Intersection Detection

References

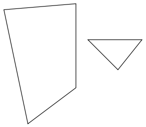
Notes based on

Tomas Moller and Eric Haines, Real-Time Rendering, A K Peters, 2nd edition, 2002. Chapter 9.

Franco Preparata and Michael Shamos, Computational Geometry: An Introduction, Springer Verlag, 1990. Chapter 2.

Introduction

Intersection detection is the problem of detecting whether two objects intersect — overlap in space. Intersection detection is carried out by performing *intersection tests*.

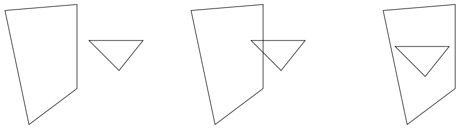


Intersection detection is a problem which occurs in computer graphics in many forms, including: clipping, view-volume culling, ray-tracing, picking and collision detection.
Intersection detection lies at the heart of *collision detection*.
Collision detection is intersection detection amongst a set of moving objects.

Three Possibilities

There are three spatial relationships between two objects:

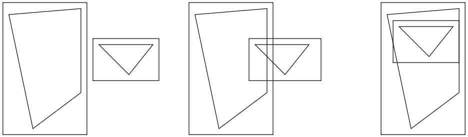
- no intersection, (disjoint, exclusion),
- partial intersection (overlap) and
- containment (inclusion).



It is important to keep all these possibilities in mind when performing intersection detection.

Bounding Volumes

The classic computer graphics technique of bounding volumes is applied in intersection detection.



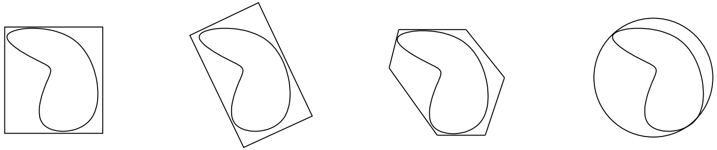
Instead of performing expensive intersection tests amongst complex objects (relatively) simpler tests using bounding volumes can be performed.
These tests allow quick ("trivially") determination of exclusion and inclusion cases, where no further work needs be done, and potential overlap cases where further work does need to be done in order to get a correct answer.

Types of Bounding Volumes

Common bounding volumes (BVs) are:

1. Axis-aligned bounding boxes (AABBs).
2. Oriented bounding boxes (OBBs).
3. Discrete oriented polytopes (k-DOPs).
4. Spheres.

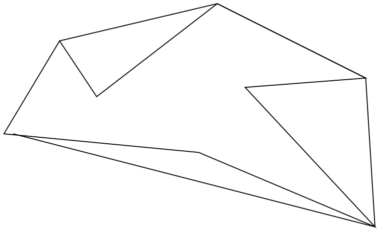
There is an obvious trade-off: tightness of fit versus complexity and cost of test.



One quantitive measure used in relation to BVs is *void volume*: the difference between the bounding volume and the object volume.

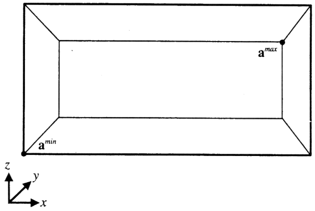
Convex Hulls

The tightest fitting bounding volume is the *convex hull*, although it is only strictly defined for polygonal objects or sets of points.



Axis-Aligned Bounding Boxes (AABBs)

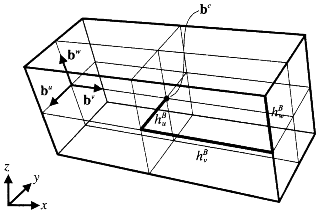
An axis-aligned bounding box (also called an *extent* or a *rectangular box*).



An axis-aligned bounding box is defined by two extreme points. For example an AABB called A is defined by $\mathbf{a}^{min}$ and $\mathbf{a}^{max}$, where $a_i^{min} \leq a_i^{max}, \forall i \in x, y, z$.

Oriented Bounding Boxes (OBBs)

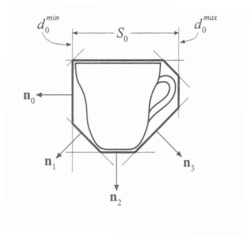
An oriented bounding box is a box which may be arbitrarily oriented (rotated). It is still a box; its faces have normals which are pairwise orthogonal.



An OBB B can be described by the center point of the box $\mathbf{b}^c$, and three normalised positively oriented vectors which describe the side directions of the box.
The vectors are $\mathbf{b}^u$, $\mathbf{b}^v$ and $\mathbf{b}^w$ and their half lengths are h_u^B, h_v^B and h_w^B.

Discrete Oriented Polytopes (k-DOPs)

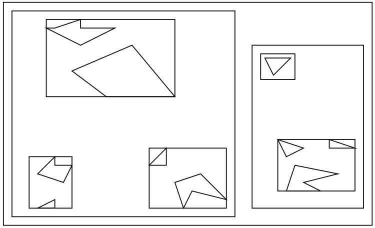
A k-DOP is the intersection of a set of pairs of parallel planes, where each pair of parallel planes is called a *slab*.



A k-DOP is defined by $k/2$ (k even) normalised normals, $\mathbf{n}_i$, $1 \leq i \leq k/2$ each with two associated scalar values d_i^{min} and d_i^{max} where $d_i^{min} < d_i^{max}$.
Each triple $(\mathbf{n}_i, d_i^m in, d_i^m ax)$ defines a slab S_i which is the volume betwen two planes $\pi_i^{min} : \mathbf{n}_i.(x) + d_i^{min} = 0$ and $\pi_i^{max} : \mathbf{n}_i.(x) + d_i^{max} = 0$.
The k-DOP volume is then the intersection of all slabs $\bigcap_{1 \leq l \leq k/2} S_l$.

Hierarchical Bounding Volumes

Bounding volumes can be placed inside other bounding volumes, recursively, thereby building *bounding volume hierarchies*.



Bounding volume hierarchies are one approach to improving collision detection performance amongst many objects or between complex objects.
Bounding volume hierarchies may reduce collision detection from an $O(n^2)$ algorithm to $O(n lg(n))$ or even $O(n)$.

Intersection Detection Principles or “Rules of Thumb”

- Perform calculations which allow *trivial acceptance* or *trivial rejection*.
- If possible, use results from above tests, even if they fail.
- Try re-ordering rejection and acceptance tests for better performance.
- Postpone expensive calculations.
- Consider reducing dimensionality of problem.
- If many objects are being tested against one object, pre-calculate values if possible.
- Perform timing tests and profiling to investigate performance.
- Make code robust (80% of work!).

Point-Point Intersection Testing

Two points $p_1(x_1, y_1, z_1)$ and $p_2(x_2, y_2, z_2)$ intersect if they are coincident:

$$x_1 = x_2, y_1 = y_2, z_1 = z_2$$

Interval-Interval Intersection Testing

Two intervals $[a^{min}, a^{max}]$ and $[b^{min}, b^{max}]$ are disjoint (do not intersect) if either $a^{min} > b^{max}$ or $b^{min} > a^{max}$.

```
    **interval_intersect**(A, B)
    returns (OVERLAP,DISJOINT)
1:  if ($a^{min} > b^{max}$ or $b^{min} > a^{max}$)
2:    return (DISJOINT);
3   else
4:    return (OVERLAP);
```

Figure: Interval Intersection Test

AABB-AABB Intersection Testing

Two axis-aligned bounding boxes (AABBs) intersect if they overlap in x or y or z. The AABB intersection test is essentially three interval intersection tests, with short-circuiting.

```
    **AABB_intersect**($A$, $B$)
    returns (OVERLAP,DISJOINT)
1:  for each $i \in x, y, z$
2:  if ($a_i^{min} > b_i^{max}$ or $b_i^{min} > a_i^{max}$)
3:    return (DISJOINT);
5:  else
4:    return (OVERLAP);
```

Figure: AABB Intersection Test

Sphere-Sphere Intersection Testing

Two spheres intersect if the distance between their centres $\mathbf{c}_1$ and $\mathbf{c}_2$ is less than the sum of their radii $r_1 + r_2$.

```
**sphere_intersect**($A$, $B$)
returns (OVERLAP,DISJOINT)
$\mathbf{l} = \mathbf{c}_2 - \mathbf{c}_1$
$d^2 = \mathbf{l}.\mathbf{l}$
if ($d^2 < (r_1 + r_2)^2$)
  return (OVERLAP);
else
  return (DISJOINT);
```

Figure: Sphere-Sphere Intersection Test

Using distance squared rather than distance avoids a square root calculation. Square roots *used* to be *very* expensive c.f. other operations, and are still *somewhat* expensive c.f. other operations.

Sphere-AABB Intersection Testing

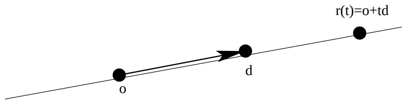
The sphere-AABB test also uses a distance test. The distance (squared) from the sphere centre to the box is accumlated in each dimension, then a distance test is applied.

```
     sphere_AABB_intersect($\mathbf{c}, r, A$)
     returns (OVERLAP,DISJOINT)
1:   $d^2 = 0$
2:   for each $i \in x, y, z$
3:     if ($c_i < a_i^{min}$)
4:       $d^2 = d^2 + (c_i - a_i^{min})^2$
5:     else if ($c_i > a_i^{max}$)
6:       $d^2 = d^2 + (c_i - a_i^{max})^2$
7:   if ($d^2 < r^2$)
8:     return (OVERLAP)
9:   else
10:    return (DISJOINT)
```

Figure: Sphere-AABB Intersection Test

Rays

Rays are directed lines. They may be finite, semi-infinite or infinite.



Rays are an important geometric object ("shape") used in intersection detection, (and, of course, ray tracing!). Rays are useful in intersection detection because:

1. They are sometimes good models of paths taken by moving objects, e.g., high speed bullets over short distances.
2. They can be used as a computationally efficient ("cheap") way to perform approximate intersection detection

Representing Rays

A ray is represented parametrically using a combination of a point (origin) **o** and a vector (direction) **d**.

$$\mathbf{r} = \mathbf{o} + t\mathbf{d}$$

If (x_o, y_o, z_o) is the ray origin and (u, v, w) is the ray direction then a point on the ray is given by

$$\begin{aligned} x &= x_o + tu \\ y &= y_o + tv \\ z &= z_o + tw \end{aligned}$$

A ray may also be specified by giving two points p_1 and p_2 in which case the equations become:

$$\begin{aligned} x &= x_1 + t(x_2 - x_1) \\ y &= y_1 + t(y_2 - y_1) \\ z &= z_1 + t(z_2 - z_1) \end{aligned}$$

It is often advantageous to normalise the ray direction. This saves repeatedly taking the magnitude (involving a `sqrt` operation) in dot products where vector projections are being calculated.

Ray-Sphere Intersection Testing

A sphere may be represented by the equation

$$(x-a)^2+(y-b)^2+(z-c)^2=r^2$$

The ray-sphere intersection is found by substituting the equations for the x, y and z coordinates of the ray into the sphere equation

$$(x_o+tu-a)^2+(y_o+tv-b)^2+(z_o+tw-c)^2=r^2$$

Rearrangement then gives

$$\begin{aligned} (u^2+v^2+w^2)t^2 + & \\ 2(u(x_o-a)+v(y_o-b)+w(z_o-c))t & + \\ (x_o-a)^2+(y_o-b)^2+(z_o-c)^2-r^2 & = 0 \end{aligned}$$

i.e. a quadratic in t. If the quadratic has real roots then the ray intersects the sphere, otherwise it does not.

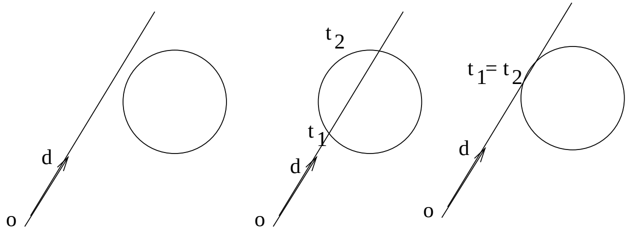
So we have

$$\begin{aligned} A &= (u^2 + v^2 + w^2)t^2 \\ B &= 2(u(x_0 - a) + v(y_0 - b) + w(z_0 - c))t \\ C &= (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 \end{aligned}$$

Solving for t we get

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

where the smallest positive root is the first intersection.



Minor Optimisation

A small optimisation can be achieved by noting that a factor of two cancels, giving

$$B = u(x_0 - a) + v(y_0 - b) + w(z_0 - c)t$$

and

$$t = \frac{-B \pm \sqrt{B^2 - AC}}{A}$$

These kinds of optimisations tend to be of decreasing value with increasing floating point performance. They are also potential source for introducting bugs and creating obscurity.

Optimised Solution

Haines optimised version of the ray-sphere intersection detection algorithm allows earlier short-circuiting.

```
    ray_sphere_intersect(o,d,c,r)
    returns (REJECT,INTERSECT,t,p)
1:  $\mathbf{l} = \mathbf{c} - \mathbf{o}$
2:  $d = \mathbf{l}.\mathbf{d}$
3:  $l^2 = \mathbf{l}.\mathbf{l}$
4:  if ($d < 0$ and $l^2 > r^2$) return (REJECT,0,$\mathbf{O}$);
5:  $m^2 = l^2 - d^2$
6:  if ($m^2 > r^2$) return (REJECT,0,$\mathbf{O}$);
7:  $q = \sqrt{r^2 - m^2}$
8:  if ($l^2 > r^2$) $t = d - q$ else $t = d + q$
9:  return (INTERSECT,$t$,$\mathbf{o} + t\mathbf{d}$);
```

Figure: Optimised Algorithm

Ray-Polygon Intersection Detection

To find the intersection point between a ray and a polygon:

1. Find the intersection point of the ray and the plane containing the polygon. If it does not exist then finished.
2. Determine if the ray-plane intersection point is inside the polgon, i.e., perform a point-polygon intersection (containment) test.

Ray-Polygon Intersection Point

The equation of a plane is

$$Ax + By + Cz + D = 0$$

The equation of a ray is

$$\begin{aligned} x &= x_o + tu \\ y &= y_o + tv \\ z &= z_o + tw \end{aligned}$$

Substituting into the plane equation we get

$$A(x_o + tu) + B(y_o + tv) + C(z_o + tw) + D = 0$$

Rearrangement gives

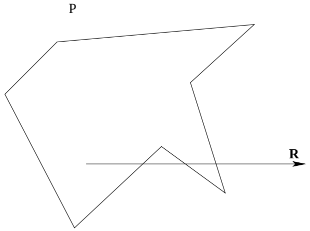
$$t = -\frac{(Ax_o + By_o + Cz_o + D)}{Au + Bv + Cw}$$

Polygon Containment Test

The polygon containment test is performed by projecting the polygon and the intersection point onto one of the coordinate planes along one of the three principal axis directions and then performing a 2D point in polygon test.
Which axis do we project along?

Point in Polygon: Crossings Test

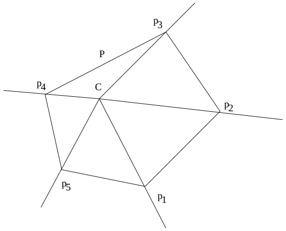
One $O(n)$ approach is to count the number of crossings or intersections between a *semi-infinite line* or *ray* and edges of the polygon.



Count the number of intersections of P with R. If the number of intersections (crossings) of is odd then z is inside, otherwise it is outside. This is a result of the Jordon curve thereom.
Special cases, e.g., vertices on the ray, need to be handled.
How can this be done?

Convex Polygons

In the case of a convex polygon an $O(lg(n))$ point-in-polygon (or *containment*) algorithm is possible.



The point q can be any internal point, e.g., the centroid of the triangle determined by any three vertices of P.
Using polar coordinates, a binary search can be performed to find the wedge which contains z. Then z can be compared against the wedge edge.
How much preprocessing is required?

Concave Polygons

Can point containment for *concave* polygons be performed in $O(lg(n))$ time?
If so, how much preprocessing is required?

Brute Force Collision Detection

A *brute force* approach to collision detection uses intersection tests of all object pairs — or *pairwise testing* — at discrete points in time where all objects are assumed to be stationary

```
for $i = 1$ to $n - 1$
    for $j$ = $i$ to $n$
        intersect(object($i$), object($j$))
```

Figure: Brute-force collision detection

There are $\frac{n(n-1)}{2}$ pairs amonst n objects, and this is the number of object-object intersection tests the brute-force algorithm performs at each point in time.
Using the big O notation, the brute-force algorithm uses $O(n^2)$ object-object intersection tests. If each intersection test can be performed in $O(1)$ time, i.e., *constant time*, then the algorithm is $O(n^2)$ time.