GAMASUTRA
The Art & Business of Making Games

PRINT

Simple Intersection Tests For Games

By miguel gomez

Whether it's your car crossing the finish line at 180 miles per hour, or a bullet tearing through the chest of your best friend, all games make use of collision detection for object interaction. This article describes some simple intersection tests for the most useful shapes: spheres and boxes.

Sweep Tests for Moving Objects

A common approach to collision detection is to simply test for whether two objects are overlapping at the end of each frame. The problem with this method is that quickly moving objects can pass through each other without detection. To avoid this problem, their trajectories can be subdivided and the objects checked for overlap at each point; however, this gets expensive if either object experienced a large displacement. On the other hand, a sweep test can efficiently determine a lower and upper bound for the time of overlap, which can then be used as more optimal starting points for the subdivision algorithm.

A Sphere-Plane Sweep Test

Figure 1 shows an example of a quickly moving sphere passing through a plane. It can be seen that $\mathbf{C_0}$ is on the positive side of the plane and $\mathbf{C_1}$ is on its negative side.

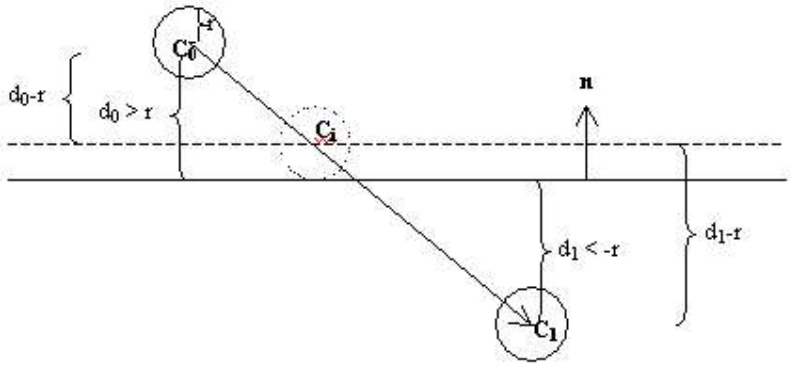

Figure 1. A sphere passes through a plane.

In general, if a sphere penetrated a plane at some point during the frame, then $d_0 > r$ and $d_1 < r$, where r is the radius of the sphere and d_0 and d_1 are the signed distances from the plane to $\mathbf{C_0}$ and $\mathbf{C_1}$, respectively. The signed distance from a point $\mathbf{C}$ to a plane can be calculated with the formula

$$d = (\mathbf{C} - \mathbf{p_0}) \cdot \mathbf{n}$$

where

$\mathbf{p_0}$ = any point on the plane

$\mathbf{n}$ = unit normal to the plane

More efficiently, we can store the plane in the form $\{\mathbf{n}, D\}$, where

$$D = -\mathbf{p_0} \cdot \mathbf{n}$$

The distance d is then calculated

$$d = \mathbf{n} \cdot \mathbf{C} + D$$

The trajectory from $\mathbf{C_0}$ to $\mathbf{C_1}$ can be parameterized with a variable u, which may be thought of as *normalized time,* since its value is 0 at $\mathbf{C_0}$ and 1 at $\mathbf{C_1}$. The normalized time at which the sphere first intersects the plane is given by

$$u_i = \frac{d_0 - r}{d_0 - d_1}$$

The center of the sphere at this time can then be interpolated with an affine combination of $\mathbf{C_0}$ and $\mathbf{C_1}$

$$\mathbf{C_i} = (1 - u_i)*\mathbf{C_0} + u_i*\mathbf{C_1}$$

This formula interpolates $\mathbf{C_i}$ correctly as long as d_0 is not equal to d_1 (which is the case if displacement has occurred), even when $r = 0$ (the case of a line segment). If desired, the parameter u can also be used to linearly interpolate the orientation of an object at this point.

In this example, it was assumed that the sphere approached the plane from the positive side and that the sphere was not already penetrating the plane at $\mathbf{C_0}$. In the case that there could have been penetration on the previous frame, the condition $|d_0| <= r$ should also be checked. Listing 1 gives an implementation of this sphere-plane sweep test.

Listing 1. A sphere-plane sweep test.

```
#include "vector.h"

class PLANE

{


        public:

        VECTOR N;
        //unit normal

        SCALAR D;
        //distance from the plane to the origin from a
        //normal and a point

        PLANE( const VECTOR& p0, const VECTOR& n ): N(n), D(-N.dot(p0))
        {}
        //from 3 points

        PLANE( const VECTOR& p0, const VECTOR& p1,
        const VECTOR& p2 ): N((p1-p0).cross(p2-p0).unit()),
        D(-N.dot(p0))
        {}
        //signed distance from the plane topoint 'p' along
        //the unit normal

        const SCALAR distanceToPoint( const VECTOR& p ) const
        {




                return N.dot(p) + D;

        }

};

const bool SpherePlaneSweep
(
const SCALAR r, //sphere radius
const VECTOR& C0, //previous position of sphere
const VECTOR& C1, //current position of sphere
const PLANE& plane, //the plane
VECTOR& Ci, //position of sphere when it first touched the plane
SCALAR& u //normalized time of collision

)

{

const SCALAR d0 = plane.distanceToPoint( C0 );
const SCALAR d1 = plane.distanceToPoint( C1 );
```

```
//check if it was touching on previous frame
if( fabs(d0) <= r )

{


        Ci = C0;
        u = 0;
        return true;

}

//check if the sphere penetrated during this frame
if( d0>r && d1<r )
{


        u = (d0-r)/(d0-d1); //normalized time
        Ci = (1-u)*C0 + u*C1; //point of first contact
        return true;

}

return false;

}
```

For the definition of the VECTOR class, please see [3].

A Sphere-Sphere Sweep Test

Figure 2 shows two spheres that collided between frames. If these spheres experienced acceleration during the frame, their trajectories will be second or higher order curves; however, usually their paths can be accurately approximated as linear segments according to the equations

$$\mathbf{A}(u) = \mathbf{A}_0 + u * \mathbf{v}_a$$

and

$$\mathbf{B}(u) = \mathbf{B}_0 + u * \mathbf{v}_b$$

where

$$\mathbf{v}_a = \mathbf{A}_1 - \mathbf{A}_0$$

$$\mathbf{v}_b = \mathbf{B}_1 - \mathbf{B}_0$$

$$0 \le u \le 1$$

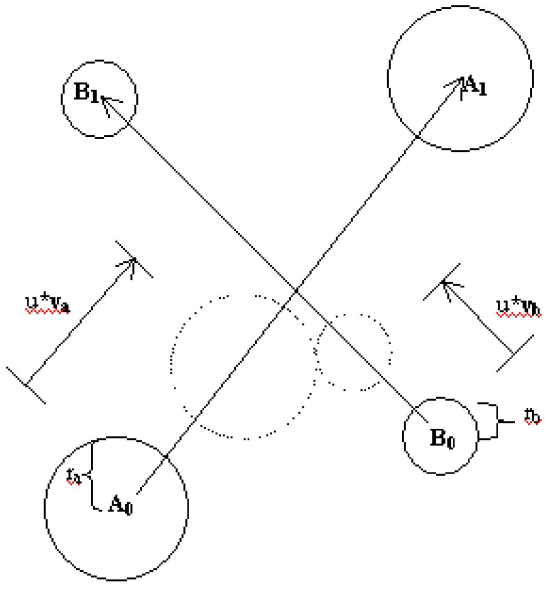

Figure 2. Two spheres that have moved between frames.

Since both spheres traveled for the same amount of time, *u* is the same for both trajectories. The square of the distance between the lines is

$$\left[\mathbf{B}(u) - \mathbf{A}(u)\right] \cdot \left[\mathbf{B}(u) - \mathbf{A}(u)\right]$$

and to calculate when they first make contact, we must solve for *u* such that

$$\left[\mathbf{B}(u) - \mathbf{A}(u)\right] \cdot \left[\mathbf{B}(u) - \mathbf{A}(u)\right] = (r_a + r_b)^2$$

This leads to the quadratic equation

$$\overline{\mathbf{AB}} \cdot \overline{\mathbf{AB}} + 2 * \mathbf{v}_{ab} \cdot \overline{\mathbf{AB}} * u + \mathbf{v}_{ab} \cdot \mathbf{v}_{ab} * u^2 = (r_a + r_b)^2$$

where

$$\overline{\mathbf{AB}} = \mathbf{B}_0 - \mathbf{A}_0$$

$$\mathbf{v}_{ba} = \mathbf{v}_b - \mathbf{v}_a$$

The vector $\mathbf{v_{ba}}$ can be thought of as the displacement of B observed by A. This equation is quadratic in *u*, so there may be no solution (the spheres never collided), one solution (they just glanced each other), or two solutions (in which case the lesser solution is when they began to overlap and the greater is when they became disjoint again). Again, it is a good idea to check for overlap at the beginning of the frame, since this will handle the case of two stationary spheres. Listing 2 shows an implementation of the sphere-sphere sweep test.

Listing 2. The sphere-sphere sweep test.

```
#include "vector.h"

template< class T >

inline void SWAP( T& a, T& b )
//swap the values of a and b

{


	const T temp = a;
	a = b;
	b = temp;

}

// Return true if r1 and r2 are real
inline bool QuadraticFormula
(
const SCALAR a,
const SCALAR b,
const SCALAR c,
SCALAR& r1, //first
SCALAR& r2 //and second roots
)

{




	const SCALAR q = b*b - 4*a*c;
	if( q >= 0 )

	{



		const SCALAR sq = sqrt(q);
		const SCALAR d = 1 / (2*a);
		r1 = ( -b + sq ) * d;
		r2 = ( -b - sq ) * d;
		return true;//real roots

	}
```

```
        else

        {




                return false;//complex roots

        }

}

const bool SphereSphereSweep
(
const SCALAR ra, //radius of sphere A
const VECTOR& A0, //previous position of sphere A
const VECTOR& A1, //current position of sphere A
const SCALAR rb, //radius of sphere B
const VECTOR& B0, //previous position of sphere B
const VECTOR& B1, //current position of sphere B
SCALAR& u0, //normalized time of first collision
SCALAR& u1 //normalized time of second collision

)

{



        const VECTOR va = A1 - A0;
        //vector from A0 to A1

        const VECTOR vb = B1 - B0;
        //vector from B0 to B1

        const VECTOR AB = B0 - A0;
        //vector from A0 to B0

        const VECTOR vab = vb - va;
        //relative velocity (in normalized time)

        const SCALAR rab = ra + rb;

        const SCALAR a = vab.dot(vab);
        //u*u coefficient

        const SCALAR b = 2*vab.dot(AB);
        //u coefficient

        const SCALAR c = AB.dot(AB) - rab*rab;
        //constant term

        //check if they're currently overlapping
        if( AB.dot(AB) <= rab*rab )

        {




                u0 = 0;
                u1 = 0;
                return true;

        }

        //check if they hit each other
        // during the frame
        if( QuadraticFormula( a, b, c, u0, u1 ) )
        {
```

```
            if( u0 > u1 )
            SWAP( u0, u1 );
            return true;

        }

        return false;

}
```

An Axis-Aligned Bounding Box (AABB) Sweep Test

Just like the name says, the faces of an axis-aligned bounding box are aligned with the coordinate axes of its parent frame (see Figure 3). In most cases AABBs can fit an object more tightly than a sphere, and their overlap test is extremely fast.

To see if A and B overlap, a separating axis test is used along the x, y, and z-axes. If the two boxes are disjoint, then at least one of these will form a separating axis. Figure 4 illustrates an overlap test in one dimension.

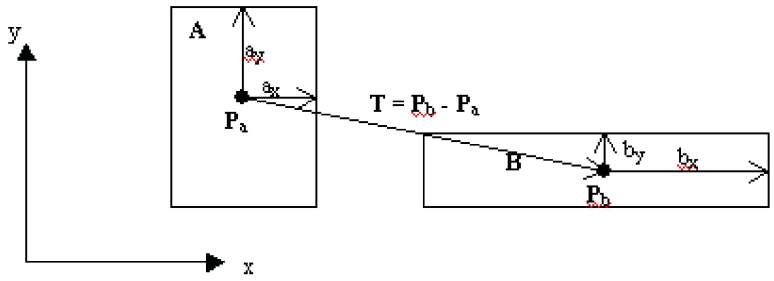

Figure 3. Axis-aligned bounding boxes in 2 dimensions.

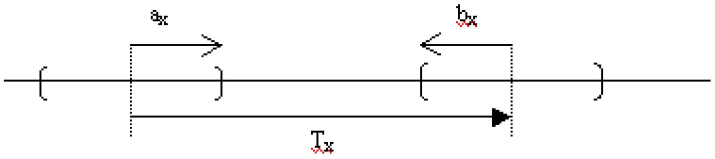

Figure 4. A separating axis test along the x-axis

In this example the x-axis forms a separating axis because

$$|T_x| > a_x + b_x$$

Note that the separating axis test will return true even if one box fully contains the other. A more general separating axis test is given in the section below on oriented bounding boxes (OBB's). Listing 3 defines an AABB class that implements this overlap test.

Listing 3. An AABB class.

```
#include "vector.h"

// An axis-aligned bounding box

class AABB
{


        public:

        VECTOR P; //position
        VECTOR E; //x,y,z extents

                AABB( const VECTOR& p,
                const VECTOR& e ): P(p), E(e)

                {}

//returns true if this is overlapping b
const bool overlaps( const AABB& b ) const
{
```

```
        const VECTOR T = b.P - P;//vector from A to B
        return fabs(T.x) <= (E.x + b.E.x)

        &&

        fabs(T.y) <= (E.y + b.E.y)

        &&

        fabs(T.z) <= (E.z + b.E.z);

}

//NOTE: since the vector indexing operator is not const,
//we must cast away the const of the this pointer in the
//following min() and max() functions
//min x, y, or z

const SCALAR min( long i ) const

{



        return ((AABB*)this)->P[i] - ((AABB*)this)->E[i];

}

//max x, y, or z
const SCALAR max( long i ) const

{



        return ((AABB*)this)->P[i] + ((AABB*)this)->E[i];

}

};
```

For more information on AABBs and their applications, please see [8].

Just like spheres, AABBs can be swept to find the first and last occurrence of overlap. In Figure 5(a), A and B experienced displacements $\mathbf{v_a}$ and $\mathbf{v_b}$, respectively, while Figure 5(b) shows B's displacement as observed by A.

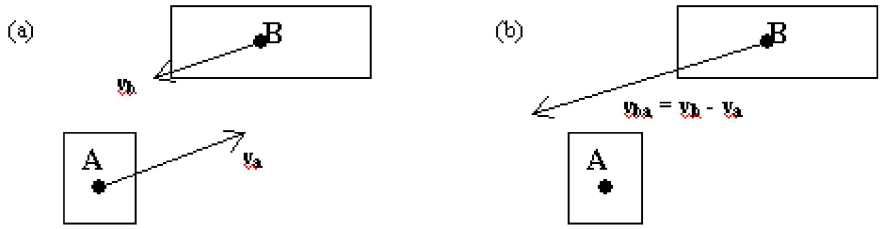

Figure 5. (a) two moving boxes (b) in A's frame of reference

Figure 6 shows B's displacement as observed by A.

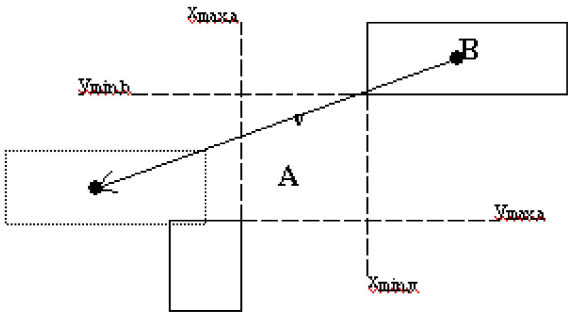

Figure 6. B can only collide with A after all extents have crossed.

In this example, the normalized times it took for the x-extents and y-extents to overlap are given by

$$u_x = \frac{x_{max,a} - x_{min,b}}{v_x}, \text{and } u_y = \frac{y_{max,a} - y_{min,b}}{v_y}$$

and it can be seen that the x-extents will cross before the y-extents. The two boxes cannot overlap until all the extents are overlapping, and the boxes will cease to overlap when any one of these extents becomes disjoint. If $u_{0,x}$, $u_{0,y}$, and $u_{0,z}$ were the times at which the x, y, and z-extents began to overlap, then the earliest time at which the boxes could have begun to overlap was

$$u_0 = \max(u_{0,x}, u_{0,y}, u_{0,z})$$

Likewise, if $u_{1,x}$, $u_{1,y}$, and $u_{1,z}$ are the times at which the x, y, and z-extents become disjoint, then the earliest time at which the boxes could have become disjoint was

$$u_1 = \min(u_{1,x}, u_{1,y}, u_{1,z})$$

In order for the two boxes to have overlapped during their displacement, the condition

$$u_0 \leq u_1$$

must have been met. Just like in the sphere sweep test, the positions of first and last overlap can be linearly interpolated with u. Listing 4 gives an implementation of this AABB sweep algorithm.

Listing 4. An AABB sweep algorithm.

```
#include "aabb.h"

//Sweep two AABB's to see if and when they first
//and last were overlapping

const bool AABBSweep
(


        const VECTOR& Ea, //extents of AABB A
        const VECTOR& A0, //its previous position
        const VECTOR& A1, //its current position
        const VECTOR& Eb, //extents of AABB B
        const VECTOR& B0, //its previous position
        const VECTOR& B1, //its current position
        SCALAR& u0, //normalized time of first collision
        SCALAR& u1 //normalized time of second collision

)

{


        const AABB A( A0, Ea );//previous state of AABB A
        const AABB B( B0, Eb );//previous state of AABB B
        const VECTOR va = A1 - A0;//displacement of A
        const VECTOR vb = B1 - B0;//displacement of B

        //the problem is solved in A's frame of reference

        VECTOR v = vb - va;
        //relative velocity (in normalized time)

        VECTOR u_0(0,0,0);
        //first times of overlap along each axis

        VECTOR u_1(1,1,1);
        //last times of overlap along each axis

        //check if they were overlapping
        // on the previous frame
        if( A.overlaps(B) )
        {
```

```
            u0 = u1 = 0;
            return true;

        }

        //find the possible first and last times
        //of overlap along each axis
        for( long i=0 ; i<3 ; i++ )
        {




            if( A.max(i)<B.min(i) && v[i]<0 )
            {


                u_0[i] = (A.max(i) - B.min(i)) / v[i];

            }

            else if( B.max(i)<A.min(i) && v[i]>0 )
            {


                u_0[i] = (A.min(i) - B.max(i)) / v[i];

            }

            if( B.max(i)>A.min(i) && v[i]<0 )
            {

                u_1[i] = (A.min(i) - B.max(i)) / v[i];

            }

            else if( A.max(i)>B.min(i) && v[i]>0 )
            {


                u_1[i] = (A.max(i) - B.min(i)) / v[i];

            }

        }

        //possible first time of overlap
        u0 = MAX( u_0.x, MAX(u_0.y, u_0.z) );

        //possible last time of overlap
        u1 = MIN( u_1.x, MIN(u_1.y, u_1.z) );

        //they could have only collided if
        //the first time of overlap occurred
        //before the last time of overlap
        return u0 <= u1;

}
```

A Box-Sphere Intersection Test

A very elegant box-sphere intersection test is described in [1]. Figure 7 shows two configurations of a sphere and a box in 2D. Sphere A is closest to an edge, whereas sphere B is closest to a corner. The algorithm calculates the square of the distance from the box to the sphere by analyzing the orientation of the sphere relative to the box in a single loop.

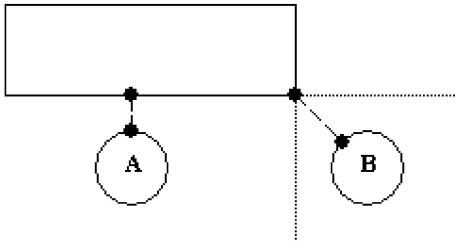

Figure 7. Closest points between a sphere and a box

If the box is not axis aligned, simply transform the center of the sphere to the box's local coordinate frame. Listing 5 gives an implementation of Arvo's algorithm.

Listing 5. Arvo's algorithm.

```
#include "aabb.h"

//Check to see if the sphere overlaps the AABB
const bool AABBOverlapsSphere ( const AABB& B, const SCALAR r, VECTOR& C )
{
	float s, d = 0;

	//find the square of the distance
	//from the sphere to the box
	for( long i=0 ; i<3 ; i++ )
	{

		if( C[i] < B.min(i) )
		{


			s = C[i] - B.min(i);
			d += s*s;

		}

		else if( C[i] > B.max(i) )
		{

			s = C[i] - B.max(i);
			d += s*s;

		}

	}
	return d <= r*r;

}
```

An Oriented Bounding Box (OBB) Intersection Test

A drawback of using an axis-aligned bounding box is that it can't fit rotating geometry very tightly.

On the other hand, an oriented bounding box can be rotated with the objects, fitting the geometry with less volume than an AABB. This requires that the orientation of the box must also be specified. Figure 8 shows a 2D example, where $\mathbf{A}^1$, $\mathbf{A}^2$, $\mathbf{B}^1$ and $\mathbf{B}^2$ are the local axes of boxes A and B.

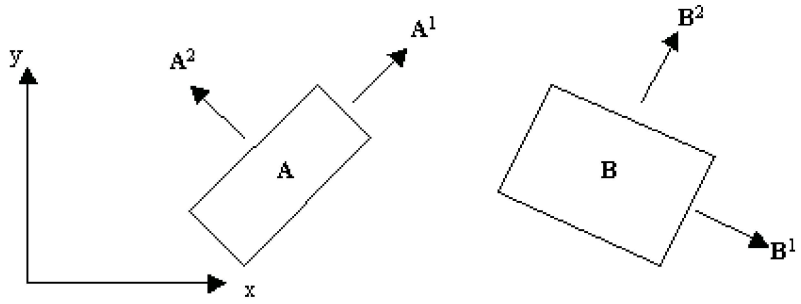

Figure 8. Oriented bounding boxes have local axes

For OBBs, the separating axis test must be generalized to three dimensions. A box's scalar projection onto a unit vector **L** creates an interval along the axis defined by **L**.

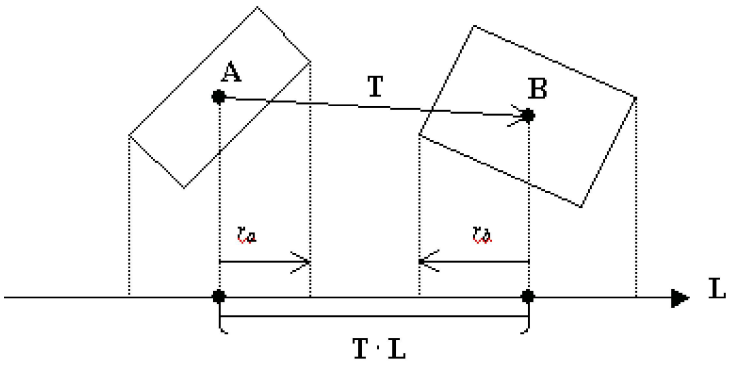

Figure 9. The vector **L** forms a separating axis.

The radius of the projection of box A onto **L** is

$$r_a = a_1\left|\mathbf{A}^1 \cdot \mathbf{L}\right| + a_2\left|\mathbf{A}^2 \cdot \mathbf{L}\right| + a_3\left|\mathbf{A}^3 \cdot \mathbf{L}\right|$$

The same is true for B, and **L** forms a separating axis if

$$\left|\mathbf{T} \cdot \mathbf{L}\right| > r_a + r_b$$

Note that **L** does not have to be a unit vector for this test to work. The boxes A and B are disjoint if none of the 6 principal axes and their 9 cross products form a separating axis. These tests are greatly simplified if **T** and B's basis vectors ($\mathbf{B}^1$, $\mathbf{B}^2$, $\mathbf{B}^3$) are transformed into A's coordinate frame.

An OBB class and an implementation of the OBB overlap test is given in Listing 6 below.

Listing 6. An OBB class.

```
#include "coordinate_frame.h"

class OBB : public COORD_FRAME
{

public:


        VeCTOR E; //extents
        OBB( const VECTOR& e ): E(e)

        {}

};

//check if two oriented bounding boxes overlap
const bool OBBOverlap
(
```

```
        //A
        VECTOR& a, //extents
        VECTOR& Pa, //position
        VECTOR* A, //orthonormal basis

        //B
        VECTOR& b, //extents
        VECTOR& Pb, //position
        VECTOR* B //orthonormal basis

)

{



        //translation, in parent frame
        VECTOR v = Pb - Pa;

        //translation, in A's frame
        VECTOR T( v.dot(A[0]), v.dot(A[1]), v.dot(A[2]) );

        //B's basis with respect to A's local frame
        SCALAR R[3][3];
        float ra, rb, t;
        long i, k;

        //calculate rotation matrix
        for( i=0 ; i<3 ; i++ )


                for( k=0 ; k<3 ; k++ )


                        R[i][k] = A[i].dot(B[k]);

                        /*ALGORITHM: Use the separating axis test for all 15 potential
                        separating axes. If a separating axis could not be found, the two
                        boxes overlap. */

                        //A's basis vectors
                        for( i=0 ; i<3 ; i++ )
                        {



                                ra = a[i];

                                rb =
                                b[0]*fabs(R[i][0]) + b[1]*fabs(R[i][1]) + b[2]*fabs(R[i][2]);

                                t = fabs( T[i] );

                                if( t > ra + rb )
                                return false;

                        }

                        //B's basis vectors
                        for( k=0 ; k<3 ; k++ )
                        {



                                ra =
                                a[0]*fabs(R[0][k]) + a[1]*fabs(R[1][k]) + a[2]*fabs(R[2][k]);

                                rb = b[k];

                                t =
                                fabs( T[0]*R[0][k] + T[1]*R[1][k] +
                                T[2]*R[2][k] );

                                if( t > ra + rb )
                                return false;

                        }
```

```
//9 cross products

//L = A0 x B0
ra =
a[1]*fabs(R[2][0]) + a[2]*fabs(R[1][0]);

rb =
b[1]*fabs(R[0][2]) + b[2]*fabs(R[0][1]);

t =
fabs( T[2]*R[1][0] -
T[1]*R[2][0] );

if( t > ra + rb )
return false;

//L = A0 x B1
ra =
a[1]*fabs(R[2][1]) + a[2]*fabs(R[1][1]);

rb =
b[0]*fabs(R[0][2]) + b[2]*fabs(R[0][0]);

t =
fabs( T[2]*R[1][1] -
T[1]*R[2][1] );

if( t > ra + rb )
return false;

//L = A0 x B2
ra =
a[1]*fabs(R[2][2]) + a[2]*fabs(R[1][2]);

rb =
b[0]*fabs(R[0][1]) + b[1]*fabs(R[0][0]);

t =
fabs( T[2]*R[1][2] -
T[1]*R[2][2] );

if( t > ra + rb )
return false;

//L = A1 x B0
ra =
a[0]*fabs(R[2][0]) + a[2]*fabs(R[0][0]);

rb =
b[1]*fabs(R[1][2]) + b[2]*fabs(R[1][1]);

t =
fabs( T[0]*R[2][0] -
T[2]*R[0][0] );

if( t > ra + rb )
return false;

//L = A1 x B1
ra =
a[0]*fabs(R[2][1]) + a[2]*fabs(R[0][1]);

rb =
b[0]*fabs(R[1][2]) + b[2]*fabs(R[1][0]);

t =
fabs( T[0]*R[2][1] -
T[2]*R[0][1] );

if( t > ra + rb )
return false;

//L = A1 x B2
ra =
a[0]*fabs(R[2][2]) + a[2]*fabs(R[0][2]);

rb =
b[0]*fabs(R[1][1]) + b[1]*fabs(R[1][0]);
```

```
        t =
        fabs( T[0]*R[2][2] -
        T[2]*R[0][2] );

        if( t > ra + rb )
        return false;

        //L = A2 x B0
        ra =
        a[0]*fabs(R[1][0]) + a[1]*fabs(R[0][0]);

        rb =
        b[1]*fabs(R[2][2]) + b[2]*fabs(R[2][1]);

        t =
        fabs( T[1]*R[0][0] -
        T[0]*R[1][0] );

        if( t > ra + rb )
        return false;

        //L = A2 x B1
        ra =
        a[0]*fabs(R[1][1]) + a[1]*fabs(R[0][1]);

        rb =
        b[0] *fabs(R[2][2]) + b[2]*fabs(R[2][0]);

        t =
        fabs( T[1]*R[0][1] -
        T[0]*R[1][1] );

        if( t > ra + rb )
        return false;

        //L = A2 x B2
        ra =
        a[0]*fabs(R[1][2]) + a[1]*fabs(R[0][2]);

        rb =
        b[0]*fabs(R[2][1]) + b[1]*fabs(R[2][0]);

        t =
        fabs( T[1]*R[0][2] -
        T[0]*R[1][2] );

        if( t > ra + rb )
        return false;

        /*no separating axis found,
        the two boxes overlap */

        return true;

    }
```

For a more complete discussion of OBBs and the separating axis test, please see [3]. Some other applications of the separating axis test are given next.

An OBB-Line Segment Test

Testing a box and a line segment for intersection requires checking only six separating axes: the box's three principal axes, and the vector cross products of these axes with **l**, the line direction. Again, the vectors used for these tests do not have to be normalized, and these tests can be simplified by transforming the line segment into the box's coordinate frame.

One application of this test is to see if a camera's line of sight is obscured. Testing every polygon in a scene could be prohibitively expensive, but if these polygons are stored in an AABB or an OBB tree, a box-segment test can quickly determine a potential set of polygons. A segment-polygon test can then be used to determine if any polygons in this subset are actually obscuring the line of sight.

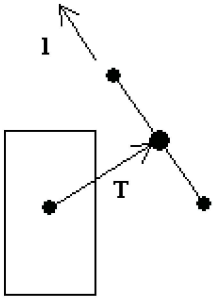

Figure 10. A line segment can be treated as a degenerate OBB.

The function in Listing 7 assumes the line segment has already been transformed to box space.

Listing 7.

```
#include "aabb.h"

const bool AABB_LineSegmentOverlap
(


        const VECTOR& l, //line direction
        const VECTOR& mid, //midpoint of the line
        // segment
        const SCALAR hl, //segment half-length
        const AABB& b //box

)

{


        /* ALGORITHM: Use the separating axis
        theorem to see if the line segment
        and the box overlap. A line
        segment is a degenerate OBB. */

        const VECTOR T = b.P - mid;
        VECTOR v;
        SCALAR r;

        //do any of the principal axes
        //form a separating axis?

        if( fabs(T.x) > b.E.x + hl*fabs(l.x) )
        return false;

        if( fabs(T.y) > b.E.y + hl*fabs(l.y) )
        return false;

        if( fabs(T.z) > b.E.z + hl*fabs(l.z) )
        return false;

        /* NOTE: Since the separating axis is
        perpendicular to the line in these
        last four cases, the line does not
        contribute to the projection. */

        //l.cross(x-axis)?

        r = b.E.y*fabs(l.z) + b.E.z*fabs(l.y);

        if( fabs(T.y*l.z - T.z*l.y) > r )
        return false;

        //l.cross(y-axis)?

        r = b.E.x*fabs(l.z) + b.E.z*fabs(l.x);

        if( fabs(T.z*l.x - T.x*l.z) > r )
        return false;

        //l.cross(z-axis)?
```

```
        r = b.E.x*fabs(l.y) + b.E.y*fabs(l.x);

        if( fabs(T.x*l.y - T.y*l.x) > r )
        return false;

        return true;

}
```

OBB to AABB conversion

Converting an OBB to an AABB merely involves calculating the extents of the OBB along the x, y, and z-axes of its parent frame. For example the extent of the OBB along the x-axis is

$$E_x = a_1\left|\mathbf{A}^1 \cdot \mathbf{x}\right| + a_2\left|\mathbf{A}^2 \cdot \mathbf{x}\right| + a_3\left|\mathbf{A}^3 \cdot \mathbf{x}\right| = a_1\left|A_x^1\right| + a_2\left|A_x^2\right| + a_3\left|A_x^3\right|$$

The extents along the y and z-axes are calculated similarly.

A Box-Plane Intersection Test

As you can see from Figure 11, a box-plane intersection test only requires checking whether or not **n** forms a separating axis.

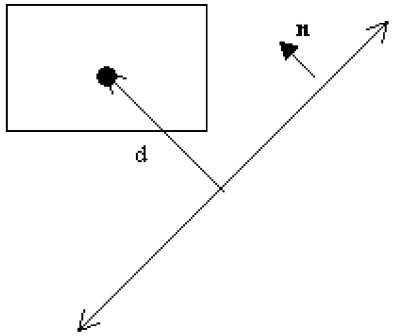

Figure 11. A box-plane test uses **n** as a separating axis.

The box and the plane overlap if the condition

$$\left|d\right| \le a_1\left|\mathbf{n} \cdot \mathbf{A}^1\right| + a_2\left|\mathbf{n} \cdot \mathbf{A}^2\right| + a_3\left|\mathbf{n} \cdot \mathbf{A}^3\right|$$

is met, where *d* is the distance from the center of the box to the plane.

Further Reading

Due to time and space, all of the useful intersection tests could not be described here. Some good polygon algorithms are given in [5] and [6]. References [4] and [8] present unique algorithms for generating and manipulating box trees. Bobic and Lander survey the subject of actual collision detection (determining a point of contact and a surface normal) in [2], [6] and [7], which is definitely an expansive, active area of research. If you plan on employing more advanced collision detection algorithms in your games, you should definitely check out all of the references given below.

Soon after receiving his degree in Physics, Miguel Gomez was lucky enough to land a job as a game programmer. Since then he has programmed physics and graphics for *PGA Tour Golf '96*, *Hyperblade*, *Microsoft Baseball 3D*, and *Destruction Derby 64*. He is currently perfecting the collision detection and fluid physics for a kayak racing title at Looking Glass Studios in Redmond. Please send questions and comments to miguel@lglass.com.

References

[1] J. Arvo. A simple method for box-sphere intersection testing. In A. Glassner, editor, *Graphics Gems*, pp. 335-339, Academic Press, Boston, MA, 1990.

[2] N. Bobic. "Advanced Collision Detection Techniques". *Game Developer* 6(5):32-42, 1999

[3] M. Gomez. "C++ Data Structures for Rigid-Body Physics". *Gamasutra,* July 2, 1999

[4] S. Gottschalk, M. C. Lin, and D. Manocha. "OBBTree: A Hierarchical Structure for Rapid Interference Detection." In *Proc. SIGGRAPH*, pp. 171-180, 1996.

[5] M. Held. "ERIT - A Collection of Efficient and Reliable Intersection Tests". *Journal of Graphics Tools*, 2(4):25-44, 1997.

[6] J. Lander. "Crashing into the New Year". *Game Developer* 6(1):21-27, 1999

[7] J. Lander. "When Two Hearts Collide". *Game Developer* 6(2):19-24, 1999

[8] G. Van den Bergen. "Efficient Collision Detection of Complex Deformable Models Using AABB Trees". *Journal of Graphics Tools*, 2(4):1-14, 1997.

Return to the full version of this article