Intersection of Convex Objects: The Method of Separating Axes

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

Created: January 28, 2001
Last Modified: June 20, 2022

Contents

1 Introduction

A set is *convex* if given any two points $\boldsymbol{P}$ and $\boldsymbol{Q}$ in the set, the line segment $(1-t)\boldsymbol{P}+t\boldsymbol{Q}$ for $t \in [0,1]$ is also in the set. In 1 dimension, both $[0,1]$ and $[0,1)$ are convex. Lines, rays and segments are convex in any dimension. In 2 dimensions, the classic examples are polygons whose interior angles are smaller than π radians. Triangles automatically satisfy this definition, so they are convex. Squares and rectangles are convex. A chevron is a quadrilateral with one vertex having interior angle larger than π radians, so it is not convex. A circle is not convex, but a disk that has a circular boundary is convex. In 3 dimensions, any 2-dimensional convex object living in a plane in 3 dimensions remains convex. Polyhedra can be partitioned into two subsets, one consisting of convex polyhedra and the other not convex. A halfspace consists of an unbounded region of points all lying on a plane or on the same side of the plane; these are convex. Solid spheres and ellipsoids are also convex.

Although the ideas in this document can be applied to unbounded convex objects, I restrict attention to the most common cases where the objects are bounded. Moreover, the objects are closed sets. In 1 dimension, the intervals $[0,1)$ and $[0,1]$ are convex and bounded. The interval $[0,1)$ is not closed because the limiting point 1 is not in the set. The interval $[0,1]$ is closed and bounded. The interval $[0,+\infty)$ is closed but not bounded. Generally, a set is *compact* if it is closed and bounded.

This document describes the *method of separating axes*, a method for determining whether two stationary convex objects are intersecting. The ideas can be extended to handle moving convex objects and are useful for predicting collisions of the objects and for computing the first time of contact. The current focus of this document is the *test intersection* geometric query which indicates whether an intersection exists or will occur when the objects are moving. The problem of computing the set of intersection is a *find intersections* geometric query and is generally more difficult to implement than the *test intersection* query. Information from the *test* query can help determine the contact set that the *find* query must construct. This document will describe in what way the *test* query information can be used.

2 Separation by Projection onto a Line

A test for nonintersection of two convex objects is simply stated: If there exists a line for which the intervals of projection of the two objects onto that line do not intersect, then the objects are do not intersect. Such a line is called a *separating line* or, more commonly, a *separating axis*.

The translation of a separating line is also a separating line, so mathematically it is sufficient to consider lines that contain the origin. However, in practice, a line is chosen that contains a point of one of the objects. This is helpful to reduce problems caused by rounding errors when computing with floating-point arithmetic.

Given a line containing the origin and with unit-length direction $\boldsymbol{D}$, the projection of a compact and convex set C onto the line is the interval

$$I = [\lambda_{\min}(\boldsymbol{D}), \lambda_{\max}(\boldsymbol{D})] = [\min\{\boldsymbol{D}\cdot\boldsymbol{X} : \boldsymbol{X} \in C\}, \max\{\boldsymbol{D}\cdot\boldsymbol{X} : \boldsymbol{X} \in C\}] \tag{1}$$

Two compact convex sets C_0 and C_1 are separated if there exists a direction $\boldsymbol{D}$ for which the projection intervals I_0 and I_1 do not intersect. Specifically they do not intersect when

$$\lambda^{(0)}_{\min}(\boldsymbol{D}) > \lambda^{(1)}_{\max}(\boldsymbol{D}) \text{ or } \lambda^{(0)}_{\max}(\boldsymbol{D}) < \lambda^{(1)}_{\min}(\boldsymbol{D}) \tag{2}$$

The superscript corresponds to the index of the convex set. Although the comparisons are made where $\boldsymbol{D}$ is unit length, the comparison results are invariant to changes in length of the vector. This follows from

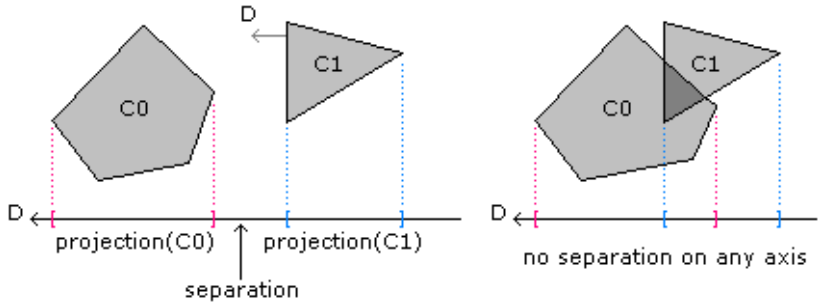
$\lambda_{\min}(t\boldsymbol{D}) = t\lambda_{\min}(\boldsymbol{D})$ and $\lambda_{\max}(t\boldsymbol{D}) = t\lambda_{\max}(\boldsymbol{D})$ for $t > 0$. The Boolean value of the pair of comparisons is also invariant when $\boldsymbol{D}$ is replaced by the opposite direction $-\boldsymbol{D}$. This follows from $\lambda_{\min}(-\boldsymbol{D}) = -\lambda_{\max}(\boldsymbol{D})$ and $\lambda_{\max}(-\boldsymbol{D}) = -\lambda_{\min}(\boldsymbol{D})$. When $\boldsymbol{D}$ is not unit length, the intervals obtained for the separating axis tests are not the projections of the object onto the line, rather they are scaled versions of the projection intervals. I make no distinction in this document between the scaled projection and regular projection. I will also use the terminology that the direction vector for a separating axis is called a *separating direction*, which is not necessarily unit length.

The ideas in this document apply to closed convex sets whether bounded or unbounded, but I restrict the discussion to the common case where the sets are convex polygons or convex polyhedra.

3 Separation of Convex Polygons in 2D

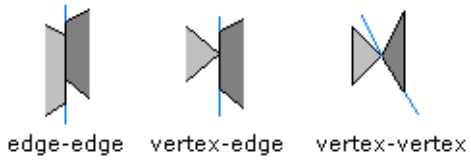
For a pair of convex polygons in 2D, only a finite set of direction vectors needs to be considered for separation tests. That set includes the normal vectors to the edges of the polygons. The left picture in Figure 1 shows two nonintersecting polygons that are separated along a direction determined by the normal to an edge of one polygon. The right picture shows two polygons that intersect; there are no separating directions.

Figure 1. Nonintersecting convex polygons (left). Intersecting convex polygons (right).



The intuition for why only edge normals must be tested is based on having two convex polygons just touching with no interpenetration. Figure 2 shows the three possible configurations: edge-edge contact, vertex-edge contact, and vertex-vertex contact.

Figure 2. Edge-edge contact (left), vertex-edge contact (middle), and vertex-vertex contact.



The blue lines are perpendicular to the separation lines that would occur for the object translated away from the other by an infinitesimal distance.

For $j = 0$ and $j = 1$, let C_j be the convex polygons with vertices $\{\boldsymbol{V}_i^{(j)}\}_{i=0}^{n_j-1}$ that are counterclockwise ordered. The number of vertices for polygon j is n_j. The direct implementation for a separation test for direction $\boldsymbol{D}$ involves computing the extreme values of the projection and compares them. That is, compute $\lambda_{\min}^{(j)}(\boldsymbol{D}) = \min_{0\le i<n_0}\{\boldsymbol{D}\cdot\boldsymbol{V}_i^{(j)}\}$ and $\lambda_{\max}^{(j)}(\boldsymbol{D}) = \max_{0\le i<n_1}\{\boldsymbol{D}\cdot\boldsymbol{V}_i^{(j)}\}$ and test the inequalities in equation 2. The algorithm is potentially slow because all the vertices are projected onto the line and then sorted to determine the extreme projection values.

Instead, we can choose the candidate separating axis to be the line with direction $\boldsymbol{D}$ and that passes through a vertex $\boldsymbol{P}$ of one of the polygons. Moreover, that vertex is an endpoint of an edge that provided the direction $\boldsymbol{D}$, and we can choose $\boldsymbol{D}$ to point to the outside of the polygon. The chosen polygon that contains $\boldsymbol{P}$ is convex, so the projection of that polygon's vertices onto the line $\boldsymbol{P}+t\boldsymbol{D}$ is an interval of the form $[T, 0]$ for some $T < 0$. We can then project the other polygon's vertices onto the line one at a time. If we encounter a vertex for which the projection is $t \le 0$, then the line is not a separating axis, but if all the projections have values $t > 0$, the line is separating and the polygons do not intersect.

This algorithm of the previous paragraph is a reasonable modification when the polygons have a large number of nonparallel edges. For triangles or rectangles, the direct implementation is a better choice. In particular, it is a better choice when the rectangles are represented by a center point, two orthonormal axes, and two half-width measurements; the projection intervals are trivial to compute with this representation. The assumption is that the polygon vertices are listed in counterclockwise order. Given an edge direction (x_0, x_1), an outward pointing normal is $(x_0, x_1)^{\perp} = (x_1, -x_0)$. Listing 1 contains pseudocode for the separating axis tests.

Listing 1. The separating axis tests for convex polygons in 2 dimensions.

```
struct Vector2 { Real x, y; };

// The vertices are listed in counterclockwise order.
struct ConvexPolygon2 { int numVertices; Vector2 vertex[]; };

Vector2 Perp(Vector2 v) { return Vector2(v.y, -v.x); }

// This function is generic in the sense that it the interface ConvexSet<N>
// has a member 'numVertices' and an array 'vertex[]' of N-tuples.
int WhichSide(ConvexSet<N> C, Vector<N> P, Vector<N> D)
{
    // The vertices are projected to the form P + t * D.  The return value is +1 if all t > 0,
    // -1 if all t < 0, but 0 otherwise, in which case the line splits the polygon projection.
    int positive = 0, negative = 0;
    for (int i = 0; i < C.numVertices ++i)
    {
        // Project a vertex onto the line.
        Real t = Dot(D, C.vertex[i] - P);
        if (t > 0)
        {
            ++positive;
        }
        else if (t < 0)
        {
            ++negative;
        }
        if (positive && negative)
        {
            // The polygon has vertices on both sides of the line, so the line is not a separating axis.
            // Time is saved by not having to project the remaining vertices.
            return 0;
        }
    }

    // Either positive > 0 or negative > 0 but not both are positive.
```

```
    return (positive > 0 ? +1 : -1);
}

// The function returns 'true' when the polygons intersect.
bool TestIntersection(ConvexPolygon2 C0, ConvexPolygon2 C1)
{
    // Test edges of C0 for separation.  Because of the counterclockwise ordering, the
    // projection interval for C0 is [T,0] where T < 0.  Determine whether C1 is on the
    // positive side of the line.
    for (int i0 = 0, i1 = C0.numVertices - 1; i0 < C0.numVertices; i1 = i0++)
    {
        Vector2 P = C0.vertex[i0];
        Vector2 D = Perp(C0.vertex[i0] - C0.vertex[i1]);  // outward pointing
        if (WhichSide(C1, P, D) > 0)
        {
            // C1 is entirely on the positive side of the line P + t * D.
            return false;
        }
    }

    // Test edges of C1 for separation.  Because of the counterclockwise ordering, the
    // projection interval for C1 is [T,0] where T < 0.  Determine whether C0 is on the
    // positive side of the line.
    for (int i0 = 0, i1 = C1.numVertices - 1; i0 < C1.numVertices; i1 = i0++)
    {
        Vector2 P = C1.vertex[i0];
        Vector2 D = Perp(C0.vertex[i0] - C0.vertex[i1]);  // outward pointing
        if (WhichSide(C0, P, D) > 0)
        {
            // C0 is entirely on the positive side of the line P + t * D.
            return false;
        }
    }

    return true;
}
```

4 Separation of Convex Polyhedra in 3D

For a pair of convex polyhedra in 3D, only a finite set of direction vectors needs to be considered for separation tests. That set includes the normal vectors to the faces of the polyhedra and vectors generated by a cross product of two edges, one from each polyhedron. The intuition is similar to that of convex polygons in 2D. If the two polyhedra are just touching with no interpenetration, then the contact is one of face-face, face-edge, face-vertex, edge-edge, edge-vertex, or vertex-vertex.

For $j = 0$ and $j = 1$, let C_j be the convex polyhedra with vertices $\{\boldsymbol{V}_i^{(j)}\}_{i=0}^{n_j-1}$, edges $\{\boldsymbol{E}_i^{(j)}\}_{i=0}^{m_j-1}$ and faces $\{\boldsymbol{F}_i^{(j)}\}_{i=0}^{\ell_j-1}$. Let the faces be planar convex polygons whose vertices are counterclockwise ordered as you view the face from outside the polyhedron. Outward pointing normal vectors can be stored with each face as a way of representing the orientation. The pseudocode for 3D that is similar to that in 2D is provided in Listing 2. It is assumed that each face has queries which allow access to the face normal and to a vertex on the face. It is also assumed that each edge has a query that allows access to a vertex on the edge.

Listing 2. The separating axis tests for convex polyhedra in 3 dimensions. The WhichSide function is the one of Listing 1 but for dimension $N = 3$.

```
struct Vector3 { Real x, y, z; }
struct Edge3 { Vector3 vertex[2]; };
```

```
struct Face3 { int numVertices; Vector3 vertex[]; Vector3 normal; };

struct ConvexPolyhedron3
{
    int numVertices, numEdges, numFaces;
    Vector3 vertex[];
    Edge3 edge[];
    Face3 face[];
};

// The function returns 'true' when the polyhedra intersect.
bool TestIntersection3 (ConvexPolyhedron C0, ConvexPolyhedron C1)
{
    // Test faces of C0 for separation.  Because of the counterclockwise ordering, the
    // projection interval for C0 is [T,0] where T < 0.  Determine whether C1 is on the
    // positive side of the line.
    for (int i = 0; i < C0.numFaces; ++i)
    {
        Vector3 P = C0.face[i].vertex[0];
        Vector3 N = C0.face[i].normal;  // outward pointing
        if (WhichSide(C1, P, N) > 0)
        {
            // C1 is entirely on the positive side of the line P + t * N.
            return false;
        }
    }

    // Test faces of C1 for separation.  Because of the counterclockwise ordering, the
    // projection interval for C1 is [T,0] where T < 0.  Determine whether C0 is on the
    // positive side of the line.
    for (int i = 0; i < C1.numFaces; ++i)
    {
        Vector3 P = C1.face[i].vertex;
        Vector3 N = C1.face[i].normal;  // outward pointing
        if (WhichSide(C0, P, N) > 0)
        {
            // C0 is entirely on the positive side of the line P + t * N.
            return false;
        }
    }

    // Test cross products of pairs of edge directions, one edge direction from each polyhedron.
    for (int i0 = 0; i0 < C0.numEdges; ++i0)
    {
        Vector3 D0 = C0.edge[i0].vertex[1] - C0.edge[i0].vertex[0];
        Vector3 P = C0.edge[i0].vertex[0];
        for (int i1 = 0; i1 < C1.numEdges; ++i1)
        {
            Vector3 D1 = C1.edge[i1].vertex[1] - C1.edge[i1].vertex[0];
            Vector3 N = Cross(D0, D1);

            if (N != Vector3(0, 0, 0))
            {
                int side0 = WhichSide(C0, P, N);
                if (side0 == 0)
                {
                    continue;
                }

                int side1 = WhichSide(C1, P, N);
                if (side1 == 0)
                {
                    continue;
                }

                if (side0 * side1 < 0)
                {
                    // The projections of C0 and C1 onto the line P + t * N are on
                    // opposite sides of the projection of P.
                    return false;
                }
            }
```

```
        }
    }

    return true;
}
```

5 Separation of Convex Polygons in 3D

For a pair of convex polygons in 3D, again only a finite set of direction vectors needs to be considered for separation tests. For $j = 0$ and $j = 1$, let C_j be the convex polygons with vertices $\{\boldsymbol{V}_i^{(j)}\}_{i=0}^{m_j-1}$ with the index wrap-around convention $\boldsymbol{V}_{m_j}^{(j)} = \boldsymbol{V}_0^{(j)}$. Each set of vertices is necessarily coplanar. The edges for the polygons are $\boldsymbol{E}_i^{(j)} = \boldsymbol{V}_{i+1}^{(j)} - \boldsymbol{V}_i^{(j)}$ for $0 \le i < m_j$. Let $\boldsymbol{N}^{(j)}$ be normal vectors for those planes, chosen so that when an observer on the side of the plane to which the normal is directed, the observer sees the triangle vertices listed in counterclockwise order.

The polygons normals are potential separating directions. If either normal direction separates the polygons, then no intersection occurs. If neither normal direction separates the polygons, then two possibilities exist for the remaining potential separating directions.

The first case is that the polygons are coplanar—effectively the 2D case—and the remaining potential separating directions are those vectors in the common plane and that are perpendicular to the polygon edges, $\boldsymbol{E}_i^{(j)} \times \boldsymbol{N}^{(j)}$ for $0 \le j \le 1$ and $0 \le i < m_j$, a total of $2m_j$ vectors. By the convention for choosing the normal vectors, each cross product is in the plane of the polygon and points to the outside of the polygon for its corresponding edge.

The second case is that the polygon planes are not parallel and that each polygon is split by the plane of the other polygon. The remaining potential separating directions are the cross products $\boldsymbol{E}_i^{(0)} \times \boldsymbol{E}_j^{(1)}$ for $0 \le i < m_0$ and $0 \le j < m_1$, a total of $m_0 m_1$ vectors.

Listing 3 contains pseudocode for the test-intersection query.

Listing 3. The separating axis tests for convex polygons in 3 dimensions. The function WhichSide is the one defined in Listing 2.

```
struct ConvexPolygon3
{
    int numVertices;
    Vector3 vertex[];
    Vector3 normal;  // ... to the plane of the polygon
};

bool TestIntersection(ConvexPolygon3 C0, ConvexPolygon3 C1)
{
    // Test the normal to the plane of C0 for separation.  The projection
    // of C0 onto the normal line P + t * N produces the degenerate
    // interval [0,0].
    Vector3 P = C0.vertex[0];
    Vector4 N = C0.normal;
    if (WhichSide(C1, P, N) != 0)
    {
        // C1 is entirely on one side of the plane of C0.
        return false;
    }
```

```
// Test the normal to the plane of C1 for separation.  The projection
// of C1 onto the normal line P + t * N produces the degenerate
// interval [0,0].
P = C1.vertex[0];
N = C1.normal;
if (WhichSide(C0, P, N) != 0)
{
    // C0 is entirely on one side of the plane of C1.
    return false;
}

// If the planes of the polygons are parallel but separated, the previous
// code will generate a return (when testing the normal to the plane of C0).
// Therefore, the remaining cases are that the planes are not parallel or
// they are the same plane.  The distinction is made simply be testing
// whether the normals are nonparallel or parallel, respectively.
Vector3 N0xN1 = Cross(C0.normal, C1.normal);
if (N0xN1 != Vector3(0, 0, 0))  // The planes are not parallel.
{
    for (int i0 = 0; i0 < C0.numEdges; ++i0)
    {
        Vector3 D0 = C0.edge[i0].vertex[1] - C0.edge[i0].vertex[0];
        Vector3 P = C0.edge[i0].vertex[0];
        for (int i1 = 0; i1 < C1.numEdges; ++i1)
        {
            Vector3 D1 = C1.edge[i1].vertex[1] - C1.edge[i1].vertex[0];
            Vector3 N = Cross(D0, D1);

            if (N != Vector3(0, 0, 0))
            {
                int side0 = WhichSide(C0, P, N);
                if (side0 == 0)
                {
                    continue;
                }

                int side1 = WhichSide(C1, P, N);
                if (side1 == 0)
                {
                    continue;
                }

                if (side0 * side1 < 0)
                {
                    // The projections of C0 and C1 onto the line P + t * N are on
                    // opposite sides of the projection of P.
                    return false;
                }
            }
        }
    }
}
else  // The polygons are coplanar.
{
    // Test edges of C0 for separation.
    for (int i0 = 0; i0 < C0.numEdges; ++i0)
    {
        Vector3 P = C0.edge[i0].vertex[0];
        Vector3 D = Cross(C0.edge[i0], C0.normal);  // outward pointing
        if (WhichSide(C1, P, D) > 0)
        {
            // C1 is entirely on the positive side of the line P + t * D.
            return false;
        }
    }

    // Test edges of C1 for separation.
    for (int i1 = 0; i1 < C1.numEdges; ++i1)
    {
        Vector3 P = C1.edge[i1].vertex[0];
        Vector3 D = Cross(C1.edge[i1], C1.normal);  // outward pointing
```

```
            if (WhichSide(C0, P, D) > 0)
            {
                // C0 is entirely on the positive side of the line P + t * D.
                return false;
            }
        }
    }

    return true;
}
```

6 Separation of Moving Convex Objects

The method of separating axes can be extended to handle convex objects moving with constant linear velocity and no angular velocity. If C_0 and C_1 are convex objects with linear velocities $\boldsymbol{W}_0$ and $\boldsymbol{W}_1$, then it can be determined via projections whether the objects will intersect for some time $T \geq 0$. Moreover, if they do, the first time of contact can be computed. Without loss of generality, it is enough to work with a stationary object C_0 and a moving object C_1 with velocity $\boldsymbol{W}$ since one can always use $\boldsymbol{W} = \boldsymbol{W}_1 - \boldsymbol{W}_0$ to perform the calculations as if C_0 were not moving.

If the C_0 and C_1 are initially intersecting, then the first time of contact is $T = 0$. The set of intersection is itself convex, but computing the set is complicated depending on the nature of the objects themselves. For example, if C_0 and C_1 are convex polygons in 2 dimensions and that overlap initially, the set of intersection is a convex polygon. However, if C_0 is a convex polygon and C_1 is a solid ellipse that overlap in a region of positive area, the set of intersection has a boundary consisting of line segments and elliptical arcs. For many physics simulations, the objects are initially placed so that they are not overlapping.

Let the convex objects be separated initially. The projection of C_1 onto a line with direction $\boldsymbol{D}$ is an interval that is moving with speed $s = (\boldsymbol{W} \cdot \boldsymbol{D})/|\boldsymbol{D}|^2$. If the projection interval of C_1 moves away from the projection interval of C_0, then the two objects will never intersect. The interesting cases are when the projection intervals for C_1 move towards those of C_0.

The intuition for how to predict an intersection is much like that for selecting the potential separating directions in the first place. If the two convex objects intersect at a first time $T_{\text{first}} > 0$, then their projections are not separated along any line. An instant before first contact, the objects are separated. Consequently there must be at least one separating direction for the objects for $T_{\text{first}} - \varepsilon$ for small $\varepsilon > 0$. Similarly, if the two convex objects intersect at a last time $T_{\text{last}} > 0$, then their projections are also not separated at that time along any line, but an instant after last contact, the objects are separated. Consequently there must be at least one separating direction for the objects for $T_{\text{last}} + \varepsilon$ for small $\varepsilon > 0$. Both T_{first} and T_{last} can be tracked as each potential separating axis is processed. After all directions are processed, if $T_{\text{first}} \leq T_{\text{last}}$, then the two objects intersect with first contact time T_{first}. It is also possible that $T_{\text{first}} > T_{\text{last}}$ in which case the two objects cannot intersect. This algorithm is attributed to Ron Levine in a post to the SourceForge game developer algorithms mailing list [1].

Let S_0 and S_1 denote the set of potential separating directions and let $\boldsymbol{W}_0$ and $\boldsymbol{W}_1$ denote the velocities for C_0 and C_1, respectively. Listing 4 contains pseudocode for testing for intersection of two moving convex objects.

Listing 4. The separating axis tests to determine first time of contact and point (or points) of contact

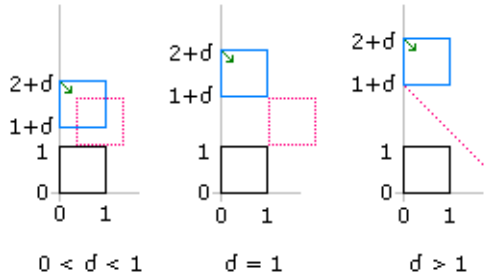
for convex objects moving with constant linear velocities but no angular velocity. The first time of contact tFirst and the last time of contact tLast are valid only when TestIntersection returns true.

```
bool TestIntersection(Convex C0, Convex C1, Real& tFirst, Real& tLast)
{
    Vector W = C1.W - C0.W;  // process as if C0 is stationary, C1 is moving
    Set<Vector> S = Union(C0.S, C1.S);  // all potential separating axes
    tFirst = -infinity;
    tLast = infinity;
    for each D in S do
    {
        Real speed = Dot(D, W);

        // Project C0 onto the separating axis at time 0.
        Real min0 = min(Dot(D, C0.vertices[i]);  // min computed over all i
        Real max0 = max(Dot(D, C0.vertices[i]);  // max computed over all i

        // Project C1 onto the separating axis at time 0.
        Real min1 = min(Dot(D, C1.vertices[i]);  // min computed over all i
        Real max1 = max(Dot(D, C1.vertices[i]);  // max computed over all i

        if (max1 < min0)
        {
            // The interval(C1) is initially on the left of the interval(C0).
            if (speed <= 0)
            {
                // The intervals are moving apart.
                return false;
            }

            // Update the first time of contact.
            Real t = (min0 - max1) / speed;
            if (t > tFirst)
            {
                tFirst = t;
            }

            // Update the last time of contact.
            t = (max0 - min1) / speed;
            if (t < tLast)
            {
                tLast = t;
            }

            // Test whether the first and last times of contact are valid.
            if (tFirst > tLast)
            {
                return false;
            }
        }
        else if (max0 < min1)
        {
            // The interval(C1) is initially on the right of the interval(C0).
            if (speed >= 0)
            {
                // The intervals are moving apart
                return false;
            }

            // Update the first time of contact.
            Real t = (max0 - min1) / speed;
            if (t > tFirst)
            {
                tFirst = t;
            }

            // Update the last time of contact.
            t = (min0 - max1) / speed;
            if (t < tLast)
            {
                tLast = t;
            }
```

```
70
71              // Test whether the first and last times of contact are valid.
72              if (tFirst > tLast)
73              {
74                  return false;
75              }
76          }
77          else
78          {
79              // The interval(C0) and interval(C1) overlap.  It is possible that
80              // the objects separate at a later time, so the last time of contact
81              // potentially needs updating.
82              if (speed > 0)
83              {
84                  // Update the last time of contact.
85                  Real t = (max0 - min1) / speed;
86                  if (t < tLast)
87                  {
88                      tLast = t;
89                  }
90
91                  // Test whether the first and last times of contact are valid.
92                  if (tFirst > tLast)
93                  {
94                      return false;
95                  }
96              }
97              else if (speed < 0)
98              {
99                  // Update the last time of contact.
100                 Real t = (min0 - max1) / speed;
101                 if (t < tLast)
102                 {
103                     tLast = t;
104                 }
105
106                 // Test whether the first and last times of contact are valid.
107                 if (tFirst > tLast)
108                 {
109                     return false;
110                 }
111             }
112         }
113     }
114     return true;
115 }
```

The following example illustrates the ideas. The first box is the unit cube $0 \le x \le 1$ and $0 \le y \le 1$ and is stationary. The second box is initially $0 \le x \le 1$ and $1 + \delta \le y \le 2 + \delta$ for some $\delta > 0$. Let its velocity be $(1, -1)$. Whether or not the second box intersects the first box depends on the value of δ. The only potential separating axes are $(1, 0)$ and $(0, 1)$. Figure 3 shows the initial configuration for three values of δ, one where there will be an edge-edge intersection, one where there will be a vertex-vertex intersection, and one where there is no intersection.

Figure 3. Edge-edge intersection predicted (left). Vertex-vertex intersection predicted (middle). No intersection predicted (right).



The black box is stationary. The blue box is moving. The green vector indicates the direction of motion. The red boxes indicate where the moving box first touches the stationary box. In the right image in the figure, the dotted red line indicates that the moving box will miss the stationary box. For $\boldsymbol{D} = (1, 0)$, the pseudocode produces min0 = 0, max0 = 1, min1 = 0, max1 = 1, and speed = 1. The projected intervals are initially overlapping. Because the speed is positive, t = (max0-min1)/speed = 1 < tLast = INFINITY and tLast is updated to 1. For $\boldsymbol{D} = (0, 1)$, the pseudocode produces min0 = 0, max0 = 1, min1 = 1+delta, max1 = 2+delta, and speed = -1. The moving projected interval is initially on the right of the stationary projected interval. Because the speed is negative, t = (max0-min1)/speed = delta > tFirst = 0 and tFirst is updated to delta. The next block of code sets t = (min0-max1)/speed = 2+delta. The value tLast is not updated because $2 + \delta < 1$ cannot happen for $\delta > 0$. On exit from the loop over potential separating directions, $t_{\text{First}} = \delta$ and $t_{\text{last}} = 1$. The objects intersect if and only if $t_{\text{first}} \le t_{\text{last}}$, or $\delta \le 1$. This condition is consistent with the images in Figure 3. The left image has $\delta < 1$ and the middle image has $\delta = 1$, intersections occurring in both cases. The right image has $\delta > 1$ and no intersection occurs.

7 Contact Set for Moving Convex Objects

Given two moving convex objects C_0 and C_1 with velocities $\boldsymbol{W}_0$ and $\boldsymbol{W}_1$ that are initially not intersecting, the material in the last section showed how to compute the first time of contact T, if it exists. Assuming it does, the sets $C_0 + T\boldsymbol{W}_0 = \{\boldsymbol{X} + T\boldsymbol{W}_0 : \boldsymbol{X} \in C_0\}$ and $C_1 + T\boldsymbol{W}_1 = \{\boldsymbol{X} + T\boldsymbol{W}_1 : \boldsymbol{X} \in C_1\}$ are just touching with no interpenetration. Figure 2 shows the various configurations for 2D.

The TestIntersection function of Listing 4 can be modified to keep track of which vertices, edges, and/or faces are projected to the endpoints of the projection interval. At the first time of contact, this information can be used to determine how the two objects are positioned with respect to each other. In 2D, if the contact is vertex-edge or vertex-vertex, the contact set is a single point, a vertex. If the contact is edge-edge, the contact set is a line segment that contains at least one vertex. In 3D, if the contact is vertex-vertex, vertex-edge, or vertex-face, then the contact set is a single point, a vertex. If the contact is edge-edge, the contact set is a single point (the intersection of the two lines containing the edges) or a line segment (the edges are on the same line). If the contact is edge-face, then the contact set is a line segment. Otherwise, the contact is face-face and the contact set is the intersection of two planar convex polygons.

8 Example: Two Moving Triangles in 2D

Consider two triangles, $\langle \boldsymbol{U}_0, \boldsymbol{U}_1, \boldsymbol{U}_2 \rangle$ and $\langle \boldsymbol{V}_0, \boldsymbol{V}_1, \boldsymbol{V}_2 \rangle$, both having counterclockwise ordering. For the sake of indexing notation, define $\boldsymbol{U}_3 = \boldsymbol{U}_0$ and $\boldsymbol{V}_3 = \boldsymbol{V}_0$. The edge directions are $\boldsymbol{E}_i = \boldsymbol{U}_{i+1} - \boldsymbol{U}_i$ and $\boldsymbol{F}_i = \boldsymbol{V}_{i+1} - \boldsymbol{V}_i$ for $0 \le i \le 2$. Define $(x, y)^\perp = (y, -x)$. Outward pointing normals to the edges are $\boldsymbol{N}_i = \boldsymbol{E}_i^\perp$ and $\boldsymbol{M}_i = \boldsymbol{F}_i^\perp$ for $0 \le i \le 2$. The six normals are the potential separating directions. Let the triangle velocities be $\boldsymbol{W}_0$ and $\boldsymbol{W}_1$.

In Listing 4 for testing for intersection of two moving convex objects, C_0 and C_1 represent the two triangles. The calculation of the minimum and maximum projections for the triangles can be computed so that additional information is known about how the two triangles are oriented with respect to each other. Three cases occur for the projection:

1. Two vertices project to the minimum of the interval and one vertex projects to the maximum.
2. One vertex projects to the minimum of the interval and two vertices project to the maximum.
3. One vertex projects to the minimum, one vertex projects to the maximum, and one vertex projects to an interior point of the interval defined by the minimum and maximum.

A flag can be associated with each triangle to indicate which of these three cases occurs for a given potential separating direction $\boldsymbol{D}$; call the flag values M21, M12, and M11. It is also necessary to remember the indices of the vertices that project to the extreme values. Listing 5 shows a convenient data structure.

Listing 5. A data structure to support the computation of contact points. The flag Mij refers to i points projecting to the minimum and j points projecting to the maximum; for example, M21 indicates that 2 points project to the minimum and 1 point projects to the maximum. The number index[0] is the index into the array of three triangle vertices for a vertex that maps to the min projection. The number index[2] is the index of that vertex that maps to the max projection. The number index[1] is that of the remaining vertex that can map to the minimum, maximum or some number between them depending on the orientation of the triangle.

```
enum ProjectionMap { M21, M12, M11 };
struct Configuration { ProjectionMap map; int index[3]; Real min, max; };
```

In the TestIntersection function of Listing 4, two configuration objects are declared, cfg0 for the U-triangle (the C0 polygon) and cfg1 for the V-triangle (the C1 polygon). To illustrate for the specific case $\boldsymbol{D} = \boldsymbol{N}_0$, the lines 12, 13, 16 and 17 of Listing 4 are shown in Listing 6 for the specific case of two triangles.

Listing 6. Lines 12, 13, 16 and 17 of Listing 4, where C0 is the $\boldsymbol{U}$-triangle and C1 is the $\boldsymbol{V}$-triangle, where U[i] denotes $\boldsymbol{U}_i$ and V[i] denotes $\boldsymbol{V}_i$.

```
Real min0 = min(Dot(D,U[i]));  // min computed over i = 0,1,2
Real max0 = max(Dot(D,U[i]));  // max computed over i = 0,1,2
Real min1 = min(Dot(D,V[i]));  // min computed over i = 0,1,2
Real max1 = max(Dot(D,V[i]));  // max computed over i = 0,1,2
```

This code is replaced by that of Listing 7.

Listing 7. The replacement code specific for computing the extreme projections of the $\boldsymbol{U}$-triangle onto the outer-pointing normal perpendicular to the edge of $\langle \boldsymbol{U}_0, \boldsymbol{U}_1 \rangle$. The edge direction is $\boldsymbol{E}_0 = \boldsymbol{U}_1 - \boldsymbol{U}_0$ and the normal is $\boldsymbol{N}_0 = \boldsymbol{E}_0^{\perp}$.

```
Configuration cfg0;  // the configuration for the U-triangle
Configuration cfg1;  // the configuration for the V-triangle

// U2 maps to minimum, U0 and U1 map to maximum
cfg0.map = M12;
cfg0.index[0] = 2;  cfg0.index[1] = 0;  cfg0.index[2] = 1;
cfg0.min = -Dot(N0,E2);
cfg0.max = 0;

// Compute min and max of interval for second triangle.  Keep track of
// vertices that project to min and max.
Real d0 = Dot(N0,V0-U0);  d1 = Dot(N0,V1-U0);  d2 = Dot(N0,V2-U0);
if (d0 <= d1)
{
    if (d1 <= d2)  // d0 <= d1 <= d2
    {
        if (d0 != d1)
        {
            cfg1.map = ( d1 != d2 ? M11 : M12 );
        }
        else
        {
            cfg1.map = M21;
        }
        cfg1.index[0] = 0;  cfg1.index[1] = 1;  cfg1.index[2] = 2;
        cfg1.min = d0;  cfg1.max = d2;
    }
    else if (d0 <= d2)  // d0 <= d2 < d1
    {
        if (d0 != d2)
        {
            cfg1.map = M11;
            cfg1.index[0] = 0;  cfg1.index[1] = 2;  cfg1.index[2] = 1;
        }
        else
        {
            cfg1.map = M21;
            cfg1.index[0] = 2;  cfg1.index[1] = 0;  cfg1.index[2] = 1;
        }
        cfg1.min = d0;  cfg1.max = d1;
    }
    else  // d2 < d0 <= d1
    {
        cfg1.map = (d0 != d1 ? M12 : M11);
        cfg1.index[0] = 2;  cfg1.index[1] = 0;  cfg1.index[2] = 1;
        cfg1.min = d2;  cfg1.max = d1;
    }
}
else
{
    if (d2 <= d1)  // d2 <= d1 < d0
    {
        if (d2 != d1)
        {
            cfg1.map = M11;
            cfg1.index[0] = 2;  cfg1.index[1] = 1;  cfg1.index[2] = 0;
        }
        else
        {
            cfg1.map = M21;
            cfg1.index[0] = 1;  cfg1.index[1] = 2;  cfg1.index[2] = 0;
        }

        cfg1.min = d2;  cfg1.max = d0;
    }
```

```
    else if (d2 <= d0)  // d1 < d2 <= d0
    {
        cfg1.map = (fD2 != fD0 ? M11 : M12);
        cfg1.index[0] = 1;  cfg1.index[1] = 2;  cfg1.index[2] = 0;
        cfg1.min = fD1;  cfg1.max = fD0;
    }
    else  // d1 < d0 < d2
    {
        cfg1.map = M11;
        cfg1.index[0] = 1;  cfg1.index[1] = 0;  cfg1.index[2] = 2;
        cfg1.min = fD1;  cfg1.max = fD2;
    }
}
```

Similar blocks of code can be written for other potential separating directions.

The conditional statements starting at line 19 of Listing 4 comparing minima and maxima must be modified to keep track of the relative location of the moving interval to the stationary one. The input configurations are for the current potential separating axis. If this axis direction becomes the new candidate for first time of contact, we need to remember its configurations to be used in the function that computes the contact set. Listing 8 contains pseudocode for tracking the configurations. It also shows how we must keep track of how the intervals intersect (if at all) along a separating axis.

Listing 8. The replacement code for the comparisons of the extremes of the projections. We need to keep track of the configurations for the current separating axis test. We also need to keep track of global configurations for the two triangles related to the first time of contact (if any). Finally, we also need to keep track of the relative position of the $\boldsymbol{U}$-triangle projection interval and the $\boldsymbol{V}$-triangle projection interval; that is, we need to how the intervals approach each other for the current separating axis test. The enumeration for this is Side.

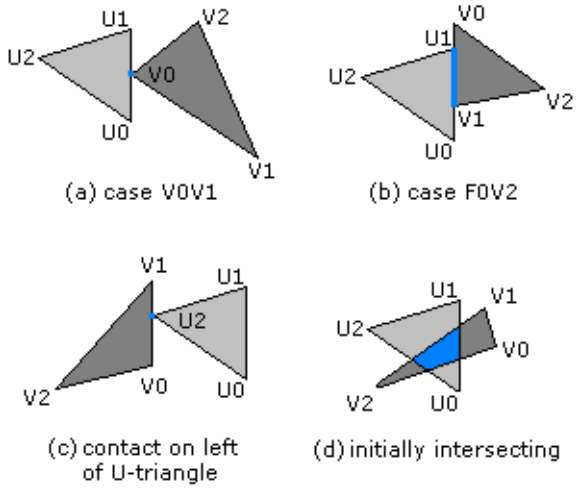
```
enum Side = { NONE, LEFT, RIGHT };

// The following thee variables are global, outside the loop over all
// potential separating axes.  The contact configurations need not be
// initialized, because they are consumed only when there is separation
// at which time the contact configurations are assigned values.
Side side = NONE;
Configuration contactCfg0, contactCfg1;

// Replacement code for the current separating axis test.  The complete
// code has a loop over all potential separating axis tests.
if (cfg1.max < cfg0.min)
{
    // The V-interval is initially on the left of the U-interval.
    if (speed <= 0)
    {
        // The intervals are moving apart.
        return false;
    }

    // Update the first time of contact.
    Real t = (cfg0.min - cfg1.max) / speed;
    if (t > tFirst)
    {
        tFirst = t;
        side = LEFT;
        contactCfg0 = cfg0;
        contactCfg1 = cfg1;
    }

    // Update the last time of contact.
```

```
        t = (cfg0.max - cfg1.min) / speed;
        if (t < tLast)
        {
            tLast = t;
        }

        // Test whether the first and last times of contact are valid.
        if (tFirst > tLast)
        {
            return false;
        }
    }
    else if (cfg0.max < cfg1.min)
    {
        // The V-interval is initially on the right of the U-interval.
        if (speed >= 0)
        {
            // The intervals are moving apart.
            return false;
        }

        // Update the first time of contact.
        Real t = (cfg0.max - cfg1.min) / speed;
        if (t > tFirst)
        {
            tFirst = t;
            side = RIGHT;
            contactCfg0 = cfg0;
            contactCfg1 = cfg1;
        }

        // Update the last time of contact.
        t = (cfg0.min - cfg1.max)/speed;
        if (t < tLast)
        {
            tLast = t;
        }

        // Test whether the first and last times of contact are valid.
        if (tFirst > tLast)
        {
            return false;
        }
    }
    else
    {
        // The U-interval and V-interval overlap.  It is possible that
        // the objects separate at a later time, so the last time of contact
        // potentially needs updating.
        if (speed > 0)
        {
            // Update the last time of contact.
            Real t = (cfg0.max - cfg1.min) / speed;
            if (t < tLast)
            {
                tLast = t;
            }

            // Test whether the first and last times of contact are valid.
            if (tFirst > tLast)
            {
                return false;
            }
        }
        else if (speed < 0)
        {
            // Update the last time of contact.
            T = (cfg0.min - cfg1.max) / speed;
            if (t < tLast)
            {
                tLast = t;
            }
```

```
        // Test whether the first and last times of contact are valid.
        if (tFirst > tLast)
        {
            return false;
        }
    }
}
```

If $\boldsymbol{N}_0$ is the last separating axis an instant before first time of contact, then the TConfig string contains one of V0V1, V1V0, V0V2, V2V0, V1V2, V2V1, V0F1, F1V0, V1F2, F2V1, V2F0, or F0V2. Figure 4 illustrates four configurations that match the pseudocode (given after the figure).

Figure 4. Four configurations for the first time of contact. Although the labels use the original vertex names, the vertices are actually the original ones moved by their corresponding velocities.



The following pseudocode shows how to construct the contact set. Again, the use of strings is just for illustration and most likely not how you would implement this.

```
if ( side == RIGHT )
{
    if ( TConfig[0] == 'V' )
    {
        // vertex-edge configuration  [Figure 8.1 (a)]
        contactSet = { Vi + TFirst*W1 };  // i is number in TConfig[1]
    }
    else  // TConfig[0] == 'F'
    {
        // Edge-edge configuration.  See discussion after this code.
        // Vi0 is the first vertex and Vi1 is the last vertex of edge Fi
        // where i is the number in TConfig[1].  [Figure 8.1 (b)]
        min = Dot(E0,Vi1-U0+TFirst*W)/Dot(E0,E0);
        max = Dot(E0,Vi0-U0+TFirst*W)/Dot(E0,E0);
        I = intersection([0,1],[min,max]);  // guaranteed not empty
        contactSet = U0 + TFirst*W1 + I*E0;  // point or line segment
    }
}
else if ( side == LEFT )
```

```
{
    // vertex-edge configuration  [Figure 4 (c)]
    contactSet = { U2 + TFirst*W0 };
}
else  // triangles were initially intersecting
{
    // Intersection set is a convex set: a point, a line segment, or a
    // convex polygon with at most six sides.  [Figure 8.1 (d)]
    Point UMove[3] = { U0+TFirst*W0, U1+TFirst*W0, U2+TFirst*W0 };
    Point VMove[3] = { V0+TFirst*W1, V1+TFirst*W1, V2+TFirst*W1 };
    contactSet = TriTriIntersection(UMove,VMove);
}
```

In the case of edge-edge contact, after motion the two triangles touch at either a point or line segment. Let T denote the contact time. For the sake of argument, suppose that the contact edge for the second triangle is $\boldsymbol{F}_0$. The touching edges are parameterized by $\boldsymbol{U}_0 + T\boldsymbol{W}_1 + s\boldsymbol{E}_0$ for $s \in [0,1]$ and $\boldsymbol{V}_0 + T\boldsymbol{W}_0 + s\boldsymbol{E}_0$ for $s \in [\mu_0, \mu_1]$ where

$$\mu_0 = \frac{(\boldsymbol{V}_1 + T\boldsymbol{W}_1) - (\boldsymbol{U}_0 + T\boldsymbol{W}_0)}{|\boldsymbol{E}_0|^2} \text{ and } \mu_1 = \frac{(\boldsymbol{V}_0 + T\boldsymbol{W}_1) - (\boldsymbol{U}_0 + T\boldsymbol{W}_0)}{|\boldsymbol{E}_0|^2}.$$

The overlap of the two edges occurs for $\bar{s} \in I = [0,1] \cap [\mu_0, \mu_1] \neq \emptyset$. The corresponding points in the contact set are $\boldsymbol{U}_0 + T\boldsymbol{W}_0 + \bar{s}\boldsymbol{E}_0$.

In the event the two triangles are initially overlapping, the contact set is more expensive to construct. It can be a single point, a line segment, or a convex polygon with at most six sides. This set can be constructed by standard methods involving Boolean operations on polygons.

9 Example: Two Moving Triangles in 3D

Consider two triangles, $\langle \boldsymbol{U}_0, \boldsymbol{U}_1, \boldsymbol{U}_2 \rangle$ and $\langle \boldsymbol{V}_0, \boldsymbol{V}_1, \boldsymbol{V}_2 \rangle$. For the sake of indexing notation, define $\boldsymbol{U}_3 = \boldsymbol{U}_0$ and $\boldsymbol{V}_3 = \boldsymbol{V}_0$. The edges are $\boldsymbol{E}_i = \boldsymbol{U}_{i+1} - \boldsymbol{U}_i$ and $\boldsymbol{F}_i = \boldsymbol{V}_{i+1} - \boldsymbol{V}_i$ for $0 \leq i \leq 2$. A normal for the first triangle is $\boldsymbol{N} = \boldsymbol{E}_0 \times \boldsymbol{E}_1$ and a normal for the second triangle is $\boldsymbol{M} = \boldsymbol{F}_0 \times \boldsymbol{F}_1$. If the triangles are not coplanar, then the potential separating directions are $\boldsymbol{N}$, $\boldsymbol{M}$, and $\boldsymbol{E}_i \times \boldsymbol{F}_j$ for $0 \leq i \leq 2$ and $0 \leq j \leq 2$. If the triangles are parallel, but are not in the same plane, then $\boldsymbol{N}$ is a separating direction and the other directions need not be tested. Moreover, if $\boldsymbol{N}$ and $\boldsymbol{M}$ do not separate non-coplanar triangles, then the vectors $\boldsymbol{E}_i \times \boldsymbol{F}_j$ cannot be zero. If the triangles are coplanar, then the potential separating directions are $\boldsymbol{N} \times \boldsymbol{E}_i$ and $\boldsymbol{N} \times \boldsymbol{F}_i$ for $0 \leq i \leq 2$. This is exactly the 2D situation discussed earlier.

If $\boldsymbol{D}$ is a potential separating direction, then the block for computing the intervals of projection is more complex than that of its 2D counterpart. Both triangles are projected onto the separating axis. Each projection interval must be sorted to determine the appropriate configuration. The left/right relationship of the two projection intervals must be determined. The set of configurations for the projection of a single triangle consists of the following:

- **3**: All three vertices project to the same point. This happens when $\boldsymbol{D}$ is a normal vector to one of the triangles.
- **2-1**: Two vertices project to the minimum point of the interval, one vertex projects to the maximum point.
- **1-2**: One vertex projects to the minimum point of the interval, two vertices project to the maximum point.

- **1-1-1**: The three vertices project to distinct points in the interval.

The tables below summarize the possibilities. The symbols U, E, and N refer to the first triangle's vertices, edges, and full triangle. The indices after the U or E symbols indicate the specific vertex or edge that is involved in the contact. This information is calculated during the sorting of the projected points. The symbols V, F, and M refer to the second triangle's vertices, edges, and full triangle.

U left \ V right	3	2-1	1-2	1-1-1
3	NM	NF_j	NV_j	NV_j
1-2	E_iM	E_iF_j	E_iV_j	E_iV_j
2-1	U_iM	U_iF_j	U_iV_j	U_iV_j
1-1-1	U_iM	U_iF_j	U_iV_j	U_iV_j

V left \ U right	3	2-1	1-2	1-1-1
3	MN	ME_i	MU_i	MU_i
1-2	F_jN	F_jE_i	F_jU_i	F_jU_i
2-1	V_jN	V_jE_i	V_jU_i	V_jU_i
1-1-1	V_jN	V_jE_i	V_jU_i	V_jU_i

The intersection set for any table entry containing U_i or V_j is just that point. Each table contains 12 such cases. The intersection set for any table entry containing E_i or F_j but not containing U_i or V_j is a line segment (possibly degenerating to a point). Each table contains 3 such cases. Finally, the table entries NM and MN correspond to coplanar triangles that intersect. The intersection set is a point, line segment, or a convex polygon with at most six sides.

Pseudocode for handling the sorting and storing information for later code that determines contact sets is given below.

```
typedef enum { m3, m21, m12, m111 } ProjectionMap;

typedef struct
{
    ProjectionMap map;  // how vertices map to projection interval
    int index[3];       // the sorted indices of the vertices
    float min, max;     // the interval is [min,max]
}
Config;

Config GetConfiguration (Point D, Point U[3])
{
    // D is potential separating direction
    // U[3] are the triangle vertices

    Configuration config;
    d0 = Dot(D,U[0]), d1 = Dot(D,U[1]), d2 = Dot(D,U[2]);
    if ( d0 <= d1 )
    {
        if ( d1 <= d2 )  // d0 <= d1 <= d2
        {
            config.index[0] = 0;  config.index[1] = 1;  config.index[2] = 2;
            config.min = d0;  config.max = d2;
            if ( d0 != d1 )
                config.map = ( d1 != d2 ? m111 : m12 );
            else
                config.map = ( d1 != d2 ? m21 : m3 );
        }
        else if ( d0 <= d2 )  // d0 <= d2 < d1
        {
            config.index[0] = 0;  config.index[1] = 2;  config.index[2] = 1;
            config.min = d0;  config.max = d1;
            config.map = ( d0 != d2 ? m111 : m21 );
        }
        else  // d2 < d0 <= d1
        {
```

```
            config.index[0] = 2;  config.index[1] = 0;  config.index[2] = 1;
            config.min = d2;  config.max = d1;
            config.map = ( d0 != d1 ? m12 : m111 );
        }
    }
    else
    {
        if ( d2 <= d1 )  // d2 <= d1 < d0
        {
            config.index[0] = 2;  config.index[1] = 1;  config.index[2] = 0;
            config.min = d2;  config.max = d0;
            config.map = ( d2 != d1 ? m111 : m21 );
        }
        else if ( d2 <= d0 )  // d1 < d2 <= d0
        {
            config.index[0] = 1;  config.index[1] = 2;  config.index[2] = 0;
            config.min = d1;  config.max = d0;
            config.map = ( d2 != d0 ? m111 : m12 );
        }
        else  // d1 < d0 < d2
        {
            config.index[0] = 1;  config.index[1] = 0;  config.index[2] = 2;
            config.min = d1;  config.max = d2;
            config.map = m111;
        }
    }
    return config;
}
```

Pseudocode for determining how the projection intervals relate to each other is given below.

```
bool Update (Config UC, Config VC, float speed,
    Side& side, Config& TUC, Config& TVC, float& TFirst, float& TLast)
{
    if ( VC.max < UC.min )  // V-interval initially on 'left' of U-interval
    {
        if ( speed <= 0 ) return false;  // intervals moving apart
        T = (UC.min - VC.max)/speed;
        if ( T > TFirst ) { TFirst = T; side = LEFT; TUC = UC; TVC = VC; }
        T = (UC.max - VC.min)/speed;  if ( T < TLast ) TLast = T;
        if ( TFirst > TLast ) return false;
    }
    else if ( UC.max < VC.min )  // V-interval initially on 'right' of U-interval
    {
        if ( speed >= 0 ) return false;  // intervals moving apart
        T = (UC.max - VC.min)/speed;
        if ( T > TFirst ) { TFirst = T; side = RIGHT; TUC = UC; TVC = VC; }
        T= (UC.min - VC.max)/speed;  if ( T < TLast ) TLast = T;
        if ( TFirst > TLast ) return false;
    }
    else  // U-interval and V-interval overlap
    {
        if ( speed > 0 )
        {
            T = (UC.max - VC.min)/speed;
            if ( T < TLast ) TLast = T;  if ( TFirst > TLast ) return false;
        }
        else if ( speed < 0 )
        {
            T = (UC.min - VC.max)/speed;
            if ( T < TLast ) TLast = T;  if ( TFirst > TLast ) return false;
        }
    }
    return true;
}
```

It is assumed that the following routines exist for use in contact determination:

- Intersection of two line segments, call it SegSegIntersection.

- Intersection of line segment and triangle that are coplanar, call it SegTriIntersection.
- Intersection of triangle and triangle that are coplanar, call it CoplanarTriTriIntersection.
- Intersection of two stationary triangles, call it TriTriIntersection. This routine will contain a call to the coplanar intersection routine if the triangles happen to be coplanar.

Pseudocode for computing the contact set is given below.

```
ContactSet GetFirstContact (Point U[3], Point W0, Point V[3], Point W1,
    Side side, Config TUC, Config TVC, float TFirst)
{
    // move triangles to first contact
    Point UTri[3] = { U[0]+TFirst*W0, U[1]+TFirst*W0, U[2]+TFirst*W0 };
    Point VTri[3] = { V[0]+TFirst*W1, V[1]+TFirst*W1, V[2]+TFirst*W1 };
    Segment USeg, VSeg;

    if ( side == RIGHT )  // V-interval on right of U-interval
    {
        if ( TUC.map == m21 || TUC.map == m111 )
            return UTri[TUC.index[2]];

        if ( TVC.map == m12 || TVC.map == m111 )
            return VTri[TVC.index[0]];

        if ( TUC.map == m12 )
        {
            USeg = <UTri[TUC.index[1]],UTri[TUC.index[2]]>;
            if ( TVC.map == m21 )
            {
                VSeg = <VTri[TVC.index[0]],VTri[TVC.index[1]]>;
                return SegSegIntersection(USeg,VSeg);
            }
            else  // TVC.map == m3
            {
                return SegTriIntersection(USeg,VTri);
            }
        }
        else  // TUC.map == m3
        {
            if ( TVC.map == m21 )
            {
                VSeg = <VTri[TVC.index[0]],VTri[TVC.index[1]]>;
                return SegTriIntersection(VSeg,UTri);
            }
            else  // TVC.map == m3
            {
                return CoplanarTriTriIntersection(UTri,VTri);
            }
        }
    }
    else if ( side == LEFT )  // V-interval on left of U-interval
    {
        if ( TVC.map == m21 || TVC.map == m111 )
            return VTri[TVC.index[2]];

        if ( TUC.map == m12 || TUC.map == m111 )
            return UTri[TUC.index[0]];

        if ( TVC.map == m12 )
        {
            VSeg = <VTri[TVC.index[1]],VTri[TVC.index[2]]>;
            if ( TUC.map == m21 )
            {
                USeg = <UTri[TUC.index[0]],UTri[TUC.index[1]]>;
                return SegSegIntersection(USeg,VSeg);
            }
            else  // TUC.map == m3
            {
```

```
                return SegTriIntersection(VSeg,UTri);
            }
        }
        else  // TVC.map == m3
        {
            if ( TUC.map == m21 )
            {
                USeg = <UTri[TUC.index[0]],UTri[TUC.index[1]]>;
                return SegTriIntersection(USeg,VTri);
            }
            else  // TUC.map == m3
            {
                return CoplanarTriTriIntersection(UTri,VTri);
            }
        }
    }
    else  // triangles were initially intersecting
    {
        return TriTriIntersection(UTri,VTri);
    }
}
```

The pseudocode that puts all this together is

```
bool TrianglesIntersect (Point U[3], Point W0, Point V[3], Point W1,
    float& TFirst, float& TLast, ContactSet& contact)
{
    W = W1 - W0;
    S = set of all potential separating axes;
    TFirst = 0;  TLast = INFINITY;
    side = NONE;
    Config TUC, TVC;

    for each D in S do
    {
        speed = Dot(D,W);
        Config UC = GetConfiguration(D,U);
        Config VC = GetConfiguration(D,V);
        if ( !Update(UC,VC,speed,side,TUC,TVC,TFirst,TLast) )
            return false;
    }

    contact = GetFirstContact(U,W0,V,W1,side,TUC,TVC,TFirst);
    return true;
}
```

References

[1] Ron Levine. Collision of moving objects. On the game developer algorithms list at `www.sourceforce.net`, November 2000.