first triangle found:
L = 8 / 1 / 2 / **2** / 4 / 5 / 6 / 7
R = 2 / **4** / 4 / 5 / 6 / 7 / 8 / 1
Updates the list after the next triangle found:
L = 8 / 1 / 2 / **2** / **2** / 5 / 6 / 7
R = 2 / **5** / 4 / 5 / 6 / 7 / 8 / 1
Do until the number of triangles is lower then n-2



Predicados Geométricos

Introdução aos Predicados

Now-ancient books on computing frequently use *flow charts*, which conveniently introduce predicates. At the time when FORTRAN in particular, and imperative programming in general, were at the forefront of computing, the use of flow charts was widespread. A flow chart illustrates rather pointedly the path that control may take during computation. This path is sketched using straight lines that connect rectangles and diamonds. Assignment statements appear inside rectangles and if-statements appear inside diamonds. Other elements also exist, but we concentrate here on the parts where the linear path of program control is broken, or *branches*. The functions that are evaluated and that decide the path taken at such branches are called *predicates*. Flow charts have since been replaced by pseudo-code, where changing the linear program control appears in the form of an indentation.

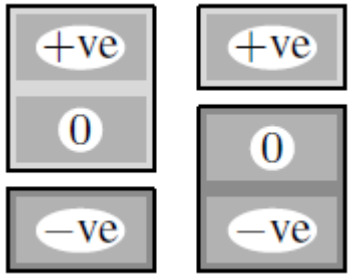
System design has gone back to schematics with the advance of techniques for object-oriented design. One such popular visual language and accompanying methodology, the Unified Modeling Language, promotes that system design should be tackled at a higher granularity. Objects and the messages they pass to each other are identified, but the advance of UML did not supplant—it merely enlarged—pseudo-code and the algorithm design that it captures.

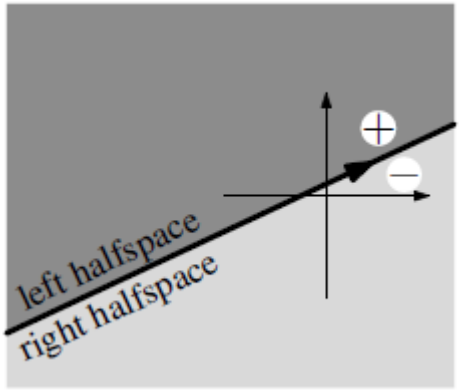
The objective of this section is to argue that crafting good geometric predicates and using them properly is at the center of geometric computing.

Tipo de Retorno de um Predicado

Geralmente pensa-se em predicados como funções com um tipo de retorno Booleano. O tipo booleano pode identificar se um contador atingiu algum limite, se uma tolerância predeterminada tem sido satisfeita, ou se o final de uma lista foi atingido. Tais predicados surgem na computação geométrica, mas um tipo adicional de teste é frequentemente necessário. Devido esse teste geométrico ter três resultados possíveis, refere-se a ele como um teste com ramificação ternária. Ainda mais frequente, tem-se o interesse em formar um predicado binário de três resultados possíveis.

A necessidade por três ramos em um teste pode ser vista quando considera-se uma linha orientada separando o plano. O plano é separado em pontos que se encontram no plano-médio positivo e pontos que se encontram no plano-médio negativo, tão bem como aqueles que se encontram na própria linha. Uma biblioteca geométrica oferecerá tal resposta ternária para os clientes, e o programador da aplicação irá decidir como os predicados devem ser formados.





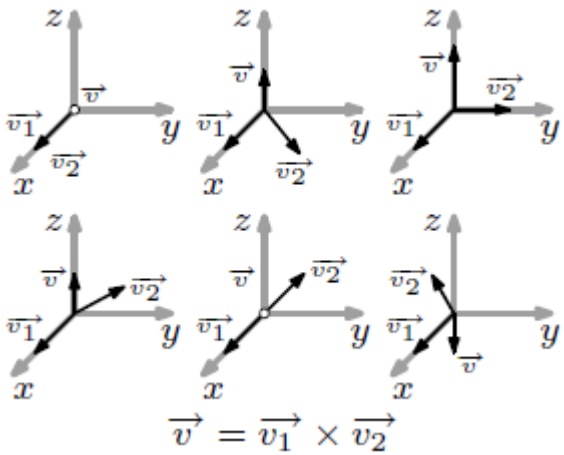
An application might quite suitably need to capture only two cases, the set of points lying on the positive halfplane or the line and the set of points lying in the negative halfplane, for example. But the geometric tests should be offered in such a way that if the application programmer wishes to provide different handling for each of the three cases, it is possible to do so.

Just as we refer to an interval being open if it does not include its extremities and refer to it as closed if it does, we can also talk about either open or closed halfspaces. A left open halfspace consists of the points lying to the left of the line, not including the points on the line itself. A left closed halfspace does include the points on the line. Whether open or closed, we define the boundary of the halfspace as the points on the line. Thus, a closed halfspace includes its boundary and an open halfspace does not. The interior of an interval is the corresponding open interval. A set is termed regular if it is equal to the closure of its interior—an interval is regular if it is closed. By thinking of the predicate as a ternary rather than as a binary predicate we simplify the design of a predicate and leave the decision of choosing among the different representable sets to the client.

O Predicado Orientação no Plano

Determinar a orientação de um ponto em relação a linha definida por dois outros pontos é facilmente definida recorrendo a uma função que nos levará momentaneamente para uma terceira dimensão.

O Produto Vetorial. Existe mais de uma forma de definir o produto vetorial de dois vetores $\mathbf{v}_1$ e $\mathbf{v}_2$. Neste contexto, toma-se a visão clássica (em computação gráfica) que o produto vetorial $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2$ é um vetor que é simultaneamente ortogonal a $\mathbf{v}_1$ e $\mathbf{v}_2$, que obedece a regra da mão direita com relação aos dois vetores, e cuja magnitude é relacionada com as magnitudes dos dois vetores por $|\overrightarrow{v}| = |\overrightarrow{v_1}||\overrightarrow{v_2}| \sin\theta$



Para desenvolver uma intuição sobre produto vetoriais precisa-se apenas considerar como ele varia quando um dos dois vetores, por exemplo $\mathbf{v}_2$, move. Considere posicionar $\mathbf{v}_1$ tal que ele coincida com o eixo *x* positivo. Se $\mathbf{v}_2$ também coincide com o eixo *x*, o produto vetorial será o vetor zero. Isso é natural, pois os dois vetores não definem um plano, ou alternativamente, o paralelogramo que eles definem tem área zero. Agora considere que $\mathbf{v}_2$ gira em direção ao eixo *y*. A magnitude do produto vetorial aumenta até atingir um máximo quando $\mathbf{v}_1$ e $\mathbf{v}_2$ são ortogonais. Como $\mathbf{v}_2$ gira além de *y*, a magnitude de $\mathbf{v}$ retrai, atinge zero quando $\mathbf{v}_2 = -\mathbf{v}_1$, e quando $\mathbf{v}_2$ passa por $-x$, a direção de $\mathbf{v}$ é alinhada com o eixo $-z$.

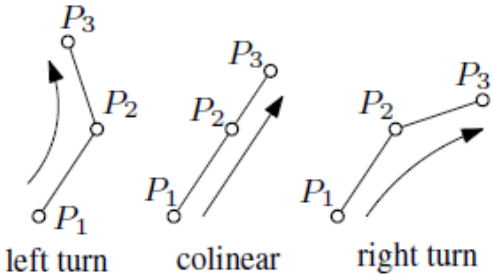
O Projeto do Predicado Orientação 2D

Alguém poderia argumentar que um predicado que reporta se três pontos são colineares seria necessário. Porém, ao invés de implementar tal predicado por si próprio, é mais conveniente implementar um predicado mais geral que determinará também colinearidade. Tal predicado de orientação no plano pode ter a seguinte assinatura

```
SIGN orient2d(const Point* _p1, const Point* _p2, const Point* _p3);
```

onde o tipo do retorno é definido como

```
enum SIGN {
  NEGATIVE = -1,
  ZERO = 0,
  POSITIVE = 1
};
```



Se necessário a implementação de vários predicados convenientes é agora simples. Os seguintes predicados binários delegam a requisição que eles recebem para a função `orient2d`, tal como

```
bool isLeftSide(const Point* _p1, const Point* _p2, const Point* _p3) {
  return orient2d(_p1,_p2,_p3) == POSITIVE;
}
bool areColinear(const Point* _p1, const Point* _p2, const Point* _p3) {
  return orient2d(_p1,_p2,_p3) == ZERO;
}
bool isRightSide(const Point* _p1, const Point* _p2, const Point* _p3) {
  return orient2d(_p1,_p2,_p3) == NEGATIVE;
}
```

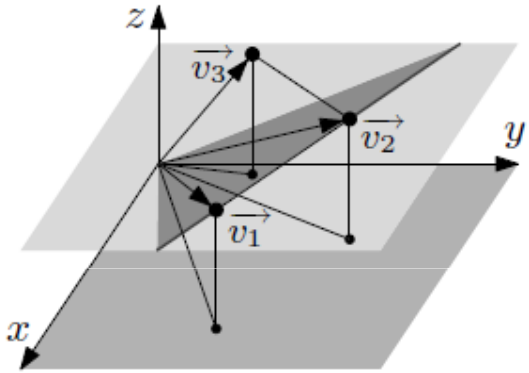
Forma Matricial do Predicado Orientação 2D

Como a linha orientada P_2P_3 divide o plano em pontos encontrando-se sobre, a esquerda e a direita da linha, o sinal da expressão

$$\overrightarrow{P_1P_2} \times \overrightarrow{P_2P_3}$$

identifica a localização do ponto P_3. Se o sinal for positivo, P_3 está a esquerda; se ele for zero, P_3 está sobre a linha; e se ele for negativo, P_3 está a direita da linha. O produto vetorial acima avalia o determinante

$$\begin{vmatrix} x_2 - x_1 & x_3 - x_2 \\ y_2 - y_1 & y_3 - y_2 \end{vmatrix},$$



que por um lado pode ser expandido no determinante 3x3

$$\begin{vmatrix} x_1 & x_2 - x_1 & x_3 - x_2 \\ y_1 & y_2 - y_1 & y_3 - y_2 \\ 1 & 0 & 0 \end{vmatrix},$$

onde os dois valores x_1 e y_1 podem ser arbitrariamente escolhido. Somando a primeira coluna com a segunda e o resultado da segunda com a terceira, obtém-se a equivalente forma homogênea

$$\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{vmatrix}.$$

Interpretando essa expressão como três vetores em 3D ao invés de três pontos em 2D, o teste particular a ser usado irá depender se está testando a inclusão do ponto em questão em um plano-médio aberto ou fechado. Se deseja determinar se um ponto encontra-se no plano-médio esquerdo aberto, testa-se `orient2d(..)==POSITIVE`., e no plano-médio esquerdo fechado, testa-se `orient2d(..)==NEGATIVE`

O Predicado em qual Lado do Círculo

Da mesma forma que dois pontos definem naturalmente uma linha que separa o plano em duas regiões, além da linha separando elas, três pontos no plano P_1, P_2 e P_3 definem um círculo que divide o plano em duas regiões, além do próprio círculo.

Matrix Form of the Side of Circle Predicate

A circle with center (x_c, y_c) and radius r in the plane has the equation

$$(x - x_c)^2 + (y - y_c)^2 = r^2,$$

which expands to

$$(x^2 + y^2) - 2(xx_c + yy_c) + (x_c^2 + y_c^2 - r^2) = 0.$$

More generally,

$$A(x^2 + y^2) + Bx + Cy + D = 0$$

is the equation of a circle in the plane provided that $A \neq 0$.

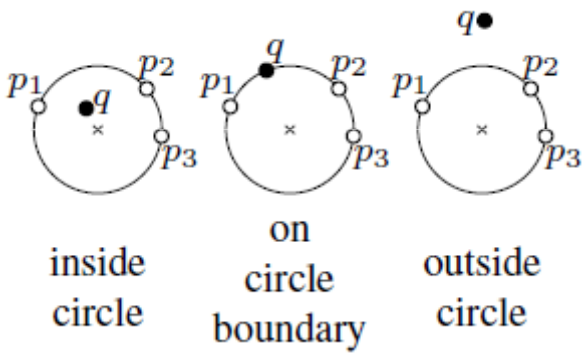
The equation above can be written as the determinant

$$\begin{vmatrix} x^2 + y^2 & x & y & 1 \\ x_1^2 + y_1^2 & x_1 & y_1 & 1 \\ x_2^2 + y_2^2 & x_2 & y_2 & 1 \\ x_3^2 + y_3^2 & x_3 & y_3 & 1 \end{vmatrix} = 0,$$

where

$$A = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}, \qquad B = \begin{vmatrix} x_1^2 + y_1^2 & y_1 & 1 \\ x_2^2 + y_2^2 & y_2 & 1 \\ x_3^2 + y_3^2 & y_3 & 1 \end{vmatrix},$$

$$C = \begin{vmatrix} x_1^2 + y_1^2 & x_1 & 1 \\ x_2^2 + y_2^2 & x_2 & 1 \\ x_3^2 + y_3^2 & x_3 & 1 \end{vmatrix}, \qquad D = \begin{vmatrix} x_1^2 + y_1^2 & x_1 & y_1 \\ x_2^2 + y_2^2 & x_2 & y_2 \\ x_3^2 + y_3^2 & x_3 & y_3 \end{vmatrix}.$$



It is clear that the determinant in Eq. (2.1) vanishes if the point $P(x, y)$ coincides with any of the three points $P_1(x_1, y_1)$, $P_2(x_2, y_2)$, or $P_3(x_3, y_3)$. Moreover, we know from § 2.2 that $A \neq 0$ if and only if the three given points are not colinear.

It is also clear that all the points lying either inside or outside the circle generate a positive determinant and that the points lying on the other side generate a negative determinant. Since exchanging any two rows in Eq. (2.1) would flip the sign of the determinant, the order of the three given points does matter. Clients of this predicate would likely rather not be careful in selecting a particular order for the three points and so it would be appropriate to take a small efficiency hit and compute the 3×3 determinant for the orientation of the three points in addition to computing the 4×4 determinant in Eq. (2.1). And so a point $P(x, y)$ can be classified with respect to a circle defined by three points by evaluating the following equation:

$$\begin{vmatrix} x^2+y^2 & x & y & 1 \\ x_1^2+y_1^2 & x_1 & y_1 & 1 \\ x_2^2+y_2^2 & x_2 & y_2 & 1 \\ x_3^2+y_3^2 & x_3 & y_3 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

$$= \text{side\_of\_circle}(P,P_1,P_2,P_3) \begin{cases} < 0 & \text{inside,} \\ = 0 & \text{on the circle boundary,} \\ > 0 & \text{outside.} \end{cases}$$

Precisão Numérica
Aritmética Exata e Adaptativa

Por que utilizar Aritmética Exata?

Fonte: Ricardo Marques

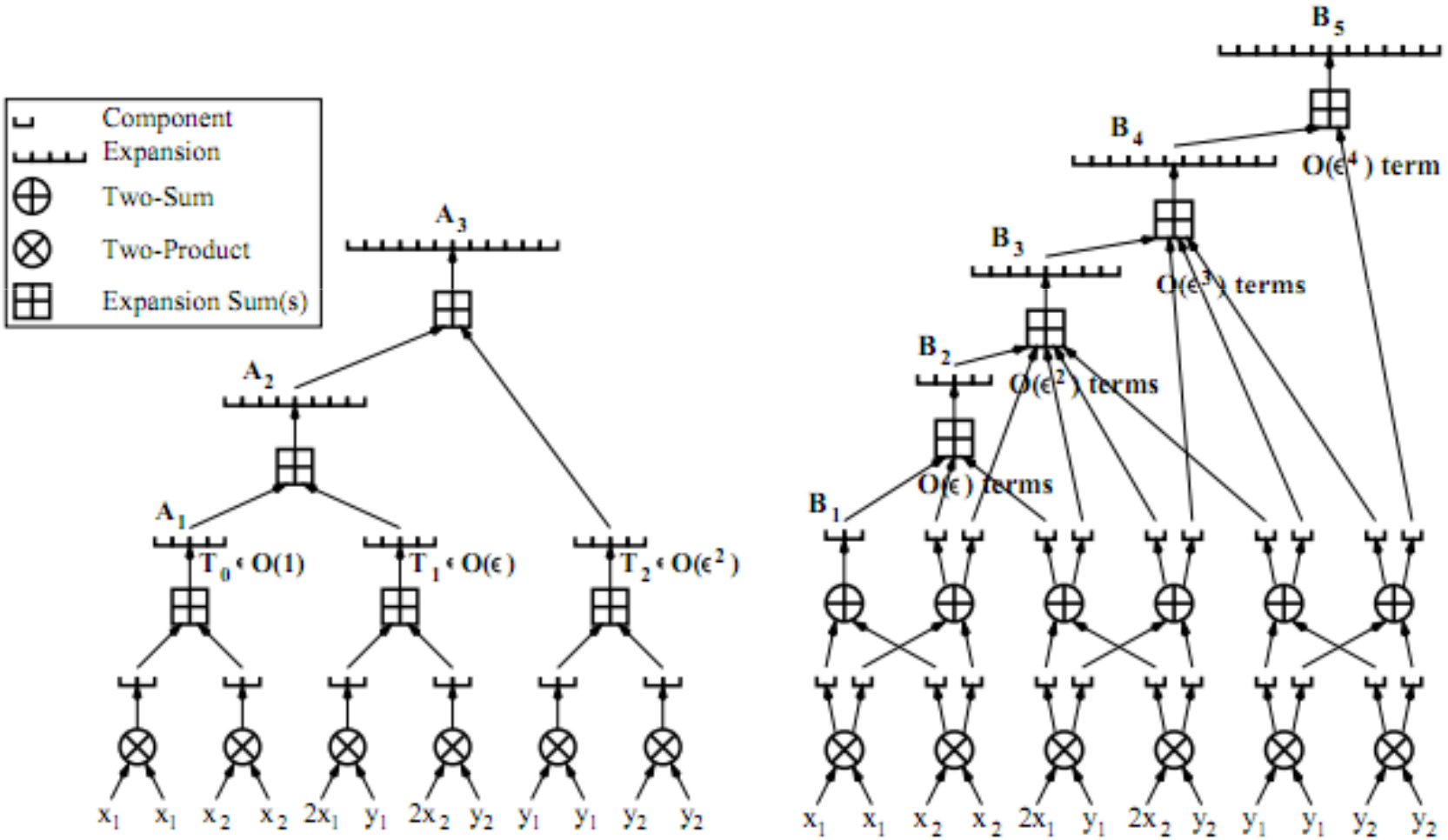
- Utilizar tolerâncias *hard-coded* não resolve!
  - Uma tolerância de 1e-07 pode ser suficiente para modelos de dimensões pequenas.
  - Mas se o modelo possuir dimensões de centenas de km, 1e-07 é inexpressivo. Nesse caso, 1e+00 é uma tolerância bem mais aceitável, representando um erro relativo de 1e-05, ou seja, 1 cm.
  - Ao mesmo tempo, uma tolerância de 1e+00 pode não faz sentido em modelos pequenos.
- Com Aritmética Exata, tolerâncias não são mais necessárias.

O que é Aritmética Exata?

Fonte: Ricardo Marques

- Aritmética Exata é uma técnica para se fazer cálculos com alto nível de precisão
  - 1e-08 é zero? -3.1415e-10 é zero?
- Evita erros de arredondamento:
  - 1e+08 + 1e-16 = 1e+08 ???
  - Nos operadores de aritmética exata, todo número (double) de entrada é quebrado em duas componentes (numéricas) não-sobrejacentes e com ordem de grandezas diferentes.
  - Ao se utilizar sucessivos operadores, componentes podem ser quebradas novamente. Ao fim, todas as componentes geradas são unidas, minimizando erro numérico.

O que é Aritmética Exata Adaptativa?



Numerical analysis is clearly the first place to look for answers about accuracy in a world of approximations. A lot is known about the accuracy of the output of a computation given the accuracy of the input. For example, to get precise answers with linear problems one would have to perform computations using four to five times the precision of the initial data. In the case of quadratic problems, one would need forty to fifty times the precision. Sometimes there may be guidelines that help one improve the accuracy of results. Given the problems with floating point arithmetic, one could try other types of arithmetic.

Bounded Rational Arithmetic. This is suggested in [Hoff89] and refers to restricting numbers to being rational numbers with denominators that are bounded by a given fixed integer. One can use the method of continued fractions to find the best approximation to a real by such rationals.

Infinite Precision Arithmetic. Of course, there are substantial costs involved in this.

"Exact" Arithmetic. This does not mean the same thing as infinite precision arithmetic. The approach is described in [Fort95]. The idea is to have a fixed but relatively small upper bound on the bit-length of arithmetic operations needed to compute geometric predicates. This means that one can do integer arithmetic. Although one does not get "exact" answers, they are reliable. It is claimed that boundary-based **faceted** modelers supporting regularized set operators can be implemented with minimal overhead (compared with floating point arithmetic). Exact arithmetic works well for linear objects but has problems with smooth ones. See also [Yu92] and [CuKM99].

Interval Analysis. See Chapter 18 for a discussion of this and also [HuPY96a] and [HuPY96b].

Just knowing the accuracy is not always enough if it is w
like. Geometric computations often involve many steps. Rathe
accuracy only after data structures and algorithms have bee
perhaps also use accuracy as one criterion for choosing the
algorithms.

One cause for the problem indicated in Figure 5.48 is that
ent computations to establish a common fact. The question
segment intersected the edge of the cube was answered twice –
f and second by using the face **g**. The problem is in the redundancy in the representation of the edge and the fact that the intersection is determined from a collection of isolated computations. If one could represent geometry in a nonredundant way, then one would be able to eliminate quite a few inconsistency problems. Furthermore, the problem shown in Figure 5.48 would be resolved if, after one found the intersection with face **f**, one would check for intersections with all the faces adjacent to **f** and then resolve any inconsistencies.

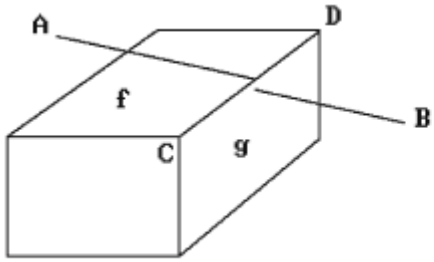


Figure 5.48. Intersection inconsistencies due to round-off errors.

Max K. Agoston
Computer Graphics and Geometric Modeling
Springer 2004

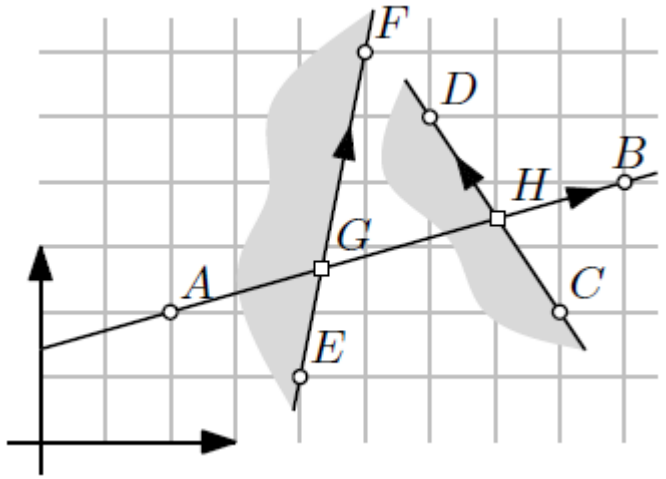
Sherif Ghali
Introduction to Geometric Computing
Springer
2008

But how do we know, when starting the design of a system, whether floating point numbers are adequate? The answer is sometimes easy. It is clear that interactive computer games, or systems that need to run in real time in general, cannot afford to use data types not provided by the hardware. This restricts the usable data types to int, long, float, and/or double. It is also clear that systems that perform Boolean operations on polygons or solids, such as the ones discussed in Chapter 28, will need to use an exact number type. In general, however, this is an important decision that needs to be made for each individual system. Genericity is a powerful device at our disposal to attempt to delay the choice of number type as long as possible, but to generate one executable or program, the various compromises have to be weighed and the decision has to be made.

At this time there is no silver bullet to determine whether to sacrifice efficiency and use an exact number type. A simple rule of thumb is to consider the compromise between speed and accuracy. If the system requirements suggest speed, then we have to sacrifice accuracy, and vice versa. The answer is of course easy if neither is required, but it is more often the case that both are.

Sherif Ghali
Introduction to Geometric Computing
Springer
2008

This theme is the topic of Chapter 7, but lest this issue appear to be of mere theoretical interest, an example is warranted. Consider clipping a segment AB in the plane by the two positive (i.e., left) halfspaces of CD and EF. In theory the resulting clipped segment does not depend on the order of the two clipping operations. The code below uses the type **float** to compute the final intersection point directly (Gd) by intersecting AB and EF, and also computes the ostensibly identical point indirectly (Gi) by first computing AH then intersecting AH and EF.



Sherif Ghali
Introduction to Geometric Computing
Springer
2008

```
int main()
{
   const Point_E2f A(2,2), B(9,5);
   const Point_E2f C(8,2), D(6,5);
   const Point_E2f E(4,1), F(5,6);

   const Segment_E2f AB(A,B), CD(C,D), EF(E,F);

   const Point_E2f Gd = intersection_of_lines(AB, EF);
   const Point_E2f H = intersection_of_lines(AB, CD);

   const Segment_E2f AH(A,H);
   const Point_E2f Gi = intersection_of_lines(AH, EF);

   // assert( Gd == Gi ); // fails

   print( Gd.x() );
   print( Gi.x() );

   print( Gd.y() );
   print( Gi.y() );
}
```

Sherif Ghali
Introduction to Geometric Computing
Springer
2008

After finding that the coordinates differ, we print the sign bit, the exponent, and the mantissa of the two x-coordinates then those of the y-coordinates (see § 7.5).

```
0 10000001 00011010000000000000000
0 10000001 00011010000000000000001
0 10000000 10000100000000000000000
0 10000000 10000100000000000000000
```

And so we see that the direct computation of the x-coordinate leads to a mantissa with a long trail of zeros, whereas the indirect computation leads to an ending least-significant bit of 1. This single-bit difference suffices for the equality operator to conclude that the two points are not equal. Performing the same computation using a combination of indirect steps only reduces the quality of the resulting floating point coordinates.

Predicados com Aritmetica exata