Chap 13. Intersection Test Methods

0. Intersection testing is needed to click the mouse on an object, or to determine whether two objects collide.

0. Given its wide range of uses, intersection testing will remain in the application stage for the foreseeable future.

0. One picking method, supported by OpengL, is to render all objects into a tiny pick window. All the objects that overlap the window are returned in a list, along with the minimum and maximum z-depth found for each object. This system is able to pick on lines and vertices in addition to surfaces. This pick method is implemented entirely on the host processor and does not use the graphics accelerator.

0. Intersection testing for picking is more efficient. Comparing a ray against a triangle is normally faster than sending the triangle through the OpengL pipeline.

0. Intersection testing methods can compute the location, normal vector, texture coordinates, and other surface data.

0. The picking problem can be solved efficiently by using a bounding volume hierarchy. The ray from the camera position through the picked pixel is recursively tested whether it intersects bounding volumes in the hierarchy.

0. In collision detection algorithms, the system must decide whether two primitive objects collide. The primitive objects include triangles, spheres, Axis-aligned Bounding Boxes (CAABBs), Oriented Bounding Boxes (OBBs), and Discrete Oriented Polytopes (k-DOPs).
13.1 Hardware-Accelerated Picking

1. Hanrahan and Haeberli [320] presented a hardware-accelerated picking method. The scene is rendered into the Z-buffer with lighting off and with each polygon having a unique color value that is used as an identifier. The image is stored off-screen and is used for rapid picking. When the user clicks on a pixel, the color identifier is looked up in this image and the polygon is immediately identified. The major disadvantage is that the entire scene must be rendered in a separate pass to support the picking.

2. It is possible to find the relative location of a point inside a triangle using the color buffer. Each triangle is rendered using Gouraud interpolation, where the three vertices are colored with red (255, 0, 0), green (0, 255, 0), and blue (0, 0, 255). Say the pixel has a color (23, 192, 40). The corresponding barycentric coordinates are (23/255, 192/255, 40/255).

3. The normals can also be encoded into RGB colors. The interval [-1, 1] per normal component is transformed to the interval [0, 1] so it can be rendered. After rendering the triangles using Gouraud interpolation, the normal at any point can be read from the color buffer.

4. Another method of using graphics hardware for picking is to render the identifiers into the stencil buffer. When Z-buffering is used, also using the stencil buffer comes for free if the 8 bits of the stencil buffer share the same word as the 24 bits of the Z-buffer. So, pick identifiers can be used at no extra cost. The limitation is that there are only 256 identifiers available.

⇒ Split the scene into 255 parts, render and identify a range of polygons, split it again into 255 parts, and repeat.
13.2 Definitions and Tools

0. A ray, \( \mathbf{r}(t) = \mathbf{p} + t \mathbf{d}, \) where \( \| \mathbf{d} \| = 1 \)

origin point, direction vector

0. An implicit surface, \( f(c) = f(p_x, p_y, p_z) = 0. \)

Ex: \( p_x^2 + p_y^2 + p_z^2 = r^2, \) a sphere with radius \( r. \)

0. An explicit surface,

\[
\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = f(\rho, \phi) = \begin{pmatrix} f_x(\rho, \phi) \\ f_y(\rho, \phi) \\ f_z(\rho, \phi) \end{pmatrix}.
\]

Ex: a sphere located at the origin with radius \( r \)

\[
f(\rho, \phi) = \begin{pmatrix} r \sin \rho \cos \phi \\ r \sin \rho \sin \phi \\ r \cos \rho \end{pmatrix}
\]

Ex: A triangle \( \mathbf{v}_0 \mathbf{v}_1 \mathbf{v}_2 \) can be represented as

\[
h(\mathbf{v}) = (1-u-v) \mathbf{v}_0 + u \mathbf{v}_1 + v \mathbf{v}_2, \text{ for } 0 \leq u, v, u + v \leq 1
\]

Definitions:

0. Axis-aligned Bounding Box (AABB) is a box whose faces have normals that coincide with the standard basis axes.

0. Oriented Bounding Box (OBB) is a box whose faces have normals that are all pairwise orthogonal.

0. Discrete Oriented Polytope (\( k\)-DOP) is defined by \( k/2 \) normalized normals (orientations), \( 0 \leq i \leq k/2, \) and with each \( i \) two associated scalar values \( d_i^{\text{min}} \) and \( d_i^{\text{max}}, \) where \( d_i^{\text{min}} < d_i^{\text{max}}. \) Each triple \( (i, d_i^{\text{min}}, d_i^{\text{max}}) \) describes a slab \( S_e, \) the volume between two planes. The \( k\)-DOP volume is \( \sum_{i \leq S \leq k/2} S_e. \)
Figure 13.2. A three-dimensional AABB, called $A$, with its extreme points, $a^\text{min}$ and $a^\text{max}$, and the axes of the standard basis.

Figure 13.3. A three-dimensional OBB, called $B$, with its center point, $b^c$, and its normalized, positively oriented side vectors, called $b^u$, $b^v$, and $b^w$. The half-lengths of the sides, $h_u^B$, $h_v^B$, and $h_w^B$, are the distances from the center of the box to the center of the faces, as shown.

Figure 13.4. An example of a two-dimensional 8-DOP for a tea cup, with all normals, $n_i$, shown along with the zero'th slab, $S_1$, and the "size" of the slab: $d_1^\text{min}$ and $d_1^\text{max}$.
13.3 Bounding Volume Creation

0. Given a collection of objects, finding a tight fitting bounding volume is important to minimizing intersection costs. It is easy to create AABBs and k-DOPs.

Sphere Creation

0. A fast algorithm is to find a bounding sphere of AABB. This sometimes gives a poor fit, and the fit can often be improved by starting with the center of the AABB as the center of the sphere BV, go through all vertices again and find the farthest vertex from the center.

0. Ritter [653] presents a simple algorithm that creates a near-optimal bounding sphere. Find the pair of vertices that produces the largest side of the AABB. Take the sphere's center at the midpoint between the two extreme vertices, and go through all the other vertices and check its distance to the center of the sphere. If the vertex is outside the sphere's radius r, move the sphere's center toward the vertex by (d-r)/2, set the radius to (d+r)/2, and continue. The new sphere encloses the vertex and the existing sphere.

0. Welzl [798] presents a randomized algorithm with expected linear-time complexity. The idea is to find a supporting set of points defining a sphere. When a vertex is found to be outside the current sphere, its location is added to the supporting set, the new sphere is computed, and the entire list is run through again, and the process repeats until the sphere contains all vertices. This algorithm guarantees that an optimal bounding sphere is found.
OBB Creation

1. Gottschalk [284] presents a method that first computes the convex hull of an object. Given \( n \) triangles, \( \Delta p_k q_k r_k \), \( 0 \leq k < n \), on the convex hull, \( \Delta_k \) is the area of triangle \( k \), and the total area of the convex hull is \( \Delta \sum_k \). The centroid of triangle \( k \) is \( \mathbf{m}^k = (\mathbf{p}_k + \mathbf{q}_k + \mathbf{r}_k) / 3 \). The centroid of the whole convex hull is

\[
\mathbf{m}^H = \frac{1}{\Delta} \sum_{k=0}^{n-1} \Delta_k \mathbf{m}^k.
\]

2. A 3x3 covariance matrix is computed

\[
C = [c_{ij}] = \left( \frac{1}{\Delta} \sum_{k=0}^{n-1} \Delta_k \left( 9m_i^k m_j^k + p_i^k p_j^k + q_i^k q_j^k + r_i^k r_j^k \right) - m_i^H m_j^H \right),
\]

\[0 \leq i, j < 3\] (13.6)

The eigenvectors of \( C \) are computed and normalized. These vectors are the direction vectors, \( \mathbf{a}_i^u \), \( \mathbf{a}_i^v \), and \( \mathbf{a}_i^w \). We then compute the center and the half-lengths of the OBB. This is done by projecting the points of the convex hull onto the direction vectors and finding the minimum and maximum along each direction.

3. When computing the OBB, the most demanding operation is the convex hull computation, which takes \( O(n \log n) \) time. (But, one can use a fast approximation algorithm.) The basis calculation takes at most linear time, and the eigenvector computation takes constant time.

4. Eberly [199] samples the set of possible directions of the box, and uses the axes whose box is smallest as a starting point. Then Powell's direction set method [635] is used to find the minimum volume box.
Separating Axis Theorem (SAT): For any two arbitrary, convex, disjoint polyhedra, A and B, there exists a separating axis where the projections of the polyhedra, which form intervals on the axis, are also disjoint. If A and B are disjoint, then they can be separated by an axis that is orthogonal (i.e., by a plane parallel) to (i) a face of A, (ii) a face of B, or (iii) An edge from each polyhedron (e.g., cross product).

0. This theorem is used in deriving the box/line segment overlap test, the triangle/box overlap test, and the OBB/OBB overlap test.

0. A common technique for optimizing intersection tests is to make some simple calculations early on that can determine whether the ray or object totally misses the other object. Such a test is called a ‘rejection test’, and if the test succeeds, the intersection is said to be rejected.

0. Another approach often used is to project the 3D objects onto the best orthogonal plane (xy, xz, or yz), and solve the problem in two dimensions instead.

0. Due to numerical imprecision, we often use a very small number ε, and its value varies from test to test. Often an epsilon is chosen that works for the programmer’s problem cases, as opposed to doing careful roundoff error analysis. The code in another setting may well break because of differing conditions.
13.4 Rules of Thumb

Before we begin studying the specific intersection methods, here are some rules of thumb that can lead to faster, more robust, and more exact intersection tests. These should be kept in mind when designing, inventing, and implementing an intersection routine:

- Perform computations and comparisons that might trivially reject or accept various types of intersections to obtain an early escape from further computations.

- If possible, exploit the results from previous tests.

- If more than one rejection or acceptance test is used, then try changing their internal order (if possible), since a more efficient test may result. Do not assume that what appears to be a minor change will have no effect.

- Postpone expensive calculations (especially trigonometric functions, square roots, and divisions) until they are truly needed (see Section 13.7 for an example of delaying an expensive division).

- The intersection problem can often be simplified considerably by reducing the dimension of the problem (for example, from three dimensions to two dimensions or even to one dimension). See Section 13.8 for an example.

- If a single ray or object is being compared to many other objects at a time, look for precalculations that can be done just once before the testing begins.

- Whenever an intersection test is expensive, it is often good to start with a sphere around the object to give a first level of quick rejection.

- Make it a habit always to do timing comparisons on your computer, and use real data and testing situations for the timings.

- Finally, try to make your code robust. This means it should work for all special cases and that it will be insensitive to as many floating point precision errors as possible. Be aware of any limitations it may have.
13.5 Ray/Sphere Intersection

Algebraic Approach to Ray/Quadric Intersection

1. When a ray \( \mathbf{r(t)} = \mathbf{o} + t\mathbf{d} \) is intersected with a quadric, we get a quadratic equation \( at^2 + bt + c = 0 \), which can be solved for the smaller \( t > 0 \).

2. The same approach can be applied to other implicit surfaces.

Geometric Approach to Ray/Sphere Intersection

1. \( \mathbf{l} = \mathbf{c} - \mathbf{o} \): the vector from the ray origin to the sphere center.
2. \( l^2 = \mathbf{l} \cdot \mathbf{l} < r^2 \): the ray origin is inside the sphere.
3. \( \mathbf{s} = \mathbf{l} \cdot \mathbf{d} \): the projection of \( \mathbf{l} \) onto the ray direction.

```plaintext
RaySphereIntersect(o, d, c, r)
    returns ((REJECT, INTERSECT), t, p)
1:    l = c - o
2:    s = l . d
3:    l^2 = l . l
4:    if(s < 0 and l^2 > r^2) return (REJECT, 0, 0);
5:    m^2 = l^2 - s^2
6:    if(m^2 > r^2) return (REJECT, 0, 0);
7:    q = sqrt(r^2 - m^2)
8:    if(l^2 > r^2) t = s - q
9:    else t = s + q
10:   return (INTERSECT, t, o + td);
```

Figure 13.6. The notation for the geometry of the optimized ray/sphere intersection. In the left figure, the ray intersects the sphere in two points, where the distances are \( t = s \pm q \) along the ray. The middle case demonstrates a rejection made when the sphere is behind the ray origin. Finally, at the right, the ray origin is inside the sphere, in which case the ray always hits the sphere.
13.6 Ray/Box Intersection

13.6.1 Slabs Method

0. The intersection with each slab produces \( t_i^{\text{min}} \) and \( t_i^{\text{max}} \),
\[
\begin{align*}
    t_i^{\text{min}} &= \max t_i^{\text{min}}, \quad t_i^{\text{max}} &= \min t_i^{\text{max}} \\
\end{align*}
\]

0. \( t_i^{\text{min}} \leq t_i^{\text{max}} \iff \) the ray intersects the box.

Figure 13.7. The left figure shows a two-dimensional OBB (Oriented Bounding Box) formed by two slabs, while the right shows two rays that are tested for intersection with the OBB. All \( t \)-values are shown, and they are subscripted with \( v \) for the light gray slab and with \( u \) for the other. The extreme \( t \)-values are marked with boxes. The left ray hits the OBB since \( t^{\text{min}} < t^{\text{max}} \), and the right ray misses since \( t^{\text{max}} < t^{\text{min}} \).

\[
\text{RayOBBIntersect}(o, d, A) \\
\quad \text{return} \{(\text{REJECT, INTERSECT}), t\};
\]

\[
1: \quad t^{\text{min}} = -\infty \\
2: \quad t^{\text{max}} = \infty \\
3: \quad p = a^t - o \\
4: \quad \text{for each } i \in \{u, v, w\} \\
5: \quad e = a^i \cdot p \\
6: \quad f = a^i \cdot d \\
7: \quad \text{if}(|f| > e) \\
8: \quad t_1 = (e + h_i)/f \\
9: \quad t_2 = (e - h_i)/f \\
10: \quad \text{if}(t_1 > t_2) \text{ swap}(t_1, t_2); \\
11: \quad \text{if}(t_1 > t^{\text{min}}) \ t^{\text{min}} = t_1 \\
12: \quad \text{if}(t_2 < t^{\text{max}}) \ t^{\text{max}} = t_2 \\
13: \quad \text{if}(t^{\text{min}} > t^{\text{max}}) \text{ return } (\text{REJECT, 0}); \\
14: \quad \text{if}(t^{\text{max}} < 0) \text{ return } (\text{REJECT, 0}); \\
15: \quad \text{else if}(-e - h_i > 0 \text{ or } -e + h_i < 0) \text{ return } (\text{REJECT, 0}); \\
16: \quad \text{if}(t^{\text{min}} > 0) \text{ return } (\text{INTERSECT, } t^{\text{min}}); \\
17: \quad \text{else return } (\text{INTERSECT, } t^{\text{max}});
\]
Plane equation: \( a^{r} \cdot [x - (a^{c} + t \cdot \hat{n} \cdot a^{r})] = 0 \)

Ray/Plane intersection:
\[
\begin{align*}
    a^{r} \cdot [0 + t \cdot dl - a^{c} + t \cdot \hat{n} \cdot a^{c}] &= 0 \\
    (a^{r} \cdot dl) t &= a^{r} \cdot (a^{c} - 0) + t \cdot (a^{r} \cdot \hat{n} \cdot a^{c}) \\
    t &= \frac{a^{r} \cdot 1P \pm t_{c}}{a^{r} \cdot dl}
\end{align*}
\]

13.6.2 Woo's Method

Woo [SP18] introduced some smart optimizations for finding the intersection between a ray and an AABB. The back-facing planes are discarded from consideration. We compute the intersection distances to front-facing planes and take the largest of these distances. Finally, the actual intersection point is computed, and if it is located on the corresponding face of the box, it is a real hit. Whether the slabs method or Woo's method is faster is an open question. The methods are comparable in performance, with each strongly affected by various factors.

Figure 13.8. Woo's method for computing the intersection between a ray and an AABB. The candidate planes, which are front-facing and marked with fat lines, are intersected with the ray, and their intersection points are marked with gray dots. Because the left intersection point is farthest from the ray origin, it is selected as a (potential) point of intersection.
13.6.3 Line Segment/Box Overlap Test

Figure 13.9. Left: The x-axis is tested for overlap between the box and the line segment. Right: The a-axis is tested. Note that $h_y|w_x| + h_z|w_y|$ and $|c_yw_z - c_zw_y|$ are shown in this illustration as if $\mathbf{x}$ is normalized. When this is not the case, then both the shown values should be scaled with $|\mathbf{x}|$. See text for notation.

$$\sqrt{w_y^2 + w_z^2}$$

```
RayAABBOverlap(c, w, B) {
    returns ({OVERLAP, DISJOINT});
    1:  $v_x = |w_x|$
    2:  $v_y = |w_y|$
    3:  $v_z = |w_z|$
    4:  if($|c_x| > v_x + h_x$) return DISJOINT;
    5:  if($|c_y| > v_y + h_y$) return DISJOINT;
    6:  if($|c_z| > v_z + h_z$) return DISJOINT;
    7:  if($|c_yw_z - c_zw_y| > h_yv_z + h_zv_y$) return DISJOINT;
    8:  if($|c_xw_z - c_zw_x| > h_xv_z + h_zv_x$) return DISJOINT;
    9:  if($|c_yw_y - c_zw_y| > h_xv_y + h_yv_x$) return DISJOINT;
   10: return OVERLAP;
}
```

0. The cross product axis $\mathbf{a} = \mathbf{w} \times (1, 0, 0) = (0, w_x, -w_y)$.
    The extent of the box projected onto $\mathbf{a}$ is $h_x/w_x + h_z/w_y$.
    The projection of the line segment direction $\mathbf{w}$ onto $\mathbf{a}$
    is 0, since $\mathbf{w} \cdot (\mathbf{w} \times (1, 0, 0)) = 0$.
    The projection of $\mathbf{c}$ onto $\mathbf{a}$ is $\mathbf{c} \cdot \mathbf{a} = c_yw_z - c_zw_y$.
    Thus, the test for the axis $\mathbf{a}$ becomes
    $|c_yw_z - c_zw_y| > h_y/w_x + h_z/w_y$.
generates three axes to test: \((1, 0, 0)\), \((0, 1, 0)\), and \((0, 0, 1)\). Finally, the axes formed from the cross product between line segment direction and the box axes should be tested.

The test for the axis \((1, 0, 0)\) is shown below, and Figure 13.9 shows this as well. The other two axes are similar.

\[
|c_x| > |w_x| + h_x \tag{13.14}
\]

If this test is true then the line segment and the box do not overlap.

The cross product axis \(\mathbf{a} = \mathbf{w} \times (1, 0, 0) = (0, w_z, -w_y)\) is tested as follows. The extent of the box projected onto \(\mathbf{a}\) is \(h_y|w_z| + h_z|w_y|\). Next we need to find the projected length of the line segment, and the projection of the line segment center. The projection of the line segment direction \(\mathbf{w}\) onto \(\mathbf{a}\) is zero, since \(\mathbf{w} \cdot (\mathbf{w} \times (1, 0, 0)) = 0\). The projection of \(\mathbf{c}\) onto \(\mathbf{a}\) is \(\mathbf{c} \cdot \mathbf{a} = \mathbf{c} \cdot (0, w_z, -w_y) = c_y w_z - c_z w_y\). Thus, the test becomes:

\[
|c_y w_z - c_z w_y| > h_y|w_z| + h_z|w_y|. \tag{13.15}
\]

Again, if this test is true, then there is no overlap. The other two cross product axes are similar.
13.7 Ray/Triangle Intersection

0. The normals of triangles are often not stored. Thus, the normals must be computed if needed.

0. Ray/triangle intersection tests usually compute the intersection point between the ray and the plane. The intersection point and the triangle vertices are then projected on the xy, yz, or xz-plane where the area of the projection is maximum. The problem is reduced to testing if a 2D point is inside a 2D triangle. There are several methods using this 3D→2D reduction.

13.7.1. Intersection Algorithm

0. A point \( t(u, v) \) on a triangle:

\[
t(u, v) = (1-u-v)v_0 + uv_1 + uv_2,
\]

where \( 0 \leq u, v, 1-u-v \leq 1 \). \((u, v)\): barycentric coordinate

0. Intersection with a ray:

\[
\vec{0} + td \perp = (1-u-v)v_0 + uv_1 + uv_2.
\]

\[
\begin{bmatrix}
-dl & v_1-v_0 & v_2-v_0 \\
\end{bmatrix}
\begin{bmatrix}
t \\
u \\
v
\end{bmatrix}
= \vec{0} - v_0
\]

This linear system of equation can be solved by Cramer's rule:

\[
\begin{bmatrix}
t \\
u \\
v
\end{bmatrix}
= \frac{1}{\det(-dl, \epsilon_1, \epsilon_2)}
\begin{bmatrix}
\det(\epsilon_1, \epsilon_1, \epsilon_2) \\
\det(-dl, \epsilon_1, \epsilon_2) \\
\det(-dl, \epsilon_1, \epsilon_2)
\end{bmatrix}
\]

where \( \epsilon_1 = v_1-v_0, \epsilon_2 = v_2-v_0, \vec{0} = \vec{0} - v_0 \).
Figure 13.11. Translation and change of base of the ray origin.

\[
\begin{pmatrix}
    t \\
    u \\
    v
\end{pmatrix}
= \frac{1}{(d \times e_2) \cdot e_1}
\begin{pmatrix}
    (s \times e_1) \cdot e_2 \\
    (d \times e_2) \cdot s \\
    (s \times e_1) \cdot d
\end{pmatrix}
= \frac{1}{p \cdot e_1}
\begin{pmatrix}
    q \cdot e_2 \\
    p \cdot s \\
    q \cdot d
\end{pmatrix},
\]

where \( p = d \times e_2 \) and \( q = s \times e_1 \).

### 13.7.2 Implementation

```plaintext
RayTriIntersect(o, d, v_0, v_1, v_2)
returns ([REJECT, INTERSECT], u, v, t);
1: e_1 = v_1 - v_0
2: e_2 = v_2 - v_0
3: p = d \times e_2
4: a = e_1 \cdot p
5: if(a > -\epsilon \text{ and } a < \epsilon) \text{ return (REJECT, 0, 0, 0);}
6: f = 1/a
7: s = o - v_0
8: u = f(s \cdot p)
9: \text{ if}(u < 0.0 \text{ or } u > 1.0) \text{ return (REJECT, 0, 0, 0);}
10: q = s \times e_1
11: v = f(d \cdot q)
12: \text{ if}(v < 0.0 \text{ or } u + v > 1.0) \text{ return (REJECT, 0, 0, 0);}
13: t = f(e_2 \cdot q)
14: return (INTERSECT, u, v, t);
```

0. This method is the fastest ray/triangle intersection routine that does not need to store the normals.
Rearranging the terms gives:

$$
\begin{pmatrix}
  v_1 - v_0 & v_2 - v_0 & -d \\
  -d & v_1 - v_0 & v_2 - v_0
\end{pmatrix}
\begin{pmatrix}
  u \\
  v \\
  t
\end{pmatrix} = o - v_0.
$$

(13.18)

This means the barycentric coordinates \((u, v)\) and the distance \(t\) from the ray origin to the intersection point can be found by solving the linear system of equations above.

The above can be thought of geometrically as translating the triangle to the origin and transforming it to a unit triangle in \(y\) and \(z\) with the ray direction aligned with \(x\). This is illustrated in Figure 13.11. If \(M = \begin{pmatrix}
  d & v_1 - v_0 & v_2 - v_0
\end{pmatrix}\) is the matrix in Equation 13.18, then the solution is found by multiplying Equation 13.18 with \(M^{-1}\).

Denoting \(\hat{e}_1 = v_1 - v_0, \hat{e}_2 = v_2 - v_0,\) and \(s = o - v_0,\) the solution to Equation 13.18 is obtained by using Cramer's rule:

$$
\begin{pmatrix}
  u \\
  v \\
  t
\end{pmatrix} = \frac{1}{\det(-d, \hat{e}_1, \hat{e}_2)}
\begin{pmatrix}
  \det(s, \hat{e}_1, -d) \\
  \det(s, \hat{e}_2, -d) \\
  \det(s, \hat{e}_1, \hat{e}_2)
\end{pmatrix}.
$$

(13.19)

Figure 13.11. Translation and change of base of the ray origin.
13. Ray/Polygon Intersection

0. A polygon of \( n \) vertices: \( \{v_0, v_1, \ldots, v_{n-1}\} \)

  The plane of the polygon: \( \Pi_p: \mathbf{n}_p \cdot \mathbf{x} + d_p = 0 \).

  The intersection with a ray
  \[ \mathbf{n}_p \cdot (\mathbf{o} + t \mathbf{d}) + d_p = 0 \]
  \[ t = -d_p - \frac{\mathbf{n}_p \cdot \mathbf{o}}{\mathbf{n}_p \cdot \mathbf{d}} \]

0. The problem is reduced to 2D. All vertices and

  \( \mathbf{p} = \mathbf{o} + t \mathbf{d} \) are projected to one of the \( xy \), \( yz \), or \( xz \)-

  planes where the area of the projected polygon is maximized.

![Figure 13.12](image)

**Figure 13.12.** Orthographic projection of polygon vertices and intersection point \( \mathbf{p} \) onto

the \( xy \)-plane, where the area of the projected polygon is maximized. This is an example

of using dimension reduction to obtain simpler calculations.

0. A 2D bounding box is sometimes profitable.

  If the point is outside the box, then reject and return.

  This was found to be a better approach than using

  a 3D bounding box for a polygon.

0. The point-in-polygon test can be done by counting

  the number of crossings between the 2D polygon and

  a ray from \( \mathbf{p} \) to infinity.
The Crossings Test

1. **Jordan Curve Theorem**: A point is inside a polygon if a ray from this point in an arbitrary direction crosses an odd number of polygon edges.

2. The crossing algorithm is the fastest test that does not use preprocessing. It works by shooting a ray from \( P \) along the positive \( x \)-axis.

3. The test point \( P \) can be thought of as being at the origin. If the \( y \)-coordinates of a polygon edge have different signs, the \( x \)-coordinates are checked. If both are positive, then there is a crossing. If they differ in sign, the \( x \)-coordinate of the intersection must be computed, and if it is positive, there is a crossing.

4. The following code runs faster than the above algorithm since we avoid the division needed for the \( x \)-intercept value. Line 8 tests whether the \( x \)-intercept is positive.

The test point is \( P = (tx, ty) \).

```c
bool PointInPolygon(t, P) {
    return ((TRUE, FALSE));
}
```

```c
1: bool inside = FALSE
2: e0 = v_n-1
3: e1 = v_0
4: bool y0 = (e0_y >= t_y)
5: for i = 1 to n
6:     bool y1 = (e1_y >= t_y)
7:     if(y0 != y1)
8:         if(((e1_y - t_y)(e0_x - e1_x) <= (e1_x - t_x)(e0_y - e1_y)) = y1)
9:             inside = ~inside
10:     y0 = y1
11:     e0 = e1
12:     e1 = v_i
13: return inside;
```
13.9 Plane/Box Intersection Detection

0. Test two extreme points of a box against a plane. The two extreme points form a diagonal of the box, which is the most aligned to the normal in of the plane.

Figure 13.15. Here, the $v_{min}$ and $v_{max}$ vertices are shown for three AABBs (in two dimensions) for a given plane. If the pair of these vertices are on the same side of the plane, then the AABB does not intersect the plane; otherwise, it does. Note that if $v_{min}$ is tested against the plane first and is found to be on the same side as the plane normal, then the AABB is “outside,” i.e., in the positive half-space of the plane.

13.9.1 AABB

0. If $B = AABB$ defined by $l_{b_{min}}$ and $l_{b_{max}}$. Find the most aligned diagonal with vertices $v_{min}$ and $v_{max}$.

0. If $n \cdot v_{min} + d > 0$, then $n \cdot v_{max} + d > 0 \Rightarrow$ no need to test $v_{max}$.

```cpp
bool PlaneAABBIntersect(B, π)
returns({OUTSIDE, INSIDE, INTERSECTING});
1:   for each i ∈ {x, y, z}
2:     if($n_i ≥ 0$)
3:       $v_{min}^i = b_{min}^i$
4:       $v_{max}^i = b_{max}^i$
5:     else
6:       $v_{min}^i = b_{max}^i$
7:       $v_{max}^i = b_{min}^i$
8:     if(($n \cdot v_{min} + d > 0$) return (OUTSIDE);
9:     if(($n \cdot v_{max} + d < 0$) return (INSIDE);
10:    return (INTERSECTING); ```
A faster way to get the sign bits of the plane normal components and use a Look-Up Table (LUT) to retrieve $v_{\min}$ and $v_{\max}$. For example, \( \mathbf{n} = (-1, 0.5, 2) \) gives a bitmask of 011. The box coordinates are stored in an array \( \{0, \ldots, 5\} \) as \((b_{x, \min}, b_{y, \min}, b_{z, \min}, b_{x, \max}, b_{y, \max}, b_{z, \max})\). The LUT entry for bitmask 011 (i.e., index 3) would be \((0, 4, 5)\), meaning \(v_{\max} = (b_{x,\min}, b_{y,\max}, b_{z,\max})\). \(v_{\min}\) can be retrieved using the bitmask 100 (the inversion of 011).

13.9.2 OBB

We transform the normal of the plane so that it lies in the coordinate system of the OBB.

\[
\mathbf{n}' = (\mathbf{h}_u \cdot \mathbf{n}, \mathbf{h}_v \cdot \mathbf{n}, \mathbf{h}_w \cdot \mathbf{n}).
\]

Using \(\mathbf{n}'\) instead of \(\mathbf{n}\), the rest of the test is the same.

Another way to test is to project the axes of the OBB onto the normal of the plane. Half the length of the projection \(r = \mathbf{h}_u \cdot (\mathbf{n} \cdot \mathbf{h}_u) + \mathbf{h}_v \cdot (\mathbf{n} \cdot \mathbf{h}_v) + \mathbf{h}_w \cdot (\mathbf{n} \cdot \mathbf{h}_w)\).

No intersection \(\iff |\mathbf{h}_u \cdot \mathbf{n} + d| > r\)

\(\Rightarrow\) The distance from the center of \(\mathbf{B}\) to the plane is greater than \(r\), then they do not intersect.

Figure 13.16. The extents of the OBB are projected onto the normal of the plane. Half the “size” of the OBB along the normal’s direction is denoted \(r\). If the distance from the center of the OBB to the plane is greater than \(r\), then they do not intersect.
13.10 Triangle/Triangle Intersection

0. The deepest levels of a collision detection algorithm typically have a routine for testing triangle-triangle intersection.

0. The separating axis theorem can be used to derive a triangle-triangle overlap test. However, this approach does not produce the line segment of intersection. We present two other methods that produce it.

0. Given two triangles $T_1 = \triangle u_1u_2u_3$ and $T_2 = \triangle v_1v_2v_3$, determine whether they intersect.

13.10.1 Interval Overlap Method (introduced by Möller)

0. The plane equation $T_2: n_2 \cdot x + d_2 = 0$ is computed:

$$n_2 = (V_1 - V_0) \times (V_2 - V_0)$$

$$d_2 = -n_2 \cdot V_0$$

0. The signed distances of $u_i$'s from $T_2$ (multiplied by $||n_2||$)

$$d_{u_i} = n_2 \cdot u_i + d_2, \quad \text{for } i = 0, 1, 2.$$

0. If $\sum d_{u_i} < 0$, for all $i$, or $\sum d_{u_i} > 0$, for all $i$, $\Rightarrow$ No intersection.

These two early rejection tests save many computations.

0. If all $d_{u_i} = 0$, the triangles are coplanar. $\Rightarrow$ Reduced to 2D.

0. Otherwise, the intersection of $T_1$ and $T_2$ is a line

$$l = 0 + t \cdot d_1$$

where $d_1 = n_1 \times n_2$ and $0$ is some point on it.

0. Triangles $T_1$ and $T_2$ intersect $l$ in some intervals on $l$.

$T_1$ and $T_2$ intersect $\Leftrightarrow$ These intervals overlap.
Figure 13.17. Triangles and the planes in which they lie. Intersection intervals are marked in gray in both figures. Left: The intervals along the line $L$ overlap as well as the triangles. Right: There is no intersection; the intervals do not overlap.

Figure 13.18. The geometrical situation. Points $u_i$ are the vertices of $T_1$; $\pi_1$ and $\pi_2$ are the planes in which $T_1$ and $T_2$ lie; $d_{ui}$ are the signed distances from $u_i$ to $\pi_2$; $k_i$ are the projections of $u_i$ onto $\pi_2$; and $p_{ui}$ are the projections of $u_i$ onto $l$, which is the line of intersection.

Assume that $u_0$ and $u_2$ lie on the same side of $\pi_2$, and that $u_1$ lies on the other side. The vertices are projected onto $p_{u_2} = dl \cdot (u_2 - o)$ on the line $l$. The $t$-value at the intersection point is $t_1 = p_{u_2} + \left( p_{u_1} - p_{u_0} \right) \frac{du_2}{du_0 - du_1}$.

t_2$ is computed in a similar way. $[t_1, t_2]$ represents the intersection between $T_1$ and $l$. morning glory 🌸
Optimizations

0. Intervals can be translated without changing the result. Equation 13.26 can be simplified into

\[ P_{ui} = all \cdot u_i, \quad \text{for } i=0,1,2, \]

which means that we project onto a line passing through the origin. One can project onto a coordinate axis as well.

\[ p_{ui} = \begin{cases} u_{ix}, & \text{if } |d_x| = \max(|d_x|, |d_y|, |d_z|) \\ u_{iy}, & \text{if } |d_y| = \max(|d_x|, |d_y|, |d_z|) \\ u_{iz}, & \text{if } |d_z| = \max(|d_x|, |d_y|, |d_z|) \end{cases}, \quad i = 0,1,2. \]

Implementation

1. Compute the plane equation of Triangle 2.
2. Trivially reject if all points of Triangle 1 are on same side.
3. Compute the plane equation of Triangle 1.
4. Trivially reject if all points of Triangle 2 are on same side.
5. Compute intersection line and project onto largest axis.
6. Compute intervals for each triangle.
7. Intersect the intervals.

0. When computing the t-values in Equation 13.29, it is possible to compute the corresponding points of intersection.
0. Robustness problems arise when the triangles are nearly coplanar or when an edge is nearly co-planar to the other triangle (especially when the edge is close to an edge of the other triangle). Many techniques have been proposed for handling these cases. However, it is not an easy problem to deal with them in a reliable way.

morning glory 🌸
13.10.2 ERIT's Method

0. Held [351] presented ERIT (Efficient and Reliable Intersection Tests), which includes the triangle/triangle test outlined below:

1. Compute \( \pi_2: n_2 \cdot x + d_2 \), the plane in which \( T_2 \) lies.

2. Trivially reject if all points of \( T_1 \) are on the same side of \( \pi_2 \) (also store the signed distances, \( d_{u_i} \), as in the previous algorithm).

3. If the triangles are coplanar, use the coplanar triangle-triangle test used in the interval overlap method.

4. Compute the intersection between \( \pi_2 \) and \( T_1 \), which clearly is a line segment that is coplanar with \( \pi_2 \). This situation is illustrated in Figure 13.19.

5. If this line segment intersects or is totally contained in \( T_2 \), then \( T_1 \) and \( T_2 \) intersect; otherwise, they do not.

$$
T_2 \quad T_1
$$

Figure 13.19. This figure depicts the way in which the ERIT method determines whether two triangles intersect. The intersection points, \( p \) and \( q \), between triangle \( T_2 \) and triangle \( T_1 \) are computed. If the line between \( p \) and \( q \) is totally contained in \( T_2 \) or if it intersects the edges of \( T_2 \), then the triangles intersect. They are disjoint otherwise. (Illustration after Held [351].)

$$
p = u_{i_1} + \frac{d_{u_{i_1}}}{d_{u_{i_1}} - d_{u_{i_2}}} (u_{i_2} - u_{i_1}) \quad \text{and similarly for } q.
$$

0. Step 5 is realized by projecting \( T_2 \), \( p \), and \( q \) onto the xy, yz, or zx-plane where the area of \( T_2 \) is maximized. \( \Rightarrow \) Reduced to a 2D problem.

0. The interval overlap method and the ERIT method are comparable in speed. On a Pentium Pro, ERIT's method was faster.
13.11 Triangle/Box Overlap

0. We test on an AABB, defined by a center $c$ and a vector of half lengths $h$, against a triangle $\Delta u_0 u_1 u_2$. To simplify the test, we translate the vertices $v_i = u_i - c$, for $i = 0, 1, 2$.

![Diagram](image)

**Figure 13.20.** Notation used for the triangle-box overlap test. To the left the initial position of the box and the triangle is shown, while at the right, the box and the triangle has been translated so that the box center coincides with the origin.

0. Based on the SAT, we test the following:

1. [3 tests] $e_0 = (1, 0, 0)$, $e_1 = (0, 1, 0)$, $e_2 = (0, 0, 1)$ (the normals of the AABB). In other words, test the AABB against the minimal AABB around the triangle.

2. [1 test] $n$, the normal of $\Delta u_0 u_1 u_2$. We use a fast plane/AABB overlap test (see Section 13.9.1), which tests only the two vertices of the box diagonal whose direction is most closely aligned to the normal of the triangle.

3. [9 tests] $a_{ij} = e_i \times f_j$, $i, j \in \{0, 1, 2\}$, where $f_0 = v_1 - v_0$, $f_1 = v_2 - v_1$, and $f_2 = v_0 - v_2$, i.e., edge vectors. These tests are very similar and we will only show the derivation of the case where $i = 0$ and $j = 0$ (see below).

0. For the axis $a_0 = e_0 \times f_0 = (0, -f_{0z}, f_{0y}) = a$, we have

$$p_0 = a \cdot v_0 = (0, -f_{0z}, f_{0y}) \cdot v_0 = v_0 z v_1 y - v_0 y v_1 z,$$
$$p_1 = a \cdot v_1 = (0, -f_{0z}, f_{0y}) \cdot v_1 = v_0 z v_1 y - v_0 y v_1 z = p_0,$$
$$p_2 = a \cdot v_2 = (0, -f_{0z}, f_{0y}) \cdot v_2 = (v_1 y - v_0 y) v_2 z - (v_1 z - v_0 z) v_2 y.$$  

⇒ We compare $\min(p_0, p_2)$ and $\max(p_0, p_2)$ against $\pm r$.

$$r = h_1 |a_0| + h_2 |a_0| + h_3 |a_2| = h_1 |a_0| + h_2 |a_0|$$

⇒ Test: if $\min(p_0, p_2) > r$ or $\max(p_0, p_2) < -r$ return false;
13.12 BV/BV Intersection Tests

- BV provides simpler intersection tests and makes more efficient rejections.
- More complex BVs often have tighter fit.

![Diagram showing bounding volumes]

Figure 13.21. The efficiency of a bounding volume can be estimated by the "empty" volume; the more empty space, the worse the fit. A sphere (left), an AABB (middle left), an OBB (middle right), and a k-DOP (right) are shown for an object, where the OBB and the k-DOP clearly have less empty space than the others.

### 13.12.1 Sphere/Sphere Intersection

```cpp
bool Sphere_intersect(c1, r1, c2, r2)
returns({OVERLAP, DISJOINT});
1: l = c1 - c2
2: l^2 = l \cdot l
3: if(l^2 > (r1 + r2)^2) return (DISJOINT);
4: return (OVERLAP);
```

### 13.12.2 Sphere/Box Intersection

```cpp
bool SphereAABB_intersect(c, r, A)
returns({OVERLAP, DISJOINT});
1: d = 0
2: for each i \in \{x, y, z\}
3: if(c_i < a_i^{\text{min}})
4: d = d + (c_i - a_i^{\text{min}})^2;
5: else if(c_i > a_i^{\text{max}})
6: d = d + (c_i - a_i^{\text{max}})^2;
7: if(d > r^2)
8: return (DISJOINT);
9: return (OVERLAP);
```
13.12.3 AABB/AABB Intersection

```c
bool AABB_intersect(A, B)
returns({OVERLAP, DISJOINT});
1: for each i ∈ {x, y, z}
2: if(a_{i,min} > b_{i,max} or b_{i,min} > a_{i,max})
3: return (DISJOINT);
4: return (OVERLAP);
```

13.12.4 k-DOP/k-DOP Intersection

- **K-DOP** was named by Klosowski et al. \cite{433}.
- Kaye and Kajiya used this concept in ray tracing.
- AABB is a special case of a b-DOP.
- As k increases, the k-DOP increasingly resembles the convex hull, which is the tightest fitting convex BV.
- Computing the Minkowski sum of k-DOPs is easy. For k-DOPs at arbitrary positions, the Minkowski sum reduces the intersection test to a problem of testing the containment of a point with respect to a k-DOP.

```c
kDOP_intersect(d_{1,min}^{A}, ..., d_{k/2}^{A,min},
                 d_{1,max}^{A}, ..., d_{k/2}^{A,max},
                 d_{1,min}^{B}, ..., d_{k/2}^{B,min},
                 d_{1,max}^{B}, ..., d_{k/2}^{B,max})
returns({OVERLAP, DISJOINT});
1: for each i ∈ {1, ..., k/2}
2: if(d_{i,min}^{B} > d_{i,max}^{A} or d_{i,min}^{A} > d_{i,max}^{B})
3: return (DISJOINT);
4: return (OVERLAP);
```

Warning: The statement "According they might … be disjoint!" right above the pseudo code in page 601 is obviously wrong. In the case of k-DOPs, one needs to consider only the k/2 directions of the k-DOP.
Hierarchical view frustum culling is essential for rapid rendering of the scene graph. It is essential to support an efficient intersection test between the view frustum and a bounding volume.

Figure 13.24. The illustration on the left is an infinite pyramid, which is cropped by the parallel near and far planes in order to construct a view frustum. The names of the other planes are also shown, and the position of the camera is at the apex of the pyramid.

The exclusion/inclusion/intersection test determines whether the BV is outside/inside/intersecting the view frustum. Sometimes the intersection test is too costly to compute. In this case, the BV is classified as "probably-inside". The simplified algorithm is an exclusion/inclusion test.

Objects to be excluded can erroneously be included. Such mistakes cost extra time. On the other hand, objects to be included should never be excluded; otherwise rendering error will occur.

Figure 13.25 explains the exclusion/inclusion test using the concept of the Minkowski sum.
Figure 13.25. The upper left image shows a frustum (light gray) and a general bounding volume (dark gray), where a point $p$ relative to the object has been selected. By tracing the point $p$ where the object moves on the outside (upper right) and on the inside (lower left) of the frustum, as close as possible to the frustum, the frustum/BV can be reformulated into testing the point $p$ against an outer and an inner volume. This is shown on the lower right. If the point $p$ is outside the dark gray volume, then the BV is outside the frustum. The BV intersects the frustum if $p$ is inside the dark gray area, and the BV is inside the frustum if $p$ is inside the light gray area.

13.13.1 Frustum/Sphere Intersection

Figure 13.26 shows a frustum/BV test. For the sake of efficiency, we may use the approximation as in the rightmost of Figure 13.26.

Figure 13.26. At the left, a frustum and a sphere are shown. The exact frustum/sphere test can be formulated as testing $p$ against the dark and light gray volumes in the middle figure. At the right is a reasonable approximation of the volumes in the middle. If the center of the sphere is located outside a rounded corner, but inside all outer planes, then it will be incorrectly classified as intersecting even though it is outside the frustum.
Most frustums are symmetric around the direction.

To reduce the number of planes to be tested, an octant test can be added. We need to test against only three planes instead of six planes. (This approach is more useful for non-spherical BVs since the sphere/plane test is already so fast and there is no improvement.)

Bishop et al. [65] suggest the following optimizations. If a BV is found to be fully inside a frustum plane, its children are also inside the plane. Thus the plane test can be omitted for all children.

**Figure 13.27.** The left figure shows how a three-dimensional frustum is divided into eight octants. The other figure shows a two-dimensional example, where the sphere center $c$ is in the upper right octant. The only planes that then need to be considered are the right and the far planes (thick black lines). Note that the small sphere is in the same octant as the large sphere. This technique also works for arbitrary objects, but the following condition must hold: The minimum distance, called $d$, from the frustum center to the frustum planes must be larger than the radius of a tight bounding sphere around the object. If that condition is fulfilled, then the bounding sphere can be used to find the octant.

### 13.13.2 Frustum/Cylinder Intersection

The cylinder is treated as a line segment, and the frustum is increased in size by the sphere's radius.

A better solution is to take the Minkowski sum of the frustum and the circular disk of the cylinder. The frustum planes are increased by different radii.
13.3.3 Frustum/Box Intersection

0. We test the bounding box against each plane of the frustum. Instead of checking all six corners of the bounding box, only two extreme corners are needed.

0. If the box is outside one plane, the box is outside the frustum. If the box is inside all six planes, then the box is inside the frustum. Otherwise, it is considered as intersecting the frustum (even though it might actually be slightly outside). See Fig 13.28.

```cpp
bool FrustumAABBIntersect(\(\pi^0, \ldots, \pi^5\), B)
returns({OUTSIDE, INSIDE, INTERSECTING});
1: intersecting = false;
2: for k = 0 to 5
3:   for each i \(\in\) \{x, y, z\}
4:     if (\(n^k_i \geq 0\))
5:       \(v^\text{min}_i = b^\text{min}_i\)
6:       \(v^\text{max}_i = b^\text{max}_i\)
7:     else
8:       \(v^\text{min}_i = b^\text{max}_i\)
9:       \(v^\text{max}_i = b^\text{min}_i\)
10:   if ((\(n^k \cdot v^\text{min} + d^k\) > 0) return OUTSIDE;
11:   if ((\(n \cdot v^\text{max} + d\) >= 0) intersecting = true;
12:   if (intersecting == true) return INTERSECTING;
13: else return INSIDE;
```

Figure 13.28. The bold black lines are the planes of the frustum. When testing the box (left) against the frustum using the presented algorithm, the box can be incorrectly classified as intersecting when it is outside. For the situation in the figure, this happens when the box's center is located in the dark gray areas.
13.15 Line/Line Intersection Tests

13.15.1 Two Dimensions

First Method

1. Two lines: \( \mathbf{r}_1(s) = \mathbf{o}_1 + s \mathbf{d}_1 \) and \( \mathbf{r}_2(t) = \mathbf{o}_2 + t \mathbf{d}_2 \).

Let \( a^\perp = (-a_y, a_x) \), where \( a = (a_x, a_y) \).

<table>
<thead>
<tr>
<th>Step</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \mathbf{r}_1(s) = \mathbf{r}_2(t) )</td>
</tr>
<tr>
<td>2</td>
<td>( \mathbf{o}_1 + s \mathbf{d}_1 = \mathbf{o}_2 + t \mathbf{d}_2 )</td>
</tr>
</tbody>
</table>
| 3    | \( \begin{cases} 
    s \mathbf{d}_1 \cdot \mathbf{d}_2^\perp = (\mathbf{o}_2 - \mathbf{o}_1) \cdot \mathbf{d}_2^\perp \\
    t \mathbf{d}_2 \cdot \mathbf{d}_1^\perp = (\mathbf{o}_1 - \mathbf{o}_2) \cdot \mathbf{d}_1^\perp 
\end{cases} \) |
| 4    | \( \begin{cases} 
    s = \frac{(\mathbf{o}_2 - \mathbf{o}_1) \cdot \mathbf{d}_2^\perp}{\mathbf{d}_1 \cdot \mathbf{d}_2^\perp} \\
    t = \frac{(\mathbf{o}_1 - \mathbf{o}_2) \cdot \mathbf{d}_1^\perp}{\mathbf{d}_2 \cdot \mathbf{d}_1^\perp} 
\end{cases} \) |

0. The above is essentially the result of applying Cramer's rule to \( [\mathbf{d}_1 - \mathbf{d}_2] \begin{bmatrix} s \\ t \end{bmatrix} = [\mathbf{o}_2 - \mathbf{o}_1] \). 

0. If \( \mathbf{d}_1 \cdot \mathbf{d}_2^\perp = 0 \), the lines are parallel and no intersection occurs.

0. For lines of infinite length, all values of \( s \) and \( t \) are valid.

0. For line segments

\( \mathbf{r}_1(s) = \mathbf{p}_1 + s(\mathbf{p}_2 - \mathbf{p}_1) \) connecting \( \mathbf{p}_1 \) and \( \mathbf{p}_2 \), and

\( \mathbf{r}_2(t) = \mathbf{q}_1 + t(\mathbf{q}_2 - \mathbf{q}_1) \) connecting \( \mathbf{q}_1 \) and \( \mathbf{q}_2 \),

a valid intersection occurs if and only if \( 0 \leq s, t \leq 1 \).

0. For rays with origins, the valid range is \( s \geq 0 \) and \( t \geq 0 \).

0. The point of intersection is obtained by plugging \( s \) into \( \mathbf{r}_1 \) (or by plugging \( t \) into \( \mathbf{r}_2 \)).
Second Method

0. Antonio [21] describes another method that does more compares and early rejections and thus avoids the expensive calculations (divisions).

0. A solution to \( \mathbf{r}_1(s) = \mathbf{r}_2(t) \) is given as follows

\[
\begin{align*}
  s &= \frac{-c \cdot a^\perp}{b \cdot a^\perp} = \frac{c \cdot a^\perp}{a \cdot b^\perp} = \frac{d}{f} \\
  t &= \frac{c \cdot b^\perp}{a \cdot b^\perp} = \frac{e}{f}.
\end{align*}
\]

where \( a = q_2 - q_1, \ b = p_2 - p_1, \ c = p_1 - q_1, \ d = c \cdot a^\perp, \ e = c \cdot b^\perp, \ f = a \cdot b^\perp. \)

Note that \( a^\perp \cdot b = -a \cdot b^\perp. \)

0. The denominators for both \( s \) and \( t \) are the same.

0. To test if \( 0 \leq s \leq 1 \), the following code is used.

```python
1: if(f > 0)  
2:     if(d < 0 or d > f) return NO_INTERSECTION;  
3:     else  
4:         if(d > 0 or d < f) return NO_INTERSECTION;
```

The same can be done for \( t = \frac{e}{f} \).

13.15.2 Three Dimensions

0. \( 0 \leq s \leq d_1 = \alpha_2 + t d_2 \)

\[\begin{align*}
  \{ & <d_1, d_1> S - <d_1, d_2> t = <\alpha_2 - \alpha_1, d_1> \\
  & -<d_1, d_2> S + <d_2, d_2> t = <\alpha_1 - \alpha_2, d_2> \}
\]

This equation can be solved by Cramer's rule.

0. If the lines are skew, then the \( s \) and \( t \) parameters represent the points of minimum distance.

This formula is simpler than the one in the book!
13.17 Dynamic Intersection Testing

0. A dynamic intersection test considers moving objects. To simplify calculations, we consider the relative motion between two objects. Assume object A moves with velocity $v_A$ and object B with velocity $v_B$. Then A moves with velocity $v = v_A - v_B$ with respect to the motion of B.

13.17.1 Sphere/Plane

0. Let $C(t)$ denote the center of a sphere of radius $r$ at time $t$, and $\Pi_r: n \cdot x + d_r = 0$ denote the offset of a plane by distance $r$. Then the intersection of sphere/plane is characterized by the solution of $n \cdot C(t) + d_r = 0$.

0. The material in the textbook is a special case of this.

13.17.2 Sphere/Sphere

0. Let $S_A$ denote a sphere of radius $r_A$ with its center moving along a path $C_A(t)$, and $S_B$ with radius $r_B$ with its center moving along a path $C_B(s)$. Note that the motions are independent. They can be made dependent by setting $S = f(t)$, for example, $S = t$.

0. The sphere/sphere intersection is then reduced to testing the inclusion of $(C_A(t) - C_B(s))$ in the region $x^2 + y^2 + z^2 < (r_A + r_B)^2$.

0. Equivalently to testing $< (C_A(t) - C_B(s), C_A(t) - C_B(s)) > - (r_A + r_B)^2 < 0$.

0. The material in the textbook is a special case of this.
the trajectory of center \( c(t) \) is contained in the offset of a polygon whose boundary consists of polygons, cylinders, and spheres.

Figure 13.33. In the left figure, a sphere moves towards a polygon. In the right figure, a ray shoots at an “inflated” version of the polygon. The two intersection tests are equivalent.

Other Bounding Volumes

0. For Line Swept Spheres (LSS, also called capsules) and Rectangle Swept Spheres (RSS, also called lozenges), it is easy to compute the minimum distance.

Figure 13.35. A Line Swept Sphere (LSS) and Rectangle Swept Sphere (RSS).