Intersection of a Box and a Cone or Cone Frustum

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: January 4, 2015
Last Modified: November 29, 2018

Contents

1 Introduction

The original version of this document was entitled *Intersection of an Oriented Box and a Cone*. A test-intersection query for the oriented box and cone is based on projection of the box onto a sphere centered at the cone vertex and having radius 1. The cone projects onto a spherical disk and the oriented box projects onto a spherical convex polygon. If the spherical disk and the spherical convex polygon intersect, then the cone and the oriented box intersect. In fact, the projection concept applies even when the box is axis aligned, and it applies to a cone frustum. The original document has been modified to contain the details for both aligned and oriented boxes and for a cone or a cone frustum.

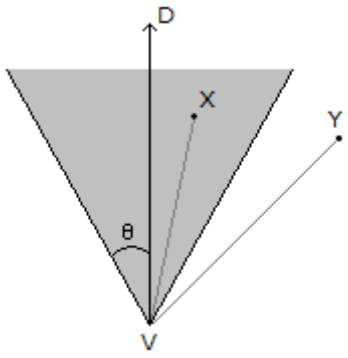
2 Definitions of Cone and Cone Frustum

An *infinite, single-sided, solid cone* has a vertex $\mathbf{V}$, an axis ray whose origin is $\mathbf{V}$ and unit-length direction is $\mathbf{D}$ and an acute cone angle $\theta \in (0, \pi/2)$. A point $\mathbf{X}$ is inside the cone when $\mathbf{X} = \mathbf{V}$ or when the angle between $\mathbf{D}$ and $\mathbf{X} - \mathbf{V}$ is in $[0, \theta]$. Algebraically, the containment is defined by

$$F(\mathbf{X}) = \mathbf{D} \cdot \frac{(\mathbf{X} - \mathbf{V})}{|\mathbf{X} - \mathbf{V}|} - \cos(\theta) \geq 0 \tag{1}$$

where the left-most equality defines the function F. The point is on the cone when it is the vertex or when the inequality of equation (1) is replaced by an equality. Figure 1 shows a 2D cone, which is sufficient to illustrate the quantites in 3D.

Figure 1. A 2D view of a single-sided cone. $\mathbf{X}$ is inside the cone and $\mathbf{Y}$ is outside the cone.



Because of the constraint on θ, both $\cos(\theta)$ and $\sin(\theta)$ are positive. The single-sided cone is *finite* when you specify a maximum height $h_{\max} > 0$ measured along the cone axis, in which case $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) \leq h_{\max}$. A *cone frustum* occurs when you specify a minimum height $h_{\min}$ and a maximum height $h_{\max}$ with $0 \leq h_{\min} < h_{\max}$. You may think of the infinite single-sided cone having height $h = +\infty$. The *supporting plane* for the single-sided cone is $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) = 0$. Any candidates for containment in the cone must satisfy the condition $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) \geq 0$. The algorithm described in this document also applies to an infinite cone that is truncated to minimum height. In a sense, this is an infinite cone frustum. The classifications based on minimum

height $h_{\min}$ and maximum height $h_{\max}$ with $0 \le h_{\min} < h_{\max} \le \infty$ are: infinite cone where $h_{\min} = 0$ and $h_{\max} = \infty$, infinite cone frustum where $h_{\min} > 0$ and $h_{\max} = \infty$ and cone frustum where $h_{\min} \ge 0$ and $h_{\max} < \infty$.

3 Definitions of Oriented Box and Aligned Box

An *oriented box* has a center $\mathbf{C}$, unit-length axis directions $\mathbf{U}_i$ that form a right-handed orthonormal set and extents $e_i > 0$ that measure half the edge length in each dimension i with $0 \le i \le 2$. A point $\mathbf{P}$ in the box may be written as

$$\mathbf{P} = \mathbf{C} + \sum_{i=0}^{2} x_i \mathbf{U}_i \tag{2}$$

where $x_i = \mathbf{U}_i \cdot (\mathbf{P} - \mathbf{C})$ and $|x_i| \le e_i$. In our implementation, the box corners are

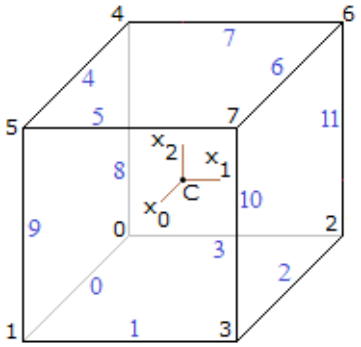
$$\mathbf{K}_j = \mathbf{C} + \sum_{i=0}^{2} \sigma_{ij} e_i \mathbf{U}_i \tag{3}$$

for $0 \le j < 8$ and for

$$\sigma_{ij} = \begin{cases} +1, & j\&(1 \ll i) \neq 0 \\ -1, & j\&(1 \ll i) = 0 \end{cases} \tag{4}$$

That is, σ_{ij} is 1 when bit i of j is 1; it is -1 when bit i of j is 0. Figure 2 shows a 3D box in the coordinate system of the x_i.

Figure 2. A 3D box in the coordinate system of the x_i. The j-indices of the corners $\mathbf{K}_j$ are shown in the figure with black numbers. The 12 edges are labeled with blue numbers.



An *aligned box* is defined by a minimum point $\mathbf{a}$ and a maximum point $\mathbf{b}$. A point $\mathbf{X}$ in the box satisfies the conditions $\mathbf{a} \le \mathbf{X} \le \mathbf{b}$, where the comparisons are componentwise. The aligned box can be transformed to an oriented box by choosing $\mathbf{C} = (\mathbf{b}+\mathbf{a})/2$ and extents $\mathbf{e} = (\mathbf{b}-\mathbf{a})/2$. The axis directions are $\mathbf{U}_0 = (1,0,0)$, $\mathbf{U}_1 = (0,1,0)$ and $\mathbf{U}_2 = (0,0,1)$.

The algorithm described here requires labeling of vertices, edges and faces to support simple graph operations. The vertices are labeled with integers v with $0 \le v \le 7$, as shown in Figure 2. The edges are labeled as shown in Table 1.

Table 1. The edges and labels for a box. The edges are unordered, so they are listed with minimum vertex label first and maximum vertex label second.

label	0	1	2	3	4	5	6	7	8	9	10	11
edge	$\{0,1\}$	$\{1,3\}$	$\{2,3\}$	$\{0,2\}$	$\{4,5\}$	$\{5,7\}$	$\{6,7\}$	$\{4,6\}$	$\{0,4\}$	$\{1,5\}$	$\{3,7\}$	$\{2,6\}$

The faces are labeled as shown in Table 2. The vertex labels are listed so that the first label is the minimum of all labels for that face. The remaining labels are counterclockwise ordered based on viewing the face from outside the box. The edge labels are for pairs of consecutive vertices and the edges are counterclockwise ordered based on viewing the face from outside the box.

Table 2. The faces and labels for a box.

label	0	1	2	3	4	5
vertices	$\{0,4,6,2\}$	$\{1,3,7,5\}$	$\{0,1,5,4\}$	$\{2,6,7,3\}$	$\{0,2,3,1\}$	$\{4,5,7,6\}$
edges	$\{8,7,11,3\}$	$\{1,10,5,9\}$	$\{0,9,4,8\}$	$\{11,6,10,2\}$	$\{3,2,1,0\}$	$\{4,5,6,7\}$

4 Test-Intersection Query by Projection

A set S is said to be *convex* if for any pair of points in S, the segment connecting them is in S. That is, if $\mathbf{X}_0 \in S$ and $\mathbf{X}_1 \in S$, then $(1-t)\mathbf{X}_0 + t\mathbf{X}_1 \in S$ for all $t \in [0,1]$. In 3-dimensional space, we can classify a convex object as *linear*, *planar* or *volumetric*. The linear convex objects are lines, rays and segments. A planar convex object lives in a plane and has intrinsic dimension 2; for example, a plane is convex (and infinite) and a triangle is a planar convex object. A volumetric object has intrinsic dimension 3; for example, a sphere, a cone, and an oriented box are all volumetric convex objects when viewed as solids.

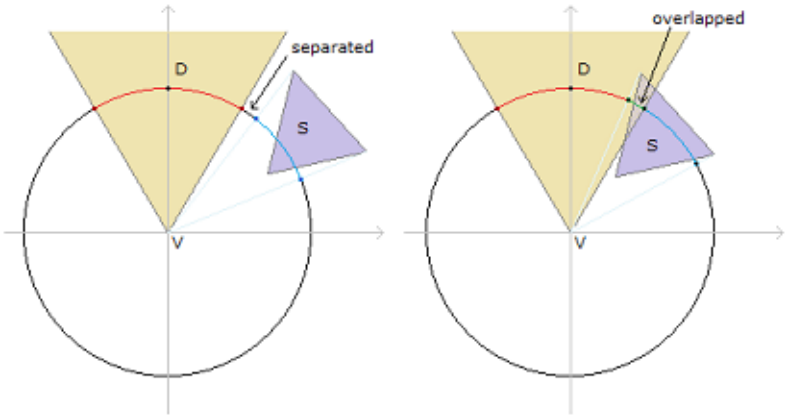
We want to formulate *test-intersection* queries that determine whether a single-sided cone and a convex object overlap. The typical computer graphics application is lighting of objects by a spot light that is modeled as a cone. An infinite cone is used when there is no attenuation of the light; otherwise, a finite cone is used. Because the lighting calculations are expensive, a broad-phase culling is applied first to eliminate objects that are not visible to the spot light. Each object has an easy-to-update convex bounding volume. The broad-phase culling rejects objects whose bounding volumes are not intersected by the infinite cone.

The abstract problem reduces to the following. The left-hand side of equation (1) has a dot product of two unit-length vectors which is $\cos(\phi)$ where ϕ is the angle between the two vectors. In the context of the cone definition, the comparison $\cos(\phi) \ge \cos(\theta) > 0$ is equivalent to $0 < \phi \le \theta$. The normalization of $\mathbf{X} - \mathbf{V}$ is a projection of the difference onto a unit sphere. The problem is formulated in Euclidean geometry, but the projection converts the problem to spherical geometry. The dot product of two points on the unit sphere is the great-circle distance between the two points, which is the angle between the points.

If S is a convex set of points that represent the object, project those points onto the unit sphere whose center is $\mathbf{V}$. The solid cone itself projects onto a spherical disk whose spherical radius is θ. The cone and convex object intersect when the two projections overlap. To determine there is overlap, we need only find one point in the overlap set. Equivalently as a spherical distance problem, we need only find one point in the projection of the convex object that is within a distance θ of the spherical point corresponding to the cone axis direction. This is a fancy way of stating that we need to find $\mathbf{X} \in S$ whose projection $(\mathbf{X}-\mathbf{V})/|\mathbf{X}-\mathbf{V}|$ forms an angle with $\mathbf{D}$ by an angle no larger than θ; however, the formulation as a projection allows us to derive an algorithm to locate the point $\mathbf{A}$ whose projection $(\mathbf{A}-\mathbf{V})/|\mathbf{A}-\mathbf{V}|$ is closest to $\mathbf{D}$. This amounts to computing $\mathbf{A}$ that maximizes the dot product of the projection with $\mathbf{D}$.

Figure 3 illustrates separation and overlap of the projections in 2D.

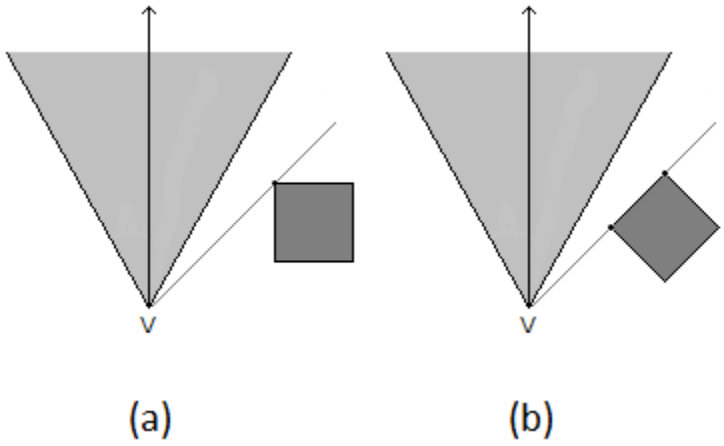
Figure 3. The projection of cone and convex object onto the unit circle centered at $\mathbf{V}$. The left image shows that the circular arcs of projection are separated, so the cone and convex object do not intersect. The right image shows that the circular arcs of projection are overlapped, so the cone and convex object do intersect.



An important observation for an infinite cone is the following. If the cone axis intersects the convex object, then the object and cone intersect.

If the cone axis does not intersect the convex object, the point $\mathbf{P}$ of the object that minimizes the angle between $\mathbf{P}-\mathbf{U}$ and the cone axis is on the boundary of the object. If the object is a convex polyhedron, $\mathbf{P}$ is either a vertex or an edge-interior point or a face-interior point. If it is a face-interior point, then there are infinitely many points that attain the minimum angle; the cone axis must be parallel to the face. Algebraically, $\mathbf{D}\cdot(\mathbf{P}-\mathbf{V})/|\mathbf{P}-\mathbf{V}|$ is the maximum dot product. The ray that generates the infinitely many points attaining maximum is $\mathbf{X}(t) = \mathbf{V}+t(\mathbf{P}-\mathbf{V})/|\mathbf{P}-\mathbf{V}|$, so $\mathbf{D}\cdot(\mathbf{X}(t)-\mathbf{V})/|\mathbf{X}(t)-\mathbf{V}| = \mathbf{D}\cdot(\mathbf{P}-\mathbf{V})/|\mathbf{P}-\mathbf{V}|$; therefore, all the points on the ray attain maximum dot product. Two of the points on the ray must lie on an edge of the polyhedron. Therefore, we can restrict our search for a point attaining minimum angle to edges of the convex polyhedron. This avoids having to compute explicitly the spherical polygon of projection, instead searching relevant edges of the convex polyhedron. Figure 4 shows two configurations, one with a unique maximizer and on with infinitely many maximizers.

Figure 4. A 2-dimensional illustration of the points that maximize $F(\mathbf{X})$. Configuration (a) shows a box for which the maximizer is a single point. Configuration (b) shows a box for which there are infinitely many maximizers.



5 Intersection with an Oriented Box

The algorithm involves several tests that are inexpensive to compute but that lead to either a quick rejection (no intersection) or quick acceptance (intersection). The last paragraph of the previous section describes the motivation for these. The oriented box is a convex polyhedron. When testing for intersection with an infinite cone, we can process the box as a whole. However, when the cone is truncated by one or both of $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) = h_{\min}$ and $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) = h_{\max}$, the box must be clipped against the relevant planes before searching for a point that minimizes the angle with the cone axis. The clipped box is a convex polyhedron and can be viewed as the intersection of the box with an *infinite slab* bounded by the two planes. The steps of the next section must be applied in the order they appear.

5.1 Quick Rejectance Test: Box Outside the Slab

The infinite cone projects onto the cone axis in a set of points $\mathbf{Q} = \mathbf{V} + h\mathbf{D}$, where $h \in [h_{\min}, h_{\max}] \subset [0, +\infty)$. The projection of point $\mathbf{P}$ is $h = \mathbf{D} \cdot (\mathbf{P} - \mathbf{V})$. The box projection onto the cone axis also results in an interval of h-values, say $[b_{\min}, b_{\max}] \subset \mathbb{R}$. If the two projection intervals do not overlap in an interval of positive length, then the box and cone do not intersect in a region of positive volume.

The projection of a box point $\mathbf{P}$ onto the cone axis has projection scalar

$$h = \mathbf{D} \cdot (\mathbf{C} - \mathbf{V}) + \sum_{i=0}^{2} x_i \mathbf{D} \cdot \mathbf{U}_i \tag{5}$$

The extreme values occur when $x_i = \sigma_i e_i$ where $|\sigma_i| = 1$. The maximum value of h occurs by choosing

$\sigma_i = \text{Sign}(\mathbf{D} \cdot \mathbf{U}_i)$. The minimum value of h occurs by choosing $\sigma_i = -\text{Sign}(\mathbf{D} \cdot \mathbf{U}_i)$. Therefore,

$$b_{\min} = \mathbf{D} \cdot (\mathbf{C} - \mathbf{V}) - \sum_{i=0}^{2} e_i |\mathbf{D} \cdot \mathbf{U}_i|, \quad b_{\max} = \mathbf{D} \cdot (\mathbf{C} - \mathbf{V}) + \sum_{i=0}^{2} e_i |\mathbf{D} \cdot \mathbf{U}_i| \tag{6}$$

The box and cone do not intersect in a region of positive volume when $b_{\min} \geq h_{\max}$ or $b_{\max} \leq h_{\min}$. Listing 1 shows pseudocode for computing the box height interval.

Listing 1. Pseudocode for computing the height interval of projection of the box onto the cone axis. The code is shown for an aligned box. The GTEngine implementation for the test-intersection query of an oriented box and cone transforms space so that the oriented box is aligned, after which the code of this listing is used.

```
void ComputeBoxHeightInterval(AlignedBox3 box, Cone3 cone, Real& boxMinHeight, Real& boxMaxHeight)
{
    Vector3 C = (box.max + box.min) / 2;
    Vector3 e = (box.max - box.min) / 2;
    Vector3 CmV = C - cone.V;
    Real DdCmV = Dot(cone.D, CmV);
    Real radius = e[0] * abs(cone.D[0]) + e[1] * abs(cone.D[1]) + e[2] * abs(cone.D[2]);
    boxMinHeight = DdCmV - radius;
    boxMaxHeight = DdCmV + radius;
}
```

5.2 Quick Acceptance Test: Cone Axis Intersects the Box

After the quick-rejectance test, we can test whether the relevant linear component of the cone axis intersects the box. If it does, the test-intersection query is complete. The linear component of the cone axis for an infinite cone or infinite cone frustum is the ray $\mathbf{V} + h\mathbf{D}$ for $h \in [h_{\min}, +\infty)$. For a finite cone frustum, the linear component is the segment $\mathbf{V} + h\mathbf{D}$ for $h \in [h_{\min}, h_{\max}]$. Therefore, we need to execute a test-intersection query for ray-box or for segment-box. These are described in Intersection of a Line and a Box. Listing 2 shows pseudocode for this test.

Listing 2. Pseudocode for testing whether the relevant linear component of the cone axis intersects the box. The code is shown for an aligned box. The GTEngine implementation for the test-intersection query of an oriented box and cone transforms space so that the oriented box is aligned, after which the code of this listing is used.

```
bool ConeAxisIntersectsBox(AlignedBox3 box, Cone3 cone)
{
    if (cone.maxHeight < infinity)
    {
        Segment3 segment;
        segment.P0 = cone.V + cone.minHeight * cone.D;
        segment.P1 = cone.V + cone.maxHeight * cone.D;
        if (SegmentIntersectsBox(segment, box))
        {
            return true;
```

```
        }
    }
    else
    {
        Ray3 ray;
        ray.P = cone.V + cone.minHeight * cone.D;
        ray.D = cone.D;
        if (RayIntersectsBox(ray, box))
        {
            return true;
        }
    }
    return false;
}
```

5.3 Box Fully Inside the Slab

If the quick-rejectance and quick-acceptance tests are inconclusive, then we next test whether the box is fully contained in the slab defined by the planes $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) = h_{\min}$ and $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V}) = h_{\max}$. In this case, the box does not have to be clipped against the planes and its edges can be processed fully. The box-inside-slab test uses information already computed, namely, the projection interval of the box onto the cone axis. The box is inside the slab when $h_{\min} \le b_{\min}$ and $b_{\max} \le h_{\max}$.

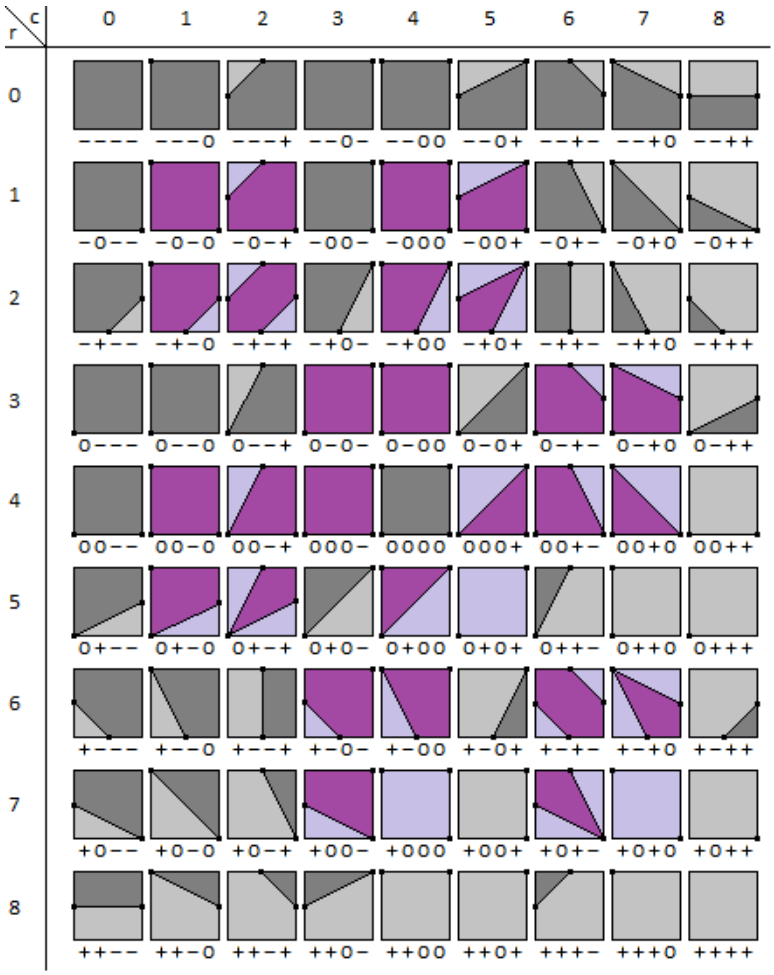
5.4 Box Straddles at Least One Slab Plane

At this point the box is intersected by one of the planes of the slab, and the intersection must be a convex polyhedron with positive volume. We know the relevant linear component of the cone axis does not intersect the polyhedron, so a polyhedron point that achieves the maximum dot product $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V})/|\mathbf{X} - \mathbf{V}|$ must occur on a polyhedron face. As noted previously, such a point can be found on a polyhedron edge.

A straightforward approach to finding the maximizer is to clip the box against the slab planes. This requires some data structures and graph algorithms to compute the convex polyhedron of intersection. However, we only need to know the edges and subedges of the box that lie inside the slab. We can use an algorithm similar to Marching Cubes that finds plane-edge intersections, but we process only box faces one at a time. The scalar values at the vertices of the box are signed h-distances to a plane. The underlying continuous function is linear, as compared to general Marching Cubes where the function is bilinear on a box face. Moreover, we allow for zero-valued h-distances at the vertices, so there are 3 possible sign values for each of 4 vertices of the face, so it turns out that the table has 81 entries. We can use the table to locate the plane-edge intersections and partition the corresponding edges into subedges that must be searched for the maximizer.

Consider the process corresponding to the $h_{\max}$ plane. A box vertex $\mathbf{K}$ has $h = \mathbf{D} \cdot (\mathbf{K} - \mathbf{V})$ and signed h-distance $d = h - h_{\max}$. A positive d indicates $\mathbf{K}$ is strictly outside the slab, a zero d indicates $\mathbf{K}$ is on the plane and a negative d indicates $\mathbf{K}$ is strictly inside the slab. The table of possible sign configurations is shown in Figure 5.

Figure 5. The 81 possible sign configurations for a box face.



The mapping from a table entry (r, c) to a 1-dimensional array index is $i = c + 9r$. Each entry has a list of 4 signs, $s_0s_1s_2s_3$ that correspond to the 4 face vertices $v_0v_1v_2v_3$ listed in counterclockwise order starting with the lower-left corner of the square shown in the entry. For example, the row 3 and column 2 entry has signs $0 - -+$. The vertex v_0 is the lower-left corner and has sign 0. The vertex v_1 is the lower-right corner and has sign -1. The vertex v_2 is the upper-right corner and has sign -1. The vertex v_3 is the upper-left corner and has sign $+1$. The vertices or edge points where the h-distance is 0 are marked as black dots. Segments connecting the black dots have constant h-distance of 0. The dark gray portions of the faces have negative h-distances and are part of the convex polyhedron obtained by clipping the box. The light gray portions of the faces have positive h-distances and are not part of the convex polyhedron.

The table has 30 entries that are drawn in shades of violet. With real-valued arithmetic (exact arithmetic), these configurations are not possible. For example, in row 1 and column 2, two diagonally opposite vertices have h-distances of 0; that is, those vertices are on the plane. The other two vertices have negative h-

distances, so they lie below the plane. This is not possible because the plane contains the line through vertices v_1 and v_3. The plane can be completely determined by one more vertex. If that vertex is v_0 where the h-distance is negative, the flatness of the plane forces the h-distance at v_2 to be positive. The pattern of signs contradicts this fact. When computing with floating-point arithmetic and its inherent rounding errors, the configuration is possible because of the errors. The likelihood that the plane is nearly coincident with the face is high, so in an implementation we can be conservative and specify that the face should not have any clipping applied. The face of row 1 and column 1 becomes part of the convex polyhedron; the face of row 7 and column 7 also becomes part of the convex polyhedron even though two vertices have positive h-distances. The idea is that the four h-distances of the vertices are nearly 0, so just treat them as if they are all zero.

The invalid configurations drawn as shades of violet occur because of floating-point rounding errors. However, it is not possible to say that a valid configuration computed with floating-point arithmetic represents the true configuration computed with exact arithmetic. For example, the entry in row 0 and column 6 has signs $--+-$. Without an analysis of the magnitudes of the h-distances, it might or might not be the case that rounding errors have changed the classification from its theoretical value.

We can process the $h_{\min}$ similarly, except that in order to tag the box points inside the slab as having negative h-distances regardless of the plane we are processing, we choose the signed h-distance to be $d = h_{\min} - h$.

The two planes can be processed independently. The list of candidate subedges that might contain the maximizer of the dot products is initially empty. We could partition box edges a plane at a time, but when both planes intersect the same box edge, we have a maximum of 3 subedges for which at most 1 subedge becomes a candidate. A simple vertex-edge graph data structure could be chosen to represent the clipped box edges, but the complexity of managing the graph can be avoided by storing a fixed-size array of points: the 8 vertices of the box, the (potential) 12 edge-interior points obtained by intersection with the $h_{\min}$ plane and the (potential) 12 edge-interior points obtained by intersection with the $h_{\max}$ plane. For a single edge, we determine whether or not it is intersected by 1 plane, 2 planes or neither plane, storing any points of intersection in the preallocated array. Simple constant arrays can be stored to associate edge-interior point indices with the appropriate edges and to associate edges with the appropriate faces; see Tables 1 and 2. Pseudocode for the processing is shown in Listing 3.

Listing 3. The processing of box edges to determine box-plane intersections at edge-interior point.

```
void ComputeCandidatesOnBoxEdges(Cone3 cone, Vector3 vertices[32], Edge edges[12],
    Real pmin[8], Real pmax[8], int& numCandidates, Edge candidates[])
{
    for (int i = 0; i < 8; ++i)
    {
        Real h = Dot(cone.D, vertices[i]);
        pmin[i] = cone.minHeight - h;
        pmax[i] = h - cone.maxHeight;
    }

    int v0 = 8;   // starting index for edge-interior points generated by hmin plane
    int v1 = 20;  // starting index for edge-interior points generated by hmax plane
    for (int i = 0; i < 12; ++i, ++v0, ++v1)
    {
        Edge edge = edges[i];

        // Process the hmin-plane.
        Real p0Min = pmin[edge[0]];
        Real p1Min = pmin[edge[1]];
        bool clipMin = (p0Min < 0 and p1Min > 0) or (p0Min > 0 and p1Min < 0);
        if (clipMin)
```

```
        {
            vertices[v0] = (p1Min * vertices[edge[0]] - p0Min * vertices[edge[1]]) / (p1Min - p0Min);
        }

        // Process the hmax-plane.
        Real p0Max = mProjectionMax[edge[0]];
        Real p1Max = mProjectionMax[edge[1]];
        bool clipMax = (p0Max < 0 and p1Max > 0) or (p0Max > 0 and p1Max < 0);
        if (clipMax)
        {
            vertices[v1] = (p1Max * vertices[edge[0]] - p0Max * vertices[edge[1]]) / (p1Max - p0Max);
        }

        // Get the candidate edges that are contained by the box edges.
        if (clipMin)
        {
            if (clipMax)
            {
                InsertEdge(numCandidates, candidates, v0, v1);
            }
            else
            {
                if (p0Min < 0)
                {
                    InsertEdge(numCandidates, candidates, edge[0], v0);
                }
                else  // p1Min < 0
                {
                    InsertEdge(numCandidates, candidates, edge[1], v0);
                }
            }
        }
        else if (clipMax)
        {
            if (p0Max < 0)
            {
                InsertEdge(numCandidates, candidates, edge[0], v1);
            }
            else  // p1Max < 0
            {
                InsertEdge(numCandidates, candidates, edge[1], v1);
            }
        }
        else
        {
            // No clipping has occurred.  If the edge is inside the box,
            // it is a candidate edge.  To be inside the box, the p*min
            // and p*max values must be nonpositive.
            if (p0Min <= 0 and p1Min <= 0 and p0Max <= 0 and p1Max <= 0)
            {
                InsertEdge(numCandidates, candidates, edge[0], edge[1]);
            }
        }
    }
}
```

The first 8 incoming vertices are the box corners minus the cone vertex. The pmin and pmax arrays are the h-distances for the two planes. These are used later in the query when connecting edges by segments, which are intersections of the planes with the box faces. The outputs numCandidates and candidates are the subedges identified as potentially containing the maximizer. The type Edge is a pair of indices into vertices that represent an edge connecting the corresponding points. The function InsertEdge(v0,v1) inserts the Edge represented by the two vertices into candidates and increments numCandidates by 1.

Now the box faces must be processed to find edges on the face that are candidates to contain the maximizer.

The table lookup is based on Figure 5. An array of 81 function pointers are stored, one per table entry. Many of the functions have empty bodies because no edges are added for a face. If a box edge has an edge-interior point of intersection with the plane, the function determines the subedge that is a candidate for containment of the maximizer and then inserts the edge into the candidates array. The invalid configurations (shown in shades of violet in the table) have functions that include subedges of the box edges regardless of the signs at the vertices. The idea is that when an invalid configuration occurs, we assume that the entire face is nearly parallel to the plane, in which case all edges of that face are candidates. Pseudocode for the processing is shown in Listing 4.

Listing 4. The processing of box faces to determine box-plane intersections on the faces.

```
void ComputeCandidatesOnBoxFaces(Vector3 vertices[32], Face faces[6], Real pmin[8], Real pmax[8],
    int& numCandidates, Edge candidates[])
{
    Real p0, p1, p2, p3;
    int b0, b1, b2, b3, index;
    for (int i = 0; i < 6; ++i)
    {
        Face face = faces[i];

        // Process the hmin-plane.
        p0 = pmin[face.v[0]];
        p1 = pmin[face.v[1]];
        p2 = pmin[face.v[2]];
        p3 = pmin[face.v[3]];
        b0 = (p0 < 0 ? 0 : (p0 > 0 ? 2 : 1));
        b1 = (p1 < 0 ? 0 : (p1 > 0 ? 2 : 1));
        b2 = (p2 < 0 ? 0 : (p2 > 0 ? 2 : 1));
        b3 = (p3 < 0 ? 0 : (p3 > 0 ? 2 : 1));
        index = b3 + 3 * (b2 + 3 * (b1 + 3 * b0));
        // 8 is the starting index for hmin plane intersections
        configuration[index](vertices, numCandidates, candidates, 8, face);

        // Process the hmax-plane.
        p0 = pmax[face.v[0]];
        p1 = pmax[face.v[1]];
        p2 = pmax[face.v[2]];
        p3 = pmax[face.v[3]];
        b0 = (p0 < 0 ? 0 : (p0 > 0 ? 2 : 1));
        b1 = (p1 < 0 ? 0 : (p1 > 0 ? 2 : 1));
        b2 = (p2 < 0 ? 0 : (p2 > 0 ? 2 : 1));
        b3 = (p3 < 0 ? 0 : (p3 > 0 ? 2 : 1));
        index = b3 + 3 * (b2 + 3 * (b1 + 3 * b0));
        // 20 is the starting index for hmax plane intersections
        configuration[index](vertices, numCandidates, candidates, 20, face);
    }
}
```

The Face type stores the 4 vertex indices for the points forming the face. It also stores the 4 indices into the edges[12] array for the box edges that bound the face. The association among the vertices, edges and faces depends on how you label the vertices. Each configuration function uses face.v indices and face.e indices to determine the edge pairs that are inserted into the candidates array.

5.5 Searching the Subedges for the Dot-Product Maximizer

Once the array of candidate edges is computed, we can search those edges for a point that maximizes $F(\mathbf{P}) = \mathbf{D} \cdot \mathbf{P}/|\mathbf{P}| - \cos(\theta)$. Note that when the search becomes active, the edges (edge endpoints) $\mathbf{X}$ have been translated by the cone vertex; that is, $\mathbf{P} = \mathbf{X} - \mathbf{V}$. During the processing of the edges, if a maximum of F on an edge is positive, we know the box intersects the cone in a region of positive volume. The search terminates, because we do not actually care about the actual maximum—only that we found a box point inside the cone.

For a single candidate edge with endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$, we first test whether $F(\mathbf{P}_0) > 0$ or $F(\mathbf{P}_1) > 0$. If one of these conditions is true, an edge endpoint is inside the cone; the box and cone intersect.

If both endpoints are outside or on the cone ($F \le 0$ for both endpoints), we can search for an interior local maximum of $\phi(t) = F(\mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0))$ for $t \in (0, 1)$. Define $\mathbf{E} = \mathbf{P}_1 - \mathbf{E}_0$. The local maxima occur when

$$\phi'(t) = \frac{(\mathbf{P}_0 \times \mathbf{D} \cdot \mathbf{P}_0 \times \mathbf{E}) + t(\mathbf{P}_1 \times \mathbf{D} \cdot \mathbf{P}_0 \times \mathbf{E})}{|\mathbf{P}_0 + t\mathbf{E}|^3} = 0 \tag{7}$$

The numerator is a linear function of t, so the local maximum is unique if it exists. The local maximum exists when $\phi'(0) > 0$ and $\phi'(1) < 0$. Listing 5 has pseudocode for the search.

Listing 5. Pseudocode for searching a candidate edge for a point that is inside the cone. To avoid the division by $|\mathbf{P}|$ in $F(\mathbf{P})$ in case the input is zero (or nearly zero), instead the code tests for positivity of the function $G(\mathbf{P}) = \mathbf{D} \cdot \mathbf{P} - (\cos\theta)|\mathbf{P}|$.

```
bool CandidatesHavePointInsideCone(Cone3 cone, Vector3 vertices[32], int numCandidates, Edge candidates[])
{
    for (int i = 0; i < numCandidates; ++i)
    {
        Edge edge = candidates[i];
        Vector3 P0 = vertices[edge[0]];
        Vector3 P1 = vertices[edge[1]];
        if (HasPointInsideCone(P0, P1, cone))
        {
            return true;
        }
    }
    return false;
}

bool HasPointInsideCone(Vector3 P0, Vector3 P1, Cone3 cone)
{
    // Test whether P0 or P1 is inside the cone.
    Real g = Dot(cone.D, P0) - cone.cosAngle * Length(P0);
    if (g > 0)
    {
        // X0 = P0 + V is inside the cone.
        return true;
    }

    g = Dot(cone.D, P1) - cone.cosAngle * Length(P1);
    if (g > 0)
    {
        // X1 = P1 + V is inside the cone.
        return true;
    }

    // Test whether an interior edge point is inside the cone.
    Vector3 E = P1 - P0;
```

```
    Vector3 crossP0U = Cross(P0, cone.D);
    Vector3 crossP0E = Cross(P0, E);
    Real dphi0 = Dot(crossP0E, crossP0U);
    if (dphi0 > 0)
    {
        Vector3 crossP1U = Cross(P1, cone.D);
        Real dphi1 = Dot(crossP0E, crossP1U);
        if (dphi1 < 0)
        {
            Real t = dphi0 / (dphi0 - dphi1);
            Vector3 PMax = P0 + t * E;
            g = Dot(cone.D, PMax) - cone.cosAngle * Length(PMax);
            if (g > 0)
            {
                // The edge point XMax = Pmax + V is inside the cone.
                return true;
            }
        }
    }

    return false;
}
```

5.6 The Query Implementation

Listing 6 contains pseudocode for the test-intersection query between an aligned box and a cone.

Listing 6. The main body of the test-intersection query between a cone and box. It is assumed that the edge and face data structures of Tables 1 and 2 are available globally to the functions called herein. Their names are globalEdges and globalFaces.

```
Edge globalEdges[12];
globalEdges[ 0] = { 0, 1 };
globalEdges[ 1] = { 1, 3 };
globalEdges[ 2] = { 2, 3 };
globalEdges[ 3] = { 0, 2 };
globalEdges[ 4] = { 4, 5 };
globalEdges[ 5] = { 5, 7 };
globalEdges[ 6] = { 6, 7 };
globalEdges[ 7] = { 4, 6 };
globalEdges[ 8] = { 0, 4 };
globalEdges[ 9] = { 1, 5 };
globalEdges[10] = { 3, 7 };
globalEdges[11] = { 2, 6 };

Face globalFaces[6];
globalFaces[0] = { { 0, 4, 6, 2 }, {  8,  7, 11,  3 } };
globalFaces[1] = { { 1, 3, 7, 5 }, {  1, 10,  5,  9 } };
globalFaces[2] = { { 0, 1, 5, 4 }, {  0,  9,  4,  8 } };
globalFaces[3] = { { 2, 6, 7, 3 }, { 11,  6, 10,  2 } };
globalFaces[4] = { { 0, 2, 3, 1 }, {  3,  2,  1,  0 } };
globalFaces[5] = { { 4, 5, 7, 6 }, {  4,  5,  6,  7 } };

bool TestIntersection(AlignedBox3 box, Cone3 cone)
{
    // Quick-rejectance test.  Determine whether the box is outside the slab bounded by the
    // minimum and maximum height planes.  When outside the slab, the box vertices are not
    // required by the cone-box intersection query, so the vertices are not yet computed.
    Real boxMinHeight = 0, boxMaxHeight = 0;
    ComputeBoxHeightInterval(box, cone, boxMinHeight, boxMaxHeight);
```

```
    if (boxMaxHeight <= cone.minHeight or boxMinHeight >= cone.maxHeight)
    {
        // There is no volumetric overlap of the box and the cone. The box is clipped entirely.
        result.intersect = false;
        return result;
    }

    // Quick-acceptance test.  Determine whether the cone axis intersects the box.
    if (ConeAxisIntersectsBox(box, cone))
    {
        result.intersect = true;
        return result;
    }

    // We need the box corners relative to the cone vertex from this point on.
    Vector3 vertices[32];
    vertices[0] = { box.min[0], box.min[1], box.min[2] } - cone.V;
    vertices[1] = { box.max[0], box.min[1], box.min[2] } - cone.V;
    vertices[2] = { box.min[0], box.max[1], box.min[2] } - cone.V;
    vertices[3] = { box.max[0], box.max[1], box.min[2] } - cone.V;
    vertices[4] = { box.min[0], box.min[1], box.max[2] } - cone.V;
    vertices[5] = { box.max[0], box.min[1], box.max[2] } - cone.V;
    vertices[6] = { box.min[0], box.max[1], box.max[2] } - cone.V;
    vertices[7] = { box.max[0], box.max[1], box.max[2] } - cone.V;

    // Test for box fully inside the slab.
    Edge candidates[32 * 32];
    if (cone.minHeight <= boxMinHeight and boxMaxHeight <= cone.maxHeight)
    {
        // The box is fully inside, so no clipping is necessary.
        for (int i = 0; i < 12; ++i)
        {
            candidates[i] = globalEdges[i];
        }
        return CandidatesHavePointInsideCone(cone, vertices, 12, candidates);
    }

    // Clear the candidates array.  See comments after the listing about the graph
    // implementation that uses an adjacency matrix to support fast clears and
    // edge-in-graph tests.
    int numCandidates = 0;
    ClearCandidates();

    // The box intersects at least one plane.  Compute the box-plane edge-interior intersection
    // points.  Insert the box subedges into the array of candidate edges.
    Real pmin[8], pmax[8];
    ComputeCandidatesOnBoxEdges(cone, vertices, globalEdges, pmin, pmax, numCandidates, candidates);

    // Insert any relevant box face-interior clipped edges into the array of candidate edges.
    ComputeCandidatesOnBoxFaces(vertices, globalFaces, pmin, pmax, numCandidates, candidates);

    return CandidatesHavePointInsideCone(cone, vertices, numCandidates, candidates);
}

bool TestIntersection(OrientedBox3 box, Cone3 cone)
{
    // Transform the cone and box so that the cone vertex is at the
    // origin and the box is aligned.
    Vector3 diff = box.C - cone.V;
    Vector3 xfrmBoxCenter = { Dot(box.U[0], diff), Dot(box.U[1], diff), Dot(box.U[2], diff) };
    AlignedBox3 xfrmBox;
    xfrmBox.min = xfrmBoxCenter - box.e;
    xfrmBox.max = xfrmBoxCenter + box.e;

    Cone3 xfrmCone = cone;
    xfrmCone.minHeight = cone.minHeight;
    xfrmCone.maxHeight = cone.maxHeight;
    xfrmCone.angle = cone.angle;
    xfrmCone.cosAngle = cone.cosAngle;
    for (int i = 0; i < 3; ++i)
    {
        xfrmCone.V[i] = 0;
```

```
        xfrmCone.D[i] = Dot(box.U[i], cone.D);
    }

    // Test for intersection between the aligned box and the cone.
    return TestIntersection(xfrmBox, cone);
}
```

The algorithm is designed to maintain an array of subedges that are candidates for containing the maximizer of $\mathbf{D} \cdot (\mathbf{X} - \mathbf{V})/|\mathbf{X} - \mathbf{V}| - \cos(\theta)$. Some of the subedges are on the box edges. These edges are shared by two faces, so we do not want to insert duplicates into the candidates array. The simplest thing to do is iterate over the candidates array and test whether the to-be-inserted edge already exists. If it does not, insert it. If it does, ignore it.

The candidates array is preallocated (on the stack, not on the heap) and has the worst-case size of $496 = \sum_{k=1}^{31} k$. If an edge is in the array, it is either (v0,v1) or (v1,v0) with distinct v0 and v1 but not both. Each query must clear the array, but that is accomplished by simply storing the number of used slots in numCandidates.

The test for edge-in-array requires an iteration over the candidates. To avoid this cost when an application performs a large number of queries, I also maintain an adjacency matrix representation of the graph; call this matrix A. The matrix is size 32×32 and has binary entries that are initially all 0, indicating that no edges are in the graph. When an edge (v0,v1) is to be inserted into the graph, I test for existence of the edge simply by examining A[v0][v1] to see whether it is 1. If it is not, set both A[v0][v1] and A[v1][v0] to 1.

To clear the adjacency matrix quickly and efficiently, iterate over the candidate array. The number of used elements is much smaller than the worst case 496. For each active edge (v0,v1) in the candidate array, set both A[v0][v1] and A[v1][v0] to 0.

The GTEngine files GteIntrAlignedBox3Cone3.h and GteIntrOrientedBox3Cone3.h implement this design. A sample test application is found in GeometricTools/GTEngine/Samples/Geometrics/IntersectBoxCone.