Intersection of Orthogonal View Frustum and Oriented Bounding Box using Separation Axis Testing

David Eberly
Magic Software, Inc.
http://www.magic-software.com

Created: March 19, 2000

1 Introduction

This document describes how to determine if an oriented bounding box intersects an orthogonal view frustum. The separating axis method is used to determine this.

2 Oriented Bounding Box

The oriented bounding box is represented in center–axis–extent form with center $\vec{C}$; axes $\vec{A}_0$, $\vec{A}_1$, and $\vec{A}_2$; and extents e_0, e_1, and e_2. The axes form a right–handed orthonormal system. The extents are assumed to be positive. The eight vertices of the box are $\vec{C} + \sum_{i=0}^{2} \sigma_i e_i \vec{A}_i$ where $|\sigma_i| = 1$ (eight choices on sign).

The three normal vectors for the six box faces are $\vec{A}_i$ for $0 \le i \le 2$. The three edge direction vectors for the twelve box edges are the same set of vectors.

3 Orthogonal View Frustum

The orthogonal view frustum has origin $\vec{E}$. Its coordinate axes are determined by left vector $\vec{L}$, up vector $\vec{U}$, and direction vector $\vec{D}$. The vectors in that order form a right–handed orthonormal system. The extent of the frustum in the $\vec{D}$ direction is $[n, f]$ where $0 < n < f$. The view plane is assumed to be the near plane, $\vec{D} \cdot (\vec{X} - \vec{E}) = n$. The far plane is $\vec{D} \cdot (\vec{X} - \vec{E}) = f$. The four corners of the frustum in the near plane are $\vec{E} \pm \ell\vec{L} \pm \mu\vec{U} + n\vec{D}$. The four corners of the frustum in the far plane are $\vec{E} + (f/n)(\pm\ell\vec{L} \pm \mu\vec{U} + n\vec{D})$.

The five normal vectors for the six frustum faces are $\vec{D}$ for the near and far faces, $\pm n\vec{L} - \ell\vec{D}$ for the left and right faces, and $\pm n\vec{U} - \ell\vec{D}$ for the top and bottom faces. The six edge direction vectors for the twelve frustum edges are $\vec{L}$ and $\vec{U}$ for the edges on the near and far faces, and $\pm\ell\vec{L} \pm \mu\vec{U} + n\vec{D}$. The normal and edge directions are not all unit length, but this is not necessary for the separation axis tests.

4 Separating Axis Test

Two convex polyhedra do not intersect if there exists a line with direction $\vec{M}$ such that the projections of the polyhedra onto the line do not intersect. In this case there must exist a plane with normal vector $\vec{M}$ that separates the two polyhedra. Given a line, it is straightforward to project the vertices of a polyhedron onto the line and compute the bounding interval $[\lambda_{\min}, \lambda_{\max}]$ for those projections. The two intervals obtained from the two polyhedra are easily compared for overlap.

The more difficult problem is selecting a finite set of line directions such that the intersection/nonintersection can be determined by separating axis tests using only vectors in that set. For convex polyhedra it turns out that the set consisting of face normals for the two polyhedra and vectors that are the cross product of edges, one edge from each polyhedron, is sufficient for the intersection testing. If polyhedron i has F_i faces and E_i edges, then the total number of vectors in the set is $F_0 + F_1 + E_0E_1$. It is possible that some of the vectors formed by cross products of edges are zero in which case they do not need to be tested. This happens, for example, with two axis–aligned bounding boxes. While the total number of vectors is $3 + 3 + 3 * 3 = 15$, the set has only 3 nonzero vectors.

The oriented bounding box has $F_0 = 3$ and $E_0 = 3$. The orthogonal view frustum has $F_1 = 5$ and $E_1 = 6$. The total number of vectors to test is 26. That set is

$$\{\vec{D}, \pm n\vec{L} - \ell\vec{D}, \pm n\vec{U} - \mu\vec{D}, \vec{A}_i, \vec{L} \times \vec{A}_i, \vec{U} \times \vec{A}_i, (\pm\ell\vec{L} \pm \mu\vec{U} + n\vec{D}) \times \vec{A}_i\}.$$

The separating axes that are tested will all be of the form $\vec{E} + \lambda\vec{M}$ where $\vec{M}$ is in the previously mentioned set. The projected vertices of the box have λ values $\vec{M} \cdot (\vec{C} - \vec{E}) + \sum_{i=0}^{2} \sigma_i e_i \vec{A}_i$ where $|\sigma_i| = 1$. Define $d = \vec{M} \cdot (\vec{C} - \vec{E})$ and $R = \sum_{i=0}^{2} e_i |\vec{M} \cdot \vec{A}_i|$. The projection interval is $[d - R, d + R]$.

The projected vertices of the frustum have λ values $\kappa(\tau_0 \ell \vec{M} \cdot \vec{L} + \tau_1 \mu \vec{M} \cdot \vec{U} + n\vec{M} \cdot \vec{D})$ where $\kappa \in \{1, f/n\}$ and $|\tau_i| = 1$. Define $p = \ell|\vec{M} \cdot \vec{L}| + \mu|\vec{M} \cdot \vec{U}|$. The projection interval is $[m_0, m_1]$ where

$$m_0 = \left\{ \begin{array}{ll} \frac{f}{n}\left(n\vec{M} \cdot \vec{D} - p\right), & n\vec{M} \cdot \vec{D} - p < 0 \\ n\vec{M} \cdot \vec{D} - p & n\vec{M} \cdot \vec{D} - p \geq 0 \end{array} \right\}$$

and

$$m_1 = \left\{ \begin{array}{ll} \frac{f}{n}\left(n\vec{M} \cdot \vec{D} + p\right), & n\vec{M} \cdot \vec{D} + p > 0 \\ n\vec{M} \cdot \vec{D} + p & n\vec{M} \cdot \vec{D} + p \leq 0 \end{array} \right\}.$$

The box and frustum do not intersect if for some choice of $\vec{M}$ the two projection intervals $[m_0, m_1]$ and $[d - R, d + R]$ do not intersect. The intervals do not intersect if $d + R < m_0$ or $d - R > m_1$. An unoptimized implementation will compute d, R, m_0, and m_1 for each of the 26 cases and test the two inequalities. However, an optimized implementation will save intermediate results during each test and use them for later tests.

5 Cacheing Intermediate Results

Effectively the potential separating axis directions will all be manipulated in the coordinate system of the frustum. That is, each direction is written as $\vec{M} = x_0\vec{L} + x_1\vec{U} + x_2\vec{D}$ and the coefficients are used in the various tests. The difference $\vec{C}-\vec{E}$ must also be represented in the frustum coordinates, say $\vec{C}-\vec{E} = \delta_0\vec{L}+\delta_1\vec{U}+\delta_2\vec{D}$. The table of coefficients is given below.

$\vec{M}$	$\vec{M}\cdot\vec{L}$	$\vec{M}\cdot\vec{U}$	$\vec{M}\cdot\vec{D}$	$\vec{M}\cdot(\vec{C}-\vec{E})$
$\vec{D}$	0	0	1	δ_2
$\pm n\vec{L}-\ell\vec{D}$	$\pm n$	0	$-\ell$	$\pm n\delta_0-\ell\delta_2$
$\pm n\vec{U}-\mu\vec{D}$	0	$\pm n$	$-\mu$	$\pm n\delta_1-\mu\delta_2$
$\vec{A}_i$	α_i	β_i	γ_i	$\alpha_i\delta_0+\beta_i\delta_1+\gamma_i\delta_2$
$\vec{L}\times\vec{A}_i$	0	$-\gamma_i$	β_i	$-\gamma_i\delta_1+\beta_i\delta_2$
$\vec{U}\times\vec{A}_i$	γ_i	0	$-\alpha_i$	$\gamma_i\delta_0-\alpha_i\delta_2$
$(\tau_0\ell\vec{L}+\tau_1\mu\vec{U}+n\vec{D})\times\vec{A}_i$	$-n\beta_i+\tau_1\mu\gamma_i$	$n\alpha_i-\tau_0\ell\gamma_i$	$\tau_0\ell\beta_i-\tau_1\mu\alpha_i$	$[-n\beta_i+\tau_1\mu\gamma_i]\delta_0+$
				$[n\alpha_i-\tau_0\ell\gamma_i]\delta_1+$
				$[\tau_0\ell\beta_i-\tau_1\mu\alpha_i]\delta_2$

where $0\le i\le 2$ and $|\tau_j|=1$ for $0\le j\le 1$.

The quantities α_i, β_i, γ_i, and δ_i are computed only when needed to avoid unnecessary calculations. Some products are computed only when needed and saved for later use. These include products of n, ℓ, or μ with α_i, β_i, γ_i, or δ_i. Some terms that include sums or differences of the products are also computed only when needed and saved for later use. These include $n\alpha_i\pm\ell\gamma_i$, $n\beta_i\pm\mu\gamma_i$, $\ell\alpha_i\pm\mu\beta_i$, and $\ell\beta_i\pm\mu\alpha_i$.

6 Implementation

The code assumes the existence of some basic classes. Class `Real` is a simple wrapper for your favorite precision floating point numbers. Class `Vector3` represents 3–tuples. The only member of the class which is directly accessed is the dot product `Dot`. Class `Box3` represents the oriented bounding box. Accessor `Center` represents $\vec{C}$. Accessor `Axes` returns a pointer to an array of the three box axes $\vec{A}_0$, $\vec{A}_1$, and $\vec{A}_2$. Accessor `Extents` returns a pointer to an array of the three box extents, e_0, e_1, and e_2. Class `Frustum` represents the frustum. Accessor `Origin` represents $\vec{E}$. Accessors `LVector`, `UVector`, and `DVector` represent $\vec{L}$, $\vec{U}$, and $\vec{D}$. Accessors `LBound`, `UBound`, `DMin`, and `DMax` represent ℓ, μ, n, and f, respectively. The accessors `GetDRatio`, `GetMTwoLF`, and `GetMTwoUF` return f/n, $-2\ell f$, and $-2\mu f$, respectively.

```
bool TestIntersection (const Box3& rkBox, const Frustum& rkFrustum)
{
    const Vector3* akA = rkBox.Axes();
    const Real* afE = rkBox.Extents();

    Vector3 kDiff = rkBox.Center() - rkFrustum.Origin();

    Real afA[3], afB[3], afC[3], afD[3];
    Real afNA[3], afNB[3], afNC[3], afND[3];
    Real afNApLC[3], afNAmLC[3], afNBpUC[3], afNBmUC[3];
    Real afLC[3], afLD[3], afUC[3], afUD[3], afLBpUA[3], afLBmUA[3];
    Real fDdD, fR, fP, fMin, fMax, fMTwoLF, fMTwoUF, fLB, fUA, fTmp;
    int i, j;

    // M = D
    afD[2] = kDiff.Dot(rkFrustum.DVector());
    for (i = 0; i < 3; i++)
        afC[i] = akA[i].Dot(rkFrustum.DVector());
    fR = afE[0]*Math::Abs(afC[0]) +
         afE[1]*Math::Abs(afC[1]) +
         afE[2]*Math::Abs(afC[2]);
    if ( afD[2] + fR < rkFrustum.DMin() || afD[2] - fR > rkFrustum.DMax() )
        return false;

    // M = n*L - l*D
    for (i = 0; i < 3; i++)
    {
        afA[i] = akA[i].Dot(rkFrustum.LVector());
        afLC[i] = rkFrustum.LBound()*afC[i];
        afNA[i] = rkFrustum.DMin()*afA[i];
        afNAmLC[i] = afNA[i] - afLC[i];
    }
    afD[0] = kDiff.Dot(rkFrustum.LVector());
    fR = afE[0]*Math::Abs(afNAmLC[0]) +
         afE[1]*Math::Abs(afNAmLC[1]) +
         afE[2]*Math::Abs(afNAmLC[2]);
    afND[0] = rkFrustum.DMin()*afD[0];
    afLD[2] = rkFrustum.LBound()*afD[2];
    fDdD = afND[0] - afLD[2];
    fMTwoLF = rkFrustum.GetMTwoLF();
    if ( fDdD + fR < fMTwoLF || fDdD > fR )
        return false;

    // M = -n*L - l*D
    for (i = 0; i < 3; i++)
        afNApLC[i] = afNA[i] + afLC[i];
    fR = afE[0]*Math::Abs(afNApLC[0]) +
         afE[1]*Math::Abs(afNApLC[1]) +
         afE[2]*Math::Abs(afNApLC[2]);
    fDdD = -(afND[0] + afLD[2]);
    if ( fDdD + fR < fMTwoLF || fDdD > fR )
        return false;

    // M = n*U - u*D
    for (i = 0; i < 3; i++)
    {
        afB[i] = akA[i].Dot(rkFrustum.UVector());
        afUC[i] = rkFrustum.UBound()*afC[i];
        afNB[i] = rkFrustum.DMin()*afB[i];
        afNBmUC[i] = afNB[i] - afUC[i];
    }
    afD[1] = kDiff.Dot(rkFrustum.UVector());
    fR = afE[0]*Math::Abs(afNBmUC[0]) +
         afE[1]*Math::Abs(afNBmUC[1]) +
         afE[2]*Math::Abs(afNBmUC[2]);
    afND[1] = rkFrustum.DMin()*afD[1];
    afUD[2] = rkFrustum.UBound()*afD[2];
    fDdD = afND[1] - afUD[2];
    fMTwoUF = rkFrustum.GetMTwoUF();
    if ( fDdD + fR < fMTwoUF || fDdD > fR )
        return false;

    // M = -n*U - u*D
    for (i = 0; i < 3; i++)
        afNBpUC[i] = afNB[i] + afUC[i];
    fR = afE[0]*Math::Abs(afNBpUC[0]) +
         afE[1]*Math::Abs(afNBpUC[1]) +
         afE[2]*Math::Abs(afNBpUC[2]);
    fDdD = -(afND[1] + afUD[2]);
    if ( fDdD + fR < fMTwoUF || fDdD > fR )
        return false;

    // M = A[i]
    for (i = 0; i < 3; i++)
    {
        fP = rkFrustum.LBound()*Math::Abs(afA[i]) +
             rkFrustum.UBound()*Math::Abs(afB[i]);
        afNC[i] = rkFrustum.DMin()*afC[i];
        fMin = afNC[i] - fP;
        if ( fMin < 0.0 )
            fMin *= rkFrustum.GetDRatio();
        fMax = afNC[i] + fP;
        if ( fMax > 0.0 )
            fMax *= rkFrustum.GetDRatio();
        fDdD = afA[i]*afD[0] + afB[i]*afD[1] + afC[i]*afD[2];
        if ( fDdD + afE[i] < fMin || fDdD - afE[i] > fMax )
            return false;
    }

    // M = Cross(L,A[i])
    for (i = 0; i < 3; i++)
    {
        fP = rkFrustum.UBound()*Math::Abs(afC[i]);
        fMin = afNB[i] - fP;
        if ( fMin < 0.0 )
            fMin *= rkFrustum.GetDRatio();
        fMax = afNB[i] + fP;
        if ( fMax > 0.0 )
            fMax *= rkFrustum.GetDRatio();
        fDdD = -afC[i]*afD[1] + afB[i]*afD[2];
        fR = afE[0]*Math::Abs(afB[i]*afC[0]-afB[0]*afC[i]) +
             afE[1]*Math::Abs(afB[i]*afC[1]-afB[1]*afC[i]) +
             afE[2]*Math::Abs(afB[i]*afC[2]-afB[2]*afC[i]);
        if ( fDdD + fR < fMin || fDdD - fR > fMax )
            return false;
    }

    // M = Cross(U,A[i])
    for (i = 0; i < 3; i++)
    {
        fP = rkFrustum.LBound()*Math::Abs(afC[i]);
        fMin = -afNA[i] - fP;
        if ( fMin < 0.0 )
            fMin *= rkFrustum.GetDRatio();
        fMax = -afNA[i] + fP;
        if ( fMax > 0.0 )
            fMax *= rkFrustum.GetDRatio();
        fDdD = afC[i]*afD[0] - afA[i]*afD[2];
        fR = afE[0]*Math::Abs(afA[i]*afC[0]-afA[0]*afC[i]) +
             afE[1]*Math::Abs(afA[i]*afC[1]-afA[1]*afC[i]) +
             afE[2]*Math::Abs(afA[i]*afC[2]-afA[2]*afC[i]);
        if ( fDdD + fR < fMin || fDdD - fR > fMax )
            return false;
    }

    // M = Cross(n*D+l*L+u*U,A[i])
    for (i = 0; i < 3; i++)
    {
        fLB = rkFrustum.LBound()*afB[i];
        fUA = rkFrustum.UBound()*afA[i];
        afLBpUA[i] = fLB + fUA;
        afLBmUA[i] = fLB - fUA;
    }
    for (i = 0; i < 3; i++)
    {
        fP = rkFrustum.LBound()*Math::Abs(afNBmUC[i]) +
             rkFrustum.UBound()*Math::Abs(afNAmLC[i]);
        fTmp = rkFrustum.DMin()*afLBmUA[i];
        fMin = fTmp - fP;
        if ( fMin < 0.0 )
            fMin *= rkFrustum.GetDRatio();
        fMax = fTmp + fP;
        if ( fMax > 0.0 )
            fMax *= rkFrustum.GetDRatio();
        fDdD = -afD[0]*afNBmUC[i] + afD[1]*afNAmLC[i] + afD[2]*afLBmUA[i];
        fR = 0.0;
        for (j = 0; j < 3; j++)
        {
            fR += afE[j]*Math::Abs(-afA[j]*afNBmUC[i]+ afB[j]*afNAmLC[i]
                + afC[j]*afLBmUA[i]);
        }
        if ( fDdD + fR < fMin || fDdD - fR > fMax )
            return false;
    }

    // M = Cross(n*D+l*L-u*U,A[i])
    for (i = 0; i < 3; i++)
    {
        fP = rkFrustum.LBound()*Math::Abs(afNBpUC[i]) +
             rkFrustum.UBound()*Math::Abs(afNAmLC[i]);
        fTmp = rkFrustum.DMin()*afLBpUA[i];
        fMin = fTmp - fP;
        if ( fMin < 0.0 )
            fMin *= rkFrustum.GetDRatio();
        fMax = fTmp + fP;
        if ( fMax > 0.0 )
            fMax *= rkFrustum.GetDRatio();
        fDdD = -afD[0]*afNBpUC[i] + afD[1]*afNAmLC[i] + afD[2]*afLBpUA[i];
        fR = 0.0;
        for (j = 0; j < 3; j++)
        {
            fR += afE[j]*Math::Abs(-afA[j]*afNBpUC[i]+ afB[j]*afNAmLC[i]
                + afC[j]*afLBpUA[i]);
        }
        if ( fDdD + fR < fMin || fDdD - fR > fMax )
            return false;
    }

    // M = Cross(n*D-l*L+u*U,A[i])
    for (i = 0; i < 3; i++)
    {
        fP = rkFrustum.LBound()*Math::Abs(afNBmUC[i]) +
             rkFrustum.UBound()*Math::Abs(afNApLC[i]);
        fTmp = -rkFrustum.DMin()*afLBpUA[i];
        fMin = fTmp - fP;
        if ( fMin < 0.0 )
            fMin *= rkFrustum.GetDRatio();
        fMax = fTmp + fP;
        if ( fMax > 0.0 )
            fMax *= rkFrustum.GetDRatio();
        fDdD = -afD[0]*afNBmUC[i] + afD[1]*afNApLC[i] - afD[2]*afLBpUA[i];
        fR = 0.0;
```

```
        for (j = 0; j < 3; j++)
        {
            fR += afE[j]*Math::Abs(-afA[j]*afNBmUC[i]+ afB[j]*afNApLC[i]
                - afC[j]*afLBpUA[i]);
        }
        if ( fDdD + fR < fMin || fDdD - fR > fMax )
            return false;
    }

    // M = Cross(n*D-l*L-u*U,A[i])
    for (i = 0; i < 3; i++)
    {
        fP = rkFrustum.LBound()*Math::Abs(afNBpUC[i]) +
             rkFrustum.UBound()*Math::Abs(afNApLC[i]);
        fTmp = -rkFrustum.DMin()*afLBmUA[i];
        fMin = fTmp - fP;
        if ( fMin < 0.0 )
            fMin *= rkFrustum.GetDRatio();
        fMax = fTmp + fP;
        if ( fMax > 0.0 )
            fMax *= rkFrustum.GetDRatio();
        fDdD = -afD[0]*afNBpUC[i] + afD[1]*afNApLC[i] - afD[2]*afLBmUA[i];
        fR = 0.0;
        for (j = 0; j < 3; j++)
        {
            fR += afE[j]*Math::Abs(-afA[j]*afNBpUC[i]+ afB[j]*afNApLC[i]
                - afC[j]*afLBmUA[i]);
        }
        if ( fDdD + fR < fMin || fDdD - fR > fMax )
            return false;
    }

    return true;
}
```