# An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

# Chapter 17 Intersection Testing

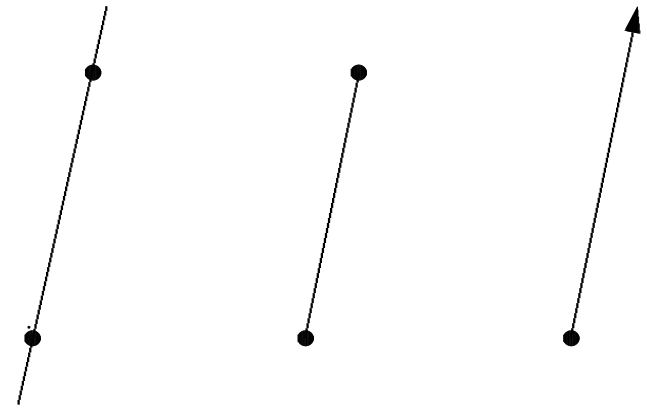
This chapter discusses the methods of determining the intersection between a ray and a surface or an object. The technique also applies to finding collisions between objects in game development.

## 17.1 Rays, Lines, and Planes

### 17.1.1 Representing Lines and Planes

For clarity of presentation, we first distinguish between a line, a ray, and a line segment. Figure 17-1 shows their characteristic, which can be described as follows:

1. A **line** is defined by two points. It is infinite in length and passes through the points, extending forever in both directions.
2. A **line segment** (**segment** for short) is defined by two end points and its length is finite.
3. A **ray** is defined by a point and a direction. It is semi-infinite, extending to infinity in one direction.



(a) Line (b) Line Segment (c) Ray

Figure 17-1 Line, Line Segment, and Ray

So we can regard a line, a segment, or a ray as a set of points. Recall that the difference between two points is a vector. We can simply define them by two points, $P_1$ and $P_2$. We let $\mathbf{D} = P_2 - P_1$ be the vector pointing from $P_1$ to $P_2$. Then all of them can be represented by a single parametric function $L(t)$,

$$L(t) = P_1 + \mathbf{D}t \tag{17.1}$$

with different restrictions on the parameter $t$:

$$\begin{array}{ll} \text{Segment:} & 0 \leq t \leq 1 \\ \text{Ray:} & 0 \leq t < \infty \\ \text{Line:} & -\infty < t < \infty \end{array} \tag{17.2}$$

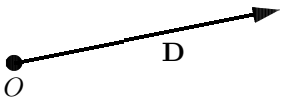
The point $P_1$ is often called the starting point or the origin of the ray. Using the classes *Vector3*, and *Point3* discussed in previous chapters, we can easily implement a ray as a class like the following:

```
class Ray {
public:
 Point3  O;     //starting point (origin)
 Vector3 D;     //direction vector
 Ray()          //default constructor
 {
   O = Point3( 0.0, 0.0, 0.0 );
   D = Vector3( 1.0, 1.0, 1.0 );    D.normalize();
 }
 //constructor
 Ray ( const Point3 &origin, const Vector3 &dir ){
   O = origin;
   D = dir;    D.normalize();
 }

 //return point L(t) at t
 Point3 getPoint ( const double t )
 {
   return O +  t * D;
 }
};
```

## 17.1.2 Ray-Plane Intersection

As discussed above, we can specify a ray by its starting position $O$ and a unit vector $\mathbf{D}$. We can consider a ray as a set of points given by

$$L(t) = O + t\mathbf{D} \qquad 0 \le t < \infty \qquad (17.3)$$




as shown in the figure on the right. One can see that a ray is actually part of a line. The line that contains the ray is referred to as a **ray-line**.

A **plane** is specified by a point $P' = (x', y', z')$ on the plane and a normal $\mathbf{n} = (n_x, n_y, n_z)$ perpendicular to it. It can be described by an equation in the form,

$$(P - P') \cdot \mathbf{n} = 0$$

impling that

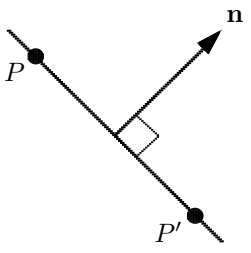
$$(x - x')n_x + (y - y')n_y + (z - z')n_z = 0$$

or in the form

$$ax + by + cz + d = 0 \qquad (17.4)$$

with

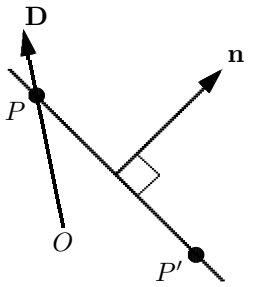
$$(a, b, c) = (n_x, n_y, n_z),$$
$$d = -(x'n_x + y'n_y + z'n_z)$$




Conversely, if a plane is described by an equation of (17.4), its normal is given by $\mathbf{n} = (a, b, c)$.

Suppose $P$ is the intersection point of the ray $L(t)$ and the plane $(P - P') \cdot \mathbf{n} = 0$ as shown below. As $P$ is on both the ray and the plane, we have

$$0 = (P - P') \cdot \mathbf{n} = (O + t\mathbf{D} - P') \cdot \mathbf{n}$$




Soving for $t$, we get

$$t = \frac{(P' - O) \cdot \mathbf{n}}{\mathbf{D} \cdot \mathbf{n}} \qquad (17.5)$$

There are three cases that imply the ray and plane do not intersect:

1. If $\mathbf{D} \cdot \mathbf{n} = 0$, the ray $L$ that contains $P$ is parallel to the plane (perpendicular to $\mathbf{n}$). So they don't intersect.
2. If $\mathbf{D} \cdot \mathbf{n} < 0$, the ray is downward relative to the plane. So they don't intersect.
3. if $t < 0$, the ray $L$ does not intersect the plane.

**Example 17.1**:

A plane passes through the points $P_1 = (3, 4, 0)$, $P_2 = (4, 4, 0)$, and $P_3 = (5, 3, -1)$. Determine whether the ray that originates from $O = (2, 1, 0)$ and passes through the point $P_0 = (1, 3, 0)$ will intersect with the plane. If yes, what is the intersection point?

**Solution**:

Here

$$\mathbf{D} = (P_0 - O) = (1, 3, 0) - (2, 1, 0) = (-1, 2, 0)$$

Let

$$\mathbf{A} = P_2 - P_1 = (4, 4, 0) - (3, 4, 0) = (1, 0, 0)$$
$$\mathbf{B} = P_3 - P_1 = (5, 3, -1) - (3, 4, 0) = (2, -1, -1)$$

Then

$$\mathbf{n} = \mathbf{A} \times \mathbf{B} = (0, 1, -1)$$
$$\mathbf{D} \cdot \mathbf{n} = (1, 2, 0) \cdot (0, 1, -1) = 2$$

So

$$t = \frac{(P_1 - O) \cdot \mathbf{n}}{\mathbf{D} \cdot \mathbf{n}}$$
$$= \frac{((3, 4, 0) - (2, 1, 0)) \cdot (0, 1, -1)}{2}$$
$$= \frac{(1, 3, 0) \cdot (0, 1, -1)}{2}$$
$$= 3/2$$

Thus, the ray intersect the plane at the point

$$P(t) = P(3/2) = (2, 1, 0) + (3/2)(-1, 2, 0) = \boxed{(1/2, 4, 0)}$$

The implementation of a plane is simple. We implement it as a class by specifying a point on it along with its normalized normal like the following:

```
class Plane {
public:
  Point3 p0;          //one point and a normal define a plane
  Vector3  n;
  Plane (){           //default is xy plane
    p0 = Point3( 0.0, 0.0, 0.0 );
    n = Vector3 ( 0, 0, 1 ) ;
  }
  Plane ( const Point3 &aPoint,  const Vector3 & aNormal )
  {
    p0 = aPoint;
    n = aNormal;
    if ( n.magnitude() == 0 )   //special case
      n = Vector3 ( 0, 0, 1 );
    else
      n.normalize();            //make it a unit vector
  }
};
```

We implement the testing of the intersection of a ray and a plane as the function **intersect_ray_plan**() presented below. The function returns **true** and the value of $t$ if they intersect and $false$ if they do not. The calculation of the parameter $t$ is a straightforward implementation of Equation (17.5).

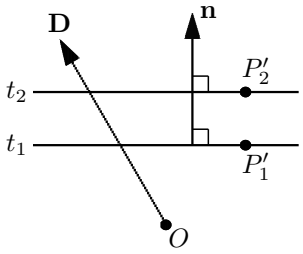
```
bool intersect_ray_plane(const Ray &ray, const Plane &plane,
                                                    double &t)
{
  t = 0;
  if ( ray.O == plane.p0 )
    return true;

  //vector from origin to p0
  Vector3 origin_to_p0 = plane.p0 - ray.O;
  double op_dot_n = origin_to_p0 * plane.n;//dot product
  double d_dot_n = ray.D * plane.n;        //dot product

  if (fabs(d_dot_n) < 0.0000001)   //ray parallel to plane
    return false;

  t =  op_dot_n / d_dot_n;
  if ( t < 0.0 )
    return false;

  return true;
}
```

### 17.1.3 Ray-Slab Intersection

A slab consists of two parallel planes as shown in the figure on the right below. Therefore, it is specified by two points and a normal. If intersection occurs, the ray intersects the slab

at two points which are specified by two values of the parameter values, $t_1$ and $t_2$.

$$L(t) = O + t\mathbf{D}$$




At intersection points, we have

$$t_1 = \frac{(P_1' - O) \cdot \mathbf{n}}{\mathbf{D} \cdot \mathbf{n}} \qquad (17.6a)$$

$$t_2 = \frac{(P_2' - O) \cdot \mathbf{n}}{\mathbf{D} \cdot \mathbf{n}} \qquad (17.6b)$$

## 17.1.4 Ray-Box Intersection

A box is bounded by six planes, which can be used both as an object and a bounding volume. We only consider the special case of boxes that have parallel faces with normals parallel to the coordinate axes; the box faces are contained in a set of slabs. In this case, the intersection of the slabs defines the box. We specify the box by two points, the minimum extent at $p_l = (x_l, y_l, z_l)$, and the maximum extent at $p_h = (x_h, y_h, z_h)$. This is shown in Figure 17-2 below.

Ray : $L(t) = O + t\mathbf{D}$
Box : minimum extent $p_l = (x_l, y_l, z_l)$
maximum extent $p_h = (x_h, y_h, z_h)$

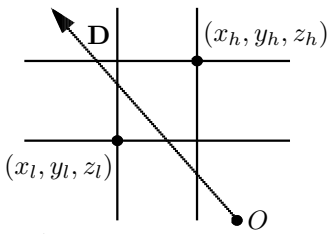
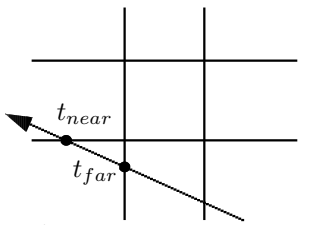
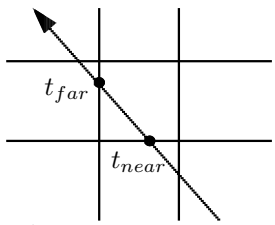



**Figure 17-2** Ray-Box Intersection

To determine whether a ray intersects a box, we examine the intersection of each pair of slabs by the ray, calculating $t_{far}$, the intersection value of $t$ at the far-side slab, and $t_{near}$, value at the near-side. If the largest overall $t_{near}$ value is larger than the smallest $t_{far}$ value then the ray misses the box, otherwise it hits the box as shown in Figure 17-3 below.




(a) min ( $t_{far}$ ) < max ( $t_{near}$ )
So ray misses box.




(b) min ( $t_{far}$ ) > max ( $t_{near}$ )
So ray hits box.

**Figure 17-3** Determining Ray-Box Intersection

Also, if the ray is parallel to an axis, we can simplify the process by first checking whether the origin of the ray lies between the corresponding parallel planes. If not, the ray will not hit the box.

Suppose we call planes parallel to $yz$-plane the $X$-planes (normal along $x$-axis), planes parallel to $zx$-plane the $Y$-planes (normal along $y$-axis), planes parallel to $xy$-plane the

$Z$-planes (normal along $z$-axis). Using Equations (17.6), we present a commonly used algorithm for determining whether the ray $L(t) = O + t\mathbf{D}$ intersects with the box discussed above:

1. Set $t_{near} = -\infty$, $t_{far} = +\infty$
2. For the pair of $X$-planes, do the following

   2.1 if ( $\mathbf{D}_x = 0$ ), the ray is parallel to the $X$-planes (normal $\mathbf{n} = \mathbf{x}$), so
   if ( $O_x < x_l$ or $O_x > x_h$ ) return **false** (origin not between planes)

   2.2 else the ray is not parallel to the planes, so calculate the ray parameter $t$ values at the intersections:
   - $t_1 = \dfrac{x_l - O_x}{\mathbf{D}_x}$ (point at which ray intersects minimum $X$-plane)
   - $t_2 = \dfrac{x_h - O_x}{\mathbf{D}_x}$ (point at which ray intersects maximum $X$-plane)
   - if ( $t_1 > t_2$ ) swap $t_1$ and $t_2$, so that $t_1$ is always at the near-plane and $t_2$ is at the far-plane
   - if ( $t_1 > t_{near}$) set $t_{near} = t_1$ (find largest $t_{near}$)
   - if ( $t_2 < t_{far}$) set $t_{far} = t_2$ (find smallest $t_{far}$)
   - if ( $t_{near} > t_{far}$ ) ray misses box so return **false**
   - if ( $t_{far} < 0$ ) box is behind ray so return **false**
3. Repeat step 2 for $Y$-planes and $Z$-planes.
4. If all tests were survived, return **true** with $t_{near}$ as parameter value at intersection point and $t_{far}$ at exit point.

We implement such a box as a class by specifying the minimum and maximum extents:

```
class Box {
public:
  Point3 pl;            //minimum extent
  Point3 ph;            //maximum extent
  Box (){ //default box center at origin
    pl = Point3( -1, -1, -1 );
    ph = Point3( 1, 1, 1 );
  }
  Box ( const Point3 &p1,  const Point3 &p2 )
  {
    assert(p1.x < p2.x && p1.y < p2.y && p1.z < p2.z);
    pl = p1;
    ph = p2;
  }
};
```

We could implement the intersection testing of a box and a ray with use of the **intersect_ray_plane**() function discussed above. However, it is a lot more efficient to implement directly using the ray-box intersection algorithm presented above. We present this implementation below, where the function **intersect_ray_box**() determines whether a ray intersects a box; if yes it returns true and ray parameters $t1$ and $t2$ at the intersection and exit points respectively:

```
bool intersect_ray_box(Ray &ray,Box &box,double &t1,double &t2)
{
  double t_near = -1.0e+10, t_far = 1.0e+10;
  double D[3];  //ray direction
  double O[3];  //ray origin
  double pl[3]; //minimum extent
  double ph[3]; //maximum extent
  ray.D.getXYZ ( D );   //put ray direction (x,y,z) in D[]
  ray.O.getXYZ ( O );   //put ray origin (x,y,z) in O[]
  box.pl.getXYZ ( pl ); //put (x,y,z) of minimum point in pl[]
  box.ph.getXYZ ( ph ); //put (x,y,z) of maximum point in ph[]

  for ( int i = 0; i < 3; i++ ) {  //test 3 planes (X, Y, Z)
    if ( D[i] == 0 ) {  //ray parallel to axis plane
      if ( O[i] < pl[i] || O[i] > ph[i] )
        return false;   //origin not between points
    } else {
       t1 = ( pl[i] - O[i] ) / D[i];
       t2 = ( ph[i] - O[i] ) / D[i];
       if ( t1 > t2 ) {  //swap t1, t2
         double temp = t1;
         t1 = t2;
         t2 = temp;
       }
       if ( t1 > t_near ) t_near = t1;   //find max (t_near)
       if ( t2 < t_far  ) t_far  = t2;   //find min (t_far)
       if (t_near > t_far) return false;
       if (t_far < 0 ) return false;
    }
  } //for
  t1 = t_near;   //returns maximum t_near (intersection)
  t2 = t_far;    //returns minimum t_far  (exit)

  return true;   //ray hits box
}
```

## 17.2 Ray-Triangle Intersection

### 17.2.1 Ray-Triangle Intersection Test

Since any surface can be approximated by triangles, ray-triangle intersection algorithms are very important in ray-tracing or game development.

A triangle is a subset of a plane; it is the region of the plane bounded by the three edges of the triangle which is defined by three vertices. Therefore, we can break up the ray-triangle intersection testing into two stages:

1. Test whether the ray intersects the plane that contains the triangle.

2. If ray-plane intersection occurs, check whether the intersection point is inside the triangle.

Recall that a plane is specified by a point $p'$ lying on the plane and a normal $\mathbf{n}$ of the plane. That is,

$$\text{Plane } = (p', \mathbf{n}) \tag{17.7}$$

Suppose a triangle is defined by three vertices, $(p_0, p_1, p_2)$ and is contained in a plane as shown in Figure 17-4 below. We refer to this plane as triangle-plane.

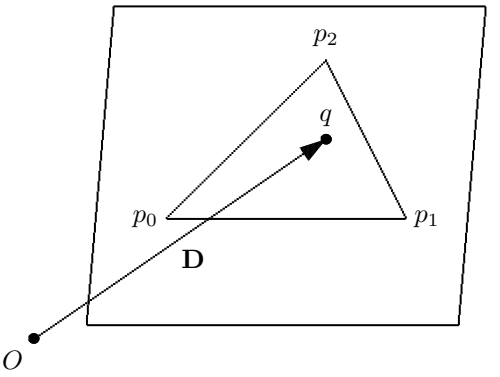



**Figure 17-4** Ray-Triangle Intersection

As the difference between two points is a vector, we can compute a unit normal $\mathbf{n}$ to the plane by the cross product of two vectors along two triangle edges originated from the same vertex:

$$\begin{aligned} \mathbf{N} &= (p_1 - p_0) \times (p_2 - p_0) \\ \mathbf{n} &= \tfrac{\mathbf{N}}{|\mathbf{N}|} \end{aligned} \tag{17.8}$$

Since $P_0$ is a point of the triangle, it is also a point of the plane containing the triangle. Therefore, the triangle-plane is specified by

$$\text{Tri-plane } = (p_0, \mathbf{n}) \tag{17.9}$$

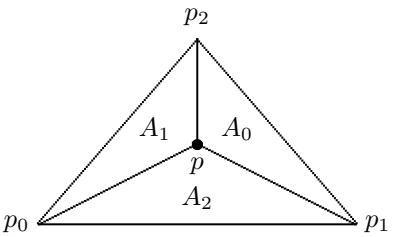
We can always substitute $p_0$ by $p_1$ or $p_2$ in (17.9) as all three points are on the plane. The intersection point $q$ is determined by the ray-plane intersection test, which calculates the ray parameter $t(q)$ at $q$ if they intersect:

$$t(q) = \frac{(P_0 - O) \cdot \mathbf{n}}{\mathbf{D} \cdot \mathbf{n}} \tag{17.10}$$

Once we have found $q$, we have to check whether it lies inside the triangle. (Of course, if the ray does not hit the plane, it does not hit the triangle either, and we do not need to make further tests.) This test is easier to be done using Barycentric coordinates.

We have learned that a point can be expressed as an affine combination of other points. Therefore, a point $p$ on a triangle-plane can be expressed as

$$p = A_0 p_0 + A_1 p_1 + A_2 p_2 \quad (17.11)$$




where $A_0, A_1, A_2$ are proportional to the areas of the corresponding subtriangles as shown in the figure above and

$$A_0 + A_1 + A_2 = 1 \quad (17.12)$$

The values $(A_0, A_1, A_2)$ are referred to as **Barycentric coordinates** of $p$. If the area of the triangle is normalized to 1, then $A_i$ $(i = 0, 1, 2)$ becomes the area of the actual subtriangle, and the coordinates are called **homogeneous Barycentric coordinates**. The criterion for $p$ to be inside the triangle is:

$$0 \leq A_i \leq 1 \quad i = 0, 1, 2 \quad (17.13)$$

Therefore, if we know the Barycentric coordinates of the hit point $q$, we can accept or reject the ray with very simple tests: Point $q$ is outside the triangle if one of the Barycentric coordinates $A_i$ is smaller than zero. So our task is to find the Barycentric coordinates of $q$.

Suppose vertices $v_0, v_1$, and $v_2$ form a triangle and $(\alpha, \beta, \gamma)$ are Barycentric coordinates of a point $q$ on the triangle-plane. Then

$$q = \alpha v_0 + \beta v_1 + \gamma v_2 \quad (17.14)$$

Substituting with $\alpha = 1 - \beta - \gamma$, we can rewrite (17.14) as

$$\beta(v_1 - v_0) + \gamma(v_2 - v_0) = q - v_0 \quad (17.15)$$

It is easier to solve for the Barycentric coordinates of (17.15) if we project the triangle in a 2D plane. Projecting both the triangle and the hit-point $q$ onto another plane does not change the Barycentric coordinates of $q$ because the projection does not change the ratios of the subtriangle areas. After projection, all computations can be performed more efficiently in 2D. For example, we can project them into the $xy$-plane ($Z$-plane). We first define three vectors:

$$\begin{aligned} \mathbf{c} &= (v_1 - v_0) \\ \mathbf{b} &= (v_2 - v_0) \\ \mathbf{h} &= (q - v_0) \end{aligned} \quad (17.16)$$

We can then express (17.15) as

$$\beta \mathbf{c} + \gamma \mathbf{b} = \mathbf{h} \quad (17.17)$$

When projected into the $Z$-plane, we obtain the following equations:

$$\begin{aligned} \beta \mathbf{c}_x + \gamma \mathbf{b}_x &= \mathbf{h}_x \\ \beta \mathbf{c}_y + \gamma \mathbf{b}_y &= \mathbf{h}_y \end{aligned} \quad (17.18)$$

So we have two equations and two unknowns $(\beta, \gamma)$, which can be easily solved. Upon solving (17.18), we obtain

$$\boxed{\beta = \frac{\mathbf{b}_x \mathbf{h}_y - \mathbf{b}_y \mathbf{h}_x}{\mathbf{b}_x \mathbf{c}_y - \mathbf{b}_y \mathbf{c}_x}} \qquad \boxed{\gamma = \frac{\mathbf{h}_x \mathbf{c}_y - \mathbf{h}_y \mathbf{c}_x}{\mathbf{b}_x \mathbf{c}_y - \mathbf{b}_y \mathbf{c}_x}} \quad (17.19)$$

If either $\beta < 0$, or $\gamma < 0$, or $\alpha = 1 - \beta - \gamma < 0$, then the intersection point $q$ is outside the triangle. Otherwise it is inside.

If the triangle-plane is almost parallel to the $Z$-plane (i.e. $\mathbf{n}_z \approx 0$), then the denominator in (17.18) is close to zero and could cause significant rounding errors in the calculation. In this case, we should project it into the $X$-plane or $Y$-plane. In other words, we should first calculate the normal components $(\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z)$ and project into the axis plane associated with the largest normal component. For example if $|\mathbf{n}_y|$ is largest, we should project the triangle into the $Y$-plane.

### 17.2.2 Triangle Implementation

As a triangle can be specified by three points, we implement it as the class *Triangle* that has three *Point3* members. The following code segment shows the implementation.

```
class Triangle {
public:
  Point3 p0;    //three points determine a triangle
  Point3 p1;
  Point3 p2;

  Triangle ()   //default Triangle
  {
    p0 = Point3( 0.0, 0.0, 0.0 );
    p1 = Point3( 1.0, 0.0, 0.0 );
    p2 = Point3( 0.0, 1.0, 0.0 );
  }
  Triangle(const Point3 &v0, const Point3 &v1,
                                  const Point3 &v2)
  {
    p0 = v0;  p1 = v0;  p2 = v2;
  }
};
```

### 17.2.3 Ray-Triangle Intersection Test Implementation

Implementation of the ray-triangle intersection test consists of a few steps:

1. Construct a plane containing the triangle.
2. Perform the ray-plane intersection test. If the ray misses the plane, return **false**. Otherwise test whether the ray-plane intersection point is inside the triangle according to steps 3-6.
3. Construct the three vectors of $\mathbf{b}, \mathbf{c}, \mathbf{h}$ of (17.16).
4. Find the axis that the normal to the triangle has the largest absolute component and choose this axis-plane to be the plane of projection.
5. Project the three vectors into the axis-plane and calculate the Barycentric coordinates of the ray-plane intersection using formulas (17.18) and (17.19).
6. If any of the Barycentric coordinates is smaller than $0$, return $\mathbf{false}$. Otherwise return $\mathbf{true}$.

The function **intersect_ray_triangle**() of Listing 17-1 below shows the actual implementation:

**Program Listing 17-1** Ray-Triangle Intersection Test

```
--------------------------------------------------------------------------------
//assume that x, y, z are positive
char max_axis ( float x, float y, float z )
{
  char a;
  if ( x  > y )
    if ( x > z )
       a = 'x';
    else
       a = 'z';
  else
    if ( y > z )
       a = 'y';
    else
       a = 'z';

  return a;
}

bool intersect_ray_triangle(Ray &ray, const Triangle &triangle,
                                                      double &t )
{
  Vector3 v01 = triangle.p1 - triangle.p0;
  Vector3 v02 = triangle.p2 - triangle.p0;
  Vector3 normal = v01 ^ v02;       //normal to triangle
  Plane plane(triangle.p0, normal); //plane containing triangle
  bool hit_plane = intersect_ray_plane ( ray, plane, t );
  if ( !hit_plane ) return false;

  Point3 q = ray.getPoint( t );     //ray-plane intersection

  //check whether q is inside triangle
  double bx, by, cx, cy, hx, hy;
  double beta, gamma;
  Vector3 b, c, h;

  c = v01;
  b = v02;
  h = q - triangle.p0;

  //find the dominant axis
  char axis = max_axis ( fabs ( normal.x), fabs ( normal.y), fabs(normal.z) );

  switch ( axis ) {
    case 'x': //project on X-plane (yz)
      bx = b.y; by = b.z;
      cx = c.y; cy = c.z;
      hx = h.y; hy = h.z;
      break;
    case 'y': //project on Y-plane (zx)
      bx = b.z; by = b.x;
      cx = c.z; cy = c.x;
      hx = h.z; hy = h.x;
      break;
    case 'z': //project on Z-plane (xy)
```

```
      bx = b.x; by = b.y;
      cx = c.x; cy = c.y;
      hx = h.x; hy = h.y;
      break;
  }

  double denominator =  bx * cy - by * cx;
  beta = ( bx * hy - by * hx ) / denominator;
  gamma = ( hx * cy - hy * cx ) / denominator;

  if ( beta < 0 ) return false;
  if ( gamma < 0 ) return false;
  if ( 1 - (beta + gamma) < 0 ) return false;

  return true;
}
```

---

## 17.3 Ray-Sphere Intersection

### 17.3.1 Ray-Sphere Intersection Test

A sphere is specified by the location of its center $C$ and its radius $R$. If $P$ is a point on the sphere, we can express the sphere as

$$(P - C)^2 - R^2 = 0 \tag{17.20}$$

A ray is described by (17.3) (i.e., $P(t) = O + t\mathbf{D}$). If $P$ is an intersection point of the ray and the sphere as shown in Figure 17-5 below, it satisfies both (17.20) and (17.3). Then

$$(O + t\mathbf{D} - C)^2 - R^2 = 0 \tag{17.21}$$

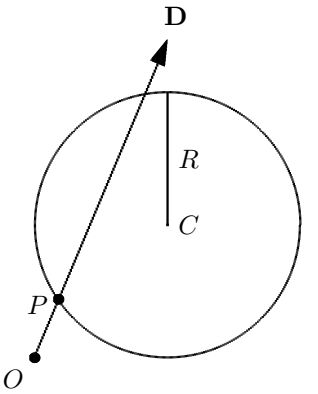



**Figure 17-5** Ray-Sphere Intersection

Equation (17.21) can be expanded and expressed in the form,

$$at^2 + bt + c = 0 \tag{17.22}$$

where

$$\begin{aligned} a &= \mathbf{D}^2 \\ b &= 2(O - C) \cdot \mathbf{D} \\ c &= (O - C)^2 - R^2 \end{aligned}$$

Equation (17.22) is a quadratic equation in $t$, and we can easily solve for $t$. Its solution is given by

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{17.23}$$

There are three different cases for the solutions of (17.23) depending on the discriminant $b^2 - 4ac$:

| | | |
|---|---|---|
| (a) | $b^2 - 4ac < 0$ | no solution, ray misses sphere |
| (b) | $b^2 - 4ac = 0$ | one solution, ray may touch sphere at one intersection point |
| (c) | $b^2 - 4ac > 0$ | two solutions, ray may intersect sphere at two points |

These three cases are shown in Figure 17-6 below. If the ray 'intersects' the sphere at two points, then the point with smaller $t$ value is nearer and is the real intersection point. However, if the larger $t$ value is negative, the ray points away from the sphere implying that the ray misses the sphere. Another situation is that the ray origin is inside the sphere. In this case, one root is positive and the other is negative; the positive root (larger $t$ value) gives the real intersection point.

For the case of (b), if the $t$ value is positive, the ray touches the sphere. If it is negative, it points away from the sphere and thus misses it (if we extend the ray backward, it then touches the sphere).

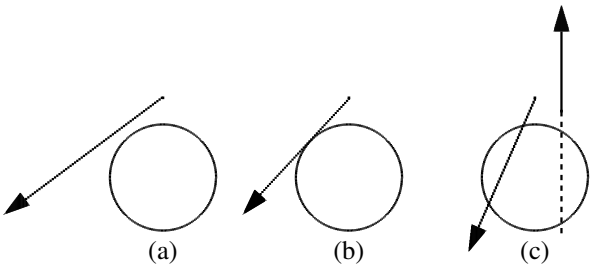



**Figure 17-6** Three Cases of Ray-Sphere Intersection

## 17.3.2 Ray-Sphere Intersection Implementation

We can represent a sphere using a class with a *Point3* data member to denote the sphere center, and a double data member to denote the radius as shown below:

```
class Sphere {
public:
  Point3 C;   //sphere center
  double R;   //sphere radius
  Sphere (){  //default
    c = Point3( 0.0, 0.0, 0.0 );
    R = 1.0;
  }
  Sphere ( double r0 ) {
    C = Point3( 0.0, 0.0, 0.0 );
    R = r0;
  }
  Sphere ( const Point3 &c0, double r0 ){
    C = c0;
    R = r0;
  }
};
```

The following function, **intersect_ray_sphere**() of Listing 17-2, implements the ray-sphere intersection test.

**Program Listing 17-2** Ray-Sphere Intersection Test

```
-------------------------------------------------------------------------------
bool intersect_ray_sphere(const Ray& ray,  const Sphere &sphere, double &t)
{
    double r = sphere.R;

    //Compute a, b and c coefficients
    double a =  ray.D * ray.D;    //dot product

    Vector3 oc = ray.O - sphere.C;
    double b = 2 * oc * ray.D;
    double c = oc * oc - r * r;

    //Find discriminant
    double disc = b * b - 4 * a * c;

    // if discriminant is negative there are no real roots, so return
    // false as ray misses sphere
    if (disc < 0)
        return false;

    // compute the roots
    double distSqrt = sqrtf(disc);
    double t0 = (-b - distSqrt ) / ( 2 * a );
    double t1 = (-b + distSqrt ) / ( 2 * a );
    // make sure t0 is smaller than t1
    if (t0 > t1) {
        // if t0 is bigger than t1, swap them around
        double temp = t0;
        t0 = t1;
        t1 = temp;
    }

    // if t1 < 0, object is in the ray's negative direction
    // as a  consequence the ray misses the sphere
```

```
    if (t1 < 0)
        return false;

    // if t0 is less than zero, the intersection point is at t1
    if (t0 < 0)  {
        t = t1;
        return true;
    } else {  // else the intersection point is at t0
        t = t0;
        return true;
    }
}
```

---

## 17.4 Ray-Tapered Cylinder Intersection

A tapered cylinder is a frustum cone with unequal base and cap as shown in Figure 17-4 (b). For simplicity, we consider the side of the tapered cylinder as a wall along the $z$-direction, with radius of 1 at $z = 0$ and a smaller radius $s$ at $z = 1$. Then the equation of the wall is given by:

$$x^2 + y^2 - (1 + (s-1)z)^2 = 0 \quad \text{for } 0 \le z \le 1 \tag{17.24}$$

When $s = 1$, it is reduced to a generic cylinder and when $s = 0$, it is a generic cone.

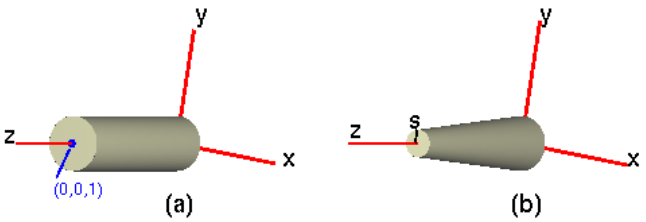



**Figure 17-7** (a) A generic cylinder (b) A tapered cylinder

There are four situations that a ray can intersect with a tapered cylinder:

1. The ray hits the wall twice.
2. The ray enters through the cap and hits the wall once.
3. The ray hits the wall and exits through the the base.
4. The ray enters the base and exits through the cap.

**Ray-Wall Intersection**

We again assume that a ray is given by (17.3) ($L(t) = O + t\mathbf{D}$). To determine whether the ray hits the wall, we substitute this into (17.24) for the intersection point, and obtain a quadratic equation of the form:

$$at^2 + 2bt + c = 0 \tag{17.25}$$

where

$$\begin{aligned} a = &\quad \mathbf{D}_x^2 + \mathbf{D}_y^2 - d^2 \\ b = &\quad O_x\mathbf{D}_x + O_y\mathbf{D}_y - Fd \\ c = &\quad O_x^2 + O_y^2 - F^2 \\ d = &\quad (s-1)\mathbf{D}_x \\ F = &\quad 1 + (s-1)O_z \end{aligned}$$

Whether the ray hits the wall or not is determined by the discriminant $Det$ of the quadratic equation (17.25), which is

$$Det = (2b)^2 - 4ac = 4(b^2 - ac) \tag{17.26}$$

We have the following situations:

1. If $Det < 0$, no intersection occurs (the ray passes by the cylinder).
2. If $Det \geq 0$, the ray hits the infinite wall and the parameter $t_{hit}$ can be found. To determine whether the ray hits the cylinder wall, we can check whether $0 \leq z \leq 1$ at $t_{hit}$. If yes, the ray hits the cylinder wall.

**Ray-Base Intersection**

To determine whether the ray hits the base, we apply the ray-plane intersection check with the plane (plane containing base) described by $z = 0$. If the ray hits the plane at point $(x, y, 0)$, then the hit spot lies within the base if $x^2 + y^2 < 1$.

**Ray-Cap Intersection**

In this case we apply the ray-plane intersection check with plane $z = 1$ (plane containing cap). If the ray hits the plane at point $(x, y, 1)$, then the hit spot lies within the cap if $x^2 + y^2 < s^2$.

## 17.5 Ray-Quadric Intersection

A *quadric* is a surface in Euclidean space consisting of points that satisfy a polynomial of degree 2, which is of the form,

$$f(x, y, z) = Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Jz + K = 0 \tag{17.27}$$

where $A, B, C, D, E, F, G, H, J, K$ are some constants. Examples of quadrics include spheres, cylinders, ellipsoids, paraboloids, hyperboloids, and cones.

To determine whether the ray hits the quadric, we can again substitute the ray equation,

$$(x(t), y(t), z(t)) = L(t) = O + t\mathbf{D} \tag{17.28}$$

into the quadric equation (17.26). We will again obtain a quadratic equation of $t$,

$$at^2 + 2bt + c = 0 \tag{17.29}$$

and the discriminant in the form of (17.26) determines whether the ray hits the object:

1. If $Det > 0$, the ray may hit the quadric surface twice (two $t_{hit}$ solutions).
2. If $Det = 0$, the ray may hit the quadric surface once (one $t_{hit}$ solution).
3. If $Det < 0$, the ray misses the quadric (no solution).

Other books by the same author

# Windows Fan, Linux Fan

by *Fore June*

*Windws Fan, Linux Fan* describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider ( ISP ) and create your own wealth. See *http://www.forejune.com/*

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

# An Introduction to Digital Video Data Compression in Java

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in java. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding. See

*http://www.forejune.com/*

January 2011

ISBN-10: 1456570870

ISBN-13: 978-1456570873

---

# An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273