PUC RIO

Universidade Federal Fluminense

Computer Graphics for Engineering

TU Graz
Graz University of Technology

ifb

numsim
Numerical simulation in technical sciences

Computational Geometry

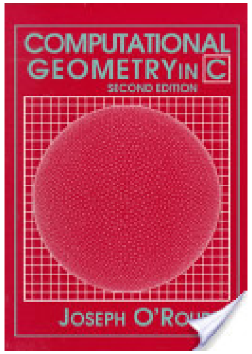
Luiz Fernando Martha
André Pereira

Graz, Austria
June 2014

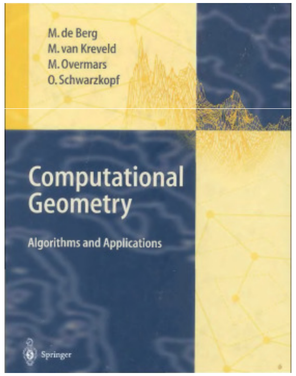
Contents

References and Sources

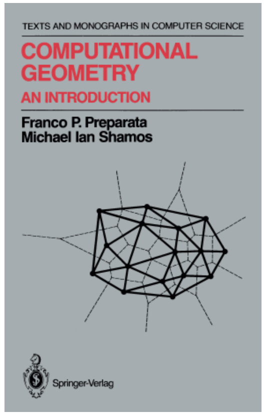
References and Sources



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



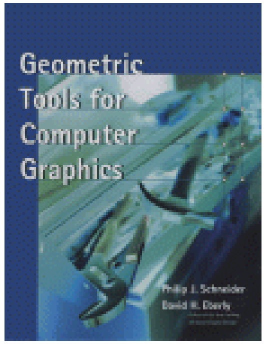
[BERG97]
M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf
Computational Geometry - Algorithms and Applications
Springer, 1997



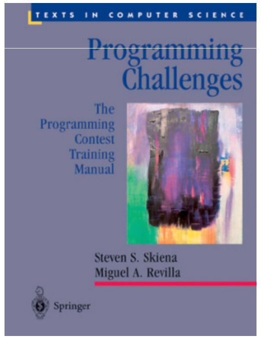
[PREPARATA85]
Franco P. Preparata, Michael Ian Shamos
Computational Geometry – An Introduction
Springer-Verlag, 1985

References and Sources

[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

Introduction and Scope

Introduction and Scope

Fonte: [OROURKE98]

Computational geometry broadly construed is the study of algorithms for solving geometric problems on a computer. The emphasis in this course is on the design of such algorithms, with somewhat less attention paid to analysis of performance.

There are many brands of geometry, and what has become known as "computational geometry" is primarily discrete and combinatorial geometry. Thus polygons play a much larger role in this course than do regions with curved boundaries. Much of the work on continuous curves and surfaces falls under the rubrics of "geometric modeling" or "solid modeling", a field with its own conferences and text, distinct from computational geometry. Of course there is substantial overlap, and there is no fundamental reason for the fields to be partitioned this way; indeed they seem to be merging to some extent [

Introduction and Scope

Fonte: [BERG97]

Computational geometry emerged from the field of algorithms design and analysis in the late 1970s. It has grown into a recognized discipline with its own journals, conferences, and a large community of active researchers. The success of the field as a research discipline can on the one hand be explained from the beauty of the problems studied and the solutions obtained, and, on the other hand, by the many application domains— computer graphics, geographic information systems (GIS), robotics, and others—in which geometric algorithms play a fundamental role.

For many geometric problems the early algorithmic solutions were either slow or difficult to understand and implement. In recent years a number of new algorithmic techniques have been developed that improved and simplified many of the previous approaches. In this textbook we have tried to make these modern algorithmic solutions accessible to a large audience.

Introduction and Scope

Fonte: [PREPARATA85]

A large number of applications areas have been the incubation bed of the discipline nowadays recognized as Computational Geometry, since they provide inherently geometric problems for which efficient algorithms have to be developed. Algorithmic studies of these and other problems have appeared in the past century in the scientific literature, with an increasing intensity in the past two decades. Only very recently, however, systematic studies of geometric algorithms have been undertaken, and a growing number of researchers have been attracted to this discipline, christened "Computational Geometry" in a paper by M. I. Shamos (1975a).

One fundamental feature of this discipline is the realization that classical characterizations of geometric objects are frequently not amenable to the design of efficient algorithms. To obviate this inadequacy, it is necessary to identify the useful concepts and to establish their properties, which are conducive to efficient computations. In a nutshell, computational geometry must reshape-whenever necessary-the classical discipline into its computational incarnation.

Introduction and Scope

Fonte: [SCHNEIDER03]

The field of computational geometry is quite large and is one of the most rapidly advancing fields in recent times. This chapter is by no means comprehensive. The general topics covered are binary space-partitioning (BSP) trees in two and three dimensions, point-in-polygon and point-in-polyhedron tests, convex hulls of finite point sets, Delaunay triangulation in two and three dimensions, partitioning of polygons into convex pieces or triangles, containment of point sets by circles or oriented boxes in two dimensions and by spheres or oriented boxes in three dimensions, area calculations of polygons, and volume calculations of polyhedra.

The emphasis is, of course, on algorithms to implement the various ideas. However, attention is given to the issues of computation when done within a floating point number system. Particular themes arising again and again are determining when points are collinear, coplanar, cocircular, or cospherical. This is easy to do when the underlying computational system is based on integer arithmetic, but quite problematic when floating-point arithmetic is used.

The Need for Data Structures

Source: Will Thacker . Lecture notes on Data Structures at Winthrop University

The Need for Data Structures

- Data structures organize data
    - This gives ***more efficient programs.***
- More powerful computers encourage more complex applications.
- More complex applications demand more calculations.
- Complex computing tasks are unlike our everyday experience.

Organization

- Any organization for a collection of records can be searched, processed in any order, or modified.
  - The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.
- A solution is said to be ***efficient*** if it solves the problem within its ***resource constraints***.
  - Space
  - Time
- The ***cost*** of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

- Select a data structure as follows
  1. Analyze the problem to determine the resource constraints a solution must meet.
  2. Determine basic operations that must be supported. Quantify resource constraints for each operation.
  3. Select the data structure that best meets these requirements.
- Some questions to ask:
  - Are all the data inserted into the structure at the beginning or are insertions interspersed with other operations?
  - Can data be deleted?
  - Are the data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

- Each data structure has costs and benefits.
- Rarely is one data structure better than another in all situations.
- A data structure requires:
  - space for each data item it stores,
  - time to perform each basic operation,
  - programming effort.
- Each problem has constraints on available time and space.
- Only after a careful analysis of problem characteristics can we know the best data structure for the task.

The Need for Data Structures

Geometric algorithms involve the manipulation of objects which are not handled at the machine language level. The user must therefore organize these complex objects by means of the simpler data types directly representable by the computer. These organizations are universally referred to as **data structures**.

The most common complex objects encountered in the design of geometric algorithms are sets and sequences (ordered sets). Data structures particularly suited to these complex combinatorial objects are well described in the standard literature on algorithms. Suffice it here to review the classification of these data structures, along with their functional capabilities and computational performance.

Let S be a set represented in a data structure and let u be an arbitrary element of a universal set of which S is a subset. The fundamental operations occurring in set manipulation are:

1. **MEMBER**(u,S). Is $u \in S$? (YES/NO answer.)
2. **INSERT**(u,S). Add u to S.
3. **DELETE**(u,S). Remove u from S.

Fonte: [PREPARATA85]

Supose now that $\{S_1, S_2, . . . , S_k\}$ is a collection of sets (with pairwise empty intersection). Useful operations on this collection are:

4. **FIND**(u). Report j, if $u \in S$.

5. **UNION**(S_i,S_j; S_k). Form the union of S_i and S_j and call it S_k.

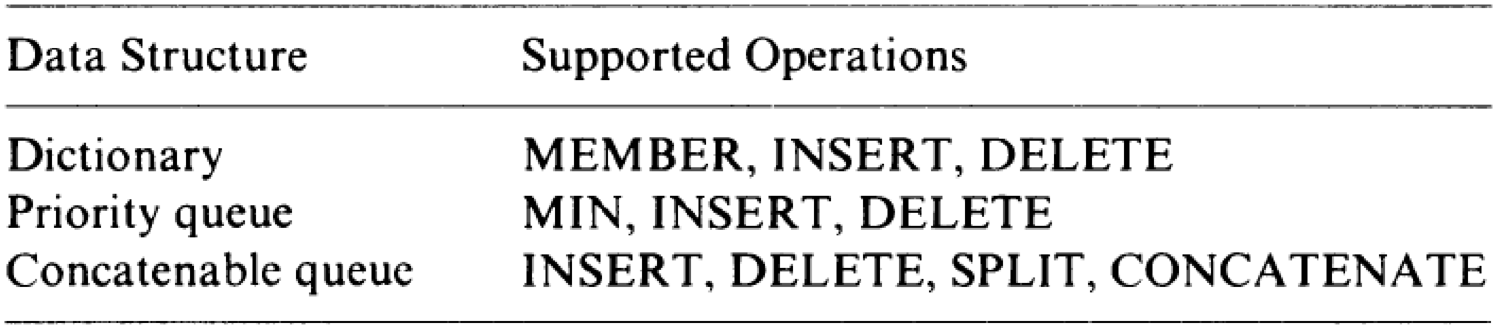
When the universal set is totally ordered, the following operations are very important:

6. **MIN**(S). Report the minimum element of S.

7. **SPLIT**(u,S). Partition S into$\{S_1,S_2\}$, so that $S_1 = \{v: v \in S \text{ and } v \leq u\}$ and $S_2 = S - S_1$.

8. **CONCATENATE**(S_1,S_2). Assuming that, for arbitrary $u' \in S_1$ and $u'' \in S_2$ we have $u' \leq u''$, form the ordered set $S = S_1 \cup S_2$.

Data structures can be classified on the basis of the operations they support (regardless of efficiency). Thus for ordered sets we have the following table.



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

For efficiency, each of these data structures is normally realized as a height-balanced binary search tree (often an AVL or a 2-3-tree). With this realization, each of the above operations is performed in time proportional to the logarithm of the number of elements stored in the data structure; the storage is proportional to the set size.

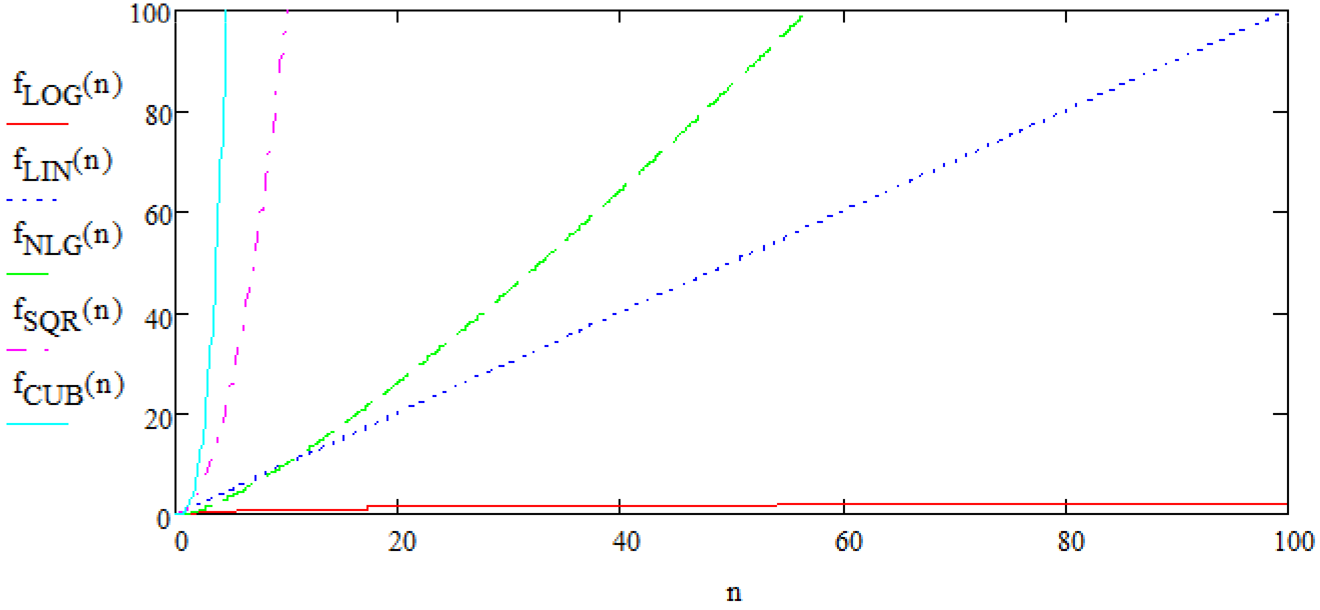
$f_{LOG}(n) := \log(n)$

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

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

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

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



The above data structures can be viewed abstractly as a linear array of elements (a **list**), so that insertions and deletions can be performed in an arbitrary position of the array. In some cases, some more restrictive modes of access are adequate for some applications, with the ensuing simplifications.

Such structures are: **Queues**, where insertions occur at one end and deletions at the other; **Stacks**, where both insertions and deletions occur at one end (the stack-top). Clearly, one and two pointers are all that is needed for managing a stack or a queue, respectively.

Unordered sets can always be handled as ordered sets by artificially imposing an order upon the elements (for example, by giving "names" to the elements and using the alphabetical order). A typical data structure for this situation is the following.

Definitions and Notations

Definitions and Notations

The objects considered in Computational Geometry are normally sets of points in Euclidean space. 3 A coordinate system of reference is assumed, so that each point is represented as a vector of cartesian coordinates of the appropriate dimension. The geometric objects do not necessarily consist of finite sets of points, but must comply with the convention to be finitely specifiable (typically, as finite strings of parameters). So we shall consider, besides individual points, the straight line containing two given points, the straight line segment defined by its two extreme points, the plane containing three given points, the polygon defined by an (ordered) sequence or points, etc.

This section has no pretence of providing formal definitions of the geometric concepts used in this book; it has just the objectives of refreshing notions that are certainly known to the reader and of introducing the adopted notation.

By E^d we denote the d-dimensional Euclidean space, i.e., the space of the d-tuples $(x_1,\ldots,x_\mathrm{d})$ of real numbers x_1, $i = 1,\ldots, d$ with metric $(\Sigma^{d}_{\mathrm{i}=1} \mathrm{x}_\mathrm{i}^2)^{1/2}$.

We shall now review the definition of the principal objects considered by Computational Geometry.

Fonte: [PREPARATA85]

Point. A d-tuple $(x_1,\ldots,x_d)$ 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.

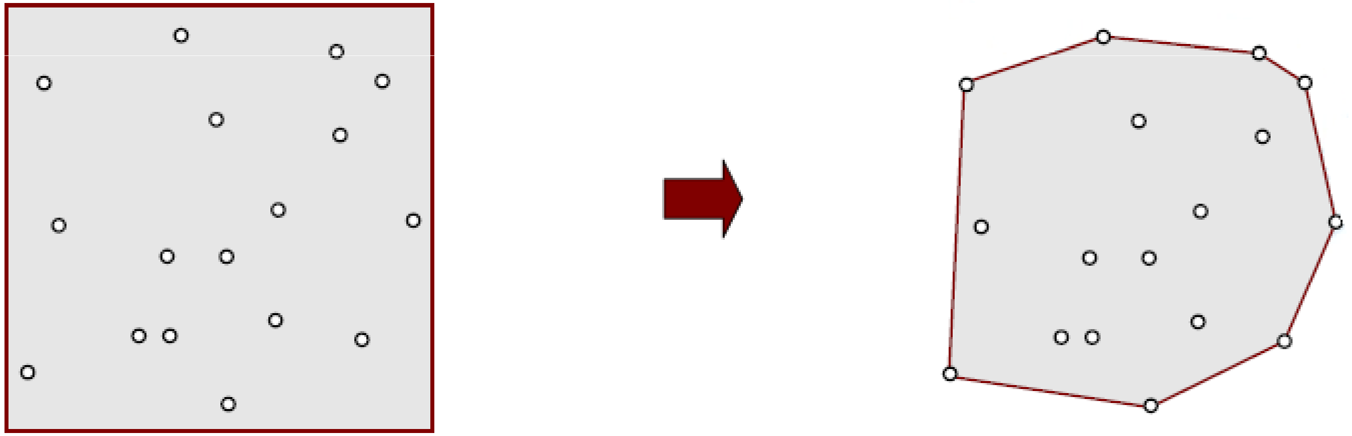
Line segment. Given two distinct points q_1 and q_2 in E^d, if in the expression $\alpha q_1+(1 - \alpha)q_2$ we add the condition $0 \leq \alpha \leq 1$, we obtain the convex combination of q_1 and q_2, i.e.,

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

This convex combination describes the straight line segment joining the two points q_1 and q_2. Normally this segment is denoted as q_1q_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.



menor objeto convexo possível!

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

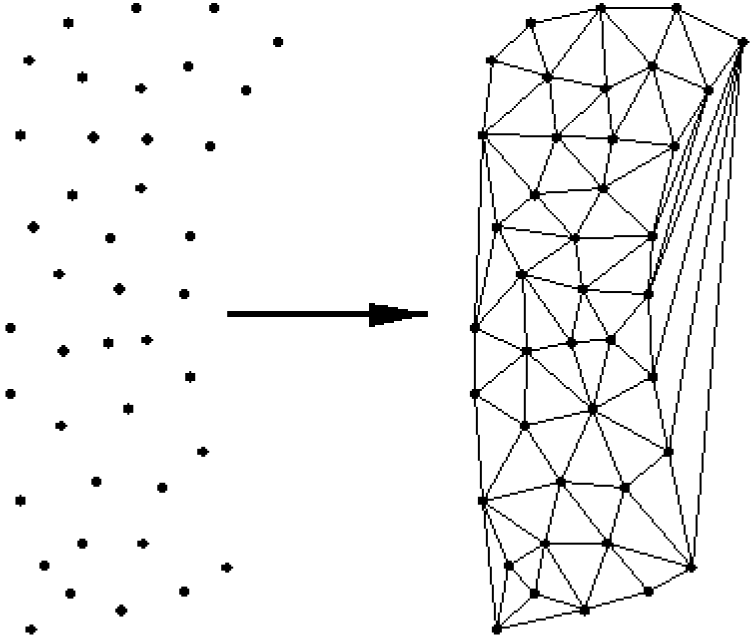
$$v \leq 2/3\, e, \qquad e \leq 3v - 6$$

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

$$v \leq 2f - 4, \qquad f \leq 2v - 4$$

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).

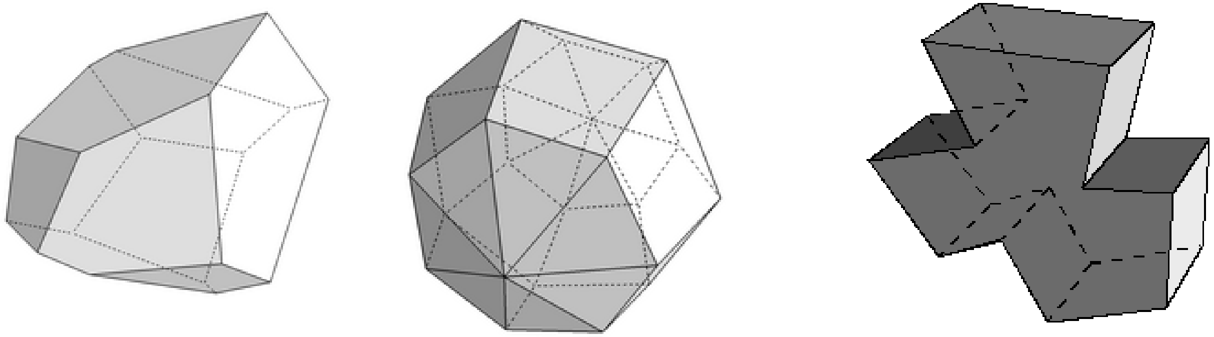


Polyhedron. In E^3 a polyhedron is defined by a finite set of plane polygons such that every edge of a polygon is shared by exactly one other polygon (adjacent polygons) and no subset of polygons has the same property. The vertices and the edges of the polygons are the vertices and the edges of the polyhedron;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

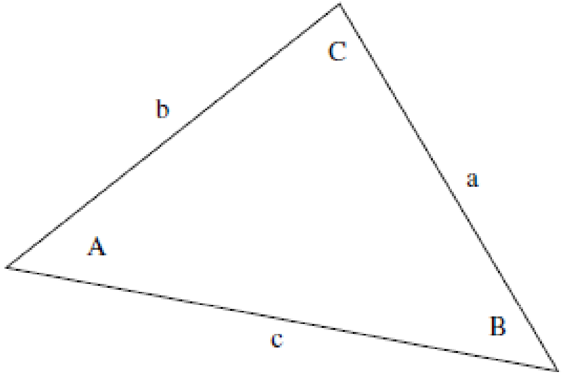
A simple polyhedron is *convex* if its interior is a convex set.

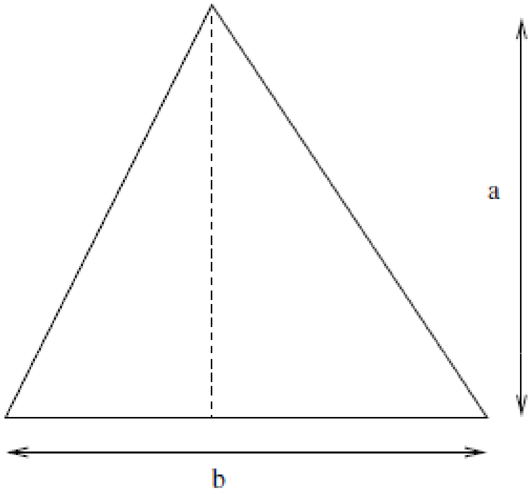


Oriented Area of Polygons

Area Computations Fonte: [SKIENA02]

We can calculate the area of a triangle, from the coordinates of its vertices, evaluating the cross product defined below. Note that this calculation is easily implementable..





$$2 \cdot A(T) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y$$

```
double signed_triangle_area(const Point* _A, const Point* _B, const Point* _C)
{
  return( (_A[X]*_B[Y] - _A[Y]*_B[X] + _A[Y]*_C[X]
         - _A[X]*_C[Y] + _B[X]*_C[Y] - _C[X]*_B[Y]) / 2.0 );
}
```

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

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

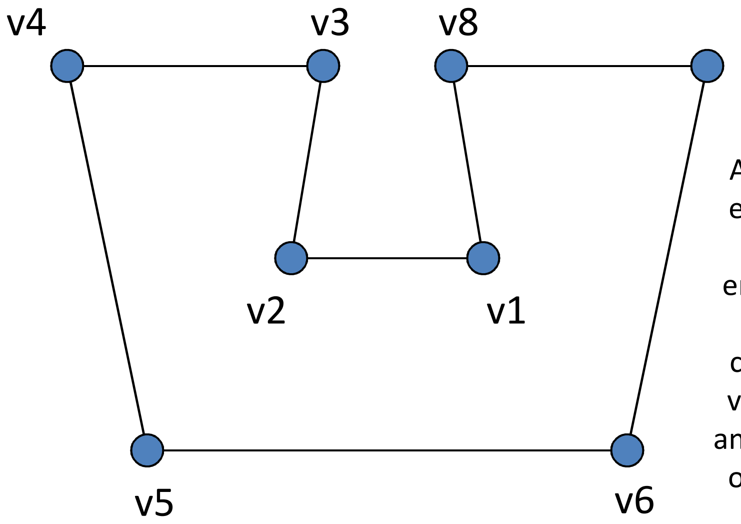
where all indices are taken modulo the number of vertices. See [O'R00] for an exposition of why this works, but it certainly leads to a simple solution:

```
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);
}
```

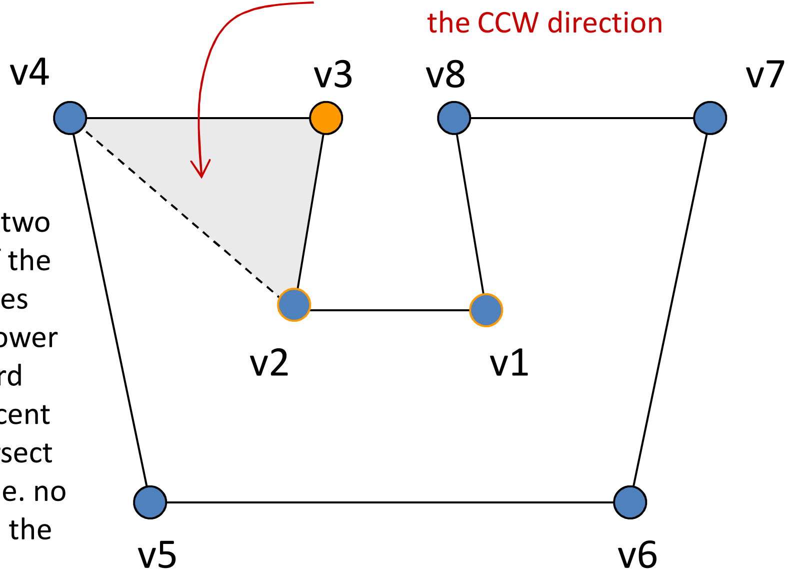
Algorithm for Polygons Tessellation

How to tessellate a face which is not convex?

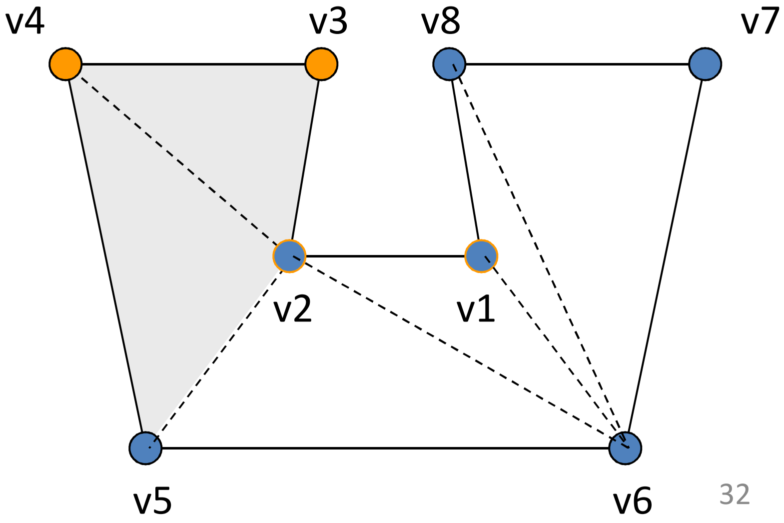
Solution from SKIENA & REVILLA, 2002, Programming Challenges, p.319



Any polygon has at least two ears. An EAR is defined if the angle between the edges emanating the vertex is lower than 180° 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).


First EAR found starting from V1 in the CCW direction

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



Geometric Predicates

Introduction to Predicates

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.

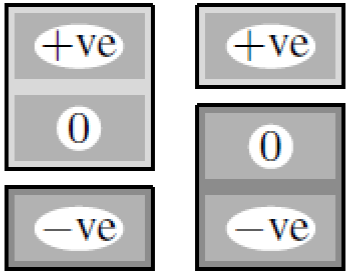
System design has gone back to schematics with the advance of techniques for object-oriented design. One such popular visual language and accompanying methodology, the Unified Modeling Language, promotes that system design should be tackled at a higher granularity. Objects and the messages they pass 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.

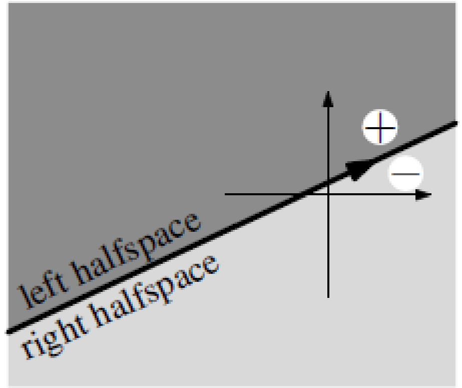
The objective of this section is to argue that crafting good geometric predicates and using them properly is at the center of geometric computing.

Return Type of a Predicate

We generally think of predicates as functions with a Boolean return type. The Boolean type might identify whether a counter has reached some bound, whether a predetermined tolerance has been satisfied, or whether the end of a list has been reached. Such predicates arise in geometric computing, but an additional type of test is frequently needed. Because this geometric test has three possible outcomes, we refer to it as a ternary branching test. Yet most often, we are interested in forming a binary predicate from the three possible outcomes.

The need for three branches in a test can be seen when we consider an oriented line splitting the plane. The plane is split into the points that lie on the positive halfplane, the points that lie on the negative halfplane, as well as those that lie on the line itself. A geometric library will offer such ternary outcomes to clients, and the application programmer will decide how the predicate should be formed.





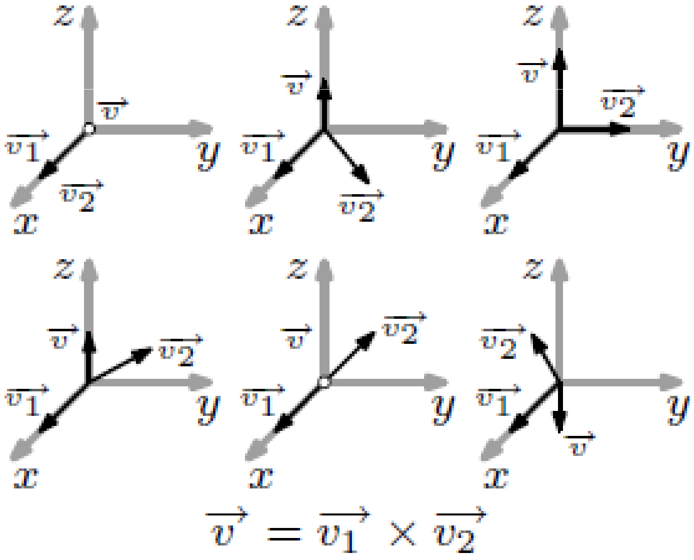
An application might quite suitably need to capture only two cases, the set of points lying on the positive halfplane or the line and the set of points lying in the negative halfplane, for example. But the geometric tests should be offered in such a way that if the application programmer wishes to provide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

The Turn Predicate (Plane Orientation)

Determining the orientation of a point with respect to the line defined by two other points is easily defined by appealing to a function that will take us momentarily to a third dimension.

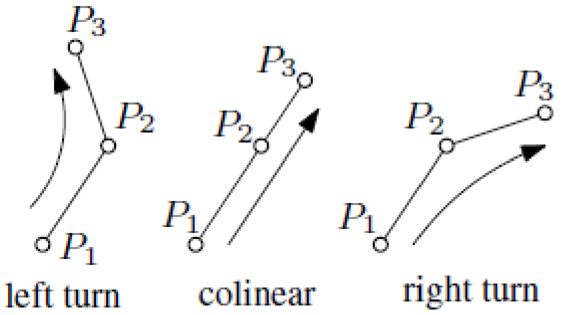
The Cross Product $|\overrightarrow{v}| = |\overrightarrow{v_1}||\overrightarrow{v_2}|\sin\theta$



The Design of an Orientation 2D Predicate

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

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



```
   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;
   }
```

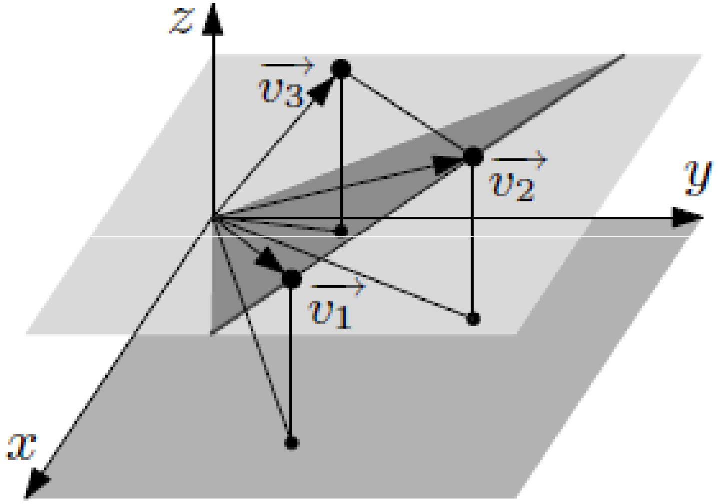
Matrix Form of the Orient2D Predicate

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

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

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

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



Side of Circle Predicate

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 $A \neq 0$.

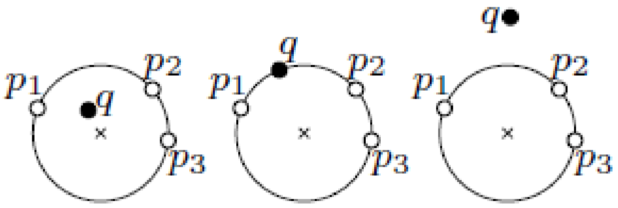
The equation above can be written as the determinant

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

where

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

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



inside circle

on circle boundary

outside circle

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:

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

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

Numerical Precision
Exact and Adaptive Arithmetic

Why using Exact Arithmetic?

Source: Ricardo Marques

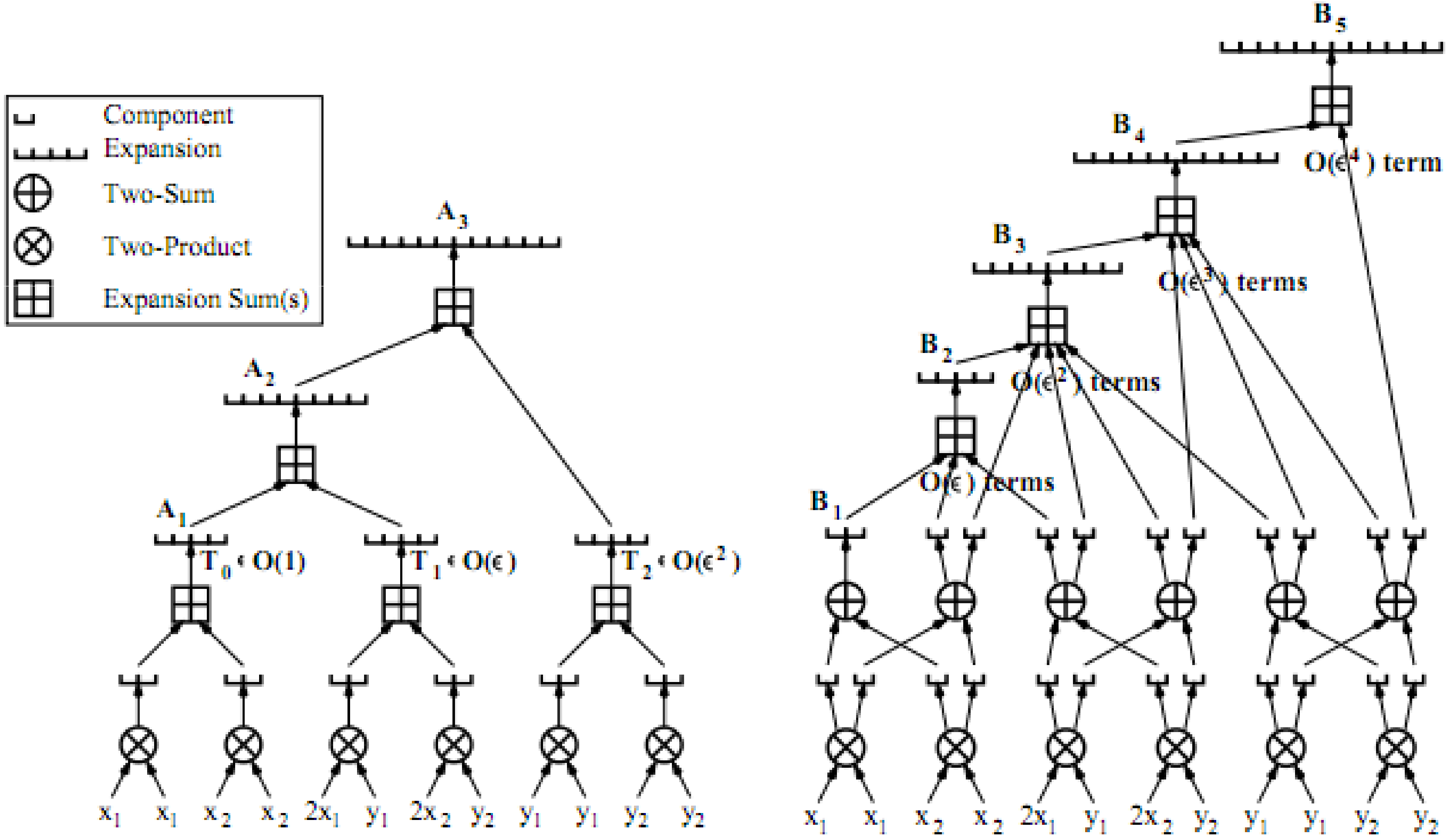
- Using *hard-coded* tolerances does not solve!
  - A tolerance of 1e-07 can be sufficient for models with dimensions relative small.
  - However, if the model has dimensions of hundreds of kilometers, 1e-07 is insignificant. In this case, 1e+00 is a much more acceptable tolerance, representing a relative error of 1e-05, or 1 cm.
  - At the same time, a tolerance of 1e+00 may not make sense on small models.
- Using Exact Aritmhmetic, tolerances are no longer needed.

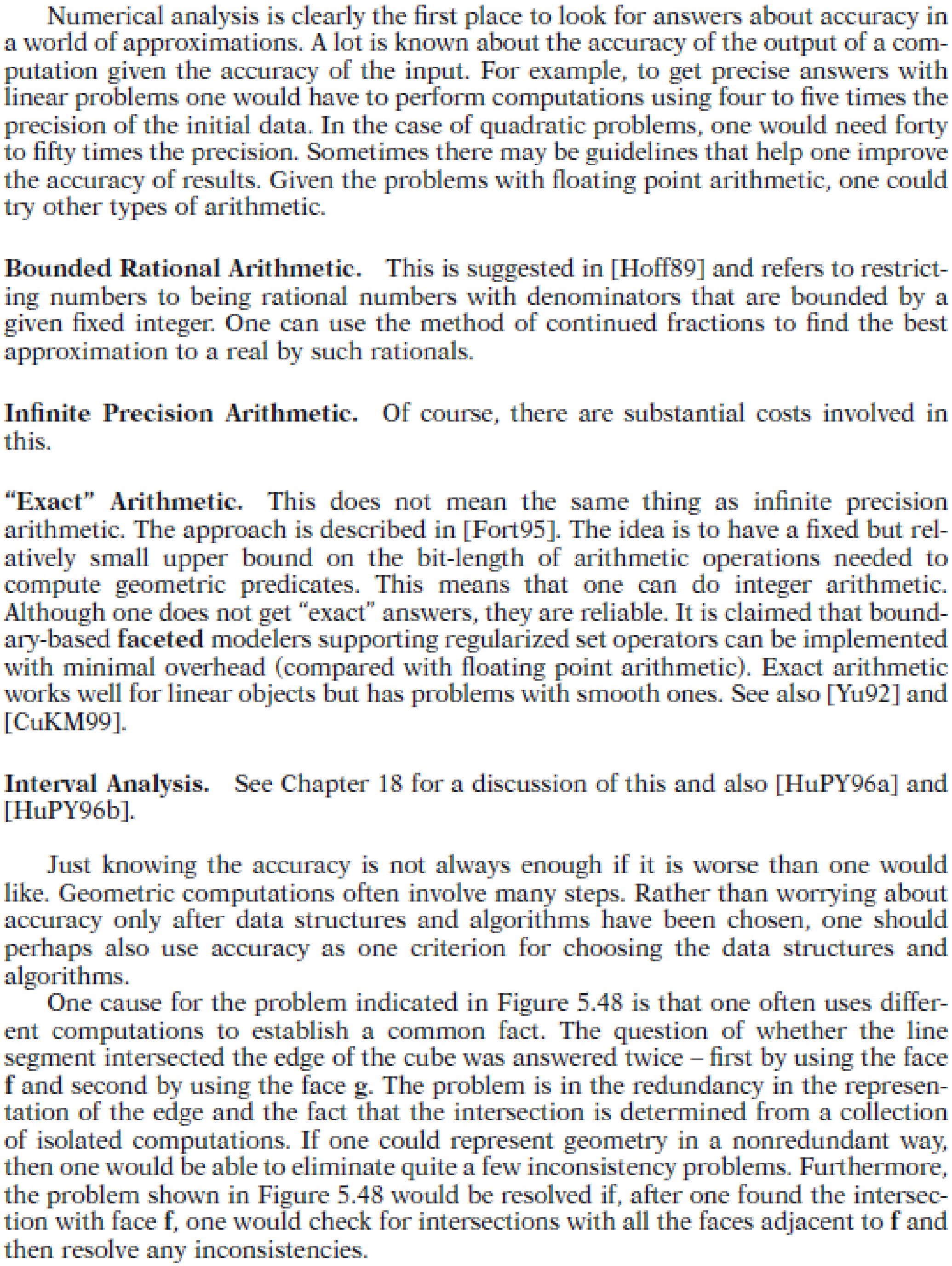
What is Exact Arithmetic?

Source: Ricardo Marques

- Exact arithmetic is a technique for performing calculations with high level of accuracy
  - Is 1e-08 zero? Is -3.1415e-10 zero?
- Avoid rounding error:
  - 1e+08 + 1e-16 = 1e+08 ???
  - Operators in exact arithmetic, every input number (double) is broken into two non-overlying components (numeric) and with different order of magnitudes.
  - By using successive operators, components can be broken again. At the end, all the generated components are joined by minimizing numerical error.

What is Adaptive/Exact Arithmetic?





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 worse than one would like. Geometric computations often involve many steps. Rather than worrying about accuracy only after data structures and algorithms have been chosen, one should perhaps also use accuracy as one criterion for choosing the data structures and algorithms.

One cause for the problem indicated in Figure 5.48 is that one often uses different computations to establish a common fact. The question of whether the line segment intersected the edge of the cube was answered twice – first by using the face **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.

Max K. Agoston
Computer Graphics and Geometric Modeling
Springer 2004

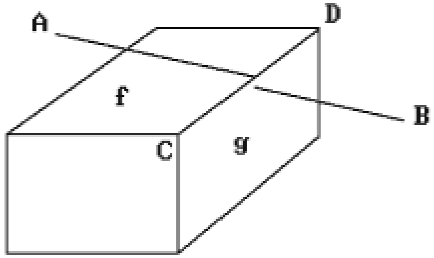


Figure 5.48. Intersection inconsistencies due to round-off errors.

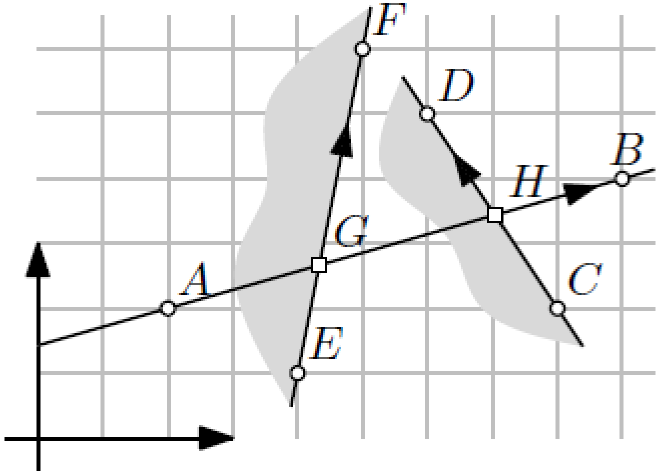
Sherif Ghali
Introduction to Geometric Computing
Springer
2008

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.

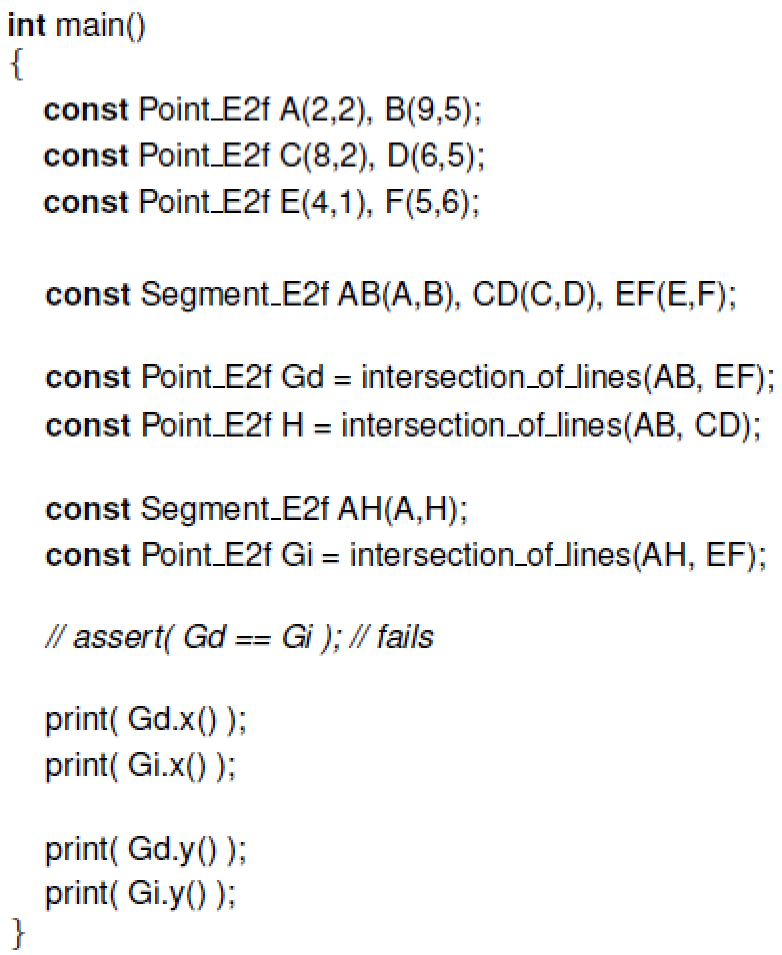
At this time there is no silver bullet to determine whether to sacrifice efficiency and use an exact number type. A simple rule of thumb is to consider the compromise between speed and accuracy. If the system requirements suggest speed, then we have to sacrifice accuracy, and vice versa. The answer is 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.

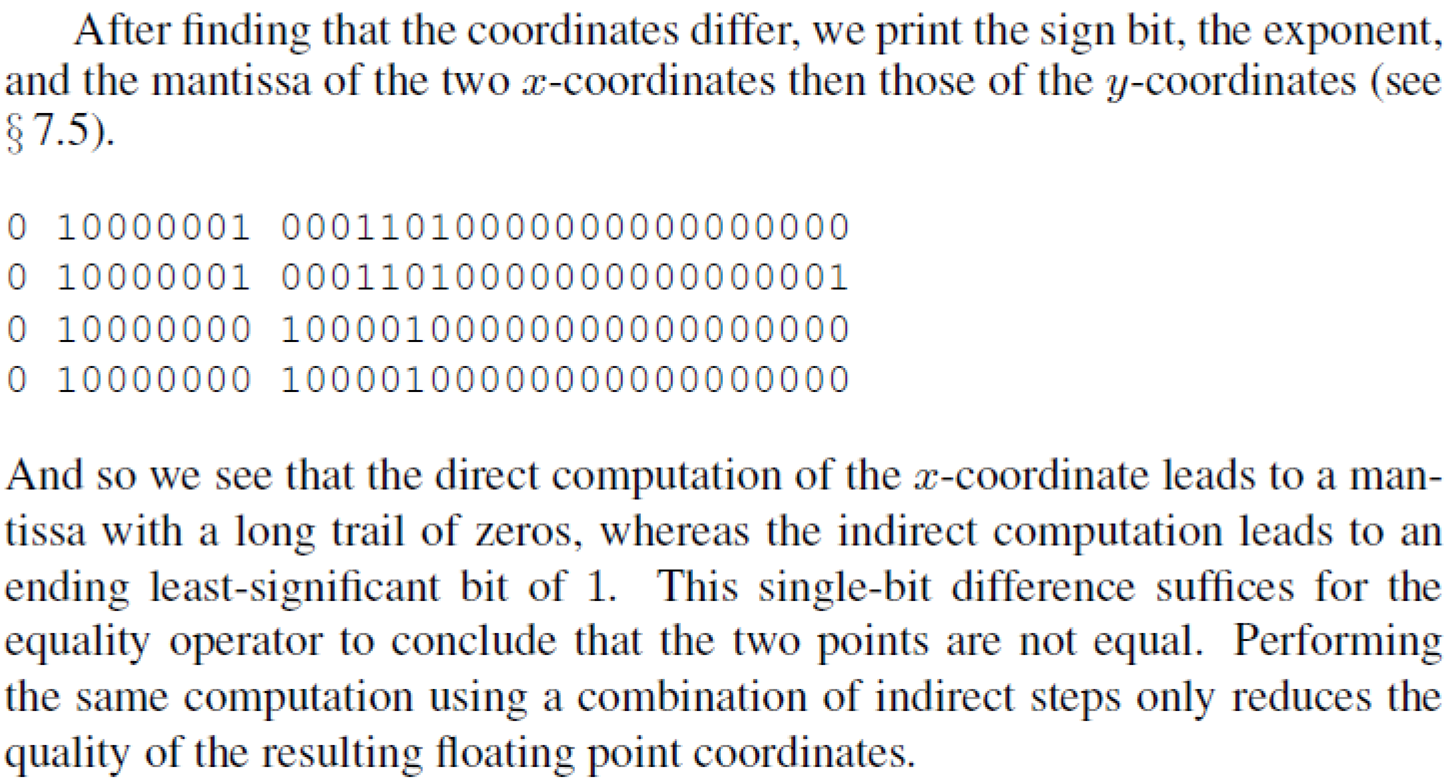


Sherif Ghali
Introduction to Geometric Computing
Springer
2008



```
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.

Closest Point at a Straight Segment

NEAREST POINT IN A STRAIGHT SEGMENT USING INTERNAL PRODUCT

Projection of C point on AB segment:

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

Closest point of C on AB segment:

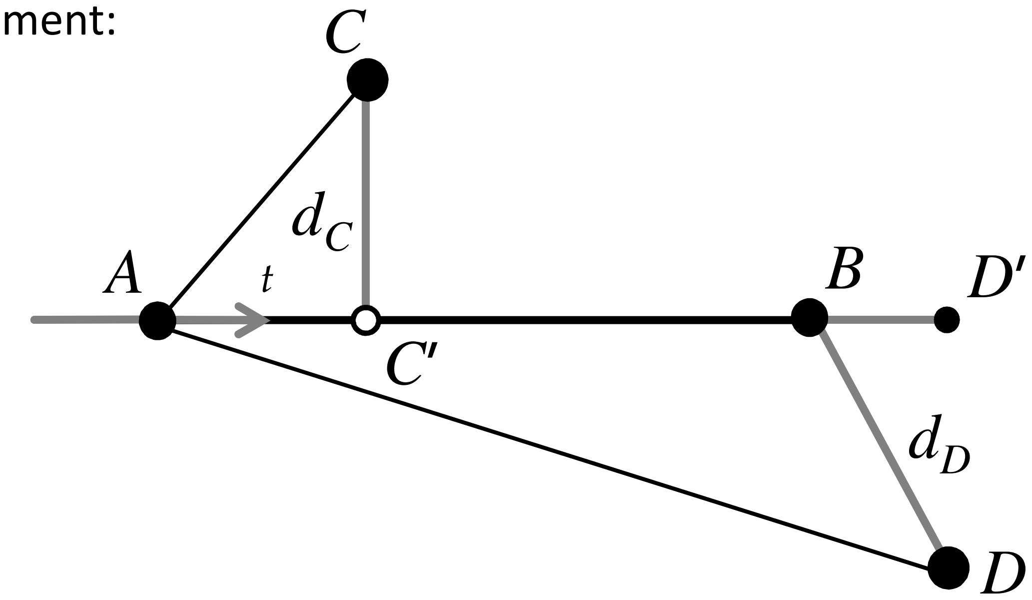
$$\boxed{P = C'}$$

Parametric value of C' point at AB segment:

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

(inner product)

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



Projection of D point on AB segment:

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

Parametric value of D' point at AB segment:

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

$$\boxed{t_{D'} > 1}$$

Closest point of D on AB segment:

$$\boxed{P = B}$$

Algorithms for Segment- Segment Intersection

Line Segments and Intersection

Fonte: [SKIENA02]

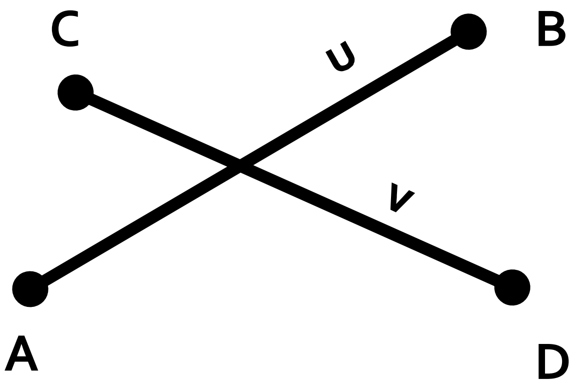
A *line segment s is the portion of a line l which lies between two given points inclusive.* Thus line segments are most naturally represented by pairs of endpoints.

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.

HOW TO TREAT THE INTERSECTION OF STRAIGHT LINES IN ROBUST AND EFFICIENT WAY?

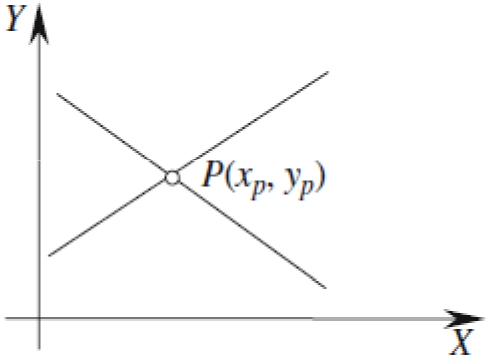


Source: Ricardo Marques

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

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

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

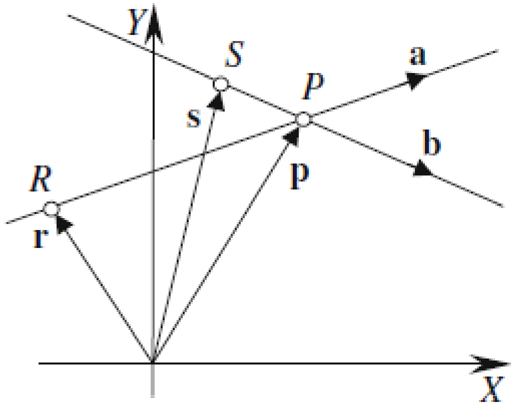
then
$$\lambda = \frac{x_b(y_S - y_R) - y_b(x_S - x_R)}{x_b y_a - x_a y_b}$$

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 $x_P = x_R + \lambda x_a$ $\quad y_P = y_R + \lambda y_a$

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

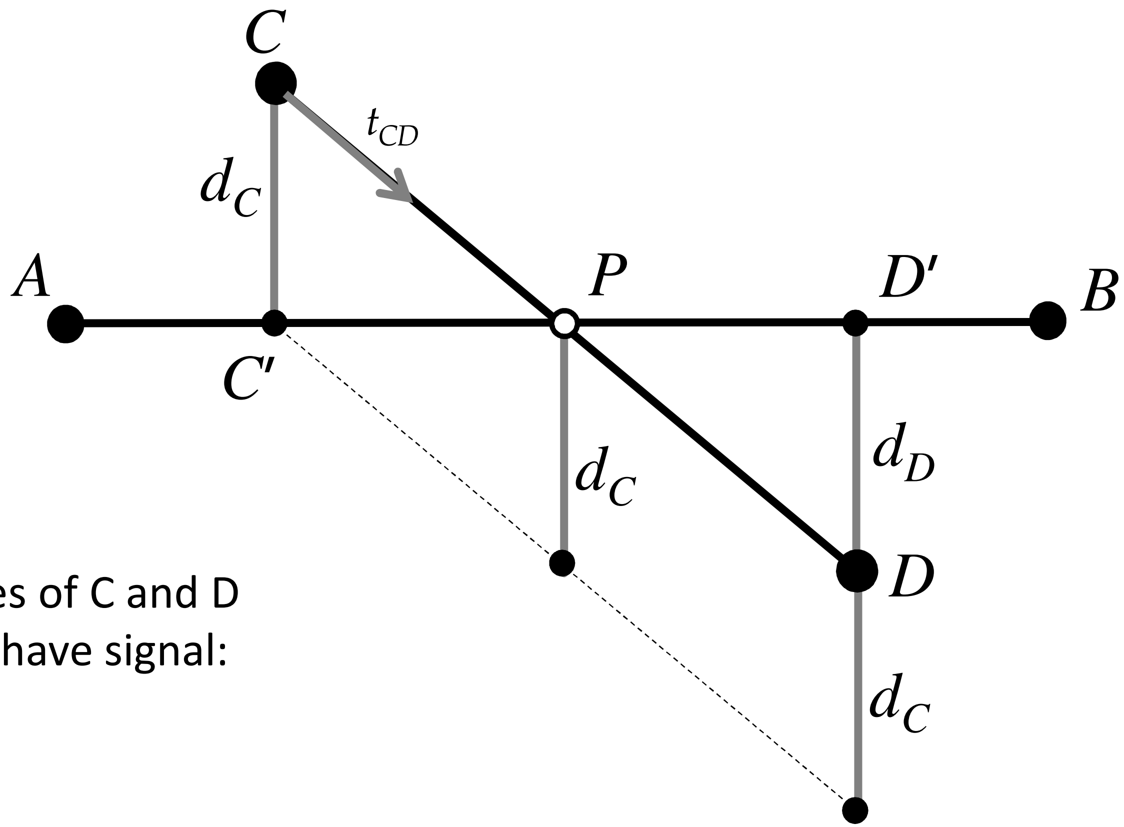
The lines are parallel if $x_b y_a - x_a y_b = 0$



INTERSECTION BASED ON PARAMETRIC REPRESENTATION OF SEGMENTS

$$\boxed{P = A + t_{AB}\left(B - A\right)}$$

$$\boxed{P = C + t_{CD}\left(D - C\right)}$$



Consider that the distances of C and D points to the AB segment have signal:

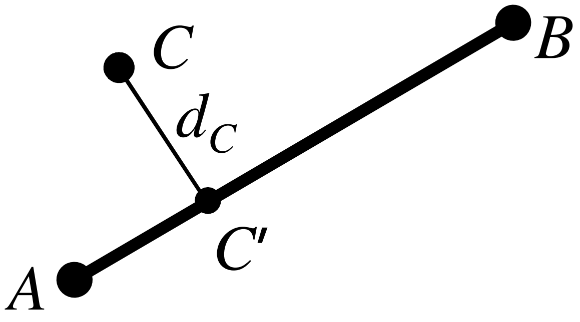
$\boxed{d_C > 0}$ $\boxed{d_D < 0}$

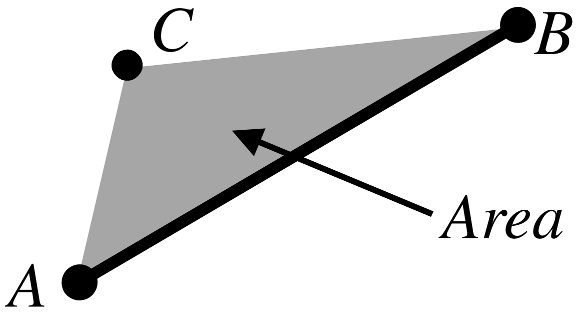
Parametric value for P point at the CD segment:

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

INTERSECTION BASED ON PARAMETRIC REPRESENTATION OF SEGMENTS

Signed distance can be replaced by a cross product





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

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

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

Definition: (double of) trianle oriented area

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

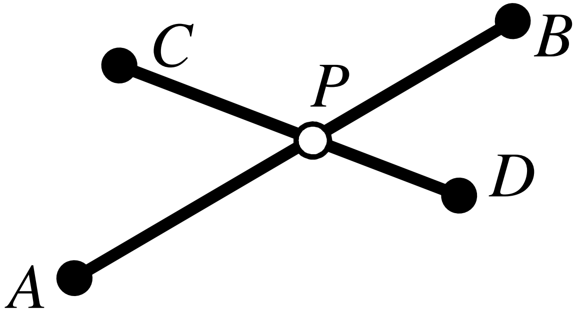
Therefore: $$d_C = \frac{orient2d(A,B,C)}{\left|\overrightarrow{AB}\right|}$$

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

Note that the signs of the distances are resolved naturally.

INTERSECTION BASED ON PARAMETRIC REPRESENTATION OF SEGMENTS

Parametric value of P point at CD segment:



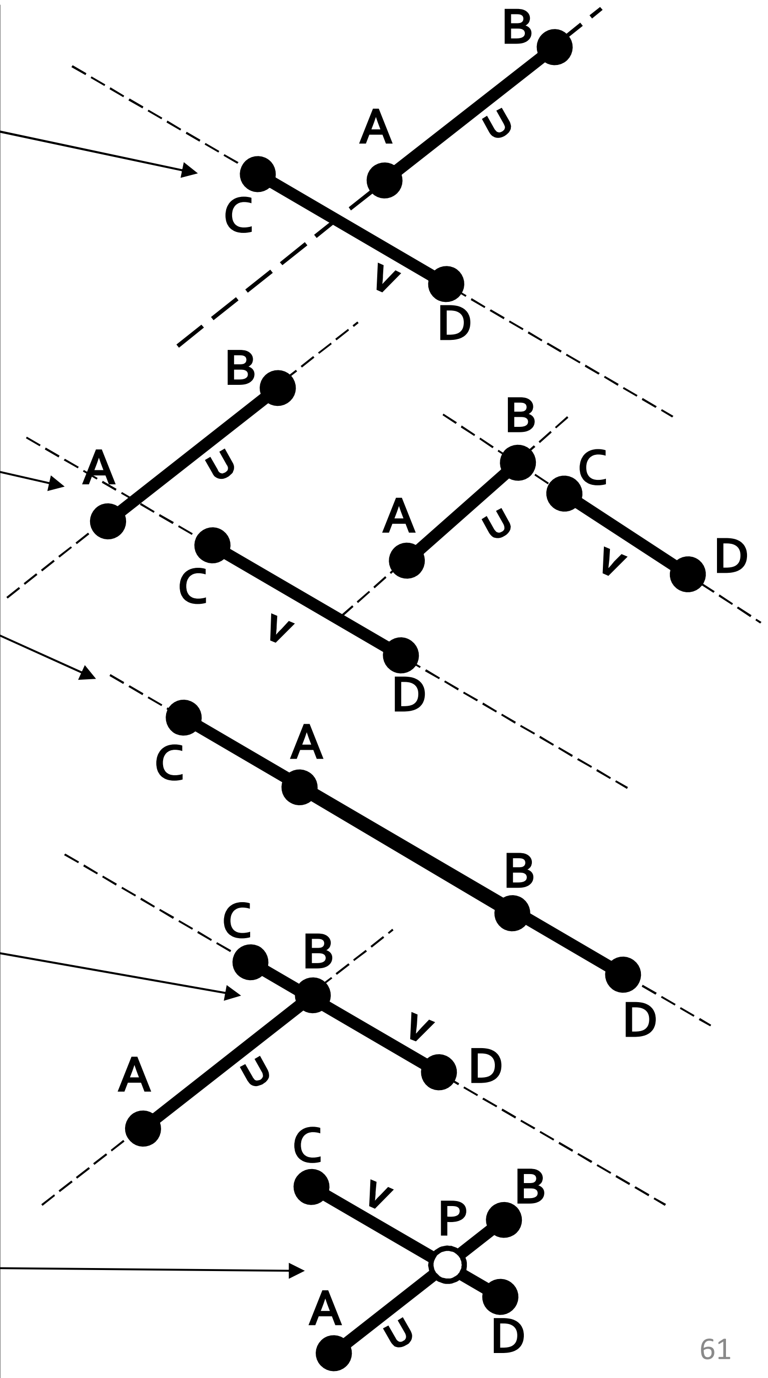
$$\boxed{t_{CD} = \frac{d_C}{d_C - d_D}}$$

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

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

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

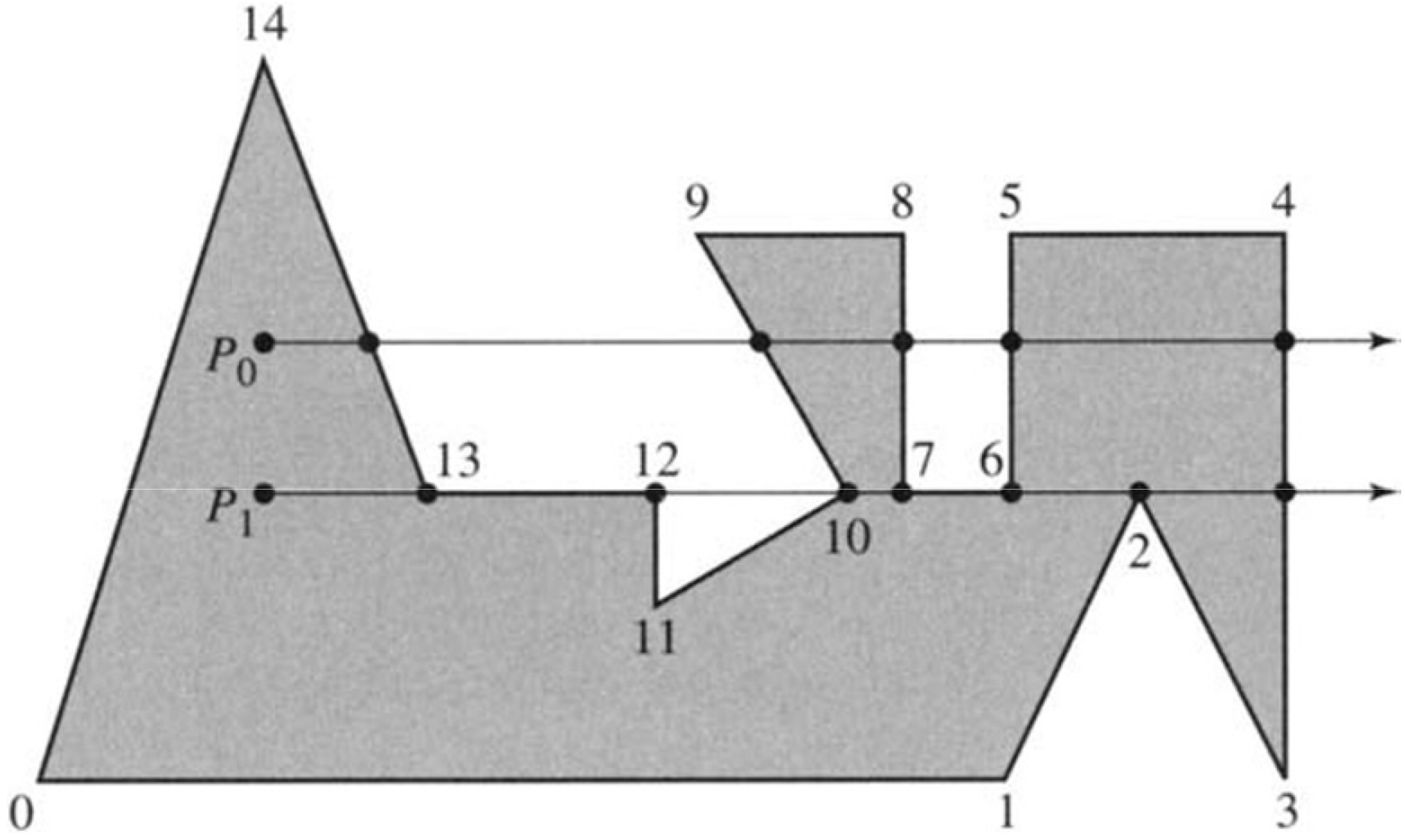
```
Segment_Segment_Intersection(u, v):
if both endpoints of u are over v then
    return false      $orient2d(C,D,A) > 0$   $orient2d(C,D,B) > 0$
end if
if both endpoints of u are under v then
    return false      $orient2d(C,D,A) < 0$   $orient2d(C,D,B) < 0$
end if
if both endpoints of v are over u then
    return false      $orient2d(A,B,C) > 0$   $orient2d(A,B,D) > 0$
end if
if both endpoints of v are under u then
    return false      $orient2d(A,B,C) < 0$   $orient2d(A,B,D) < 0$
end if
if u and v are collinear then
    return false      $orient2d(C,D,A) = 0$   $orient2d(C,D,B) = 0$
end if
              ou      $orient2d(A,B,C) = 0$   $orient2d(A,B,C) = 0$

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

if u touches v  then                        (there are many cases)
    return true       $orient2d(C,D,B) = 0$   $orient2d(C,D,A) < 0$
end if
                                                        $P = B$

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

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

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



Algorithm for
Point in Polygon Verification

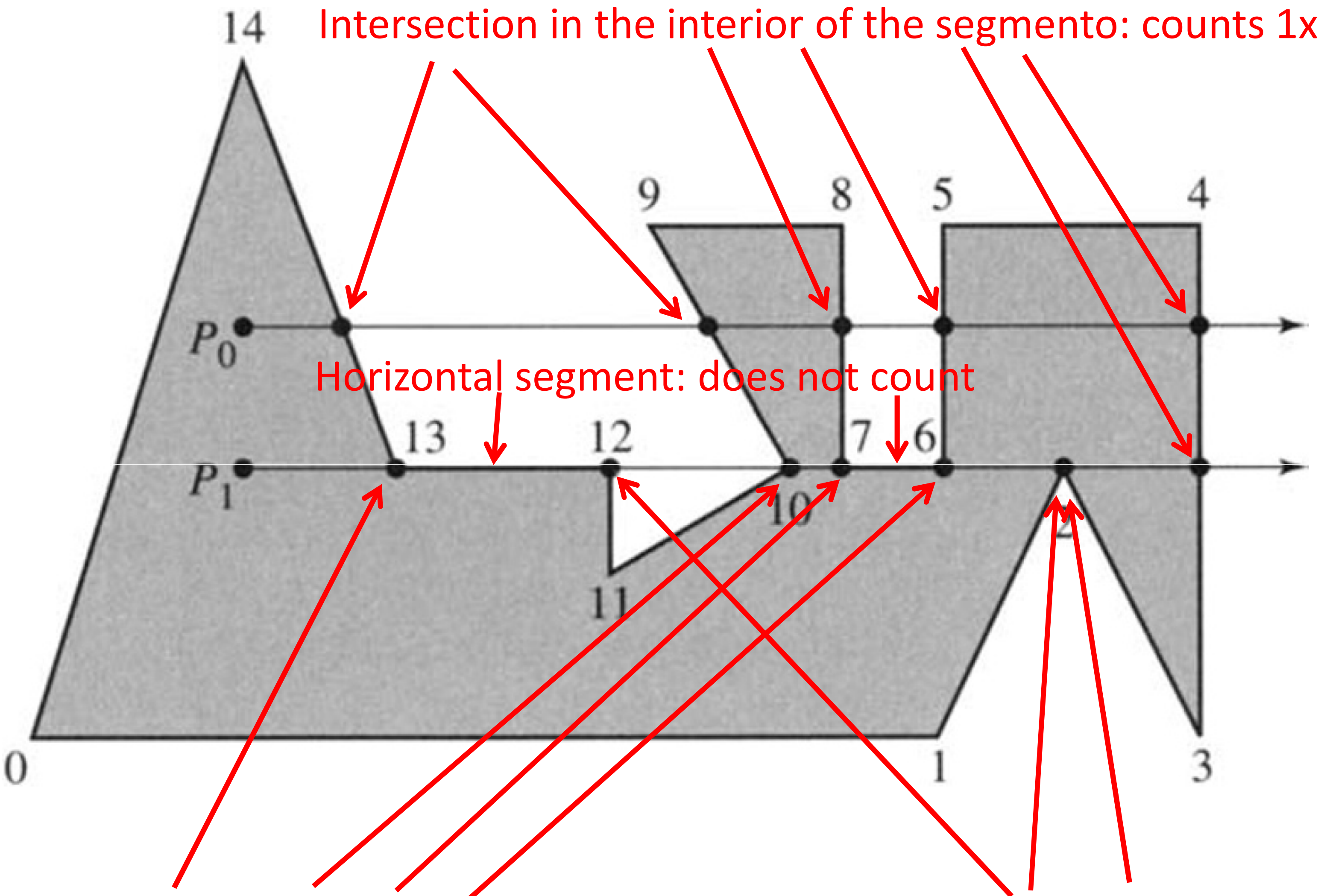
Ray Algorithm (or shot)

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



A ray that part of any point within the polygon in one direction will cut any curves on the edge of the polygon an odd number of times. If the ray cut the boundary of the polygon an even number of times, the point is outside the polygon.

Criteria for counting intersections of the ray with a boundary edge



Intersection in the interior of the segmento: counts 1x

Horizontal segment: does not count

Intersection in the inferior point
of the segment: counts 1x

Intersection in the superior point
of the segment: does not count