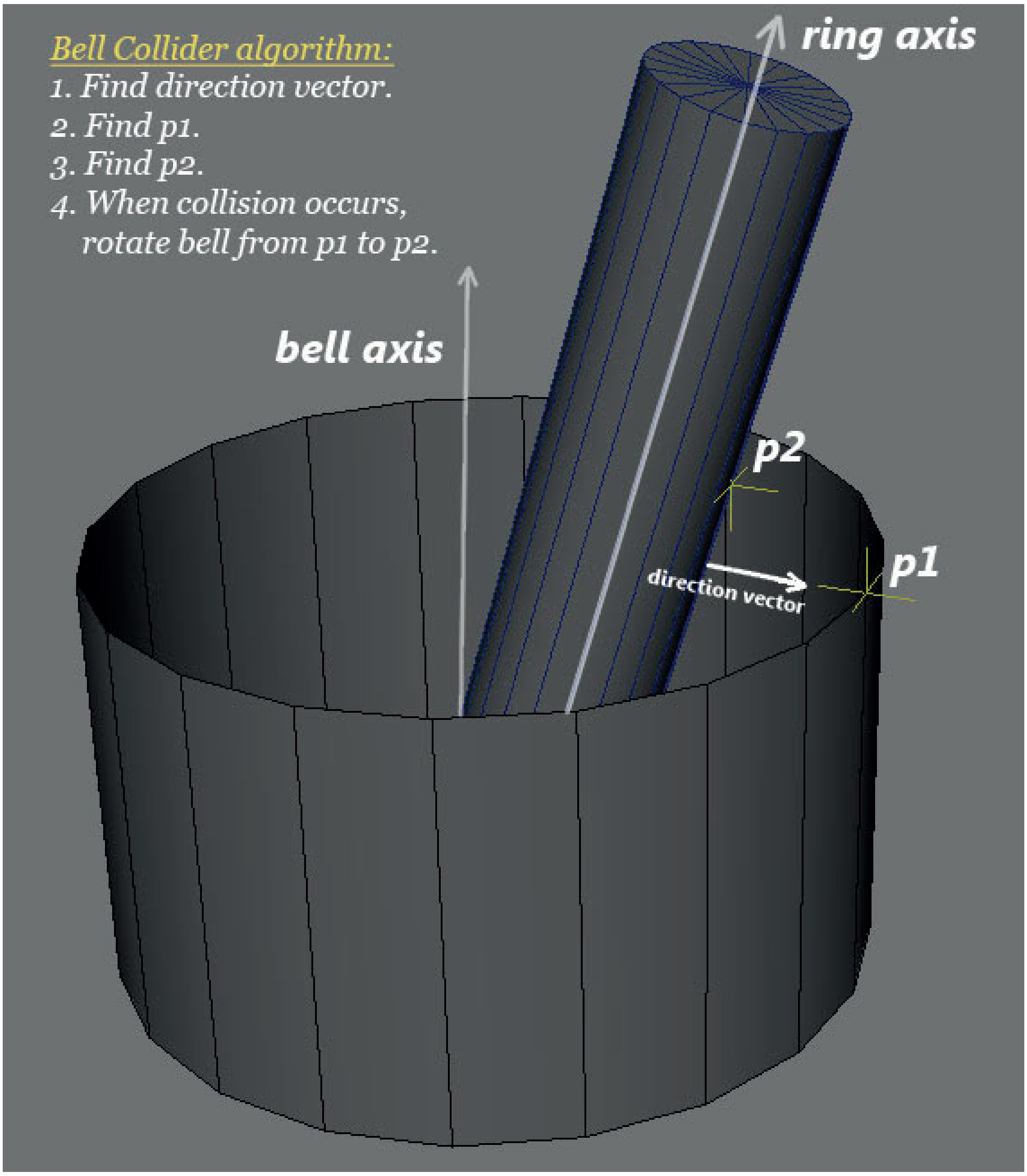
Bell Collider tutorial

(based on Dave Otte's idea)

Author: Alexander Zagoruyko
Email: azagoruyko@gmail.com

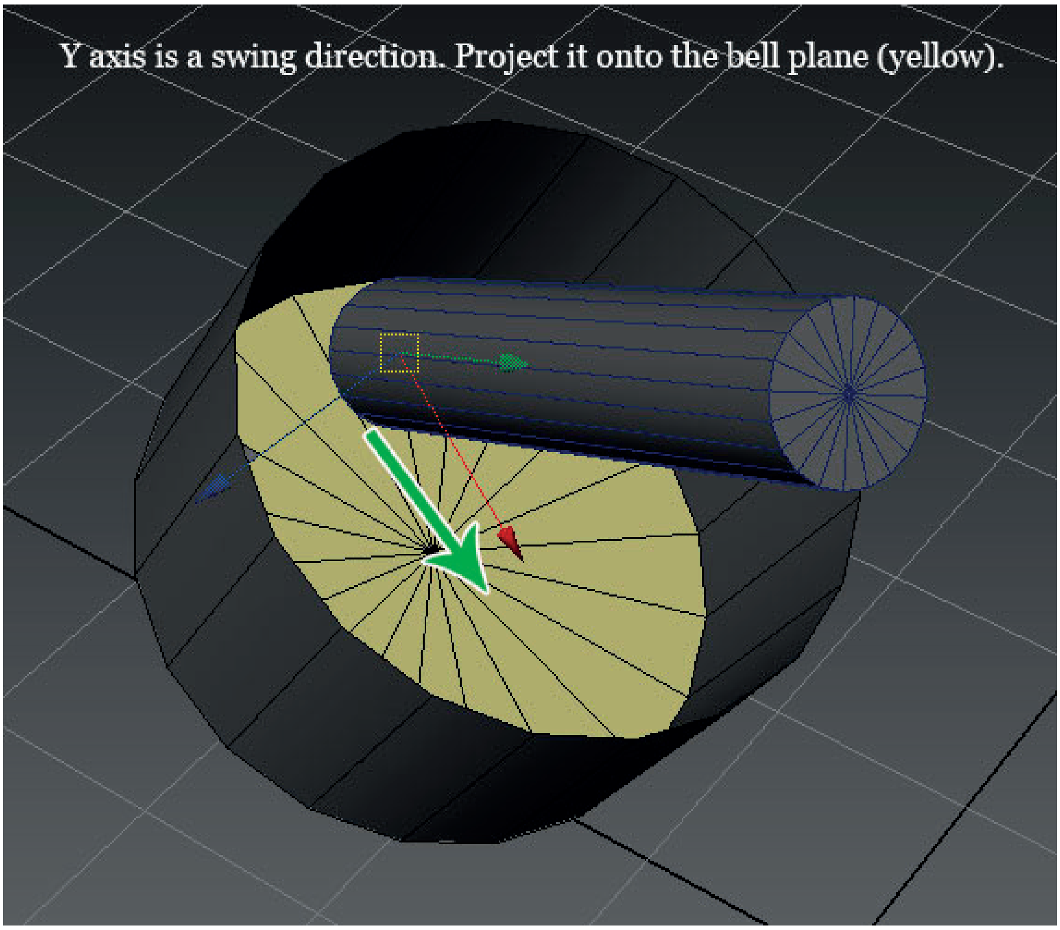
Look at the basic approach below. Here is an algorithm we need to do.



Basically, everything we need to do is find the direction vector, the points p1 and p2. After that we rotate the bell by a quaternion to match p1 with p2. Let's begin.

Step 1. Direction vector

Initially we have the bell and ring matrices. We can use one of the ring axises as a swing direction.
Assume Y axis directs towards the ring.
Actually the direction vector is a projection of the swing vector onto the bell plane.



Fat green arrow is the direction vector.

Let’s some code.

bellMatrix is the bell world matrix.
ringMatrix is the ring world matrix.

```
MVector swing(ringMatrix[1][0], ringMatrix[1][1], ringMatrix[1][2]); // Y axis
```

Bell plane is defined as a normal vector to the plane.

```
MVector bellPlane(bellMatrix[1][0], bellMatrix[1][1], bellMatrix[1][2]); // Y axis by default as well
```

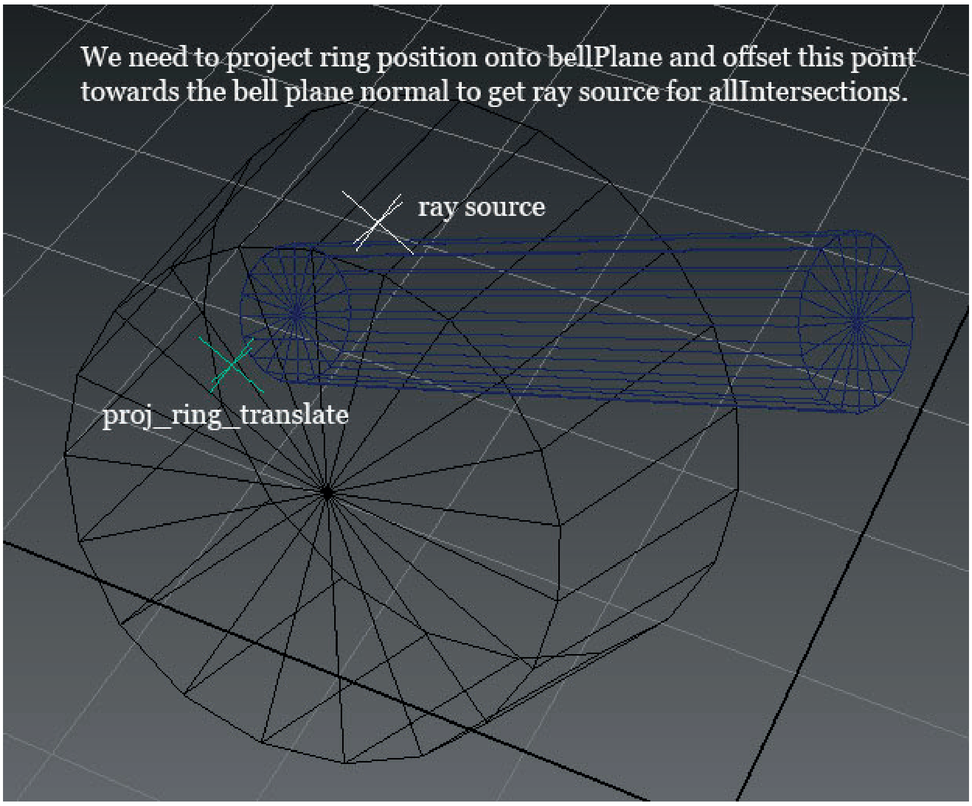
Project swing vector onto bellPlane.

```
MVector proj_swing = swing - (swing*bellPlane.normal()) * bellPlane.normal();
proj_swing.normalize();
```

proj_swing is the direction vector.

Step 2. Find p1.

I use MFnMesh::allIntersections() function to find p1 point, because our bell shape can be customized.
We need to find ray source. Ray direction equals proj_swing. We also need worldMesh of the bell as an input.



Approach:
1. Project ring position onto the bell plane.
Ring and bell positions can be retrieved from the input world matrices.

```
MVector bell_translate(bellMatrix[3][0], bellMatrix[3][1], bellMatrix[3][2]);
MVector ring_translate(ringMatrix[3][0], ringMatrix[3][1], ringMatrix[3][2]);

MVector v = ring_translate - bell_translate;
MVector n = bellPlane.normal();
double dist = v * n; // scalar product
MPoint proj_ring_translate = ring_translate - dist*n;
```

2. Offset proj_ring_translate towards the bell plane normal.

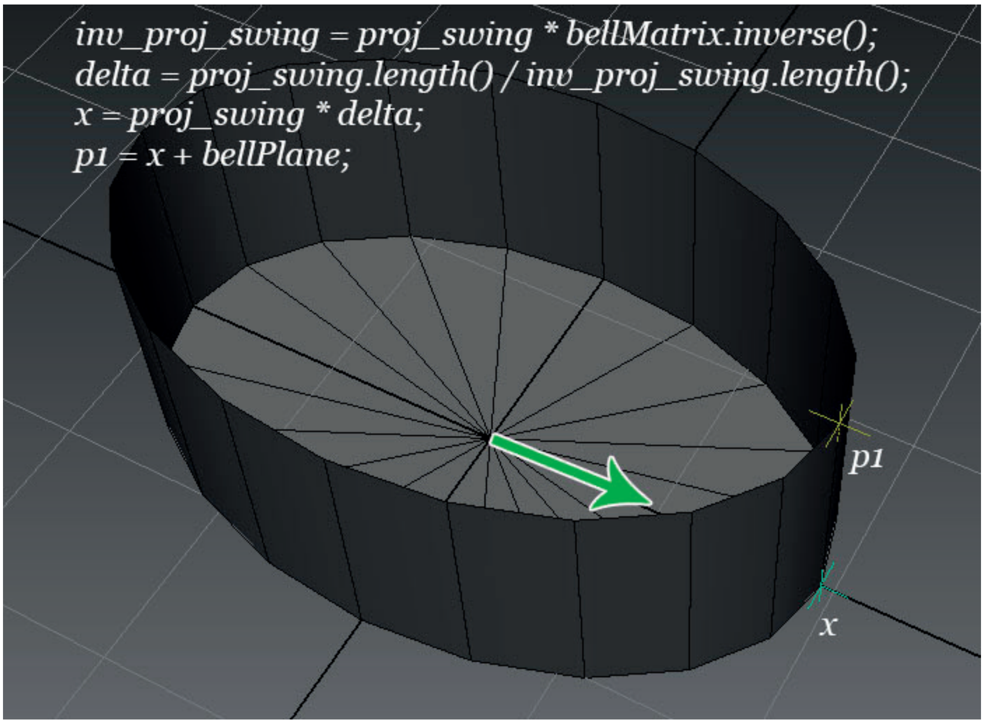
```
MFloatPoint raySource = proj_ring_translate + bellPlane*0.999; // rough
```

Consider that bellPlane is a scaled vector (Y axis). Make sure your bell and ring are the normalized cylinders with radius=1 and height=1. Pivot must be at the bottom center. Actually bellPlane.length() = cylinder height. Magic number 0.999 is used here to be sure that ray source point is not higher than the cylinder height.

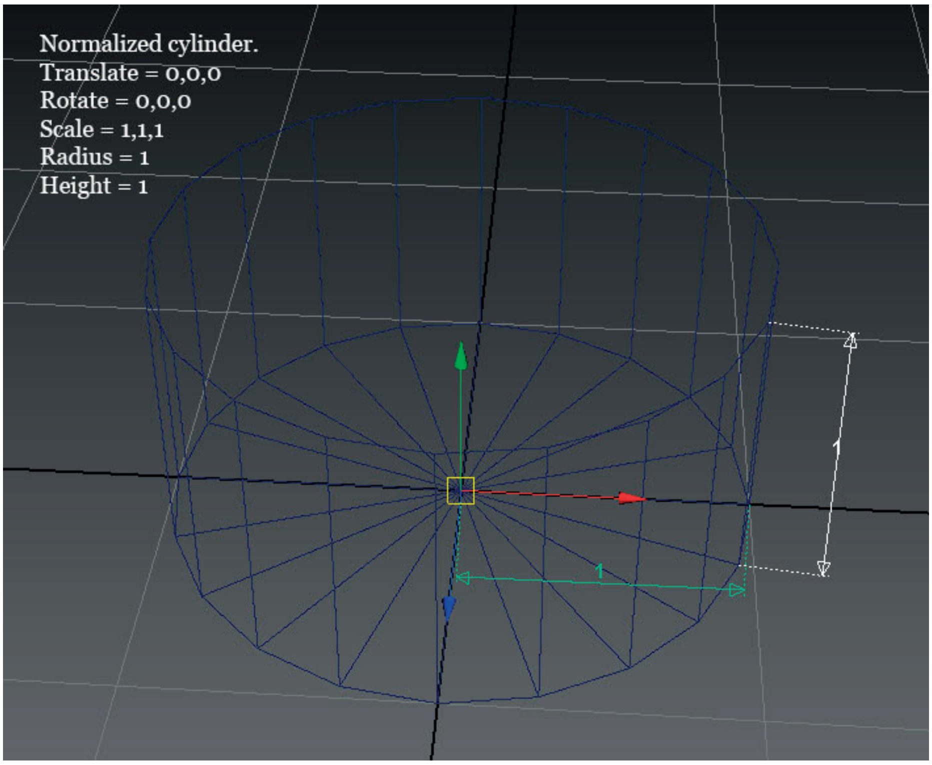
```
MFnMesh meshFn(bellMesh);
auto accel = meshFn.autoUniformGridParams();
MFloatPointArray hitPoints;
meshFn.allIntersections(raySource, MFloatVector(proj_swing), NULL, NULL, false, MSpace::kWorld,
ringPlane.length(), false, &accel, false, hitPoints, NULL, NULL, NULL, NULL, NULL);
MPoint p1 = hitPoints[0];  // be sure you've found something
```

There is another way to get p1 if you are sure that the bell shape is a cylinder (not a custom shape).
«Parent» proj_swing to bellMatrix with the transformation preserve.
Calculate delta scale and then compute the points x and p1 (see picture below).

```
MVector inv_proj_swing = proj_swing * bellMatrix.inverse();
double delta = proj_swing.length() /  inv_proj_swing.length();
MPoint x = proj_swing * delta;
MPoint p1 = x1 + bellPlane;
```



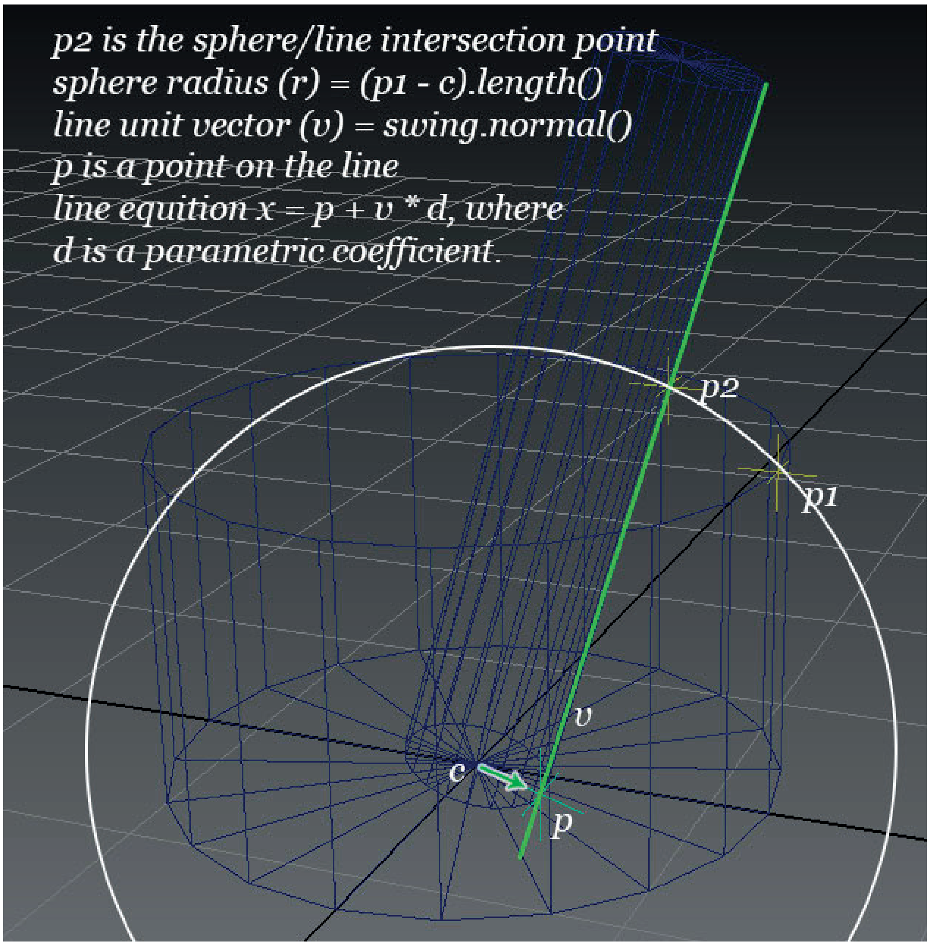
Anyway your bell and ring must be normalized cylinders as I said.



Bell and ring cylinders must be normalized.

Step 3. Find p2.

This step is a bit harder than previous ones. To find p2 we need to solve system of equitions. Look at the picture below.



At first we should find p point.
Project proj_swing onto ringPlane (the same as we do with the projection onto the bell plane).

```
MVector ringPlane(ringMatrix[1][0], ringMatrix[1][1], ringMatrix[1][2]); // ring Y axis
MVector ring_proj_swing = proj_swing - (proj_swing*ringPlane.normal()) * ringPlane.normal();
ring_proj_swing.normalize();
```

Consider ring scale.

```
MVector ring_proj_swing_inv = ring_proj_swing * ringMatrix.inverse();
double delta = ring_proj_swing.length() / ring_proj_swing_inv.length();
MPoint p = ring_translate + ring_proj_swing.normal() * delta;
```

Imagine a sphere with a center at bell_translate (c at the picture) with a radius = (p1-bell_translate).length().
p2 must be on the sphere and on the line at the same moment.
Read here about sphere/line intersection.
https://en.wikipedia.org/wiki/Line%E2%80%93sphere_intersection

Sphere equation: (x - c) * (x - c) = r*r
Line equation: x = p + v*d, where

```
double r = (p1-bell_translate).length(); // sphere radius
MPoint c = bell_translate; // sphere center
MVector v = ringPlane.normal(); // line unit vector
```

p is the point we found above.
d is the parametric line coefficient.

Combine equitions: (p + v*d - c)*(p + v*d - c) = r*r
Expand equition: d*d*v*v + 2*d*(v * (p - c)) + (p - c)*(p - c) = r*r
This is a standard form of a quadratic equition with d variable.
a*d*d + b*d + c = 0, where
a = v*v
b = 2 * (v * (p - c))
c = (p - c) * (p - c) - r*r

Expand vectors.
```
double a = v.x*v.x + v.y*v.y + v.z*v.z; // =1 as v is a unit vector
double b = 2 * (v.x * (p.x - c.x) + v.y * (p.y - c.y) + v.z * (p.z - c.z));
double c = (p.x - c.x)*(p.x - c.x) + (p.y - c.y)*(p.y - c.y) + (p.z - c.z)*(p.z - c.z) - r*r;
```

Discriminant here.
```
double delta = b*b - 4 * a * c;
```

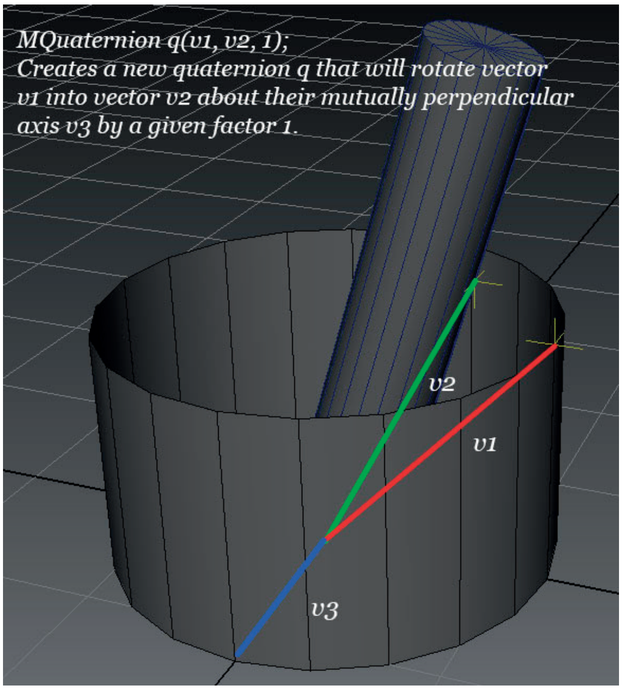
So the roots are the following.
```
double root1 = (-b + sqrt(delta)) / 2*a;
double root2 = (-b - sqrt(delta)) / 2*a;
```
We need root1, because root1 > root2 and therefore the point lies further along v than the original p point. root2 point lies on the opposite side.

Substitute d parametric coefficient with found root1.
```
MVector p2 = p + v*root1;
```

At this step we have found p1, p2 and proj_swing direction vector. I know, it’s a bit crazy, but there are no real problems here.

Step 4. Rotation.

The final step is to rotate our bell from p1 to p2. This is really easy to do.
Maya MQuaternion takes two vectors as constructor parameters and return a quaternion that rotates the first vector into another.



It looks like this.

```
MTransformationMatrix bellMatrixFn(bellMatrix);
if (proj_swing * (p1 - p2) < 0 || bellPlane*ringPlane < 0) // when collision occurs
{
	MQuaternion quat(p1 - bell_translate, p2 - bell_translate, 1);
	bellMatrixFn.rotateBy(quat, MSpace::kTransform);
}

auto euler = bellMatrixFn.rotation().asEulerRotation();
MVector rotation(MAngle(euler.x).asDegrees(), MAngle(euler.y).asDegrees(), MAngle(euler.z).asDegrees());
```

rotation is the bell rotation.
That's done.

Thanks for reading. Any feedback and comments are welcome.

email: azagoruyko@gmail.com
skype: azagoruyko