Collision Detection with Swept Spheres and Ellipsoids

Author: Jorrit Rouwé
Source code at: https://github.com/jrouwe/SweptEllipsoid

1. Introduction

Today most games use convex polygons for collision detection. They are usually stored in a tree like structure to make it possible to quickly reject a lot of polygons when performing intersection tests. In the end however, every intersection test boils down to some primitive versus a polygon test.

One commonly used intersection test is that of a swept sphere (a sphere moving along a line) or a swept ellipsoid (an ellipsoid moving along a line but not rotating) with a static polygon. A game character for example can be represented by a collection of spheres. When the character is moving through the environment we need to detect the first point for which the collection of spheres intersects with the world geometry. After this point is detected we also need an indication of where the character collided so we can compute a sliding direction that prevents the character from getting stuck. As an alternative to a number of spheres we can also use a single ellipsoid as an approximation for the volume of a character.

Polygons used for rendering usually reside somewhere where they are not accessible to the collision system (AGP memory for example) or they are compressed and interleaved with other rendering data like vertex colors. This forces us to store a separate set of polygons for collision. A cheap way of reducing the amount of memory needed for a polygon is to store it as a list of 2D vertices together with a plane equation. Of course you get the extra overhead of storing a plane equation, but when using a BSP (Binary Space Partitioning) tree for example you need to store this plane anyway.

In this article we will first look at a function that can convert a polygon consisting of 3D points into a plane and a list of 2D points. After that we will derive the intersection between a static polygon and a swept sphere and that of a static polygon with a swept ellipsoid. Finally we show that the same algorithm also works on polygons consisting of 3D points in the case that it is not possible (or not desirable) to store a polygon in 2D.

2. Projecting 3D Points on a Plane

A plane is defined by the set of points $\vec{P}$ for which $\vec{N} \cdot \vec{P} + C_p = 0$. We call $\vec{N} = (N_x, N_y, N_z)$ the plane normal and C_p the plane constant. Details on how to create a plane equation can for example be found in [1].

The distance of a point $\vec{P}$ to the plane is: $d_{plane}(\vec{P}) = \left|\vec{P} \cdot \vec{N} + C_p\right|$.

To project a point on the plane we create an orthonormal base for the plane with origin at $-C_p\vec{N}$ and axes $(\vec{U}, \vec{V}, \vec{N})$:

$$\begin{aligned} \vec{U} &= \begin{cases} (N_z, 0, -N_x) \Big/ \sqrt{{N_x}^2 + {N_z}^2} & (N_x > N_y) \\ (0, N_z, -N_y) \Big/ \sqrt{{N_y}^2 + {N_z}^2} & (N_x \le N_y) \end{cases} \\ \vec{V} &= \vec{N} \times \vec{U} \end{aligned}$$

To project a point $\vec{P}$ from world space to plane space use:

$$\vec{P}' = (\vec{P} \cdot \vec{U}, \vec{P} \cdot \vec{V})$$

To project a point $\vec{P}'$ from plane space to world space use:

$$\vec{P} = P_x'\vec{U} + P_y'\vec{V} - C_p\vec{N}$$

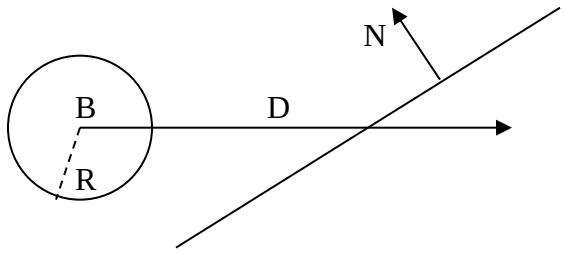
3. Swept Sphere Versus Polygon

3.1. Swept Sphere Versus Plane

The first step in determining if a swept sphere collides with a polygon is to determine the interval over which it intersects with the plane of the polygon.

We use a swept sphere of radius R and center $\vec{C}(t) = \vec{B} + t\vec{D}$, where $\vec{B}$ is the begin position of the sphere, $\vec{D}$ the translation of the sphere and t is a value in the range [0, 1].

The following image illustrates this:



The sphere intersects the plane if the distance of the center of the sphere to the plane is equal or less than R: $d_{plane}(\vec{C}(t)) \leq R$.

When $\vec{D} \cdot \vec{N}$ is zero the plane is parallel to the motion of the sphere. If $d_{plane}(\vec{B}) > R$ there is no intersection, if $d_{plane}(\vec{B}) \leq R$ the sphere intersects over the whole range t = [0, 1].

If $\vec{D} \cdot \vec{N}$ is not zero we have to solve $d_{plane}(\vec{C}(t)) = R$ for t:

$$t_1 = \frac{-R - (\vec{B} \cdot \vec{N} + C_p)}{\vec{D} \cdot \vec{N}}$$

$$t_2 = \frac{R - (\vec{B} \cdot \vec{N} + C_p)}{\vec{D} \cdot \vec{N}}$$

If both values are outside the range [0, 1] there is no intersection. We sort the two values so that $t_1 < t_2$ and clamp the values to the range [0, 1]. The interval of intersection is now t = $[t_1, t_2]$.

The following function performs this test:

```
bool PlaneSweptSphereIntersect(const Plane &inPlane, const Vector3 &inBegin, const Vector3 &inDelta,
	float inRadius, float &outT1, float &outT2)
{
	// If the center of the sphere moves like: center = inBegin + t * inDelta for t e [0, 1]
	// then the sphere intersects the plane if: -R <= distance plane to center <= R
	float n_dot_d = inPlane.mNormal.Dot(inDelta);
	float dist_to_b = inPlane.GetSignedDistance(inBegin);
	if (n_dot_d == 0.0f)
	{
		// The sphere is moving nearly parallel to the plane, check if the distance
		// is smaller than the radius
		if (Abs(dist_to_b) > inRadius)
			return false;

		// Intersection on the entire range
		outT1 = 0.0f;
		outT2 = 1.0f;
	}
	else
	{
		// Determine interval of intersection
		outT1 = (inRadius - dist_to_b) / n_dot_d;
		outT2 = (-inRadius - dist_to_b) / n_dot_d;

		// Order the results
		if (outT1 > outT2)
			Swap(outT1, outT2);

		// Early out if no hit possible
		if (outT1 > 1.0f || outT2 < 0.0f)
			return false;
```

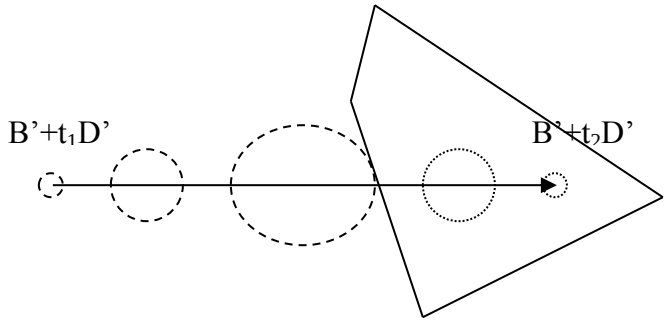
```
                // Clamp it to the range [0, 1], the range of the swept sphere
                if (outT1 < 0.0f) outT1 = 0.0f;
                if (outT2 > 1.0f) outT2 = 1.0f;
        }

        return true;
}
```

3.2. Swept Circle Versus Polygon

We have now determined the interval over which the swept sphere intersects with the plane of the polygon. To test if the sphere intersects with the polygon itself we are going to look at the problem in the space of the plane. The intersection between the sphere and the plane is a circle with a radius that varies along the path. The circle has a positive radius over our previous computed interval $[t_1, t_2]$. Outside this interval the radius is negative because the sphere does not intersect with the plane.

The following image illustrates this:



$\vec{B}$ and $\vec{D}$ are projected on the plane of the polygon in the same way as the points of the polygon (see section 2), they result in $\vec{B}'$ and $\vec{D}'$ (2D vectors) so that the center of the swept circle is described by $\vec{C}'(t) = \vec{B}' + t\vec{D}'$.

The radius of the circle is $R'(t)^2 = R^2 - d_{plane}(\vec{C}(t))^2 = r_1 t^2 + r_2 t + r_3$ with:

$$
\begin{aligned}
r_1 &= -(\vec{D}\cdot\vec{N})^2 \\
r_2 &= -2(\vec{D}\cdot\vec{N})(\vec{B}\cdot\vec{N} + C_p) \\
r_3 &= R^2 - (\vec{B}\cdot\vec{N} + C_p)^2
\end{aligned}
$$

There are three cases for which the sphere intersects with the polygon:

1. The circle intersects with the polygon at t_1.
2. The circle intersects with a vertex on the interval $[t_1, t_2]$.
3. The circle intersects with an edge on the interval $[t_1, t_2]$.

The results of all these tests will give a closest intersection fraction (the value of t for the first collision) and a collision point. This collision point can be converted back to 3D as described in section 2. It can be used to determine a collision response once the sphere hits the polygon.

We will now derive the three tests in the next sections.

3.3. Static Circle Versus 2D Polygon

In the first test we test if the circle at t_1 defined by center $\vec{C}'(t_1)$ and radius $R'(t_1)$ intersects with the polygon. This is the case when:

1. The circle center is inside the polygon. In this case $\vec{C}'(t_1)$ is the collision point and t_1 the intersection fraction.

2. The closest point from $\vec{C}'(t_1)$ to any of the edges is less than or equal to $R'(t_1)$. In this case this closest point is the collision point and t_1 the intersection fraction.

The following function combines both tests. It assumes that the vertices are (inVertices, inNumVertices) ordered counter clockwise. The center of the circle is inCenter and the radius of the circle squared is inRadiusSq. The collision point will be returned in outPoint.

```
bool PolygonCircleIntersect(const Vector2 *inVertices, int inNumVertices, const Vector2 &inCenter,
	float inRadiusSq, Vector2 &outPoint)
{
	// Check if the center is inside the polygon
	if (PolygonContains(inVertices, inNumVertices, inCenter))
	{
		outPoint = inCenter;
		return true;
	}

	// Loop through edges
	bool collision = false;
	for (const Vector2 *v1 = inVertices, *v2 = inVertices + inNumVertices - 1;
		v1 < inVertices + inNumVertices; v2 = v1, ++v1)
	{
		// Get fraction where the closest point to this edge occurs
		Vector2 v1_v2 = *v2 - *v1;
		Vector2 v1_center = inCenter - *v1;
		float fraction = v1_center.Dot(v1_v2);
		if (fraction < 0.0f)
		{
			// Closest point is v1
			float dist_sq = v1_center.GetLengthSquared();
			if (dist_sq <= inRadiusSq)
			{
				collision = true;
```

```
                                        outPoint = *v1;
                                        inRadiusSq = dist_sq;
                                }
                        }
                        else
                        {
                                float v1_v2_len_sq = v1_v2.GetLengthSquared();
                                if (fraction <= v1_v2_len_sq)
                                {
                                        // Closest point is on line segment
                                        Vector2 point = *v1 + v1_v2 * (fraction / v1_v2_len_sq);
                                        float dist_sq = (point - inCenter).GetLengthSquared();
                                        if (dist_sq <= inRadiusSq)
                                        {
                                                collision = true;
                                                outPoint = point;
                                                inRadiusSq = dist_sq;
                                        }
                                }
                        }
                }

                return collision;
}
```

The PolygonContains function checks if a point is inside the polygon:

```
bool PolygonContains(const Vector2 *inVertices, int inNumVertices, const Vector2 &inPoint)
{
        // Loop through edges
        for (const Vector2 *v1 = inVertices, *v2 = inVertices + inNumVertices - 1;
                v1 < inVertices + inNumVertices; v2 = v1, ++v1)
        {
                // If the point is outside this edge, the point is outside the polygon
                Vector2 v1_v2 = *v2 - *v1;
                Vector2 v1_point = inPoint - *v1;
                if (v1_v2.mX * v1_point.mY - v1_point.mX * v1_v2.mY > 0.0f)
                        return false;
        }

        return true;
}
```

3.4. Swept Circle Versus 2D Vertex

In the second test, we test if the swept circle intersects with any of the vertices of the polygon. For every vertex $\vec{v}$ we test if there is a t for which the distance between the center of the circle and the vertex equals the radius of the circle:

$$\left\|\vec{v} - \vec{C}'(t)\right\|^2 = R'(t)^2$$

Solving for t we get a quadratic equation $at^2 + bt + c = 0$ with:

$$
\begin{aligned}
a &= r_1 - \left\|\vec{D}'\right\|^2 \\
b &= r_2 + 2\left(\vec{D}' \cdot (\vec{v} - \vec{B}')\right) \\
c &= r_3 - \left\|\vec{v} - \vec{B}'\right\|^2
\end{aligned}
$$

Let t' be the smallest solution that lies in the interval [0, current closest fraction]. If there is a solution we store t' as the current closest intersection fraction and $\vec{v}$ as the current collision point.

3.5. Swept Circle Versus 2D Edge

In the third and final test we test if the swept circle intersects with an edge of the polygon. For every edge $\vec{v}_1$ to $\vec{v}_2$ we will test if there is a t for which the circle and the edge touch.

First we test if the circle intersects with the infinite line that the edge is part of. The distance between a point $\vec{P}$ and an infinite line through $\vec{v}_1$ and $\vec{v}_2$ is:

$$
d_{edge}(\vec{P})^2 = \left\|\vec{v}_1 - \vec{P}\right\|^2 - \frac{\left((\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_1 - \vec{P})\right)^2}{\left\|\vec{v}_2 - \vec{v}_1\right\|^2}
$$

The circle and the infinite line touch when:

$$
d_{edge}(\vec{C}'(t))^2 = R'(t)^2
$$

Solving for t we get a quadratic equation $at^2 + bt + c = 0$ with:

$$
\begin{aligned}
a &= \left\|\vec{v}_2 - \vec{v}_1\right\|^2 \left(r_1 - \left\|\vec{D}'\right\|^2 \right) + \left((\vec{v}_2 - \vec{v}_1) \cdot \vec{D}'\right)^2 \\
b &= \left\|\vec{v}_2 - \vec{v}_1\right\|^2 \left(r_2 + 2\left(\vec{D}' \cdot (\vec{v}_1 - \vec{B}')\right)\right) - 2\left((\vec{v}_2 - \vec{v}_1) \cdot \vec{D}'\right)\left((\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_1 - \vec{B}')\right) \\
c &= \left\|\vec{v}_2 - \vec{v}_1\right\|^2 \left(r_3 - \left\|\vec{v}_1 - \vec{B}'\right\|^2 \right) + \left((\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_1 - \vec{B}')\right)^2
\end{aligned}
$$

Let t' be the smallest solution that lies in the interval [0, current closest fraction]. If there is no solution we continue with the next edge.

A point on the edge is given by:

$$
\vec{p} = \vec{v}_1 + f(\vec{v}_2 - \vec{v}_1)
$$

The closest point from the circle center to the edge is when the direction from the center of the circle to the point is perpendicular to the edge:

$$\left(\vec{p} - \vec{C}'(t')\right) \cdot (\vec{v}_2 - \vec{v}_1) = 0$$

Solving for f:

$$f = \frac{\left((\vec{v}_2 - \vec{v}_1) \cdot \vec{D}'\right)t' - \left((\vec{v}_2 - \vec{v}_1) \cdot (\vec{v}_1 - \vec{B}')\right)}{\left\|\vec{v}_2 - \vec{v}_1\right\|^2}$$

If f is inside the range [0, 1] there is an intersection, we store t' as the current closest intersection fraction and $\vec{p}$ as the current closest intersection point.

3.6. Implementation

This section will combine the results of the previous sections into the full swept sphere versus static polygon test. First of all we will combine the tests from sections 3.4 and 3.5 since they share a lot of expressions.

The following function tests a swept circle with the edges and vertices of the polygon. The swept circle travels from inBegin to inBegin + inDelta. inA, inB and inC are a, b, and c of the quadratic equation of the circle radius. The collision point will be returned in outPoint and the fraction in outFraction.

```
bool SweptCircleEdgeVertexIntersect(const Vector2 *inVertices, int inNumVertices, const Vector2 &inBegin,
        const Vector2 &inDelta, float inA, float inB, float inC, Vector2 &outPoint, float &outFraction)
{
        // Loop through edges
        float upper_bound = 1.0f;
        bool collision = false;
        for (const Vector2 *v1 = inVertices, *v2 = inVertices + inNumVertices - 1;
                v1 < inVertices + inNumVertices; v2 = v1, ++v1)
        {
                float t;

                // Check if circle hits the vertex
                Vector2 bv1 = *v1 - inBegin;
                float a1 = inA - inDelta.GetLengthSquared();
                float b1 = inB + 2.0f * inDelta.Dot(bv1);
                float c1 = inC - bv1.GetLengthSquared();
                if (FindLowestRootInInterval(a1, b1, c1, upper_bound, t))
                {
                        // We have a collision
                        collision = true;
                        upper_bound = t;
                        outPoint = *v1;
                }

                // Check if circle hits the edge
                Vector2 v1v2 = *v2 - *v1;
```

```
                float v1v2_dot_delta = v1v2.Dot(inDelta);
                float v1v2_dot_bv1 = v1v2.Dot(bv1);
                float v1v2_len_sq = v1v2.GetLengthSquared();
                float a2 = v1v2_len_sq * a1 + v1v2_dot_delta * v1v2_dot_delta;
                float b2 = v1v2_len_sq * b1 - 2.0f * v1v2_dot_bv1 * v1v2_dot_delta;
                float c2 = v1v2_len_sq * c1 + v1v2_dot_bv1 * v1v2_dot_bv1;
                if (FindLowestRootInInterval(a2, b2, c2, upper_bound, t))
                {
                        // Check if the intersection point is on the edge
                        float f = t * v1v2_dot_delta - v1v2_dot_bv1;
                        if (f >= 0.0f && f <= v1v2_len_sq)
                        {
                                // We have a collision
                                collision = true;
                                upper_bound = t;
                                outPoint = *v1 + v1v2 * (f / v1v2_len_sq);
                        }
                }
        }

        // Check if we had a collision
        if (!collision)
                return false;
        outFraction = upper_bound;
        return true;
}
```

The tests are not performed on the range $[t_1, t_2]$ as described in section 3.2, but on the range [0, current closest fraction]. Numerical round off can generate solutions that are slightly lower than t_1. Solving over this larger range does not give us any false collisions since the radius of the circle becomes negative outside the range $[t_1, t_2]$ so no solutions are possible.

The following piece of code finds the lowest solution of a quadratic equation with coefficients inA, inB and inC in the interval [0, inUpperBound]. The solution is returned in outX when the function returns true.

```
bool FindLowestRootInInterval(float inA, float inB, float inC, float inUpperBound, float &outX)
{
        // Check if a solution exists
        float determinant = inB * inB - 4.0f * inA * inC;
        if (determinant < 0.0f)
                return false;

        // The standard way of doing this is by computing: x = (-b +/- Sqrt(b^2 - 4 a c)) / 2 a
        // is not numerically stable when a is close to zero.
        // Solve the equation according to "Numerical Recipies in C" paragraph 5.6
        float q = -0.5f * (inB + (inB < 0.0f? -1.0f : 1.0f) * Sqrt(determinant));

        // Both of these can return +INF, -INF or NAN that's why we test both solutions
        // to be in the specified range below
        float x1 = q / inA;
        float x2 = inC / q;

        // Order the results
        if (x2 < x1)
                Swap(x1, x2);
```

```
        // Check if x1 is a solution
        if (x1 >= 0.0f && x1 <= inUpperBound)
        {
                outX = x1;
                return true;
        }

        // Check if x2 is a solution
        if (x2 >= 0.0f && x2 <= inUpperBound)
        {
                outX = x2;
                return true;
        }

        return false;
}
```

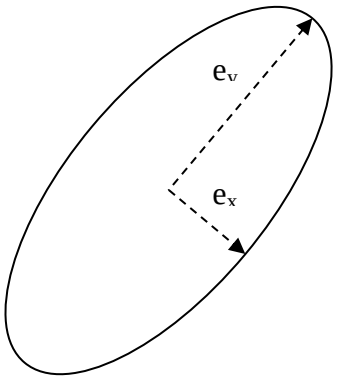
4. Swept Ellipsoid Versus Polygon

4.1. Theory

In this section we will expand the swept sphere versus static polygon test into a swept ellipsoid versus static polygon test. We will assume that the ellipsoid does not rotate.

To test a swept ellipsoid with a polygon we have to transform the polygon to the space where the ellipsoid is a unit sphere (a sphere with radius 1).

We define an ellipsoid by its three orthogonal axis $(\vec{e}_x, \vec{e}_y, \vec{e}_z)$:



The center of the ellipsoid will move according to $\vec{C}(t) = \vec{B} + t\vec{D}$, where $\vec{B}$ is the begin position of the ellipsoid, $\vec{D}$ the delta translation and t is a value in the range [0, 1].

We define a rotation / scaling matrix that transforms a unit sphere in the ellipsoid:

$$M_{unit->ellipsoid} = \begin{bmatrix} & & & 0 \\ \vec{e}_x & \vec{e}_y & \vec{e}_z & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When transforming the ellipsoid by the matrix $\left(M_{unit->ellipsoid}\right)^{-1}$ we get a unit sphere moving along $\vec{C}_{unit}(t) = \vec{B}_{unit} + t\vec{D}_{unit}$ with $\vec{B}_{unit} = \left(M_{unit->ellipsoid}\right)^{-1}\vec{B}$ and $\vec{D}_{unit} = \left(M_{unit->ellipsoid}\right)^{-1}\vec{D}$.

We need to transform the plane of the polygon with $\left(M_{unit->ellipsoid}\right)^{-1}$, this leads to the plane equation $\vec{N}' \cdot \vec{P} + C'_p = 0$ with:

$$\vec{N}' = \frac{\left(M_{unit->ellipsoid}\right)^T \vec{N}}{\left\|\left(M_{unit->ellipsoid}\right)^T \vec{N}\right\|}$$

$$C'_p = \frac{C_p}{\left\|\left(M_{unit->ellipsoid}\right)^T \vec{N}\right\|}$$

Where $\left(M_{unit->ellipsoid}\right)^T$ indicates the transpose of $M_{unit->ellipsoid}$.

With this transformed plane we can determine the interval of intersection between the unit sphere and the plane as before. If there is an intersection we need to transform our 2D polygon to the space of the unit sphere.

Projecting a point from plane space to world space can be written in matrix form:

$$\begin{pmatrix} \vec{P} \\ 1 \end{pmatrix} = M_{plane \to world} \begin{pmatrix} P'_x \\ P'_y \\ 0 \\ 1 \end{pmatrix}$$

$$M_{plane \to world} = \begin{bmatrix} \vec{U} & \vec{V} & \vec{N} & -C_p\vec{N} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let $M_{plane \to world}$ and $M_{transformed\ plane \to world}$ respectively be the matrix that takes points from the untransformed plane to world space and from the transformed plane to world space.

The transformation needed for the polygon is:

$$M_{polygon} = \left(M_{transformed\ plane \rightarrow world}\right)^{-1}\left(M_{unit -> ellipsoid}\right)^{-1}\left(M_{untransformed\ plane \rightarrow world}\right)$$

Once the polygon has been transformed we can project $\vec{B}_{unit}$ and $\vec{D}_{unit}$ on the transformed plane to form $\vec{B}'$ and $\vec{D}'$. Now we follow the same path as before to determine collision between a moving circle and a polygon.

If a collision point is found we have to transform it into world space by using the following matrix:

$$M_{collision \rightarrow world} = \left(M_{unit -> ellipsoid}\right)\left(M_{transformed\ plane \rightarrow world}\right)$$

4.2. Implementation

The swept ellipsoid moves from inBegin to inBegin + inDelta. The principal axis of the ellipsoid are inAxis1, inAxis2 and inAxis3 which should be orthogonal. When there is a collision the function returns true and the collision point will be in outPoint and the center of the sphere is inBegin + outFraction * inDelta when the sphere collides.

```
bool PolygonSweptEllipsoidIntersect(const Plane &inPlane, const Vector2 *inVertices, int inNumVertices,
        const Vector3 &inBegin, const Vector3 &inDelta, const Vector3 &inAxis1, const Vector3 &inAxis2,
        const Vector3 &inAxis3, Vector3 &outPoint, float &outFraction)
{
        // Compute matrix that takes a point from unit sphere space to world space
        // NOTE: When colliding with lots of polygons this can be cached
        Matrix unit_sphere_to_world;
        unit_sphere_to_world.Column(0) = inAxis1;
        unit_sphere_to_world.Column(1) = inAxis2;
        unit_sphere_to_world.Column(2) = inAxis3;

        // Compute matrix that takes a point from world space to unit sphere space
        // NOTE: When colliding with lots of polygons this can be cached
        Matrix world_to_unit_sphere = unit_sphere_to_world.GetInversed();

        // Compute begin and delta in unit sphere space
        // NOTE: When colliding with lots of polygons this can be cached
        Vector3 begin_uss = world_to_unit_sphere * inBegin;
        Vector3 delta_uss = world_to_unit_sphere * inDelta;

        // Transform the plane into unit sphere local space
        Plane transformed_plane;
        transformed_plane = inPlane.GetTransformedByInverse(unit_sphere_to_world);

        // Determine the range over which the unit sphere intersects the transformed plane
        float t1, t2;
        if (!PlaneSweptSphereIntersect(transformed_plane, begin_uss, delta_uss, 1.0f, t1, t2))
                return false;

        // Get matrix that transforms a point from plane space to world space
        Matrix plane_to_world = inPlane.GetPlaneToWorldMatrix();

        // Get matrix that transforms a point from the transformed plane to unit sphere space
        Matrix transformed_plane_to_unit_sphere = transformed_plane.GetPlaneToWorldMatrix();
```

```
	// Get matrix that takes a 2d polygon vertex from the original space to the space of the
	// transformed plane so that the unit sphere is still a unit sphere
	Matrix plane_to_transformed_plane = transformed_plane_to_unit_sphere.GetInversed()
		* world_to_unit_sphere * plane_to_world;

	// The radius of the circle is defined as: radius^2 = 1 - (distance plane to center)^2
	// this can be written as: radius^2 = a * t^2 + b * t + c
	float n_dot_d = transformed_plane.mNormal.Dot(delta_uss);
	float dist_to_b = transformed_plane.GetSignedDistance(begin_uss);
	float a = -n_dot_d * n_dot_d;
	float b = -2.0f * n_dot_d * dist_to_b;
	float c = 1.0f - dist_to_b * dist_to_b;

	// Get the basis vectors for the transformed plane
	const Vector3 &u = transformed_plane_to_unit_sphere.Column(0);
	const Vector3 &v = transformed_plane_to_unit_sphere.Column(1);

	// To avoid translating the polygon we subtract the translation from the begin point
	// and then later add it to the collision result again
	Vector2 trans(plane_to_transformed_plane.E(0, 3), plane_to_transformed_plane.E(1, 3));

	// Get the equation for the intersection circle between the plane and the
	// unit sphere: center = begin + t * delta
	Vector2 begin = Plane::sConvertWorldToPlane(u, v, begin_uss) - trans;
	Vector2 delta = Plane::sConvertWorldToPlane(u, v, delta_uss);

	// Transform the polygon
	Vector2 *transformed_vertices = (Vector2 *)alloca(inNumVertices * sizeof(Vector2));
	for (int i = 0; i < inNumVertices; ++i)
		transformed_vertices[i] = Transform2x2(plane_to_transformed_plane, inVertices[i]);

	// Test if sphere intersects at t1
	Vector2 p;
	if (PolygonCircleIntersect(transformed_vertices, inNumVertices,
		begin + delta * t1, a * t1 * t1 + b * t1 + c, p))
	{
		outFraction = t1;
		outPoint = unit_sphere_to_world
			*  (transformed_plane_to_unit_sphere * Vector3(p + trans));
		return true;
	}

	// Test if sphere intersects with one of the edges or vertices
	if (SweptCircleEdgeVertexIntersect(transformed_vertices, inNumVertices, begin, delta,
		a, b, c, p, outFraction))
	{
		outPoint = unit_sphere_to_world
			* (transformed_plane_to_unit_sphere * Vector3(p + trans));
		return true;
	}

	return false;
}
```

5. Using Polygons Stored as 3D Points

We stored our polygons as a list of 2D points and a plane equation, but the algorithm is the same if polygons are stored as a list of 3D points. To make the algorithm work we need to make the following changes to the equations in the previous sections:

- Compute the plane equation at run time or store it.
- Set $\vec{B}' = \vec{B}$ and $\vec{D}' = \vec{D}$ (so they become 3 vectors).
- Set $r_1 = r_2 = 0$ and $r_3 = R^2$.
- Transform the polygon by the 4x4 matrix: $M_{polygon} = \left(M_{unit->ellipsoid}\right)^{-1}$.
- Transform the collision result by the 4x4 matrix: $M_{collision \to world} = M_{unit->ellipsoid}$.

6. References

1. Nevell, *Graphics Gems III*, pp. 231-232.

2. W.H. Press, *Numerical Recipes in C, Second Edition* (http://www.library.cornell.edu/nr/bookcpdf/c5-6.pdf)

3. T. Schroeder, "Collision Detection Using Ray Casting", *Game Developer Magazine*, pp. 50-57, August 2001 (ftp://ftp.gdmag.com/pub/src/aug01.zip)

4. T. Akenine-Möller, E. Haines, *Real-Time Rendering* (http://www.realtimerendering.com/int/)