2.10

The Shortest Arc Quaternion

Stan Melax

This article shows a short routine called `RotationArc()`. Given two vectors v_0 and v_1, this function returns a quaternion q where $q^*v_0==v_1$. The implementation is fairly optimal and avoids a common numerical instability pitfall.

Motivation

You might be wondering where you would ever want to use such a function. Consider a guided missile in your video game. This is an object that uses orientation (3DOF), even though it is only the forward direction (2DOF) of this radial-symmetric object that is important for its AI. As is the case for all rigid bodies, a quaternion q is needed to reorient this object from its current direction v_0 to the direction we want it to be going v_1. Although we can choose from an infinite number of axes of rotation, it is best to choose the obvious axis of rotation that minimizes the (arc) angle of reorientation—that is, the obvious axis that is perpendicular to both vectors. This routine is also useful for implementing a "virtual trackball" for spinning objects with the mouse (as in a VRML viewer). The SpinLogo demo program included on the CD that accompanies this book uses `RotationArc()` to implement this feature.

Numerical Instability

This algorithm could easily be done by taking the normalized cross product to get an axis of rotation and then taking the `acos()` of the dot product to get the angle between the vectors. This axis and angle would be fed into the constructor of a quaternion. However, that is not a good solution, because as the vectors v_0 and v_1 get close together, the cross product (being proportional to the sin of the angle between the two vectors) becomes small and potentially unstable when we try to normalize it (see Figure 2.10.1). Deriving the angle can also cause grief. It is possible that taking the dot product of two unit length vectors that are parallel can result in a small overflow (greater than 1). This can be problematic when deriving the angle. Try executing `acos(1.00000001)`. In these cases, using the standard quaternion constructor that

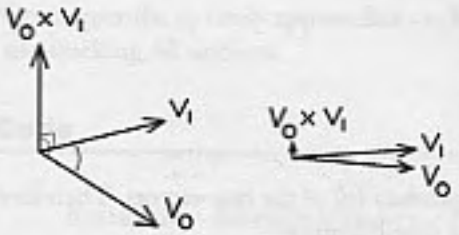

FIGURE 2.10.1. Cross product shrinks as vectors converge.

accepts an axis and an angle is not appropriate. The solution is to generate the quaternion in a more direct manner.

This problem first hit us during the development of the guided missiles for the video game MDK2 [Bioware00]. This is not an obscure problem that showed up only in our development. It has been noticed by many others and could happen to you. *Real-Time Rendering* [Moller99] briefly mentions the subject. This article provides a more thorough explanation and provides code to add to the *Game Programming Gems* math library (or to your own quaternion library). If you want to avoid introducing a nasty bug, you should use the code provided in this article for generating a quaternion from two direction vectors.

Derivation of Stable Formula

For this discussion, let $c = [c_x, c_y, c_z] = \text{cross}(v_0, v_1)$, and the quaternion q we are trying to derive has the elements q_x, q_y, q_z, q_w. The angle (unknown) between the two vectors (v_0 and v_1) is t. Let d be the dot product: $d = \text{dot}(v_0, v_1)$;

The q_x, q_y, q_z components of a quaternion q have a length that is sin of half the angle ($t/2$). As mentioned, the cross product's length is the sin of the angle t. Therefore:

$$[q_x, q_y, q_z] = [c_x, c_y, c_z]\frac{\sin(t/2)}{\sin(t)}$$

So now we have to determine a stable formula for the *sin(t/2)/sin(t)* term. Recall the half-angle formula:

$$\sin(t/2) = \sqrt{\frac{1-\cos(t)}{2}}$$

and the circle identity:

$$\sin^2 + \cos^2 = 1$$

Then:

$$\frac{\sin(t/2)}{\sin(t)} = \frac{\sqrt{(1-\cos(t))/2}}{\sqrt{1-\cos^2(t)}}$$

We know *cos(t)* to be the dot product (*d*) of the two vectors. Therefore, we replace it in the formula and continue the simplification:

$$\frac{\sqrt{(1-d)\ /\ 2}}{\sqrt{1-d^2}} = \sqrt{\frac{1-d}{2(1+d)(1-d)}} = \frac{1}{\sqrt{2(1+d)}}$$

Substituting back in:

$$[q_x, q_y, q_z] = \frac{[c_x, c_y, c_z]}{\sqrt{2(1+d)}}$$

Deriving the q_w component (angle) of the quaternion is fairly straightforward using the half-angle formula for *cos(t/2)*:

$$q_w = \cos(t\ /\ 2) = \sqrt{\frac{1+\cos(t)}{2}} = \sqrt{\frac{1+d}{2}}$$

In order to optimize our C++ function to use only one call to `sqrt()`, we multiply the inner square root term of our q_w formula by 2/2. As a result, the term within the square root is now the same as for the other quaternion elements. In other words, q_w is derived using the equivalent formula:

$$q_w = \frac{\sqrt{2(1+d)}}{2}$$

These new formulas for our quaternion elements remain stable as v_0 approaches v_1 and as the dot product d approaches 1.

Remaining Instability Condition

Note that this function still becomes numerically unstable as v_0 approaches $-v_1$. This is not surprising, because when v_0 equals $-v_1$, there is not a unique solution; any axis of rotation on the plane perpendicular to v_0 will do. Remember that v_0 and v_1 are directions, not orientations. A check could be added to the function in order to detect this case and avoid the possibility of dividing by zero. However, we did not do this, because it is so unlikely that one would be calling this function in that case. The objective of using a function such as this one is to orient an object toward a target.

Consequently, v_0 rarely approaches $-v_1$ but instead converges to v_1 after repeated missile-tracking AI updates.

Source Code

```
quaternion RotationArc(vector3 v0,vector3 v1) {
    quaternion q;
    v0.normalize();  // Skip if known to be unit length.
    v1.normalize();  // Do only if needed.
    vector3 c = CrossProduct(v0,v1);
    float   d = DotProduct(v0,v1);
    float   s = (float)sqrt((1+d)*2);
    q.x = c.x / s;
    q.y = c.y / s;
    q.z = c.z / s;
    q.w = s /2.0f;
    return q;
}
```

Virtual TrackBall

As a bonus, this article includes code for implementing a virtual trackball function. Although such functionality might not be necessary in the user interface of your finished game, it is very handy during development to be able to grab any of your game objects and spin them around with the mouse.

An easy way of implementing object spinning is to rotate the object about the *Y*-axis when the mouse is moved parallel to the *X* direction and about the *X*-axis based on vertical (*Y*) mouse movement. This method doesn't allow for rotation about the *Z*-axis (normal to window), and it simply doesn't feel "intuitive."

There are a variety of other user interface methods for spinning objects. This article explains one simple approach. The old and new mouse positions (2D [*X*, *Y*]) are converted into rays (3D) that point from the viewpoint into the window. Next, we determine where these rays would intersect a sphere around the object the user is manipulating. If a ray does not intersect the sphere, the closest point on the silhouette of the sphere is used. The sphere is rotated so that the point of intersection from the old mouse ray coincides with the point of intersection from the new mouse ray. This is achieved by passing these two points (using the center of the sphere as the origin of the coordinate system) as input to `RotationArc()`. This returns the quaternion that is used to adjust the object's orientation. The source code for the virtual trackball function and extracting the direction vectors from mouse input are available in the SpinLogo source code on the CD that accompanies this book.

References

[Bioware00] Bioware, Shiny, and Interplay Productions, *MDK2*, 2000.

[Moller99] Moller, Tomas, and Haines, Eric, *Real-Time Rendering*, A. K. Peters Ltd., 1999.