Intersection of Moving Sphere and Triangle

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

Created: January 21, 2019
Last Modified: September 11, 2020

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 (velocity $\mathbf{0}$) and the sphere is moving relative to the triangle with velocity $\mathbf{V}$. If a first time of contact exists, the contact point is the initial sphere center plus that time multiplied by the sphere's velocity, not the relative velocity.

The contact point can be computed by formulating the problem using Minkowski sums. The sphere has center $\mathbf{C}$ and radius r and is shrunk to its center point. The triangle has vertices $\mathbf{P}_i$ for $0 \le i \le 2$ and is grown by the radius r to produce a *sphere-swept volume* that is the union of all spheres of radius r and with centers at triangle points. In this setting, the sphere center travels along a ray $\mathbf{C} + t\mathbf{V}$ for $t \ge 0$. If the ray intersects the sphere-swept volume at a time $T \ge 0$ and no intersection occurs for $T < 0$, then T is the first time of contact. The corresponding intersection point $\mathbf{K}$ is the contact point.

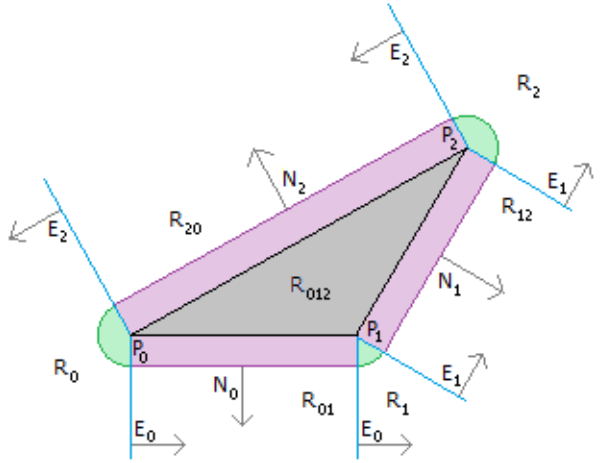
2 Description of the Triangle and its Sphere-Swept Volume

Let the triangle have vertices at positions $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$. The edge vectors are $\mathbf{E}_0 = \mathbf{P}_1 - \mathbf{P}_0$, $\mathbf{E}_1 = \mathbf{P}_2 - \mathbf{P}_1$ and $\mathbf{E}_2 = \mathbf{P}_0 - \mathbf{P}_2$. The plane of the triangle has unit-length normal vector $\mathbf{U} = \mathbf{E}_0 \times \mathbf{E}_1 / |\mathbf{E}_0 \times \mathbf{E}_1|$. Outer-pointing unit-length normal vectors $\mathbf{N}_i$ to the edges $\mathbf{E}_i$ are $\mathbf{N}_i = \mathbf{E}_i \times \mathbf{U} / |\mathbf{E}_i \times \mathbf{U}|$ for $0 \le i \le 2$.

The Voronoi regions for the triangle features (vertices, edges, interior) are those sets of points closest to the features. Region R_{012} is the set of points inside the triangle and includes the edge points. Region R_i is the set of exterior points whose closest feature of the triangle is vertex $\mathbf{P}_i$. Region R_{ij} is the set of exterior points whose closest feature of the triangle is edge $\langle \mathbf{P}_i, \mathbf{P}_j \rangle$.

Figure 1 shows a 2D illustration of the triangle, its corresponding Voronoi regions and the sphere-swept volume of the triangle.

Figure 1. A 2D illustration of the Voronoi regions of the triangle and the sphere-swept volume of the triangle. Imagine this as a top-down view of the 3D triangle and sphere-swept volume.



The triangle edges are drawn in black and the triangle interior is drawn in gray. The black edges and blue lines are the boundaries of the Voronoi regions of the triangle.

The boundary of the sphere-swept volume is decomposed into several surfaces. The boundary of the volume generated by the sphere at a vertex is a *sphere wedge*, which is that portion of the sphere between two planes that are perpendicular to the plane of the triangle and pass through the vertex. The wedges at $\mathbf{P}_0$, $\mathbf{P}_1$ and $\mathbf{P}_2$ are defined by

$$\begin{aligned}
&|\mathbf{X}-\mathbf{P}_0|^2 = r^2, \quad \mathbf{E}_2\cdot(\mathbf{X}-\mathbf{P}_0) \ge 0, \quad \mathbf{E}_0\cdot(\mathbf{X}-\mathbf{P}_0) \le 0\\
&|\mathbf{X}-\mathbf{P}_1|^2 = r^2, \quad \mathbf{E}_0\cdot(\mathbf{X}-\mathbf{P}_1) \ge 0, \quad \mathbf{E}_1\cdot(\mathbf{X}-\mathbf{P}_1) \le 0\\
&|\mathbf{X}-\mathbf{P}_2|^2 = r^2, \quad \mathbf{E}_1\cdot(\mathbf{X}-\mathbf{P}_2) \ge 0, \quad \mathbf{E}_2\cdot(\mathbf{X}-\mathbf{P}_2) \le 0
\end{aligned} \tag{1}$$

respectively.

The triangle edges generate boundaries of the sphere-swept volume that are finite half cylinders. The planes that split the finite cylinders in half are perpendicular to the plane of the triangle and contain the edge points. Define the projection matrices $M_i = I - \mathbf{E}_i\mathbf{E}_i^{\mathsf{T}}/|\mathbf{E}_i|^2$. The half cylinders for edge $\mathbf{E}_0$, $\mathbf{E}_1$ and $\mathbf{E}_2$ are defined by

$$\begin{aligned}
&|M_0(\mathbf{X}-\mathbf{P}_0)|^2 = r^2, \quad \mathbf{E}_0\times\mathbf{U}\cdot M_0(\mathbf{X}-\mathbf{P}_0) \ge 0, \quad \mathbf{E}_0\cdot(\mathbf{X}-\mathbf{P}_0) \ge 0, \quad \mathbf{E}_0\cdot(\mathbf{X}-\mathbf{P}_1) \le 0\\
&|M_1(\mathbf{X}-\mathbf{P}_1)|^2 = r^2, \quad \mathbf{E}_1\times\mathbf{U}\cdot M_1(\mathbf{X}-\mathbf{P}_1) \ge 0, \quad \mathbf{E}_1\cdot(\mathbf{X}-\mathbf{P}_1) \ge 0, \quad \mathbf{E}_1\cdot(\mathbf{X}-\mathbf{P}_2) \le 0\\
&|M_2(\mathbf{X}-\mathbf{P}_2)|^2 = r^2, \quad \mathbf{E}_2\times\mathbf{U}\cdot M_2(\mathbf{X}-\mathbf{P}_2) \ge 0, \quad \mathbf{E}_2\cdot(\mathbf{X}-\mathbf{P}_2) \ge 0, \quad \mathbf{E}_2\cdot(\mathbf{X}-\mathbf{P}_0) \le 0
\end{aligned} \tag{2}$$

respectively.

The triangle interior generates two triangular boundaries on parallel planes that are separated by the diameter $2r$. The triangles and their interiors are defined by

$$\pm\,\mathbf{U}\cdot(\mathbf{X}-\mathbf{P}_0) = r, \;\; \mathbf{E}_0\times\mathbf{U}\cdot(\mathbf{X}-\mathbf{P}_0) \le 0, \;\; \mathbf{E}_1\times\mathbf{U}\cdot(\mathbf{X}-\mathbf{P}_1) \le 0, \;\; \mathbf{E}_2\times\mathbf{U}\cdot(\mathbf{X}-\mathbf{P}_2) \le 0 \tag{3}$$

All computations in equations (1), (2) and (3) avoid normalized vectors. This allows us to compute the contact time and contact point using exact arithmetic (rational arithmetic and quadratic fields).

3 Testing for Initial Overlap

It is possible that the sphere and triangle overlap at time 0. The intersection can be a region of positive volume, in which case there is no official first contact point. It is also possible that the sphere and triangle initially intersect in a single point, in which case there is a contact point. We can handle this by computing the distance from the initial sphere center to the triangle and testing whether it is smaller or equal to the sphere radius. The sphere center is $\mathbf{C}$, the sphere radius is r and the triangle is T. The sphere and triangle overlap or are in initial contact when $\text{Distance}(\mathbf{C}, T) \le r$. Overlap occurs when the inequality is strict and initial contact occurs when the distance and radius are equal. See Distance Between Point and Triangle in 3D for details on computing the distance between a point and a triangle. The algorithm also produces the triangle point $\mathbf{K}$ closest to the input point $\mathbf{C}$, in which case $\mathbf{K}$ might as well be reported as the moving-sphere-triangle contact point with contact time $t = 0$. Source code hyperlinks are provided in the PDF.

4 Computing the Intersection of Ray and Sphere-Swept Volume

This section is relevant when the distance from the initial sphere center to the triangle is larger than the sphere radius; that is, the initial sphere and triangle are separated. The ray is $\mathbf{X}(t) = \mathbf{C} + t\mathbf{V}$ with $t \geq 0$. The first contact time $\bar{t} > 0$, if it exists, has the property that the points $\mathbf{C} + t\mathbf{V}$ for $t \in [0, \bar{t})$ are outside the sphere-swept volume and $\mathbf{K} = \mathbf{C} + \bar{t}\mathbf{V}$ is on the boundary of the sphere-swept volume.

4.1 Computing the Intersection of Ray and Sphere Wedges

The idea is illustrated for the sphere wedge generated by vertex $\mathbf{P}_0$. Define $\boldsymbol{\Delta}_0 = \mathbf{C} - \mathbf{P}_0$. Substituting the parameterized ray into the sphere equation of (1) for $\mathbf{P}_0$, we obtain the quadratic equation

$$Q(t) = |\mathbf{V}|^2 t^2 + 2(\mathbf{V} \cdot \boldsymbol{\Delta}_0)t + |\boldsymbol{\Delta}_0|^2 - r^2 = 0 \tag{4}$$

The quadratic roots are

$$t = \frac{\alpha_0 \pm \sqrt{\beta_0}}{|\mathbf{V}|^2} \tag{5}$$

where $\alpha_0 = -\mathbf{V} \cdot \boldsymbol{\Delta}_0$ and $\beta_0 = (\mathbf{V} \cdot \boldsymbol{\Delta}_0)^2 - |\mathbf{V}|^2(|\boldsymbol{\Delta}_0|^2 - r^2) = \alpha_0^2 - |\mathbf{V}|^2(|\boldsymbol{\Delta}_0|^2 - r^2)$.

In order to intersect the sphere, the roots must be real-valued, which implies $\beta_0 \geq 0$.

If the points $\mathbf{C} + t\mathbf{V}$ increase in distance from $\mathbf{P}_0$ as t increases from 0, the ray will not intersect the sphere. The argument is geometric. We have a plane $\boldsymbol{\Delta}_0 \cdot (\mathbf{X} - \mathbf{C}) = 0$ that contains the initial sphere center. If the ray is on the side of the plane opposite that of the sphere, the ray cannot intersect the sphere. This condition is characterized by

$$0 \geq \boldsymbol{\Delta}_0 \cdot (\mathbf{X}(t) - \mathbf{C}) = t\boldsymbol{\Delta}_0 \cdot \mathbf{V} = -t\alpha_0 \tag{6}$$

for all $t \geq 0$. To have a first point of contact it is necessary that $\alpha_0 \geq 0$.

Finally, the first time of contact must be positive, so we need the smallest quadratic root to be positive. The initial separation of the sphere and triangle ensure $|\boldsymbol{\Delta}_0|^2 - r^2 > 0$, in which case

$$(\alpha_0 - \sqrt{\beta_0})(\alpha_0 + \sqrt{\beta_0}) = \alpha_0^2 - \beta_0 = |\mathbf{V}|^2(|\boldsymbol{\Delta}_0|^2 - r^2) > 0 \tag{7}$$

Knowing that $\alpha_0 \geq 0$ and $\beta_0 \geq 0$, we conclude that $\alpha_0 - \sqrt{\beta_0} \geq 0$. The intersection of the ray and sphere occurs at time

$$\bar{t} = \frac{\alpha_0 - \sqrt{\beta_0}}{|\mathbf{V}|^2}, \quad \alpha_0 \geq 0, \beta_0 \geq 0 \tag{8}$$

If there is a point of intersection, we must determine whether the point is on the sphere wedge. This is simply a matter of evaluating the inequalities of equation (1),

$$\delta_{20} + \nu_2\bar{t} \geq 0, \quad \delta_{00} + \nu_0\bar{t} \leq 0 \tag{9}$$

where $\delta_{ji} = \mathbf{E}_j \cdot \boldsymbol{\Delta}_i$ and $\nu_j = \mathbf{E}_j \cdot \mathbf{V}$.

The three ray-sphere-wedge intersection tests at $\mathbf{P}_i$ are summarized by the following where $0 \leq i \leq 2$ and $j = (i + 2) \bmod 3$,

$$\bar{t} = \frac{\alpha_i - \sqrt{\beta_i}}{|\mathbf{V}|^2}, \quad \delta_{ji} + \nu_j\bar{t} \geq 0, \quad \delta_{ii} + \nu_i\bar{t} \leq 0 \tag{10}$$

4.2 Computing the Intersection of Ray and Half Cylinders

The idea is illustrated for the half cylinder generated by edge $\langle \mathbf{P}_0, \mathbf{P}_1 \rangle$. The construction effectively projects various vectors of interest onto a plane perpendicular to the edge. Define $\mathbf{\Delta}_0 = \mathbf{C} - \mathbf{P}_0$, $\widehat{\mathbf{V}} = M_0\mathbf{V}$ and $\widehat{\mathbf{\Delta}}_0 = M_0\mathbf{\Delta}_0$. Substituting the parameterized ray into the cylinder equation of (2) for the edge, we obtain the quadratic equation

$$Q(t) = |\widehat{\mathbf{V}}|^2 t^2 + 2(\widehat{\mathbf{V}} \cdot \widehat{\mathbf{\Delta}}_0)t + |\widehat{\mathbf{\Delta}}_0|^2 - r^2 = 0 \tag{11}$$

If $\widehat{\mathbf{V}} = \mathbf{0}$, the quadratic polynomial reduces to a constant which is positive because the sphere and triangle are initially separated. We can test for a zero projected velocity vector and exit early if it is. Therefore, in the remainder of this section assume that $\widehat{\mathbf{V}}$ is not the zero vector. The quadratic roots are

$$t = \frac{\alpha_0 \pm \sqrt{\beta_0}}{|\widehat{\mathbf{V}}|^2} \tag{12}$$

where $\alpha_0 = -\widehat{\mathbf{V}} \cdot \widehat{\mathbf{\Delta}}_0$ and $\beta_0 = (\widehat{\mathbf{V}} \cdot \widehat{\mathbf{\Delta}}_0)^2 - |\widehat{\mathbf{V}}|^2(|\widehat{\mathbf{\Delta}}_0|^2 - r^2) = \alpha_0^2 - |\widehat{\mathbf{V}}|^2(|\widehat{\mathbf{\Delta}}_0|^2 - r^2)$.

In order to intersect the cylinder, the roots must be real-valued, which implies $\beta_0 \geq 0$.

If the points $\mathbf{C} + t\mathbf{V}$ increase in distance from the edge as t increases from 0, the ray will not intersect the half cylinder. The argument is geometric and based on the projection of vectors to a plane perpendicular to the cylinder axis. In that plane we have a line $\widehat{\mathbf{\Delta}}_0 \cdot M_0(\mathbf{X} - \mathbf{C}) = 0$ that contains the projection of the initial sphere center. If the projected ray is on the side of the line opposite that of the projected cylinder (a circle), the ray cannot intersect the half cylinder. This condition is characterized by

$$0 \geq \widehat{\mathbf{\Delta}}_0 \cdot M_0(\mathbf{X}(t) - \mathbf{C}) = t\widehat{\mathbf{\Delta}}_0 \cdot \widehat{\mathbf{V}} = -t\alpha_0 \tag{13}$$

for all $t \geq 0$. To have a first point of contact it is necessary that $\alpha_0 \geq 0$.

Similar to the construction for the intersection of a ray and sphere wedge, the candidate contact time is

$$\bar{t} = \frac{\alpha_0 - \sqrt{\beta_0}}{|\widehat{\mathbf{V}}|^2}, \quad \alpha_0 \geq 0, \beta_0 \geq 0 \tag{14}$$

The inequalities of equation (2) must be satisfied by the candidate time to be the time of intersection,

$$\mu_0 + \omega_0\bar{t} \geq 0, \quad \delta_{00} + \nu_0\bar{t} \geq 0, \quad \delta_{01} + \nu_0\bar{t} \leq 0 \tag{15}$$

where δ_{ji} and μ_j are defined as in the ray-sphere-wedge case, and where $\mu_i = \mathbf{E}_i \times \mathbf{U} \cdot \widehat{\mathbf{\Delta}}_i$ and $\omega_i = \mathbf{E}_i \times \mathbf{U} \cdot \widehat{\mathbf{V}}$.

The three ray-half-cylinder intersection tests at edge $\langle \mathbf{P}_i, \mathbf{P}_j \rangle$ are summarized by the following where $0 \leq i \leq 2$ and $j = (i + 1) \bmod 3$,

$$\bar{t} = \frac{\alpha_i - \sqrt{\beta_i}}{|\widehat{\mathbf{V}}|^2}, \quad \mu_i + \omega_i\bar{t} \geq 0, \quad \delta_{ii} + \nu_i\bar{t} \geq 0, \quad \delta_{ij} + \nu_i\bar{t} \leq 0 \tag{16}$$

4.3 Computing the Intersection of Ray and Triangular Boundaries

The triangular boundaries are defined in equation (3), namely $\mathbf{U} \cdot (\mathbf{X} - \mathbf{P}_0) = r$ and $-\mathbf{U} \cdot (\mathbf{X} - \mathbf{P}_0) = r$. The plane equations are written so that the normal vectors point outside the sphere-swept volume.

Knowing that the sphere and triangle are initially separated, the only ray-plane intersections we need consider are those for which the ray direction $\mathbf{V}$ and the plane normal form an obtuse angle. If the initial sphere center is on the positive side of the plane with normal $\mathbf{U}$, the sphere moves away from the plane when $\mathbf{V} \cdot \mathbf{U} \geq 0$. If the dot product is negative, the ray will intersect the plane but not necessarily inside the triangle. If the initial sphere center is on the positive side of the plane with normal $-\mathbf{U}$, the sphere moves away from the plane when $\mathbf{V} \cdot (-\mathbf{U}) \geq 0$. If the dot product is negative, the ray will intersect the plane but not necessarily inside the triangle.

The analysis here applies to both planes where $\sigma \in \{-1, 1\}$. The planes are $\sigma\mathbf{U} \cdot (\mathbf{X} - \mathbf{P}_0) = r$. If $\sigma\mathbf{U} \cdot \mathbf{V} \geq 0$, the sphere moves away from the closest triangular boundary, so there will be no intersection. Now consider the case when $\sigma\mathbf{U} \cdot \mathbf{V} < 0$. Define $\mathbf{\Delta}_0 = \mathbf{C} - \mathbf{P}_0$, substitute the parameterized ray into the plane equation and solve for the candidate contact time

$$\bar{t} = \frac{\sigma r - \mathbf{U} \cdot \mathbf{\Delta}_0}{\mathbf{U} \cdot \mathbf{V}} \tag{17}$$

We are guaranteed by the geometric arguments that $\bar{t} \geq 0$; it is the case that $r < \sigma\mathbf{U} \cdot \mathbf{\Delta}$ when the initial sphere and triangle are separated. The inequality constraints of equation (3) must apply,

$$\phi_i + \psi_i \bar{t} \leq 0 \tag{18}$$

where $\phi_i = \mathbf{E}_i \times \mathbf{U} \cdot \mathbf{\Delta}_i$ and $\psi_i = \mathbf{E}_i \times \mathbf{U} \cdot \mathbf{V}$ for $0 \leq i \leq 2$.

4.4 Pseudocode for the Query

We have 8 subboundaries of the sphere-swept volume to test. For a fast no-intersection, it is convenient to apply the ray-triangular-boundaries test first. The ray-half-cylinder tests are applied second and the ray-sphere-wedge test are applied third. Listing 1 contains pseudocode for the query.

Listing 1. The query for computing first time of contact (if any) and the corresponding contact point for a moving sphere and a moving triangle. The code is unoptimized in the sense that various dot and cross products might be computed multiple times throughout the code.

```
struct Sphere
{
    Vector3 C;  // center
    Real r;     // radius
    Real rsqr;  // square of radius
};

struct Triangle
{
    Vector3 P[3];     // vertices

    // Precomputed quantities.  These can be computed on-demand by FindIntersection
    // with additional logic (Boolean variables to indicate whether or not a
    // quantity needs to be computed the first time).
    Vector3 E[3];     // edges P[1]-P[0], P[2]-P[1], P[0]-P[2]
    Real sqrLenE[3];  // Dot(E[0],E[0]), Dot(E[1],E[1]), Dot(E[2],E[2])
    Vector3 U;        // U = Cross(E[0],E[1])/Length(Cross(E[0],E[1]))
    Vector3 ExU[3];   // edge normals Cross(E[0],U), Cross(E[1],U), Cross(E[2],U)
};

enum IntersectionType
{
    OVERLAP,
    CONTACT
    NO_CONTACT
```

```
};

IntersectionType FindIntersection(Sphere sphere, Vector3 sphereVelocity,
    Triangle tri, Vector3 triangleVelocity, Real& contactTime, Vector3& contactPoint)
{
    // Test for initial overlap or contact.
    Vector3 closestTrianglePoint;
    Real sqrDistance = ComputeSquaredDistance(sphere.C, tri, closestTrianglePoint);
    if (sqrDistance <= sphere.rsqr)
    {
        contactTime = 0;
        contactPoint = closestTrianglePoint;
        return (sqrDistance < sphere.rsqr ? OVERLAP : CONTACT);
    }

    // To get here, the sphere and tri are initially separated.
    // Compute the velocity of the sphere relative to the tri.
    Vector3 V = sphereVelocity - triangleVelocity;
    Real sqrLenV = Dot(V, V);
    if (sqrLenV == 0)
    {
        // The sphere and tri are separated and the sphere is not
        // moving relative to the tri, so there is no contact.
        contactTime = 0;  // invalid
        contactPoint = { 0, 0, 0 };  // invalid
        return NO_CONTACT;
    }

    // NOTE: If the triangle precomputed quantities are not provided in the
    // Triangle data structure, they can be computed here for use by the
    // remainder of the query.

    // Compute the vectors from the tri vertices to the sphere center.
    Vector3 Delta[3];
    for (int i = 0; i < 3; ++i)
    {
        Delta[i] = sphere.C - tri.P[i];
    }

    // Determine where the sphere center is located relative to the
    // planes of the tri offset faces of the sphere-swept volume.
    Real dotUDelta0 = Dot(tri.U, Delta[0]);
    if (dotUDelta0 >= sphere.r)
    {
        // The sphere is on the positive side of Dot(U,X-C) = r.  If
        // the sphere will contact the sphere-swept volume at a
        // triangular face, it can do so only on the face of the
        // aforementioned plane.
        Real dotUV = Dot(tri.U, V);
        if (dotUV >= 0)
        {
            // The sphere is moving away from, or parallel to, the
            // plane of the tri, so there is no contact.
            contactTime = 0;  // invalid
            contactPoint = { 0, 0, 0 };  // invalid
            return NO_CONTACT;
        }

        Real tbar = (sphere.r - dotUDelta0) / dotUV;
        if (Dot(tri.ExU[0], Delta[0]) + Dot(tri.ExU[0], V) * tbar <= 0 and
            Dot(tri.ExU[1], Delta[1]) + Dot(tri.ExU[1], V) * tbar <= 0 and
            Dot(tri.ExU[2], Delta[2]) + Dot(tri.ExU[2], V) * tbar <= 0)
        {
            contactTime = tbar;
            contactPoint = sphere.C + tbar * sphereVelocity;
            return CONTACT;
        }
    }
    else if (dotUDelta0 <= -sphere.r)
    {
        // The sphere is on the positive side of Dot(-U,X-C) = r.  If
        // the sphere will contact the sphere-swept volume at a
```

```
98          // triangular face, it can do so only on the face of the
99          // aforementioned plane.
100         Real dotUV = Dot(tri.U, V);
101         if (dotUV <= 0)
102         {
103             // The sphere is moving away from, or parallel to, the
104             // plane of the tri, so there is no contact.
105             contactTime = 0;  // invalid
106             contactPoint = { 0, 0, 0 };  // invalid
107             return NO_CONTACT;
108         }
109
110         Real tbar = (-sphere.r - dotUDelta0) / dotUV;
111         if (Dot(tri.ExU[0], Delta[0]) + Dot(tri.ExU[0], V) * tbar <= 0 and
112             Dot(tri.ExU[1], Delta[1]) + Dot(tri.ExU[1], V) * tbar <= 0 and
113             Dot(tri.ExU[2], Delta[2]) + Dot(tri.ExU[2], V) * tbar <= 0)
114         {
115             contactTime = tbar;
116             contactPoint = sphere.C + tbar * sphereVelocity;
117             return CONTACT;
118         }
119     }
120     // else: The ray-sphere-swept-volume contact point (if any) cannot
121     // be on a triangular face of the sphere-swept-volume.
122
123     // The sphere is moving towards the slab between the two planes
124     // of the sphere-swept volume triangular faces.  Determine whether
125     // the ray intersects the half cylinders or sphere wedges of the
126     // sphere-swept volume.
127
128
129     // Test for contact with half cylinders of the sphere-swept volume.
130     // First, precompute some dot products required in the computations.
131     Real del[3], delp[i], nu[3];
132     for (int im1 = 2, i = 0; i < 3; im1 = i++)
133     {
134         del[i] = Dot(tri.E[i], Delta[i]);
135         delp[im1] = Dot(tri.E[im1], Delta[i]);
136         nu[i] = Dot(tri.E[i], V);
137     }
138
139     for (int i = 2, ip1 = 0; ip1 < 3; i = ip1++)
140     {
141         Vector3 hatV = V - tri.E[i] * (nu[i] / sqrLenE[i]);
142         Real sqrLenHatV = Dot(hatV, hatV);
143         if (sqrLenHatV > 0)
144         {
145             Vector3 hatDelta = Delta[i] - tri.E[i] * del[i] / tri.sqrLenE[i];
146             Real alpha = -Dot(hatV, hatDelta);
147             if (alpha >= 0)
148             {
149                 Real sqrLenHatDelta = Dot(hatDelta, hatDelta);
150                 Real beta = alpha * alpha - sqrLenHatV * (sqrLenHatDelta - sphere.rsqr);
151                 if (beta >= 0)
152                 {
153                     Real tbar = (alpha - sqrt(beta)) / sqrLenHatV;
154                     Real mu = Dot(tri.ExU[i], Delta[i]);
155                     Real omega = Dot(tri.ExU[i], hatV);
156                     if (mu + omega * tbar >= 0)
157                     {
158                         if (del[i] + nu[i] * tbar >= 0)
159                         {
160                             if (delp[i] + nu[i] * tbar <= 0)
161                             {
162                                 // The constraints are satisfied, so tbar is the first
163                                 // time of contact.
164                                 contactTime = tbar;
165                                 contactPoint = sphere.C + tbar * sphereVelocity;
166                                 return CONTACT;
167                             }
168                         }
169                     }
```

```
                }
            }
        }
    }

    // Test for contact with the sphere wedges.  We know that |V|^2 > 0 because
    // of a previous early-exit test.
    for (int im2 = 2, i = 0; i < 3; im1 = i++)
    {
        Real alpha = -Dot(V, Delta[i]);
        if (alpha >= 0)
        {
            Real sqrLenDelta = Dot(Delta[i], Delta[i]);
            Real beta = alpha * alpha - sqrLenV * (sqrLenDelta - sphere.rsqr);
            if (beta >= 0)
            {
                Real tbar = (alpha - sqrt(beta)) / sqrLenV;
                if (delp[im1] + nu[im1] * tbar >= 0)
                {
                    if (del[i] + nu[i] * tbar <= 0)
                    {
                        // The constraints are satisfied, so tbar is the
                        // first time of contact.
                        contactTime = tbar;
                        contactPoint = sphere.C + tbar * sphereVelocity;
                        return CONTACT;
                    }
                }
            }
        }
    }

    // The ray does not intersect the sphere-swept volume, so the sphere and
    // tri do not come into contact.
    contactTime = 0;  // invalid
    contactPoint = { 0, 0, 0 };  // invalid
    return NO_CONTACT;
}
```

---

5 Computing Contact Information using Exact Arithmetic

As is nearly always the case when using floating-point arithmetic to solve geometric problems, rounding errors can cause problems. For example, it is possible that the ray intersects the sphere-swept volume at a point close to the hemicircle that is shared by a sphere wedge and a half cylinder. Theoretically, the ray must intersect either the sphere wedge at one of its interior points, or the half cylinder at one of its interior points, or the hemicircle. With floating-point arithmetic, it is possible that all three tests lead to a no-intersection result. This is similar to the problem of salt-and-pepper noise when ray tracing a triangle mesh, the solution usually ensuring that the computations on an edge shared by two triangles have the same expression trees (and hoping the compiler does not optimize two equivalent trees to low-level code that has different rounding error behavior).

For the problem at hand, it is possible to use exact arithmetic to generate a theoretically correct result. This arithmetic involves a mixture of rational arithmetic and symbolic arithmetic of quadratic fields. The sphere center, sphere radius, triangle vertices and velocity vector consist of finite floating-point numbers, which means they all have exact rational representations. The symbolic arithmetic of quadratic fields involves manipulating expressions of the form $x + y\sqrt{d}$, where x, y and d are rational numbers. Specifically, the first times for the various intersections of ray with subboundaries of the sphere-swept volume are elements of quadratic fields, and they can be manipulated symbolically during comparisons to obtain the minimum

first time. Once this minimum time is computed, only then do we convert the square-root quantity to a floating-point number, which generally will be an approximation to the theoretical result.

5.1 Operations Involving Exact Arithmetic

For a single quadratic field with elements of the form $x+y\sqrt{d}$, where x, y and $d \geq 0$ are rational, the operations of interest for the problem at hand are addition, subtraction, multiplication, division and comparison to zero. The addition and subtraction are

$$(x_0 + y_0\sqrt{d}) \pm (x_1 + y_1\sqrt{d}) = (x_0 \pm x_1) + (y_0 \pm y_1)\sqrt{d} \tag{19}$$

Multiplication is

$$(x_0 + y_0\sqrt{d})(x_1 + y_1\sqrt{d}) = (x_0x_1 + y_0y_1d) + (x_0y_1 + x_1y_0)\sqrt{d} \tag{20}$$

Division is

$$\frac{x_0 + y_0\sqrt{d}}{x_1 + y_1\sqrt{d}} = \frac{x_0x_1 - y_0y_1d}{x_1^2 - y_1^2d} + \frac{y_0x_1 - x_0y_1}{x_1^2 - y_1^2d}\sqrt{d} \tag{21}$$

When d is not the square of a rational number, it is the case that $\sqrt{d}$ is irrational. The division is well defined for $x_1 + y_1\sqrt{d}$ when at least one of x_1 or y_1 is not zero; the denominator cannot be zero in this case. To see this, if $y_1 = 0$, then $x_1 \neq 0$ and the divisor is a nonzero rational number. If $y_1 \neq 0$, then $x_1 + y_1\sqrt{d} = 0$ implies $\sqrt{d} = -x_1/y_1$. The left-hand side of the last expression is irrational whereas the right-hand of the last expression is rational, which is not possible. Now if d is the square of a rational number, say $d = r^2$ where r is rational, then 0 has infinitely many representations $0 = (-yr) + y\sqrt{d}$ for any rational y.

Comparison requires several steps. Listing 2 contains pseudocode for the 6 types of comparison.

Listing 2. Comparison tests of $x + y\sqrt{d}$ to 0.

```
bool EqualZero(Real x, Real y, Real d)
{
    if (d == 0 or y == 0)
    {
        return x == 0;
    }
    else if (y > 0)
    {
        if (x >= 0)
        {
            return false;
        }
        else
        {
            return x * x - y * y * d == 0;
        }
    }
    else // y < 0
    {
        if (x <= 0)
        {
            return false;
        }
        else
        {
            return x * x - y * y * d == 0;
        }
    }
```

```
}

bool NotEqualZero(Real x, Real y, Real d)
{
    if (d == 0 or y == 0)
    {
        return x != 0;
    }
    else if (y > 0)
    {
        if (x >= 0)
        {
            return true;
        }
        else
        {
            return x * x - y * y * d != 0;
        }
    }
    else // y < 0
    {
        if (x <= 0)
        {
            return true;
        }
        else
        {
            return x * x - y * y * d != 0;
        }
    }
}

bool LessThanZero(Real x, Real y, Real d)
{
    if (d == 0 or y == 0)
    {
        return x < 0;
    }
    else if (y > 0)
    {
        if (x >= 0)
        {
            return false;
        }
        else
        {
            return x * x - y * y * d > 0;
        }
    }
    else // y < 0
    {
        if (x <= 0)
        {
            return true;
        }
        else
        {
            return x * x - y * y * d < 0;
        }
    }
}

bool GreaterThanZero(Real x, Real y, Real d)
{
    if (d == 0 or y == 0)
    {
        return x > 0;
    }
    else if (y > 0)
    {
        if (x >= 0)
        {
```

```
            return true;
        }
        else
        {
            return x * x - y * y * d < 0;
        }
    }
    else // y < 0
    {
        if (x <= 0)
        {
            return false;
        }
        else
        {
            return x * x - y * y * d > 0;
        }
    }
}

bool LessThanOrEqualZero(Real x, Real y, Real d)
{
    if (d == 0 or y == 0)
    {
        return x <= 0;
    }
    else if (y > 0)
    {
        if (x >= 0)
        {
            return false;
        }
        else
        {
            return x * x - y * y * d >= 0;
        }
    }
    else // y < 0
    {
        if (x <= 0)
        {
            return true;
        }
        else
        {
            return x * x - y * y * d <= 0;
        }
    }
}

bool GreaterThanOrEqualZero(Real x, Real y, Real d)
{
    if (d == 0 or y == 0)
    {
        return x >= 0;
    }
    else if (y > 0)
    {
        if (x >= 0)
        {
            return true;
        }
        else
        {
            return x * x - y * y * d <= 0;
        }
    }
    else // y < 0
    {
        if (x <= 0)
        {
            return false;
```

```
        }
        else
        {
            return x * x - y * y * d >= 0;
        }
    }
}
```

Source code for the arithmetic and comparisons is found in QuadraticField.h. The QuadraticField class stores a single member d. The nested class Element stores x and y for the element $x + y\sqrt{d}$, and the QuadraticField class implements the arithmetic operators and comparisons so that the Element class does not need to store a copy of d for each object of that class.

5.2 Modifications to the Pseudocode for Exact-Arithmetic Comparisons

The lines of Listing 1 are numbered and referenced here. The lines that must be modified for exact arithmetic are discussed here. The type Real is intended to be an exact arithmetic type. In GTE, this will be BSRational<UIntegerAP32>. The FindIntersection signature is modified to return the quadratic field and the quadratic field elements that represent the exact contact time and contact point. The names are contactField, contactTime and contactPoint[].

Line 33 has a call to ComputeSquaredDistance. The assumption is that this function supports exact arithmetic to compute the squared distance and the contact point exactly. The GTE query DCPQuery<Real, Vector3<Real>, Triangle3<Real>> satisfies this requirement. The block of code including lines 36 through 38 becomes

```
contactField = QuadraticField(0);
contactTime.x = 0;
contactTime.y = 0;
for (int j = 0; j < 3; ++j)
{
    contactPoint[j].x = closestTrianglePoint[j];
    contactPoint[j].y = 0;
}
return (sqrDistance < sphere.rsqr ? OVERLAP : CONTACT);
```

Line 67 computes a dot product between the normalized vector $\mathbf{U} = \mathbf{E}_0 \times \mathbf{E}_1$ and Δ_0. The normalization usually introduces floating-point rounding errors. For exact arithmetic, we need to avoid the normalization, so instead use $\mathbf{U} = \mathbf{E}_0 \times \mathbf{E}_1$. The cross products tri.ExU[] are computed exactly using $\mathbf{E}_i \times (\mathbf{E}_0 \times \mathbf{E}_1)$. The block of code including lines 67 through 93 are controlled by the Boolean expression shown next,

```
// Quadratic field elements are x + y * |U|, where |U| = sqrt(Dot(U,U)).
QuadraticField ufield(Dot(tri.U, tri.U));

// The plane is Dot(U, X-C) - r * |U| = 0 and points on the positive side
// satisfy Dot(U, X-C) - r * |U| >= 0.
QuadraticField::Element element(Dot(tri.U, Delta[0]), -sphere.r));
if (ufield.GreaterThanOrEqualZero(element))
{
    // The sphere is on the positive side...
}
```

The expression dotUV on line 74 need only be tested for nonnegativity. It does not matter whether or not $\mathbf{U}$ is normalized, so no change is required for that test.

The block of code including lines 84 through 92 can be reformulated as a loop,

```
Real tbar = (sphere.r - dotUDelta0) / dotUV;
bool foundContact = true;
for (int i = 0; i < 3; ++i)
{
    Real phi = Dot(tri.ExU[i], Delta[i]);
    Real psi = Dot(tri.ExU[i], V);
    if (phi + psi * tbar > (Real)0)
    {
        foundContact = false;
        break;
    }
}
if (foundContact)
{
    contactTime = tbar;
    contactPoint = sphere.C + tbar * sphereVelocity;
    return CONTACT;
}
```

The replacement code is shown next. To avoid the relatively expensive exact-arithmetic division by $\mathbf{U} \cdot \mathbf{V}$, the inequality constraints are multiplied first by positive number $-\mathbf{U} \cdot \mathbf{V}$. The division is deferred until we know there is contact.

```
bool foundContact = true;
for (int i = 0; i < 3; ++i)
{
    Real phi = Dot(tri.ExU[i], Delta[i]);
    Real psi = Dot(tri.ExU[i], V);
    QuadraticField::Element arg(psi * element.x - phi * dotUV, psi * element.y);
    if (ufield.GreaterThanZero(arg))
    {
        foundContact = false;
        break;
    }
}
if (foundContact)
{
    contactField = ufield;
    contactTime.x = -element.x / dotUV;
    contactTime.y = -element.y / dotUV;
    for (int j = 0; j < 3; ++j)
    {
        contactPoint[j].x = sphere.C[j] + contactTime.x * sphereVelocity[j];
        contactPoint[j].y = contactTime.y * sphereVelocity[j];
    }
    return CONTACT;
}
```

Similar code replacement applies to the block of code including lines 94 through 119, but in this case the inequality constraints are multiplied first by the positive number $\mathbf{U} \cdot \mathbf{V}$.

The block of code involving lines 153 through 169 have several comparisons that must be converted to use exact arithemtic. The divisions by $|\hat{\mathbf{V}}|^2$ are deferred until we know there is contact. The replacement block is

```
QuadraticField bfield(beta);
Real mu = Dot(tri.ExU[i], Delta[i]);
Real omega = Dot(tri.ExU[i], hatV);
QuadraticField::Element arg0(mu * sqrLenHatV + omega * alpha, -omega);
if (bfield.GreaterThanOrEqualZero(arg0))
{
    QuadraticField::Element arg1(del[i] * sqrLenHatV + nu[i] * alpha, -nu[i]);
    if (bfield.GreaterThanOrEqualZero(arg1))
    {
        QuadraticField::Element arg2(delp[i] * sqrLenHatV + nu[i] * alpha, -nu[i]);
        if (bfield.LessThanOrEqualZero(arg2))
        {
```

```
            contactField = bfield;
            contactTime.x = alpha / sqrLenHatV;
            contactTime.y = (Real)-1 / sqrLenHatV;
            for (int j = 0; j < 3; ++j)
            {
                contactPoint[j].x = sphere.C[j] + contactTime.x * sphereVelocity[j];
                contactPoint[j].y = contactTime.y * sphereVelocity[j];
            }
            return CONTACT;
        }
    }
}
```

The block of code involving lines 186 through 197 are have a couple of comparisons that must be converted to use exact arithemtic. The divisions by $|\mathbf{V}|^2$ are deferred until we know there is contact. The replacement block is

```
QuadraticField bfield(beta);
QuadraticField::Element arg0(delp[im1] * sqrLenV + nu[im1] * alpha, -nu[im1]);
if (bfield.GreaterThanOrEqualZero(arg0))
{
    QuadraticField::Element arg1(del[i] * sqrLenV + nu[i] * alpha, -nu[i]);
    if (bfield.LessThanOrEqualZero(arg1))
    {
        contactField = bfield;
        contactTime.x = alpha / sqrLenV;
        contactTime.y = (Real)-1 / sqrLenV;
        for (int j = 0; j < 3; ++j)
        {
            contactPoint[j].x = sphere.C[j] + contactTime.x * sphereVelocity[j];
            contactPoint[j].y = contactTime.y * sphereVelocity[j];
        }
        return CONTACT;
    }
}
```

Listing 3 contains pseudocode for the exact-arithmetic query.

Listing 3. The query for computing first time of contact (if any) and the corresponding contact point for a moving sphere and a moving triangle. The code uses exact arithmetic support; the template parameter Real must be an exact-arithmetic type.

```
struct Sphere
{
    Vector3 C;  // center
    Real r;     // radius
    Real rsqr;  // square of radius
};

struct Triangle
{
    Vector3 P[3];     // vertices

    // Precomputed quantities.  These can be computed on-demand by FindIntersection
    // with additional logic (Boolean variables to indicate whether or not a
    // quantity needs to be computed the first time).
    Vector3 E[3];     // edges P[1]-P[0], P[2]-P[1], P[0]-P[2]
    Real sqrLenE[3];  // Dot(E[0],E[0]), Dot(E[1],E[1]), Dot(E[2],E[2])
    Vector3 U;        // U = Cross(E[0],E[1]), [unnormalized]
    Real sqrLenU;     // Dot(U,U)
    Vector3 ExU[3];   // edge normals Cross(E[0],U), Cross(E[1],U), Cross(E[2],U)
};

enum IntersectionType
{
    OVERLAP,
```

```
    CONTACT
    NO_CONTACT
};

IntersectionType FindIntersection(Sphere sphere, Vector3 sphereVelocity,
    Triangle tri, Vector3 triangleVelocity, QuadraticField& contactField,
    QuadraticField::Element& contactTime, QuadraticField::Element contactPoint[3])
{
    // Test for initial overlap or contact.
    Vector3 closestTrianglePoint;
    Real sqrDistance = ComputeSquaredDistance(sphere.C, tri, closestTrianglePoint);
    if (sqrDistance <= sphere.rsqr)
    {
        contactField = QuadraticField(0)
        contactTime.x = 0;
        contactTime.y = 0;
        for (int j = 0; j < 3; ++j)
        {
            contactPoint[j].x = closestTrianglePoint[j];
            contactPoint[j].y = 0;
        }
        return (sqrDistance < sphere.rsqr ? OVERLAP : CONTACT);
    }

    // To get here, the sphere and tri are initially separated.
    // Compute the velocity of the sphere relative to the tri.
    Vector3 V = sphereVelocity - triangleVelocity;
    Real sqrLenV = Dot(V, V);
    if (sqrLenV == 0)
    {
        // The sphere and tri are separated and the sphere is not
        // moving relative to the tri, so there is no contact.
        contactField = QuadraticField(0);  // invalid
        contactTime.x = 0;  // invalid
        contactTime.y = 0;  // invalid
        for (int j = 0; j < 3; ++j)  // invalid
        {
            contactPoint[j].x = 0;
            contactPoint[j].y = 0;
        }
        return NO_CONTACT;
    }

    // NOTE: If the triangle precomputed quantities are not provided in the
    // Triangle data structure, they can be computed here for use by the
    // remainder of the query.

    // Compute the vectors from the tri vertices to the sphere center.
    Vector3 Delta[3];
    for (int i = 0; i < 3; ++i)
    {
        Delta[i] = sphere.C - tri.P[i];
    }

    // Determine where the sphere center is located relative to the
    // planes of the tri offset faces of the sphere-swept volume.
    QuadraticField ufield(tri.sqrLenU);
    QuadraticField::Element element(Dot(tri.U, Delta[0]), -sphere.r);
    if (ufield.GreaterThanOrEqualZero(element))
    {
        // The sphere is on the positive side of Dot(U,X-C) = r.  If
        // the sphere will contact the sphere-swept volume at a
        // triangular face, it can do so only on the face of the
        // aforementioned plane.
        Real dotUV = Dot(tri.U, V);
        if (dotUV >= 0)
        {
            // The sphere is moving away from, or parallel to, the
            // plane of the tri, so there is no contact.
            contactField = QuadraticField(0);  // invalid
            contactTime.x = 0;  // invalid
            contactTime.y = 0;  // invalid
```

```
            for (int j = 0; j < 3; ++j)  // invalid
            {
                contactPoint[j].x = 0;
                contactPoint[j].y = 0;
            }
            return NO_CONTACT;
        }

        bool foundContact = true;
        for (int i = 0; i < 3; ++i)
        {
            Real phi = Dot(tri.ExU[i], Delta[i]);
            Real psi = Dot(tri.ExU[i], V);
            QuadraticField::Element arg(psi * element.x - phi * dotUV, psi * element.y);
            if (ufield.GreaterThanZero(arg))
            {
                foundContact = false;
                break;
            }
        }
        if (foundContact)
        {
            contactField = ufield;
            contactTime.x = -element.x / dotUV;
            contactTime.y = -element.y / dotUV;
            for (int j = 0; j < 3; ++j)
            {
                contactPoint[j].x = sphere.C[j] + contactTime.x * sphereVelocity[j];
                contactPoint[j].y = contactTime.y * sphereVelocity[j];
            }
            return CONTACT;
        }
    }
    else
    {
        element.y = -element.y;
        if (ufield.LessThanOrEqualZero(element))
        {
            // The sphere is on the positive side of Dot(-U,X-C) = r.  If
            // the sphere will contact the sphere-swept volume at a
            // triangular face, it can do so only on the face of the
            // aforementioned plane.
            Real dotUV = Dot(tri.U, V);
            if (dotUV <= 0)
            {
                // The sphere is moving away from, or parallel to, the
                // plane of the tri, so there is no contact.
                contactField = QuadraticField(0);  // invalid
                contactTime.x = 0;  // invalid
                contactTime.y = 0;  // invalid
                for (int j = 0; j < 3; ++j)  // invalid
                {
                    contactPoint[j].x = 0;
                    contactPoint[j].y = 0;
                }
                return NO_CONTACT;
            }

            bool foundContact = true;
            for (int i = 0; i < 3; ++i)
            {
                Real phi = Dot(tri.ExU[i], Delta[i]);
                Real psi = Dot(tri.ExU[i], V);
                QuadraticField::Element arg(phi * dotUV - psi * element.x, -psi * element.y);
                if (ufield.GreaterThanZero(arg))
                {
                    foundContact = false;
                    break;
                }
            }
            if (foundContact)
            {
```

```
            contactField = ufield;
            contactTime.x = -element.x / dotUV;
            contactTime.y = -element.y / dotUV;
            for (int j = 0; j < 3; ++j)
            {
                contactPoint[j].x = sphere.C[j] + contactTime.x * sphereVelocity[j];
                contactPoint[j].y = contactTime.y * sphereVelocity[j];
            }
            return CONTACT;
        }
    }
    // else: The ray-sphere-swept-volume contact point (if any) cannot
    // be on a triangular face of the sphere-swept-volume.
}

// The sphere is moving towards the slab between the two planes
// of the sphere-swept volume triangular faces.  Determine whether
// the ray intersects the half cylinders or sphere wedges of the
// sphere-swept volume.

// Test for contact with half cylinders of the sphere-swept volume.
// First, precompute some dot products required in the computations.
Real del[3], delp[i], nu[3];
for (int im1 = 2, i = 0; i < 3; im1 = i++)
{
    del[i] = Dot(tri.E[i], Delta[i]);
    delp[im1] = Dot(tri.E[im1], Delta[i]);
    nu[i] = Dot(tri.E[i], V);
}

for (int i = 2, ip1 = 0; ip1 < 3; i = ip1++)
{
    Vector3 hatV = V - tri.E[i] * (nu[i] / sqrLenE[i]);
    Real sqrLenHatV = Dot(hatV, hatV);
    if (sqrLenHatV > 0)
    {
        Vector3 hatDelta = Delta[i] - tri.E[i] * del[i] / tri.sqrLenE[i];
        Real alpha = -Dot(hatV, hatDelta);
        if (alpha >= 0)
        {
            Real sqrLenHatDelta = Dot(hatDelta, hatDelta);
            Real beta = alpha * alpha - sqrLenHatV * (sqrLenHatDelta - sphere.rsqr);
            if (beta >= 0)
            {
                QuadraticField bfield(beta);
                Real mu = Dot(tri.ExU[i], Delta[i]);
                Real omega = Dot(tri.ExU[i], hatV);
                QuadraticField::Element arg0(mu * sqrLenHatV + omega * alpha, -omega);
                if (bfield.GreaterThanOrEqualZero(arg0))
                {
                    QuadraticField::Element arg1(del[i] * sqrLenHatV + nu[i] * alpha, -nu[i]);
                    if (bfield.GreaterThanOrEqualZero(arg1))
                    {
                        QuadraticField::Element arg2(delp[i] * sqrLenHatV + nu[i] * alpha, -nu[i]);
                        if (bfield.LessThanOrEqualZero(arg2))
                        {
                            contactFieldd = bfield;
                            contactTime.x = alpha / sqrLenHatV;
                            contactTime.y = (Real)-1 / sqrLenHatV;
                            for (int j = 0; j < 3; ++j)
                            {
                                contactPoint[j].x = sphere.C[j] + contactTime.x * sphereVelocity[j];
                                contactPoint[j].y = contactTime.y * sphereVelocity[j];
                            }
                            return CONTACT;
                        }
                    }
                }
            }
        }
    }
}
```

```
    // Test for contact with the sphere wedges.  We know that |V|^2 > 0 because
    // of a previous early-exit test.
    for (int im2 = 2, i = 0; i < 3; im1 = i++)
    {
        Real alpha = -Dot(V, Delta[i]);
        if (alpha >= 0)
        {
            Real sqrLenDelta = Dot(Delta[i], Delta[i]);
            Real beta = alpha * alpha - sqrLenV * (sqrLenDelta - sphere.rsqr);
            if (beta >= 0)
            {
                QuadraticField bfield(beta);
                QuadraticField::Element arg0(delp[im1] * sqrLenV + nu[im1] * alpha, -nu[im1]);
                if (bfield.GreaterThanOrEqualZero(arg0))
                {
                    QuadraticField::Element arg1(del[i] * sqrLenV + nu[i] * alpha, -nu[i]);
                    if (bfield.LessThanOrEqualZero(arg1))
                    {
                        contactFieldd = bfield;
                        contactTime.x = alpha / sqrLenV;
                        contactTime.y = (Real)-1 / sqrLenV;
                        for (int j = 0; j < 3; ++j)
                        {
                            contactPoint[j].x = sphere.C[j] + contactTime.x * sphereVelocity[j];
                            contactPoint[j].y = contactTime.y * sphereVelocity[j];
                        }
                        return CONTACT;
                    }
                }
            }
        }
    }

    // The ray does not intersect the sphere-swept volume, so the sphere and
    // tri do not come into contact.
    contactField = QuadraticField(0);  // invalid
    contactTime.x = 0;  // invalid
    contactTime.y = 0;  // invalid
    for (int j = 0; j < 3; ++j)  // invalid
    {
        contactPoint[j].x = 0;
        contactPoint[j].y = 0;
    }
    return NO_CONTACT;
}
```

---

6 Source Code and Sample Application

GTE has an implementation for both floating-point and exact arithmetic; see IntrSphere3Triangle3.h. A sample application used for testing the code is in the folder

GeometricTools/GTE/Samples/Intersection/MovingSphereTriangle

Screen captures are shown in the following figures. In the right-most images, the initial sphere is drawn in light gray. The path of motion is drawn in green. The triangle is the medial object of the sphere-swept volume (cheap alpha blending is used, ignoring correct ordering of geometric primitives). The two triangular faces are drawn in red. The half cylinders for the edges are drawn in magenta. The sphere wedges for the vertices are drawn in blue. The small black dot is the intersection of the ray of motion with the sphere-swept volume. In the right-most images, the final sphere at time of contact is drawn in dark gray. The center of

that sphere is the small sphere corresponding to the black dot. The actual sphere-triangle contact point is not highlighted, but you can see roughly where the final sphere is just touching the triangle. The contact time and contact point values are displayed at the top of the application window.

Figure 2 shows contact with one of the triangular faces.

Figure 2. Examples of the predicted contact time and point for a sphere and triangle, each moving with constant linear velocity.

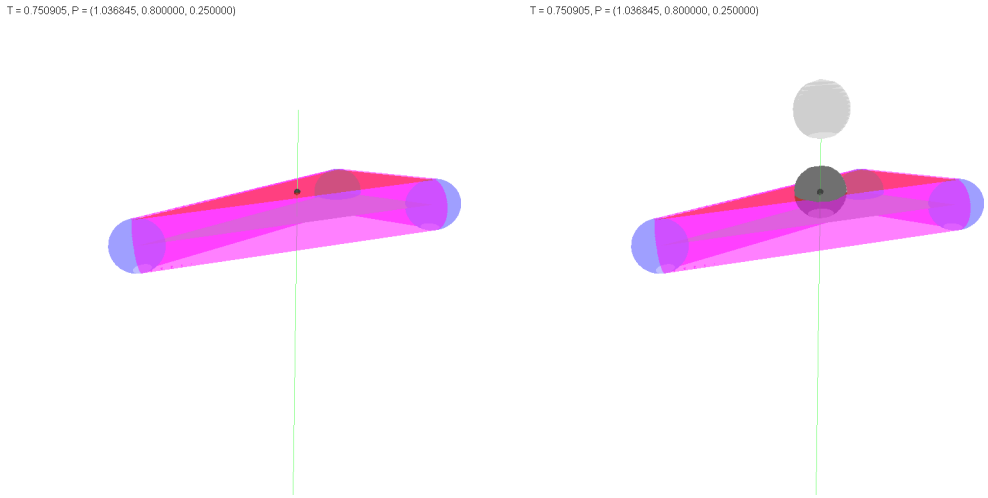


Figure 3 shows contact with one of the half cylinders.

Figure 3. Examples of the predicted contact time and point for a sphere and triangle, each moving with constant linear velocity.

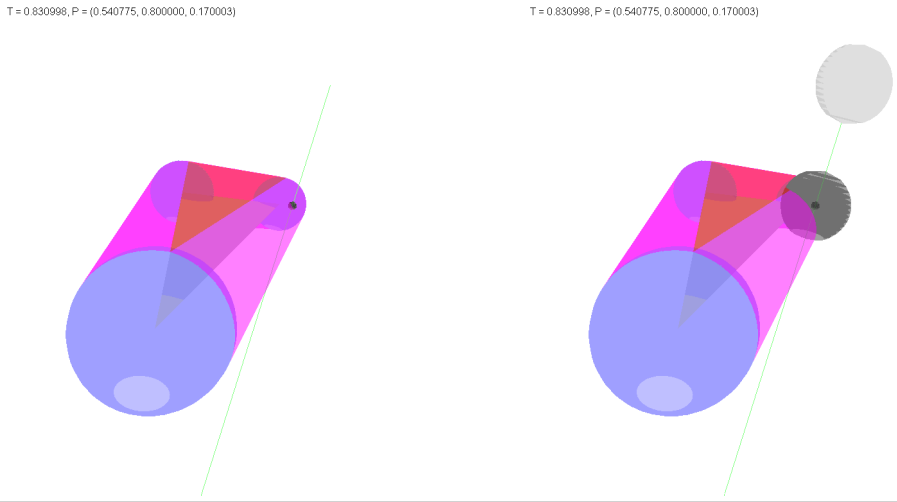


Figure 4 shows contact with one of the sphere wedges.

Figure 4. Examples of the predicted contact time and point for a sphere and triangle, each moving with constant linear velocity.