# Intersection of Moving Sphere and Triangle

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/


Created: March 21, 2010

## Contents

# 1 Introduction

This document describes how to compute the first time of contact and the point of contact between a (solid) sphere and triangle, both moving with constant linear velocities. To simplify the problem, subtract the triangle velocity from the sphere velocity so that the triangle is stationary and the sphere is moving relative to the triangle.

## 1.1 The Triangle

Let the triangle have vertices at positions $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$. Let the plane of the triangle be defined implicitly by $\mathbf{U}_2 \cdot (\mathbf{X} - \mathbf{P}_0) = 0$, where $\mathbf{U}_2$ is a unit-length vector. Choose $\mathbf{U}_2$ to be

$$\mathbf{U}_2 = \frac{(\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0)}{|(\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0)|}, \tag{1}$$

Let $\mathbf{U}_0$ and $\mathbf{U}_1$ be unit-length vectors such that $\{\mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2\}$ is a right-handed orthonormal set; that is, the vectors are all unit length, mutually perpendicular, and $\mathbf{U}_2 = \mathbf{U}_0 \times \mathbf{U}_1$. A point $\mathbf{X}$ in space may be written as

$$\mathbf{X} = \mathbf{P}_0 + y_0\mathbf{U}_0 + y_1\mathbf{U}_1 + y_2\mathbf{U}_2 \tag{2}$$

where $y_i = \mathbf{U}_i \cdot (\mathbf{X} - \mathbf{P}_0)$. The 3-tuple $(y_0, y_1, y_2)$ lists the coordinates of $\mathbf{X}$ relative to the coordinate system whose origin is $\mathbf{P}_0$ and whose axis directions are $\mathbf{U}_i$. As a 3-tuple, the projection of $\mathbf{X}$ onto the plane of the triangle is $\mathbf{X}_{\text{proj}} = \mathbf{P}_0 + y_0\mathbf{U}_0 + y_1\mathbf{U}_1$. As a 2-tuple, the projection in planar coordinates is $\mathbf{Y} = (y_0, y_1)$. For simplicity, choose

$$\mathbf{U}_0 = \frac{\mathbf{P}_1 - \mathbf{P}_0}{|\mathbf{P}_1 - \mathbf{P}_0|}, \quad \mathbf{U}_1 = \mathbf{U}_2 \times \mathbf{U}_0 \tag{3}$$

In planar coordinates, $\mathbf{P}_i$ is is represented by $\mathbf{Q}_i$,

$$\begin{aligned} &\mathbf{Q}_0 = (0, 0), \\ &\mathbf{Q}_1 = (\ell, 0), \quad \ell = |\mathbf{P}_1 - \mathbf{P}_0| \\ &\mathbf{Q}_2 = (\alpha, \beta), \quad \alpha = \mathbf{U}_0 \cdot (\mathbf{P}_2 - \mathbf{P}_0), \quad \beta = \mathbf{U}_1 \cdot (\mathbf{P}_2 - \mathbf{P}_0) \end{aligned} \tag{4}$$

The 2-tuple edge vectors are

$$\mathbf{E}_0 = \mathbf{Q}_1 - \mathbf{Q}_0 = (\ell, 0), \quad \mathbf{E}_1 = \mathbf{Q}_2 - \mathbf{Q}_1 = (\alpha - \ell, \beta), \quad \mathbf{E}_2 = \mathbf{Q}_0 - \mathbf{Q}_2 = (-\alpha, -\beta) \tag{5}$$

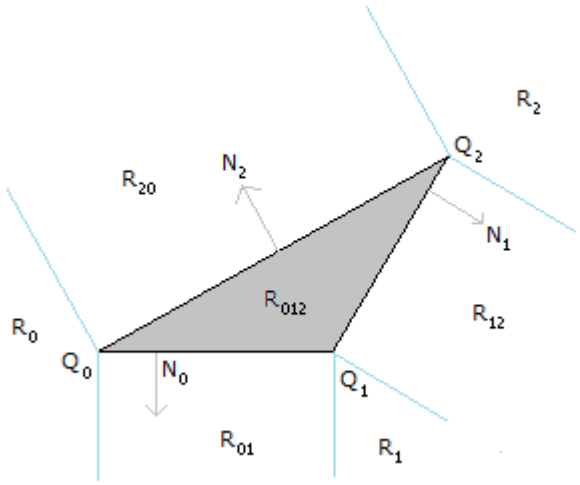
and are not necessarily unit length.

Outer-pointing and unit-length edge normals are the 2-tuples $\mathbf{N}_0$ for edge $\langle \mathbf{Q}_0, \mathbf{Q}_1 \rangle$, $\mathbf{N}_1$ for edge $\langle \mathbf{Q}_1, \mathbf{Q}_2 \rangle$, and $\mathbf{N}_2$ for edge $\langle \mathbf{Q}_2, \mathbf{Q}_0 \rangle$. They are

$$\mathbf{N}_0 = (0, -1), \quad \mathbf{N}_1 = \frac{(\beta, \ell - \alpha)}{\sqrt{\beta^2 + (\ell - \alpha)^2}}, \quad \mathbf{N}_2 = \frac{(-\beta, \alpha)}{\sqrt{\beta^2 + \alpha^2}} \tag{6}$$

Within the plane of the triangle, the Voronoi diagram is shown in Figure 1.

**Figure 1.** The Voronoi diagram of the triangle.




The plane is decomposed into seven regions. Region $R_{012}$ is the set of points inside the triangle (including the edge points). Region $R_i$ is the set of exterior points whose closest feature of the triangle is vertex $\mathbf{Q}_i$. Region $R_{ij}$ is the set of exterior points whose closest feature of the triangle is edge $\langle \mathbf{Q}_i, \mathbf{Q}_j \rangle$.

## 1.2 Distance from Point to Triangle

For any point $\mathbf{X}$ in space, the distance from the point to the triangle is the distance from the point to the closest feature of the triangle. The closest feature to $\mathbf{X}$ is the same as the closest feature to the projection of $\mathbf{X}$ onto the plane of the triangle. Let the projection be $\mathbf{Y} = (y_0, y_1)$, where $\mathbf{X} = \mathbf{P}_0 + y_0\mathbf{U}_0 + y_1\mathbf{U}_1 + y_2\mathbf{U}_2$.

If $\mathbf{Y}$ is in region $R_{012}$, then the closest triangle point to $\mathbf{X}$ is its 3-tuple projection, $\mathbf{P}_0 + y_0\mathbf{U}_0 + y_1\mathbf{U}_1$. The distance is

$$d_{012} = |y_2| \tag{7}$$

which is the vertical distance measured from $\mathbf{X}$ to its projection.

If $\mathbf{Y}$ is in region $R_i$, then the closest triangle point as a 2-tuple is $\mathbf{Q}_i$, so the closest triangle point to $\mathbf{X}$ as a 3-tuple is $\mathbf{P}_i$. The distance is

$$d_i = |\mathbf{X} - \mathbf{P}_i| \tag{8}$$

If $\mathbf{Y}$ is in region $R_{ij}$, then the closest triangle point as a 2-tuple is $\mathbf{Q}_i + s\mathbf{E}_i$, where $s = \mathbf{E}_i \cdot (\mathbf{Y} - \mathbf{Q}_i)/\mathbf{E}_i \cdot \mathbf{E}_i \in [0, 1]$. The closest triangle point as a 3-tuple is $\mathbf{P}_i + s(\mathbf{P}_j - \mathbf{P}_i)$. The distance is

$$d_{ij} = |\mathbf{X} - (\mathbf{P}_i + s(\mathbf{P}_j - \mathbf{P}_i))| \tag{9}$$

## 1.3 The Moving Sphere

Let the sphere initially have center $\mathbf{C}_0$ and let the sphere have radius $r$. Let the sphere's constant linear velocity be the nonzero vector $\mathbf{V}$. Therefore, the moving sphere has center $\mathbf{C}(t) = \mathbf{C}_0 + t\mathbf{V}$ for time $t \geq 0$.

We are interested in determining the time $T$, if any, for which the distance from $\mathbf{C}(T)$ to the triangle is exactly $r$ and for which the distance from $\mathbf{C}(t)$ for $t < T$ is larger than $r$. Such a time $T$ is called the *(first) contact time* between the sphere and triangle. The *(first) contact point* is the triangle point that is closest to $\mathbf{C}(T)$.

In fact, we will compute the first contact time $T_0$ and the last contact time $T_1$, if they exist (the sphere might never come within $r$ units of the triangle). It is necessary that $T_0 \leq T_1$ and the distance from $\mathbf{C}(T_i)$ to the triangle is exactly $r$. Assuming your query is constrained to $t \geq 0$, the following list contains all the possible results.

1. If $T_0 > 0$, the sphere and triangle are initially separated but then come in contact at time $T_0$.
2. If $T_0 = 0$ or $T_1 = 0$, the sphere and triangle are initially in contact. Your application might want to know which case.
   (a) $T_0 = 0$ and $T_1 > 0$. The sphere and triangle are initially in contact but overlap as time increases.
   (b) $T_0 < 0$ and $T_1 = 0$. The sphere and triangle are initially in contact but separate as time increases. In this situation, an application might not call this an intersection.
3. If $T_0 < 0$ and $T_1 > 0$, the sphere and triangle are initially overlapping. There are infinitely many points with $r$ units of distance of the sphere center.
4. If $T_1 < 0$, the sphere and triangle are never in contact as time increases.

As noted previously, the triangle point closest to a point may be determined by projecting the point onto the plane of the triangle and finding which Voronoi region contains it. Our point of interest is the moving sphere center, $\mathbf{C}(t)$, which may be written in terms of the coordinate system implied by the plane,

$$\mathbf{C}_0 + t\mathbf{V} = \mathbf{P}_0 + y_0(t)\mathbf{U}_0 + y_1(t)\mathbf{U}_1 + y_2(t)\mathbf{U}_2 \tag{10}$$

where $y_i(t) = \mathbf{U}_i \cdot (\mathbf{C}_0 - \mathbf{P}_0) + t\mathbf{U}_i \cdot \mathbf{V}$. The projection onto the plane is

$$\mathbf{C}_{\text{proj}}(t) = \mathbf{P}_0 + y_0(t)\mathbf{U}_0 + y_1(t)\mathbf{U}_1 \tag{11}$$

As a 2-tuple, the projected center is

$$\mathbf{L}(t) = (y_0(t), y_1(t)) = \mathbf{K} + t\mathbf{W} \tag{12}$$

We will partition the projected line into segments and two rays using the Voronoi regions of the triangle. Each linear component of the partition has an associated closest feature on the triangle, and the squared distance to that closest feature turns out to be a quadratic function of time.

## 2 Line Partitioning by the Voronoi Regions

To partition the projected line by the Voronoi regions, we must compute the intersections between the line and segments or rays that form the Voronoi diagram. The three segments of interest are the edges $\langle \mathbf{Q}_0, \mathbf{Q}_1 \rangle$, $\langle \mathbf{Q}_1, \mathbf{Q}_2 \rangle$, and $\langle \mathbf{Q}_2, \mathbf{Q}_0 \rangle$. The six rays of interest are the origin-direction pairs $\langle \mathbf{Q}_0, \mathbf{N}_0 \rangle$, $\langle \mathbf{Q}_1, \mathbf{N}_0 \rangle$, $\langle \mathbf{Q}_1, \mathbf{N}_1 \rangle$, $\langle \mathbf{Q}_2, \mathbf{N}_1 \rangle$, $\langle \mathbf{Q}_2, \mathbf{N}_2 \rangle$, and $\langle \mathbf{Q}_0, \mathbf{N}_2 \rangle$.

The partition may be viewed as an ordered set

$$S = \{[a_i, b_i], \ell_i\}_{i=0}^{n-1} \tag{13}$$

where the $i$-th linear component corresponds to the time interval $[a_i, b_i]$ with $a_i < b_i$ and where $\ell_i$ is a label that identifies the Voronoi region that contains the linear component. The elements of $S$ are ordered in the sense that $b_i = a_{i+1}$ for all $i$.

The number of components, $n$, must be at least one. The number is exactly one when the original line of centers is perpendicular to the plane of the triangle. The projected line is a single point $\mathbf{K}$, where $\mathbf{W} = \mathbf{0}$ in equation (12). This point lies in a single Voronoi region, in which case $[a_0, b_0] = [-\infty, +\infty]$.

When the line of centers is not perpendicular to the plane, the projected line must intersect at least two of the segments or rays forming the Voronoi diagram. Although a visual inspection of the placement of a line on the plane will convince you this is true, consider also that as $t$ approaches $+\infty$, the distance becomes unbounded. As $t$ approaches $-\infty$, the distance also becomes unbounded. The set $S$ must have two components corresponding to the "end rays" for $|t|$ very large. Consequently, $S$ must have at least one more element to connect the two end rays. Thus, $n$ must be at least three.

Because we have seven Voronoi regions, the set $S$ can have at most seven elements. In fact, there are examples for which the projected line intersects all the regions. In Figure 1, the line containing edge $\langle \mathbf{Q}_1, \mathbf{Q}_2 \rangle$ intersects regions $R_0$, $R_{01}$, $R_{012}$ (the triangle edges are part of this region), and $R_2$, for a total of four intersected regions. Now rotate this line just slightly in a counterclockwise direction about the edge center $(\mathbf{Q}_1 + \mathbf{Q}_2)/2$. This line intersects the four aforementioned regions, but it also intersects $R_1$, $R_{12}$, and $R_{20}$.

## 3 Computing the Contact Times

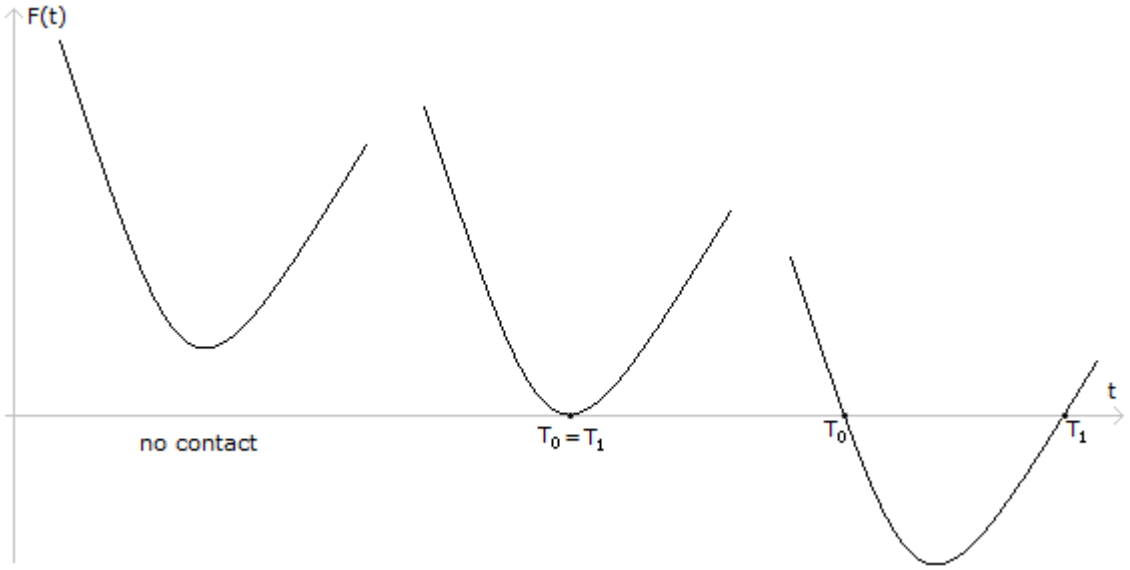
For an element of $S$, let $[a_i, b_i]$ be the time interval corresponding to that element. The corresponding centers of the moving sphere are $\mathbf{C}(t)$ for $t \in [a_i, b_i]$. Depending on the label $\ell_i$, the distance between $\mathbf{C}(t)$ and the closest triangle point is specified by one of equations (8), (9), and (7). Let the distance function be $d_i(t)$. Define the function

$$F_i(t) = d_i(t)^2 - r^2 = f_{i,0} + 2f_{i,1}t + f_{i,2}t^2 \tag{14}$$

which is a quadratic function of $t$. If a contact time $T$ (first time or last time) is in $[a_i, b_i]$, then $F_i(T) = 0$. Therefore, it is enough to compute the roots of $F_i(t)$ on $[a_i, b_i]$ or show that it does not have any.

Considering all the functions $F_i(t)$ defined for all the linear components represented by partition $S$, we have a function $F(t)$ that is piecewise quadratic. Moreover, this function is convex. Figure 2 shows the three possible graphs for $F(t)$.

**Figure 2.** The three possible graphs for $F(t)$.




Recall that the elements of the partition $S$ are ordered. The first element has interval $[a_0, b_0] = [-\infty, b_0]$. The last element has interval $[a_{n-1}, b_{n-1}] = [a_{n-1}, +\infty]$. If there is only one element, then the interval is $[a_0, b_0] = [-\infty, +\infty]$.

We may iterate over the elements from first to last to look for an interval $[a_i, b_i]$ whose quadratic function $F_i(t)$ has at least one root on the interval. If none of the functions have roots, then the sphere and triangle never intersect; see the left image of Figure 2.

If there is an index $i$ for which $F_i(t)$ has a repeated root on the interval, then the first and last contact times are that root; see the middle image of Figure 2.

If there is an index $i_0$ for which $F_{i_0}(t)$ has a single root on the interval, the the first contact time $T_0$ is that root. We may continue the iteration to find the last contact time, but in hopes of reducing the computation time, a reverse iteration may be performed starting with the last interval of the partition. Let $i_1$ be the index for which $F_{i_1}(t)$ has a single root on the interval; necessarily, $i_1 > i_0$. This root is the last contact time; see the right image of Figure 2.

# 4 Pseudocode

The following pseudocode shows how to implement the algorithm.

```
enum Label { R0, R1, R2, R01, R12, R20, R012 };
struct Element { Real tmin, tmax; Label label; };
struct Partition { int numElements; Element elements[7]; };
struct Sphere { Point3 C; Real r; };
struct ContactInfo { Real time; Point3 contact; };

struct Triangle
{
    Point3 P0, P1, P2;

    // Quantities derived from P0, P1, and P2.
    Vector3 U2;  // = Cross(P1-P0,P2-P0)/Length(Cross(P1-P0,P2-P0))
    Vector3 U0;  // = (P1-P0)/Length(P1-P0)
    Vector3 U1;  // = Cross(N,U0)
    Point2 Q0;   // = (0,0)
    Point2 Q1;   // = (L,0) = (Length(P1-P0),0)
    Point2 Q2;   // = (A,B) = (Dot(U0,P2-P0),Dot(U1,P2-P0))
    Vector2 E0;  // = Q1 - Q0 = (L,0)
    Vector2 E1;  // = Q2 - Q1 = (A-L,B)
    Vector2 E2;  // = Q0 - Q2 = (-A,-B)
    Vector2 N0;  // = (0,-1)
    Vector2 N1;  // = (B,L-A)/sqrt(B*B+(L-A)*(L-A))
    Vector2 N2;  // = (-B,A)/sqrt(B*B+A*A)

    // Implement this to compute the derived quantities.  It must be called
    // after P0, P1, and P2 are initialized.  If the triangle is used many
    // times in sphere-triangle intersection test, call this function once
    // and re-use the derived quantities in GetContact.
    void ComputeDerived ();

    // Partition projection(C+t*V) = K+t*W by the triangle's Voronoi region.
    Partition ComputePartition (Sphere sphere, Vector3 V);

    // Clip K+t*W against the region identified by 'label'.  Return 'true' when
    // there is overlap, in which case the return value 'tmin' and 'tmax' are valid
    // with tmin < tmax.
    bool GetOverlapInterval (Point2 K, Vector2 W, Label label, Real& tmin, Real& tmax);

    // When the projected line is degenerate to a point K, return the label
    // of the Voronoi region that contains K.  This is a point-in-convex-region test.
    Label GetContainingRegion (Point2 K);

    // Compute the roots of the quadratic F(t) corresponding to the element.
    void ComputeRoots (Sphere sphere, Vector3 V, Element element,
        int& numRoots, ContactInfo contact[2]);

    // Compute the roots of a2*t^2 + 2*a1*t + a0 = 0 on the interval [tmin,tmax].
    void SolveQuadratic (Real tmin, Real tmax, Real a0, Real a1, Real a2,
       int& numRoots, Real roots[2]);
};

// Return 'true' if there is contact, in which case the last parameter of the
// function call is valid.  Return 'false' if there is no contact, in which case
// the last parameter is invalid.  The first contact time/point are at index 0
// of the last parameter.  The last contact time/point are at index 1 of the
// last two parameters.
bool GetContact (Sphere sphere, Vector3 sphereVelocity, Triangle triangle, Vector3 triangleVelocity,
    ContactInfo contact[2])
{
    Vector3 V = sphereVelocity - triangleVelocity;
    Partition S = tri.ComputePartition(sphere, V);

    bool hasRoots = false;
    contact[0].time = +infinity;
    contact[1].time = -infinity;
```

```
    for (int i = 0; i < S.numElements; ++i)
    {
        // Compute the roots of F(t) corresponding to the element.  The pseudocode
        // is not optimized.  Once you detect two roots (distinct or repeated), you
        // can exit the i-loop.  However, beware of incorrect counting of roots
        // because of shared interval endpoints and of numerical round-off errors.
        int numRoots;
        ContactInfo roots[2];
        tri.ComputeRoots(sphere, V, S.elements[i], numRoots, roots);
        for (int j = 0; j < numRoots; ++j)
        {
            if (roots[j].time < contact[0].time)
            {
                contact[0] = roots[j];
                hasRoots = true;
            }
            if (roots[j].time > contact[1].time)
            {
                contact[1] = roots[j];
                hasRoots = true;
            }
        }
    }
    return hasRoots;
}

Partition Triangle::ComputePartition (Sphere sphere, Vector3 V)
{
    Partition S;

    Point3 CmP0 = sphere.C - P0;
    Point2 K(Dot(U0,CmP0), Dot(U1,CmP0));
    Vector2 W(Dot(U0,V), Dot(U1,V));
    if (W != (0,0))
    {
        // Clip the line against each Voronoi region.  This pseudocode is
        // unoptimized (there are no doubt better approaches that reduce
        // computation time).
        S.numElements = 0;
        for (each label of the seven values of Label)
        {
            Real tmin, tmax;
            if (GetOverlapInterval(K,W,label,tmin,tmax))  // must return tmax > tmin
            {
                Element& element = S.elements[S.numElements];
                element.tmin = tmin;
                element.tmax = tmax;
                element.label = R0;
                ++S.numElements;
            }
        }
        S.SortElements();  // sort so that interval times are increasing
    }
    else
    {
        S.numElements = 1;
        S.elements[0].tmin = -infinity;
        S.elements[0].tmax = +infinity;
        S.elements[0].label = GetContainingRegion(K);  // point-in-convex-region tests
    }

    return S;
}

void Triangle::ComputeRoots (Sphere sphere, Vector3 V, Element element,
    int& numRoots, ContactInfo contact[2])
{
    Real sqrRadius = sphere.r * sphere.r;
    Real tmin = element.tmin, tmax = element.tmax;
    Vector3 diff, conCoeff, linCoeff;
    Real invSqrLen, s0, s1, roots[2];
    Real a0, a1, a2;
```

```
int i;

switch (element.label)
{
case R0:
    diff = sphere.C - P0;
    a0 = Dot(diff,diff) - sqrRadius;
    a1 = Dot(V,diff);
    a2 = Dot(V,V);
    SolveQuadratic(tmin, tmax, a0, a1, a2, numRoots, roots);
    for (i = 0; i < numRoots; ++i)
    {
        contact[i].time = roots[i];
        contact[i].point = P0;
    }
    break;
case R1:
    diff = sphere.C - P1;
    a0 = Dot(diff,diff) - sqrRadius;
    a1 = Dot(V,diff);
    a2 = Dot(V,V);
    SolveQuadratic(tmin, tmax, a0, a1, a2, numRoots, roots);
    for (i = 0; i < numRoots; ++i)
    {
        contact[i].time = roots[i];
        contact[i].point = P1;
    }
    break;
case R2:
    diff = sphere.C - P2;
    a0 = Dot(diff,diff) - sqrRadius;
    a1 = Dot(V,diff);
    a2 = Dot(V,V);
    SolveQuadratic(tmin, tmax, a0, a1, a2, numRoots, roots);
    for (i = 0; i < numRoots; ++i)
    {
        contact[i].time = roots[i];
        contact[i].point = P2;
    }
    break;
case R01:
    diff = sphere.C - P0;
    edge = P1 - P0;
    invSqrLen = 1/edge.Dot(edge);
    s0 = Dot(edge,diff)*invSqrLen;
    s1 = Dot(edge,V)*invSqrLen;
    conCoeff = diff - s0*edge;
    linCoeff = V - s1*edge;
    a0 = Dot(conCoeff,conCoeff) - sqrRadius;
    a1 = Dot(conCoeff,linCoeff);
    a2 = Dot(linCoeff,linCOeff);
    SolveQuadratic(tmin, tmax, a0, a1, a2, numRoots, roots);
    for (i = 0; i < numRoots; ++i)
    {
        contact[i].time = roots[i];
        contact[i].point = P0 + (s1*roots[i]+s0)*edge;
    }
    break;
case R12:
    diff = sphere.C - P1;
    edge = P2 - P1;
    invSqrLen = 1/edge.Dot(edge);
    s0 = Dot(edge,diff)*invSqrLen;
    s1 = Dot(edge,V)*invSqrLen;
    conCoeff = diff - s0*edge;
    linCoeff = V - s1*edge;
    a0 = Dot(conCoeff,conCoeff) - sqrRadius;
    a1 = Dot(conCoeff,linCoeff);
    a2 = Dot(linCoeff,linCOeff);
    SolveQuadratic(tmin, tmax, a0, a1, a2, numRoots, roots);
    for (i = 0; i < numRoots; ++i)
    {
```

```
            contact[i].time = roots[i];
            contact[i].point = P1 + (s1*roots[i]+s0)*edge;
        }
        break;
    case R20:
        diff = sphere.C - P2;
        edge = P0 - P2;
        invSqrLen = 1/edge.Dot(edge);
        s0 = Dot(edge,diff)*invSqrLen;
        s1 = Dot(edge,V)*invSqrLen;
        conCoeff = diff - s0*edge;
        linCoeff = V - s1*edge;
        a0 = Dot(conCoeff,conCoeff) - sqrRadius;
        a1 = Dot(conCoeff,linCoeff);
        a2 = Dot(linCoeff,linCOeff);
        SolveQuadratic(tmin, tmax, a0, a1, a2, numRoots, roots);
        for (i = 0; i < numRoots; ++i)
        {
            contact[i].time = roots[i];
            contact[i].point = P2 + (s1*roots[i]+s0)*edge;
        }
        break;
    case R012:
        diff = sphere.C - P0;
        s0 = Dot(U2,diff);
        s1 = Dot(U2,V);
        a0 = s0*s0 - sqrRadius;
        a1 = s0*s1;
        a2 = s1*s1;
        SolveQuadratic(tmin, tmax, a0, a1, a2, numRoots, roots);
        for (i = 0; i < numRoots; ++i)
        {
            contact[i].time = roots[i];
            contact[i].point = sphere.C + roots[i]*V - (s1*roots[i]+s0)*U2;
        }
        break;
    }
}

void Triangle::SolveQuadratic (Real tmin, Real tmax, Real a0, Real a1, Real a2,
    int& numRoots, Real roots[2])
{
    numRoots = 0;
    if (a2 != 0)
    {
        Real discr = a1*a1 - a0*a2;
        if (discr > 0)
        {
            Real rootDiscr = sqrt(discr);
            Real invA2 = 1/a2;
            Real tmp0 = (-a1 - rootdiscr)*invA2;
            Real tmp1 = (-a1 + rootdiscr)*invA2;
            if (tmin <= tmp0 && tmp0 <= tmax)
            {
                roots[numRoots] = tmp0;
                ++numRoots;
            }
            if (tmin <= tmp1 && tmp1 <= tmax)
            {
                roots[numRoots] = tmp1;
                ++numRoots;
            }
        }
        else if (discr == 0)
        {
            Real tmp = -a1/a2;
            if (tmin <= tmp && tmp <= tmax)
            {
                roots[0] = tmp;
                roots[1] = tmp;
                numRoots = 2;
            }
```

```
        }
        // else discr < 0
    }
    else if (a1 != 0)
    {
        Real tmp = -a0/a1;
        if (tmin <= tmp && tmp <= tmax)
        {
            roots[0] = tmp;
            numRoots = 1;
        }
    }
    else if (a0 == 0)
    {
        // This happens only when the sphere is just touching the triangle
        // and neither object is moving.
        roots[0] = 0;
        roots[1] = 0;
        numRoots = 2;
    }
}
```