Numerical Robustness
(for Geometric Calculations)

Christer Ericson

Sony Computer Entertainment

Slides @ http://realtimecollisiondetection.net/pubs/

EPSILON is NOT 0.00001!

Christer Ericson

Sony Computer Entertainment

Slides @ http://realtimecollisiondetection.net/pubs/

Takeaway

- An appreciation of the pitfalls inherent in working with floating-point arithmetic.
- Tools for addressing the robustness of floating-point based code.
- Probably something else too.

THE PROBLEM

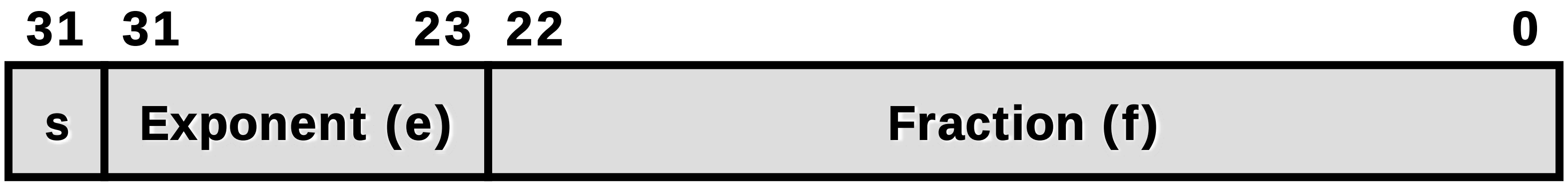
Floating-point arithmetic

Floating-point numbers

- Real numbers must be approximated
  - Floating-point numbers
  - Fixed-point numbers (integers)
  - Rational numbers
    - Homogeneous representation
- If we could work in real arithmetic, I wouldn’t be having this talk!

Floating-point numbers

- IEEE-754 single precision
  - 1 bit sign
  - 8 bit exponent (biased)
  - 23 bits fraction (24 bits mantissa w/ hidden bit)



$$V = (-1)^s \times (1.f) \times 2^{e-127}$$

- This is a **normalized** format

Floating-point numbers

- IEEE-754 representable numbers:

Exponent	Fraction	Sign	Value
0<e<255			$V = (-1)^s \times (1.f) \times 2^{e-127}$
e=0	f=0	s=0	$V = 0$
e=0	f=0	s=1	$V = -0$
e=0	f≠0		$V = (-1)^s \times (0.f) \times 2^{e-126}$
e=255	f=0	s=0	$V = +Inf$
e=255	f=0	s=1	$V = -Inf$
e=255	f≠0		$V = NaN$

Floating-point numbers

- In IEEE-754, domain extended with:
  - –Inf, +Inf, NaN
- Some examples:
  - a/0 = +Inf, if a > 0
  - a/0 = –Inf, if a < 0
  - 0/0 = Inf – Inf = ±Inf · 0 = NaN
- Known as **Infinity Arithmetic (IA)**

Floating-point numbers

- IA is a potential source of robustness errors!
  - +Inf and –Inf compare as normal
  - But NaN compares as unordered
    - NaN != NaN is true
    - All other comparisons involving NaNs are false
- These expressions are not equivalent:

```
if (a > b) X(); else Y();
```

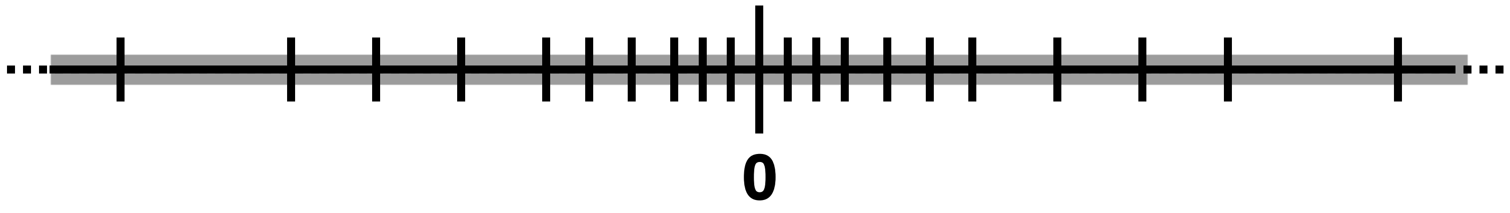
```
if (a <= b) Y(); else X();
```

Floating-point numbers

- But IA provides a nice feature too
- Allows not having to test for div-by-zero
  - Removes test branch from inner loop
  - Useful for SIMD code
- (Although same approach usually works for non-IEEE CPUs too.)

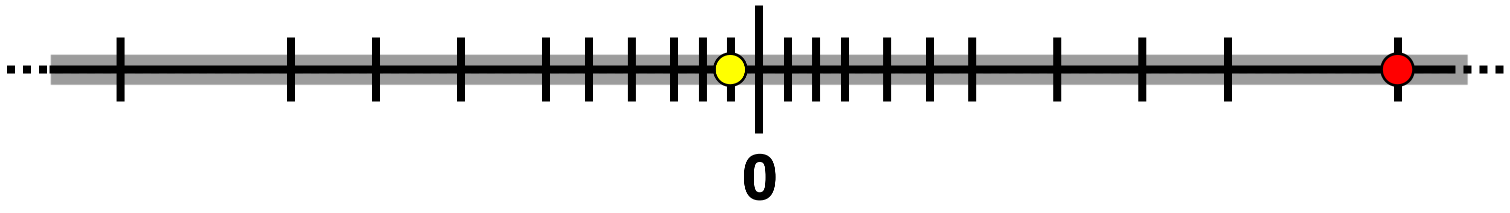
Floating-point numbers

- Irregular number line
  - Spacing increases the farther away from zero a number is located
  - Number range for exponent **k+1** has twice the spacing of the one for exponent **k**
  - Equally many representable numbers from one exponent to another



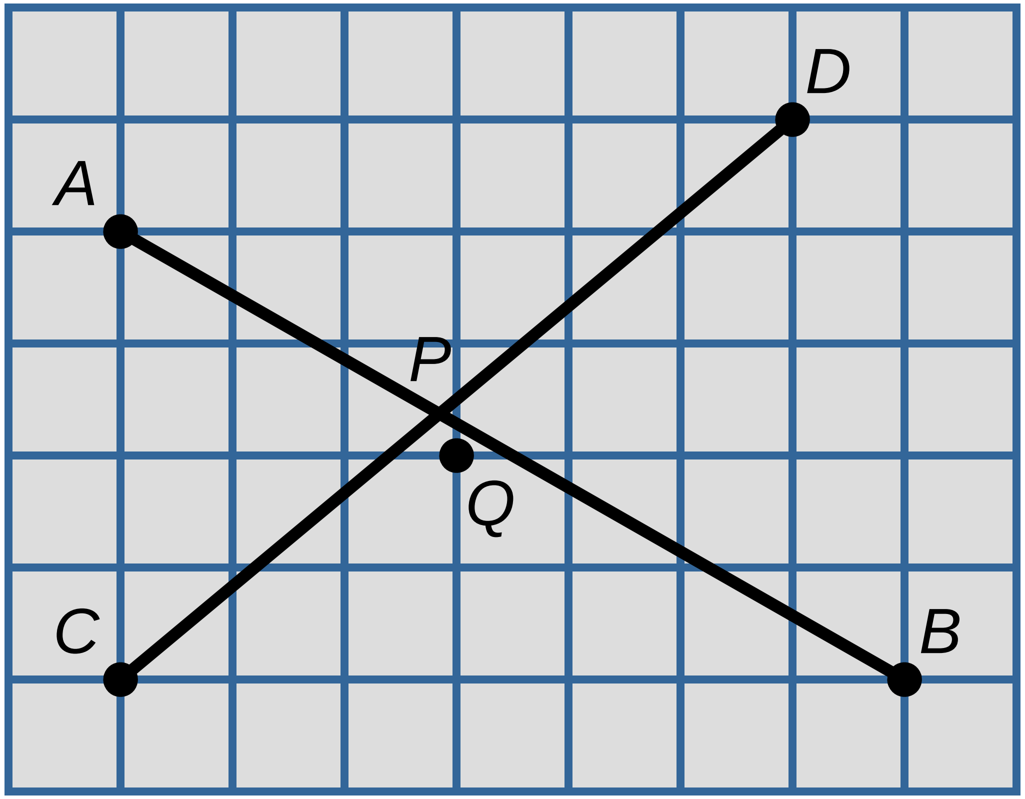
Floating-point numbers

- Consequence of irregular spacing:
  - $-10^{20} + (10^{20} + 1) = 0$
  - $(-10^{20} + 10^{20}) + 1 = 1$
- Thus, not associative (in general):
  - (a + b) + c != a + (b + c)
- Source of endless errors!



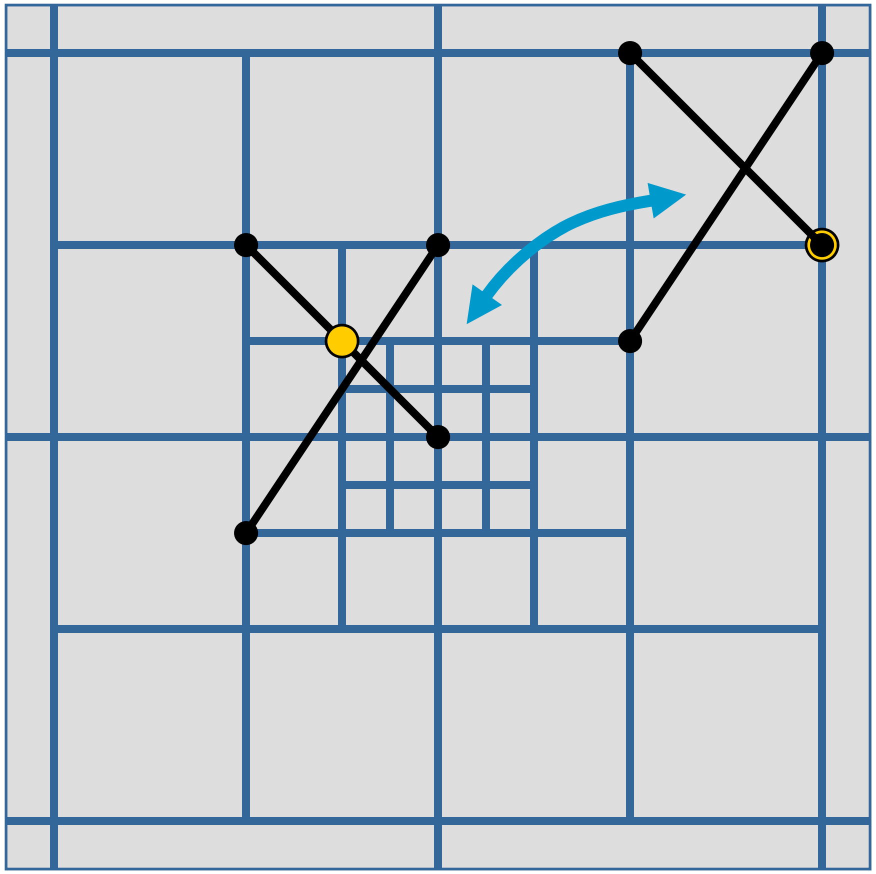
Floating-point numbers

- All discrete representations have non-representable points



The floating-point grid

- In floating-point, behavior changes based on position, due to the irregular spacing!

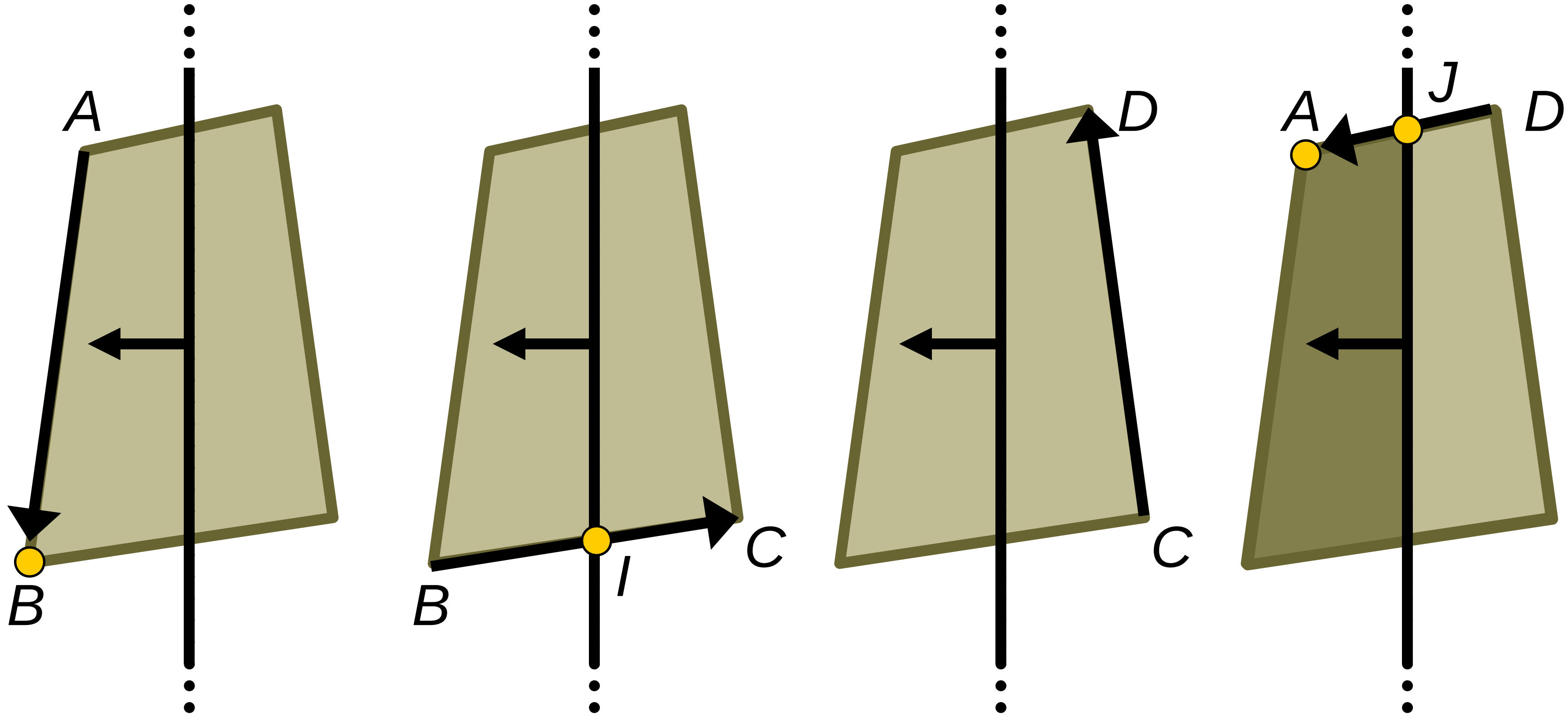


EXAMPLE

Polygon splitting

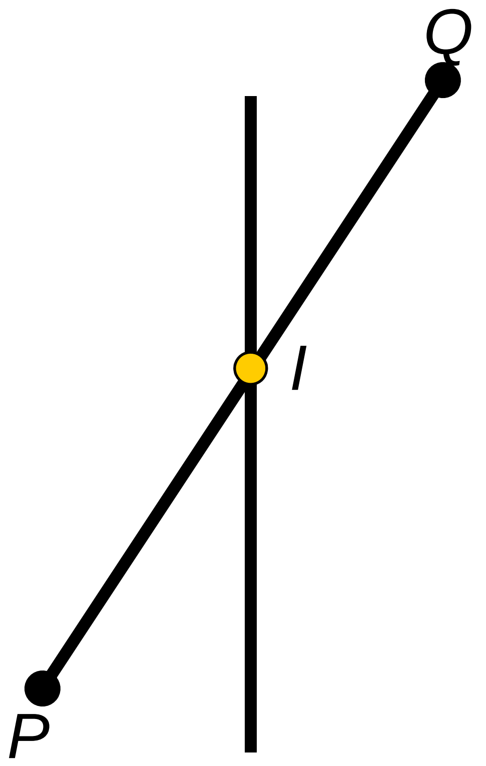
Polygon splitting

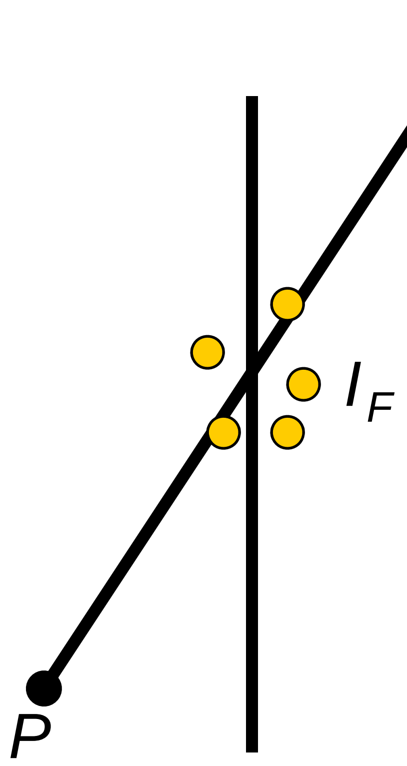
- Sutherland-Hodgman clipping algorithm



Polygon splitting

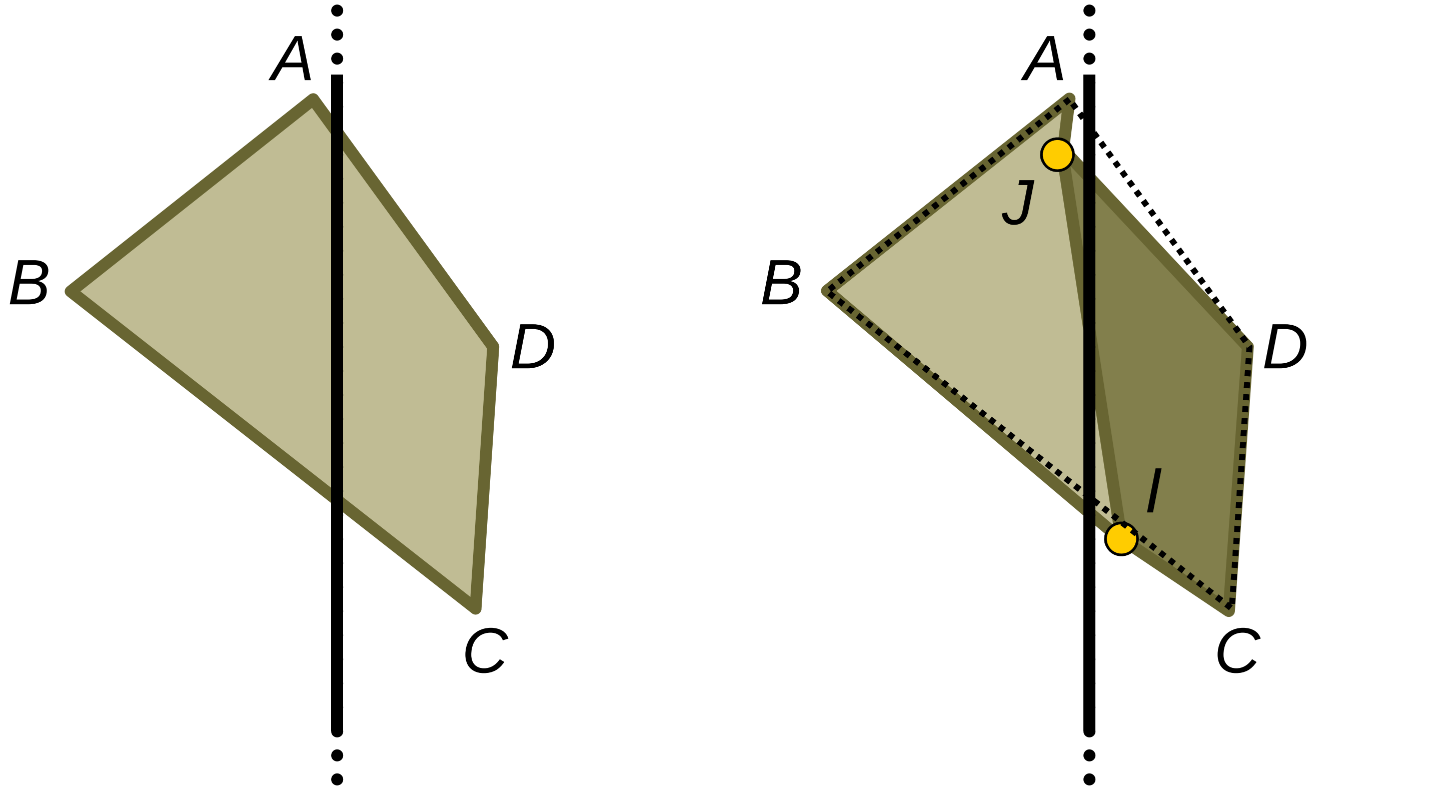
- Enter floating-point errors!





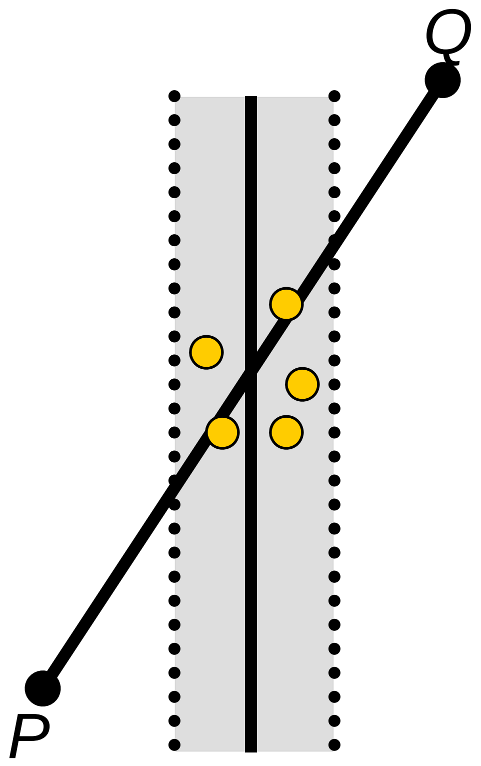
Polygon splitting

- ***ABCD*** split against a plane



Polygon splitting

- **Thick** planes to the rescue!

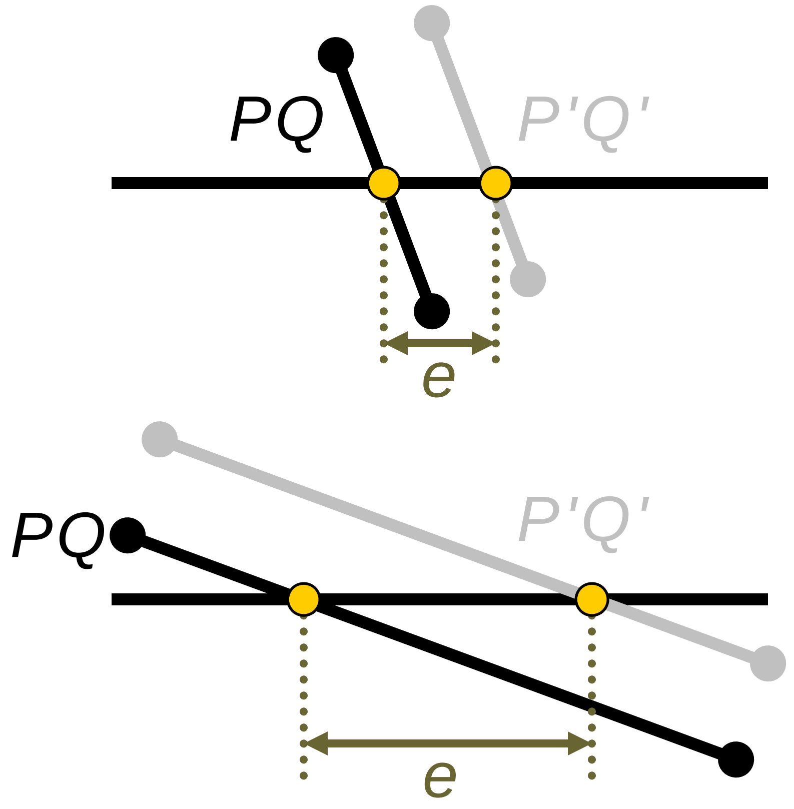


Desired invariant:
OnPlane(I, plane) = true

where:
I = IntersectionPoint(PQ, plane)

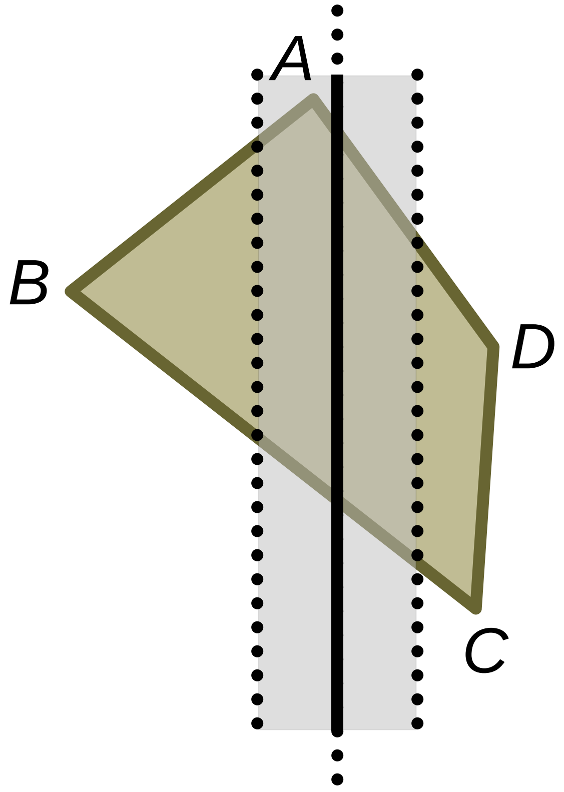
Polygon splitting

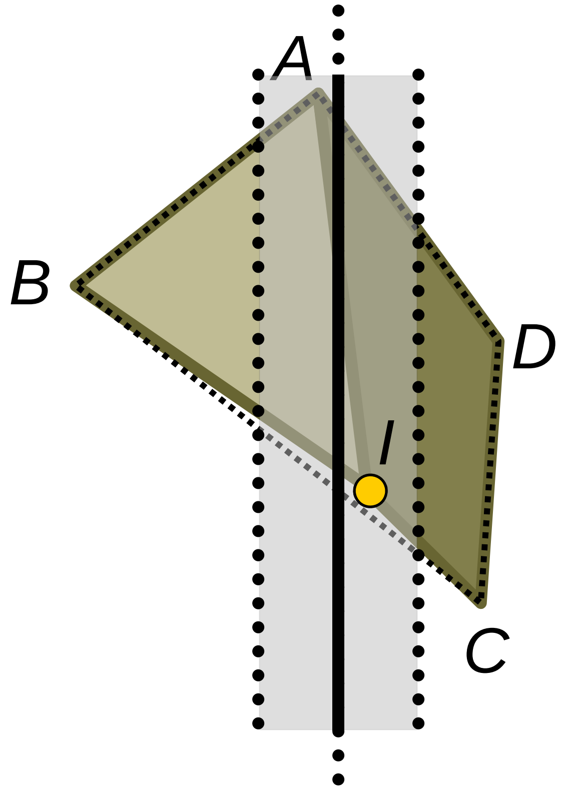
- Thick planes also help bound the error



Polygon splitting

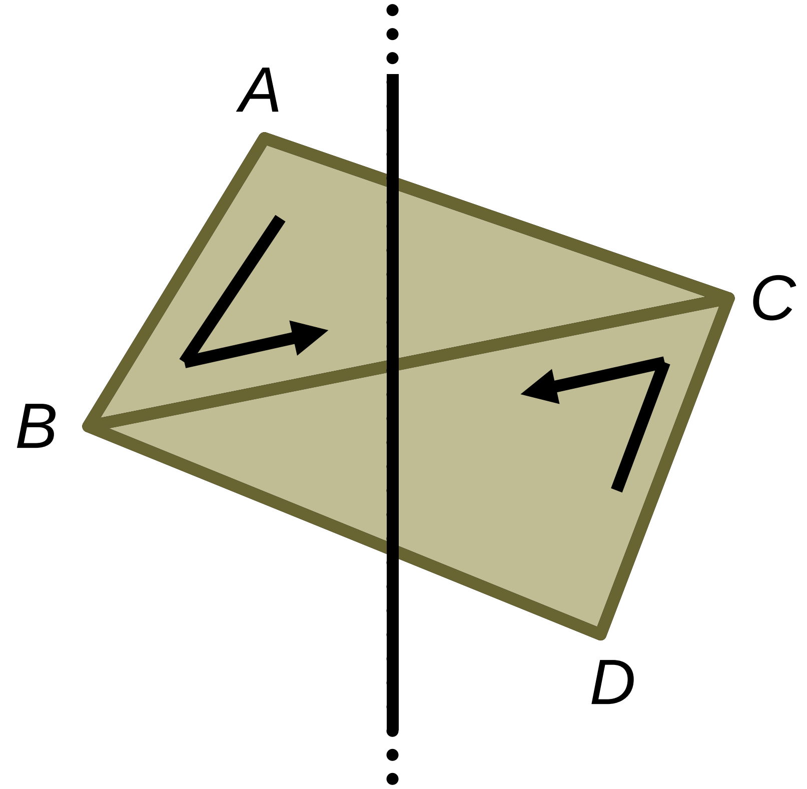
- ***ABCD*** split against a ***thick*** plane

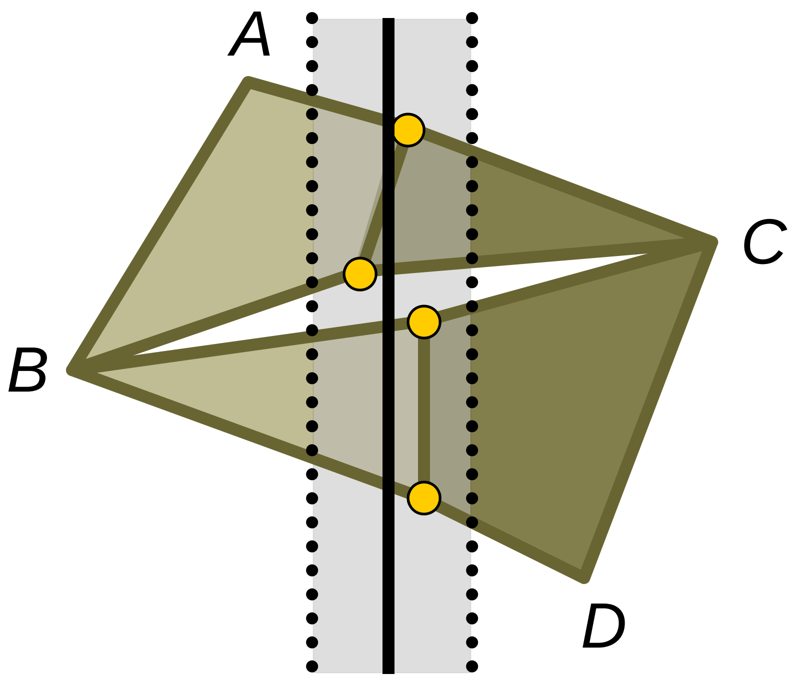




Polygon splitting

- **Cracks** introduced by inconsistent ordering





EXAMPLE

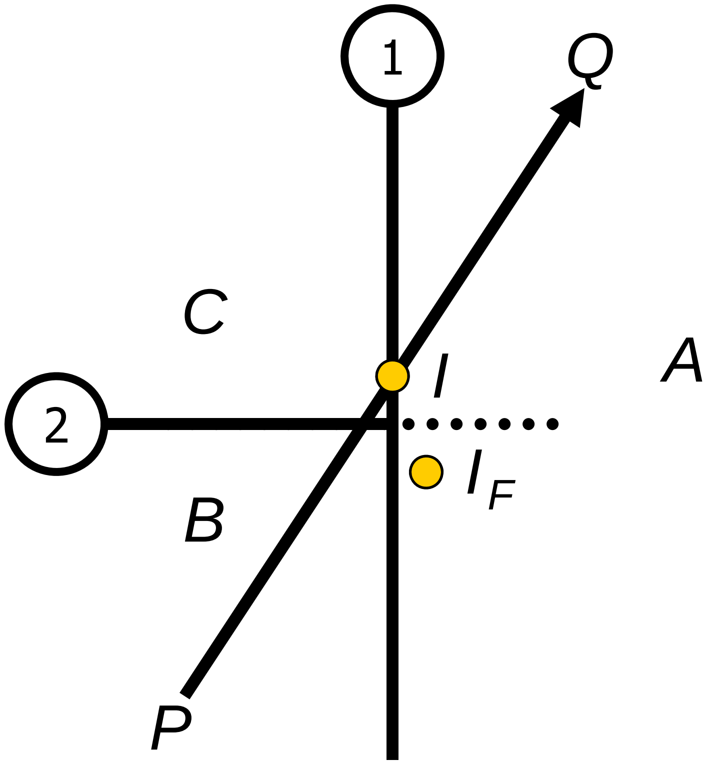
BSP-tree robustness

BSP-tree robustness

- Robustness problems for:
  - Insertion of primitives
  - Querying (collision detection)
- Same problems apply to:
  - All spatial partitioning schemes!
  - (k-d trees, grids, octrees, quadtrees, …)

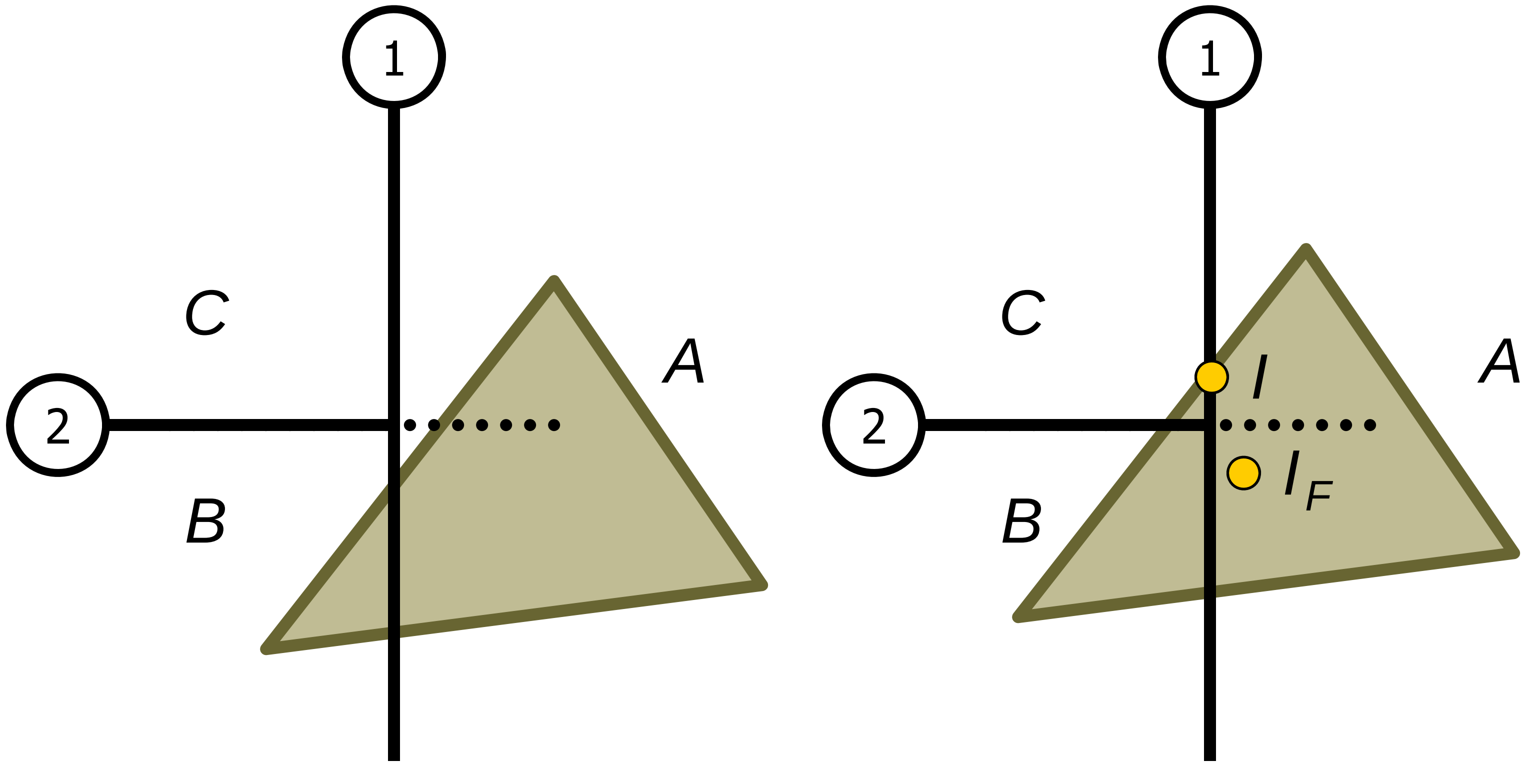
BSP-tree robustness

- Query robustness



BSP-tree robustness

- Insertion robustness



BSP-tree robustness

- How to achieve robustness?
  - Insert primitives conservatively
    - Accounting for errors in querying and insertion
  - Can then ignore problem for queries

EXAMPLE

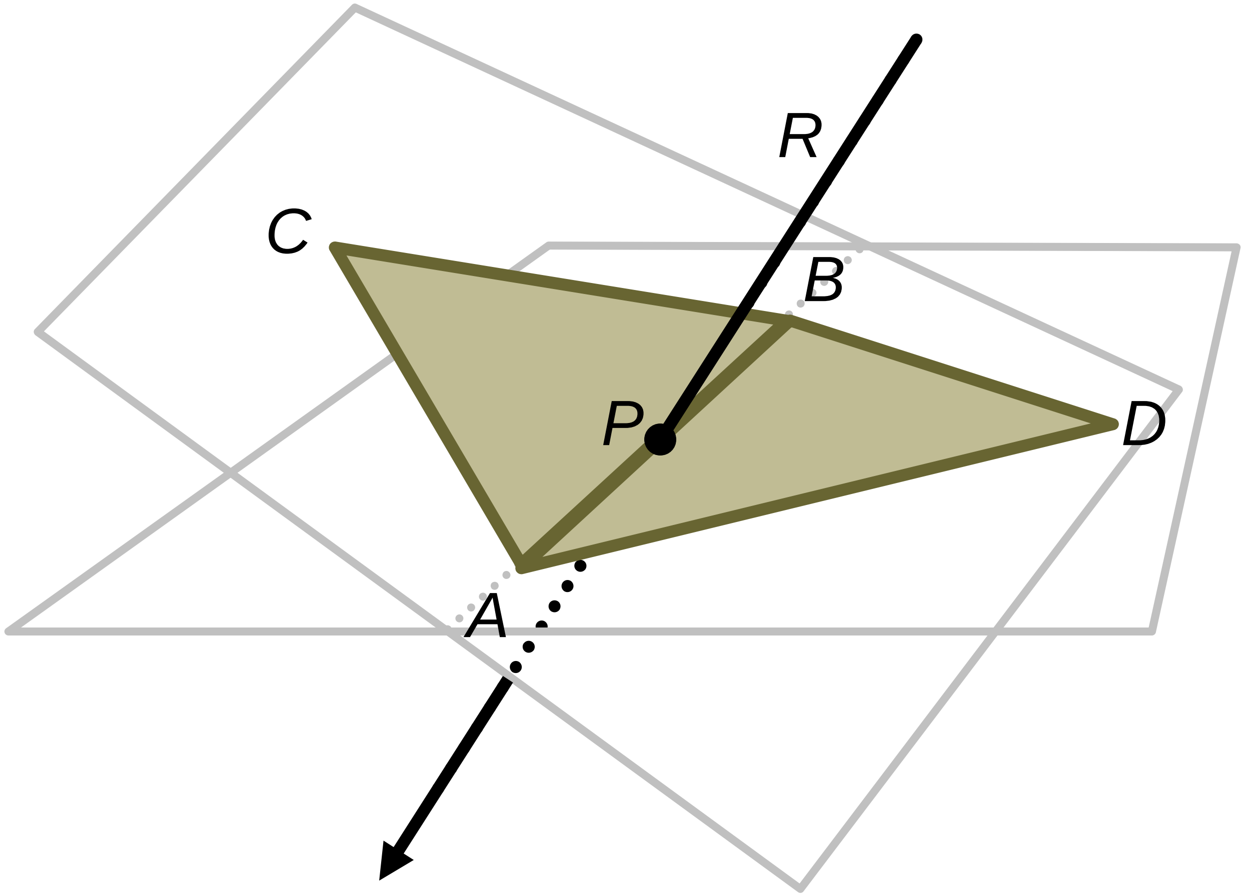
Ray-triangle test

Ray-triangle test

- Common approach:
  - Compute intersection point *P* of ray *R* with plane of triangle *T*
  - Test if *P* lies inside boundaries of *T*
- Alas, this is not robust!

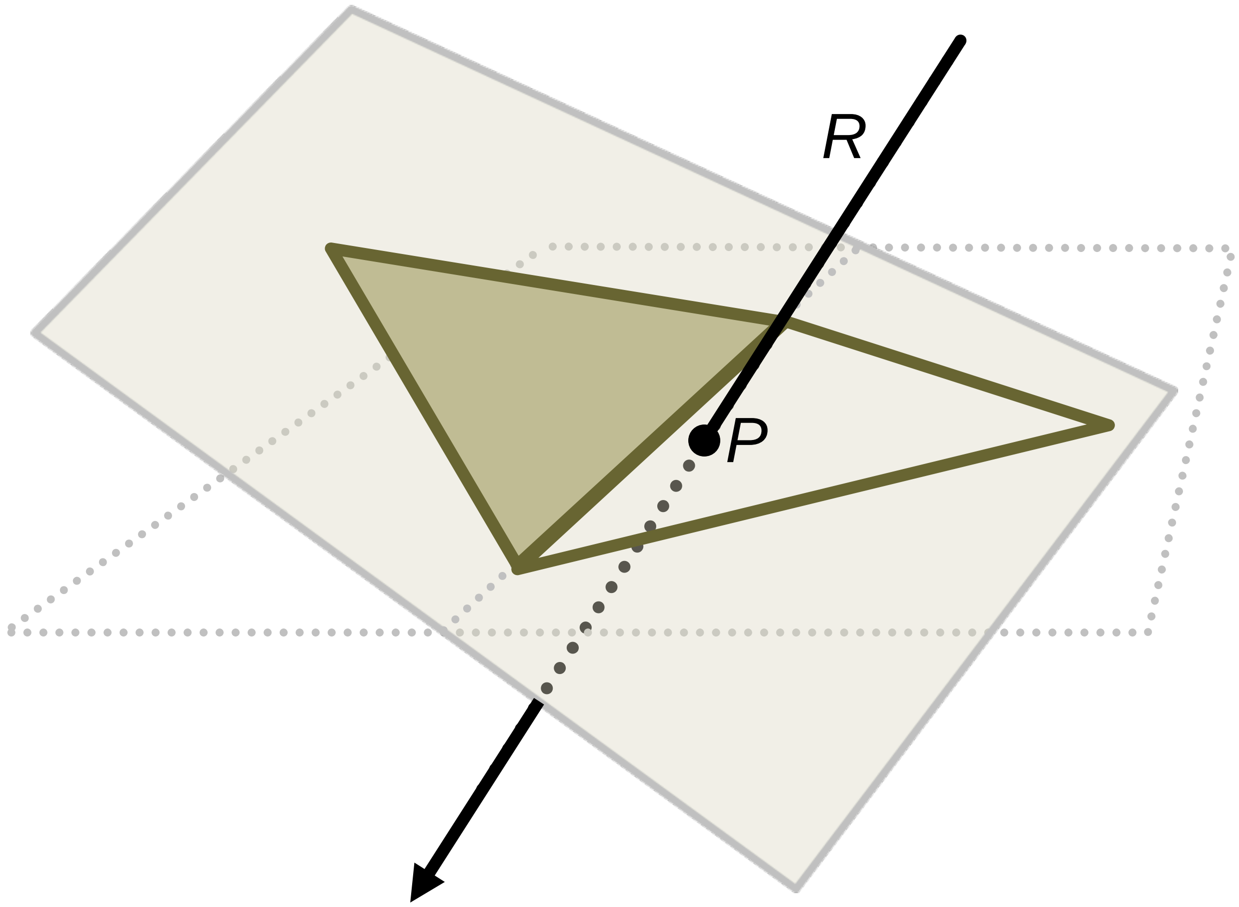
Ray-triangle test

- A problem configuration



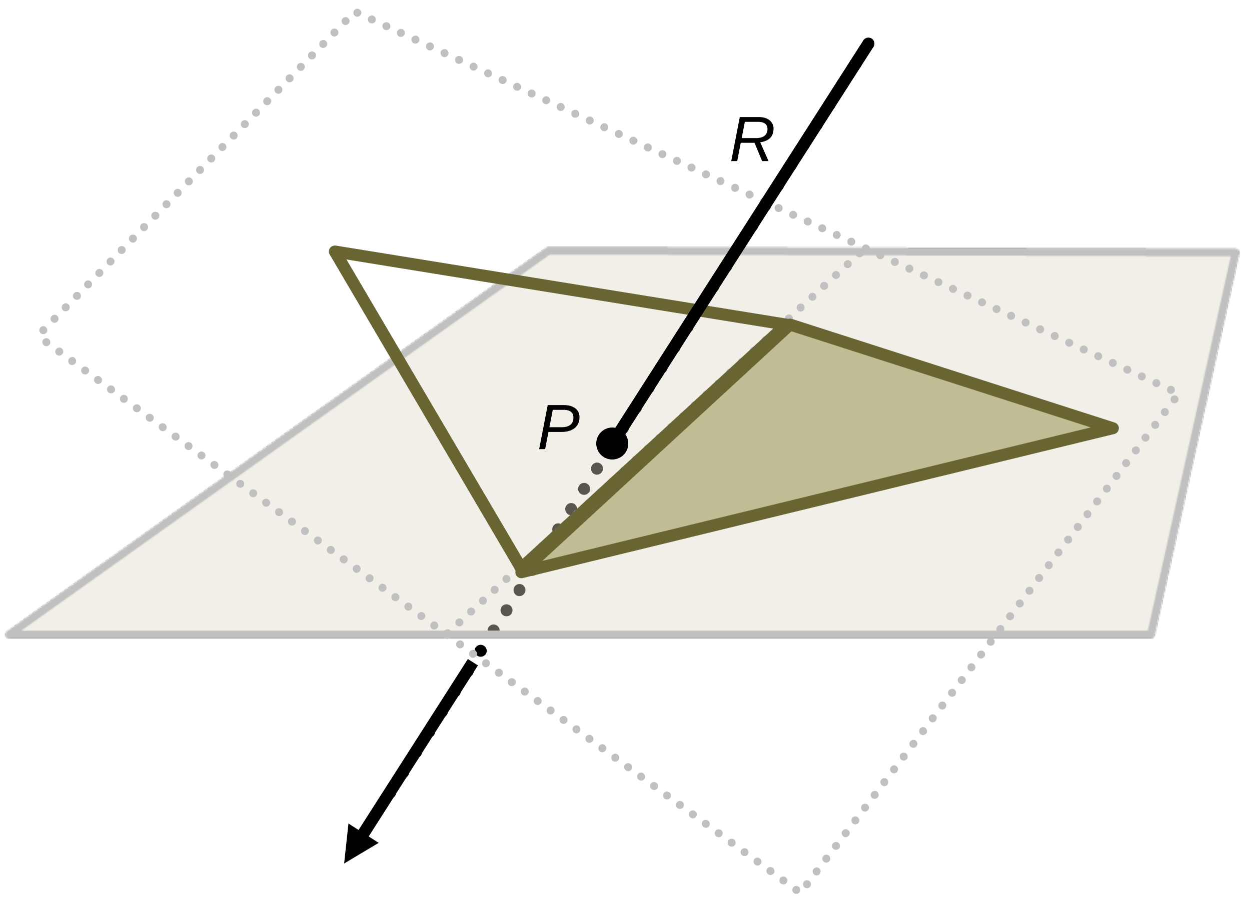
Ray-triangle test

- Intersecting *R* against one plane



Ray-triangle test

- Intersecting *R* against the other plane

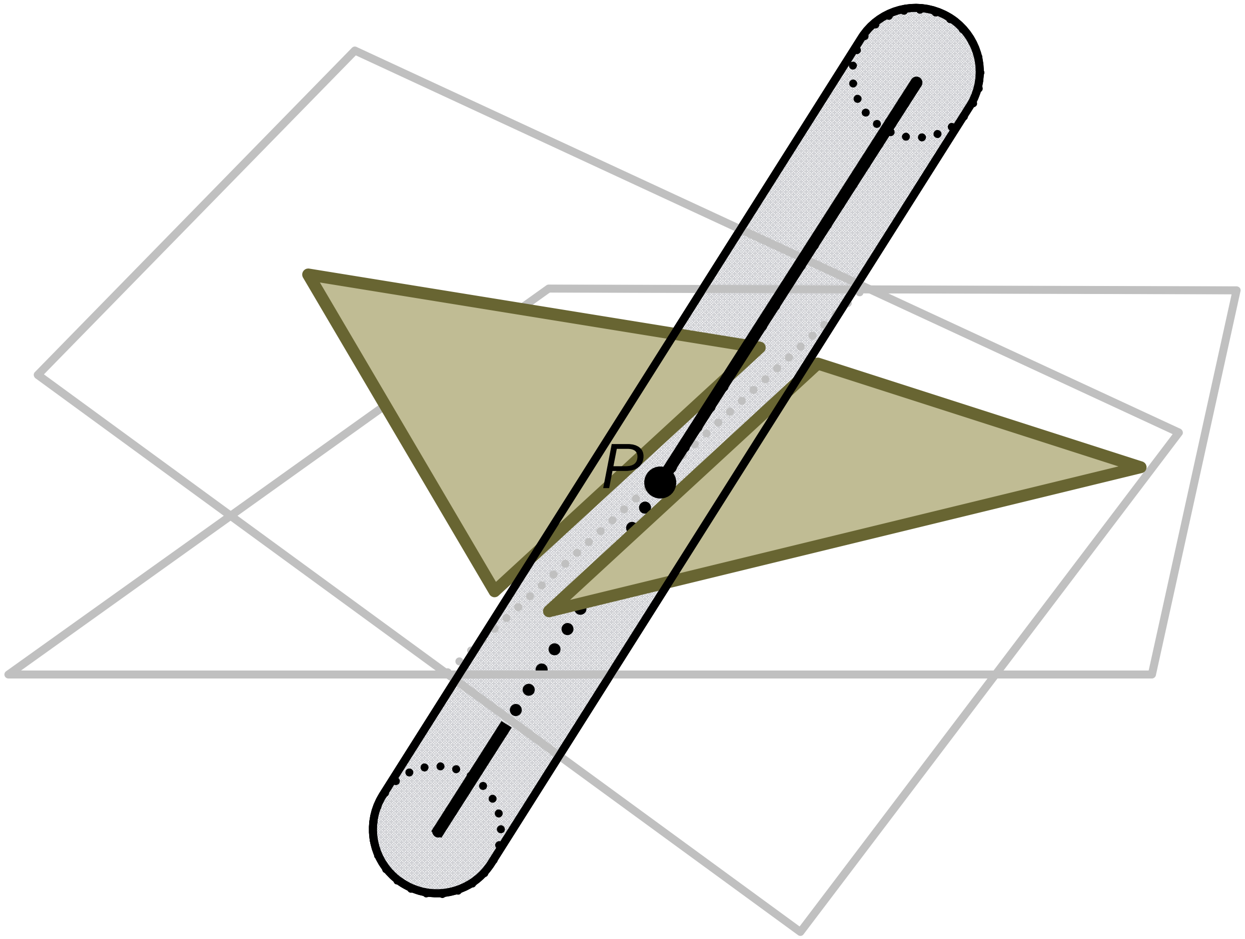


Ray-triangle test

- Robust test must **share calculations** for shared edge *AB*
- Perform test directly in 3D!
  - Let ray be $R(t)=O+t\mathbf{d}$
  - Then, sign of $\mathbf{d}\cdot(OA\times OB)$ says whether $\mathbf{d}$ is left or right of *AB*
  - If *R* left of all edges, *R* intersects CCW triangle
  - **Only then** compute *P*
- Still errors, but managable

Ray-triangle test

- "Fat" tests are also robust!



EXAMPLES SUMMARY

- Achieve robustness through…
  - (Correct) use of tolerances
  - Sharing of calculations
  - Use of fat primitives

TOLERANCES

Tolerance comparisons

- Absolute tolerance
- Relative tolerance
- Combined tolerance
- (Integer test)

Absolute tolerance

Comparing two floats for equality:

```
if (Abs(x – y) <= EPSILON) …
```

- Almost never used correctly!
- What should EPSILON be?
  - Typically arbitrary small number used! OMFG!!

Absolute tolerances

Delta step to next representable number:

Decimal	Hex	Next representable number
10.0	0x41200000	x + 0.000001
100.0	0x42C80000	x + 0.000008
1,000.0	0x447A0000	x + 0.000061
10,000.0	0x461C4000	x + 0.000977
100,000.0	0x47C35000	x + 0.007813
1,000,000.0	0x49742400	x + 0.0625
10,000,000.0	0x4B189680	x + 1.0

Absolute tolerances

Möller-Trumbore ray-triangle code:

```
#define EPSILON 0.000001
#define DOT(v1,v2) (v1[0]*v2[0]+v1[1]*v2[1]+v1[2]*v2[2])
...
// if determinant is near zero, ray lies in plane of triangle
det = DOT(edge1, pvec);
...
if (det > -EPSILON && det < EPSILON) // Abs(det) < EPSILON
    return 0;
```

- Written using doubles.
  - Change to float without changing epsilon?
  - DOT({10,10,10},{10,10,10}) breaks test!

Relative tolerance

Comparing two floats for equality:

```
if (Abs(x – y) <= EPSILON * Max(Abs(x), Abs(y)) …
```

- Epsilon scaled by magnitude of inputs
- But consider Abs(x)<1.0, Abs(y)<1.0

Combined tolerance

Comparing two floats for equality:

```
if (Abs(x – y) <= EPSILON * Max(1.0f, Abs(x), Abs(y))
   …
```

- Absolute test for Abs(x)≤1.0, Abs(y)≤1.0
- Relative test otherwise!

Floating-point numbers

- Caveat: Intel uses 80-bit format internally
  - Unless told otherwise.
  - Errors dependent on what code generated.
  - Gives different results in debug and release.

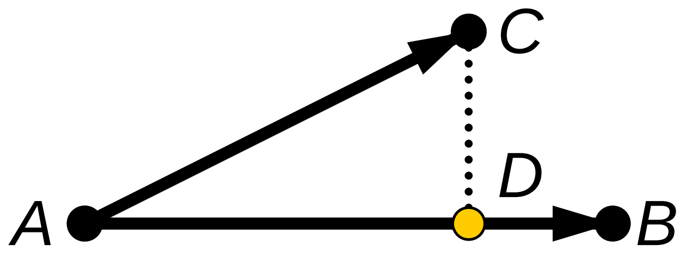
EXACT ARITHMETIC
(and semi-exact ditto)

Exact arithmetic

- Hey! Integer arithmetic is exact
  - As long as there is no overflow
  - Closed under +, –, and *
  - Not closed under / but can often remove divisions through cross multiplication

Exact arithmetic

- Example: Does *C* project onto *AB* ?



$$D = A + tAB, t = \frac{AC \cdot AB}{AB \cdot AB}$$

- Floats:

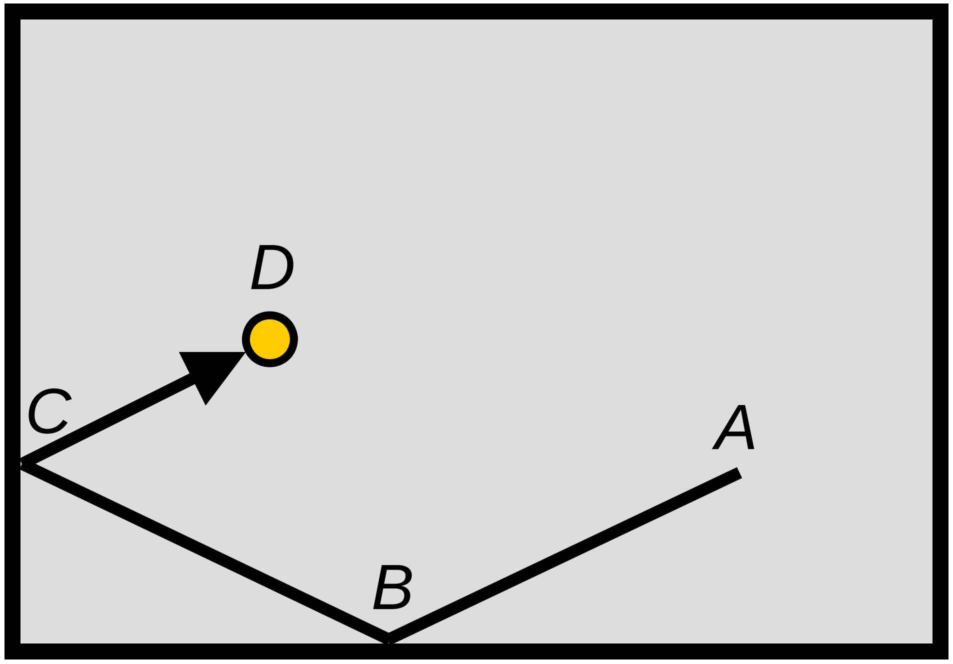
```
float t = Dot(AC, AB) / Dot(AB, AB);
if (t >= 0.0f && t <= 1.0f)
    ... /* do something */
```

- Integers:

```
int tnum = Dot(AC, AB), tdenom = Dot(AB, AB);
if (tnum >= 0 && tnum <= tdenom)
    ... /* do something */
```

Exact arithmetic

- Another example:



Exact arithmetic

- **Tests**
  - Boolean, can be evaluated exactly
- **Constructions**
  - Non-Boolean, cannot be done exactly

Exact arithmetic

- Tests, often expressed as determinant predicates. E.g.

$$P(\mathbf{u},\mathbf{v},\mathbf{w}) \equiv \begin{vmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix} \geq 0 \Leftrightarrow \mathbf{u}\cdot(\mathbf{v}\times\mathbf{w}) \geq 0$$

- Shewchuk's predicates well-known example
  - Evaluates using extended-precision arithmetic (EPA)
- EPA is expensive to evaluate
  - Limit EPA use through “floating-point filter”
  - Common filter is interval arithmetic

Exact arithmetic

- Interval arithmetic
  - $x = [1,3] = \{ x \in R \mid 1 \leq x \leq 3 \}$
  - Rules:
    - $[a,b] + [c,d] = [a+c,b+d]$
    - $[a,b] - [c,d] = [a-d,b-c]$
    - $[a,b] * [c,d] = [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)]$
    - $[a,b] / [c,d] = [a,b] * [1/d,1/c]$ for $0 \notin [c,d]$
  - E.g. $[100,101] + [10,12] = [110,113]$

Exact arithmetic

- Interval arithmetic
  - Intervals must be rounded up/down to nearest machine-representable number
  - Is a reliable calculation

References

BOOKS

- Ericson, Christer. **Real-Time Collision Detection**. Morgan Kaufmann, 2005. http://realtimecollisiondetection.net/
- Hoffmann, Christoph. **Geometric and Solid Modeling: An Introduction**. Morgan Kaufmann, 1989. http://www.cs.purdue.edu/homes/cmh/distribution/books/geo.html
- Ratschek, Helmut. Jon Rokne. **Geometric Computations with Interval and New Robust Methods**. Horwood Publishing, 2003.

PAPERS

- Hoffmann, Christoph. "**Robustness in Geometric Computations**." JCISE 1, 2001, pp. 143-156. http://www.cs.purdue.edu/homes/cmh/distribution/papers/Robustness/robust4.pdf
- Santisteve, Francisco. "**Robust Geometric Computation (RGC), State of the Art**." Technical report, 1999. http://www.lsi.upc.es/dept/techreps/ps/R99-19.ps.gz
- Schirra, Stefan. "**Robustness and precision issues in geometric computation.**" Research Report MPI-I-98-004, Max Planck Institute for Computer Science, 1998. http://domino.mpi-sb.mpg.de/internet/reports.nsf/NumberView/1998-1-004
- Shewchuk, Jonathan. "**Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates.**" Discrete & Computational Geometry 18(3):305-363, October 1997. http://www.cs.cmu.edu/~quake-papers/robust-arithmetic.ps