2.6

Smooth C^2 Quaternion-based Flythrough Paths

Alex Vlachos, ATI Research;
and John Isidore
alex@Vlachos.com and jisidoro@cs.bu.edu

In this gem, we describe a method for smoothly interpolating a camera's position and orientation to produce a flythrough with C^2 continuity. We draw on several known methods and provide a C++ class that implements the methods described here.

Introduction

Smoothly interpolating the positions of a flythrough path can easily be achieved by applying a natural cubic spline to the sample points. The orientations, on the other hand, require a little more attention. We describe a method for converting a quaternion in S^3 space (points on the unit hypersphere) into R^4 space (points in 4D space) [Johnstone99]. Once the quaternion is in R^4 space, any 4D spline can be applied to the transformed data. The resulting interpolated points can then be transformed back into S^3 space and used as a quaternion. In addition, a technique called *selective negation* is described to preprocess the quaternions in a way that produces the shortest rotation path between sample point orientations.

Camera cuts (moving a camera to a new location) are achieved by introducing phantom points around the camera cut similar to the way an open spline is padded. These additional points are needed to pad the spline to produce smooth results near the cut point. The code provided describes cut points as part of a single fly path and simplifies the overall code. Internally to the C++ class, the individual cut segments are treated as separate splines without the overhead of creating a spline for each segment.

Position Interpolation

Let's now discuss position interpolation.

Sample Points

There are two common ways to specify sample points. The first is to have each segment between control points represent a constant time (for example, each control point rep-

resents one second of time). The second is to use the control points only to define the shape of the camera path, and to have the camera move at a constant speed along this path. The code provided with this gem assumes a constant time between control points, although this code could easily be modified for the constant speed technique.

Natural Cubic Spline

A natural cubic spline is chosen due to the high degree of continuity it provides, namely C^2. However, it's important to note that any spline may be used in place of the natural cubic spline. Code for implementing this spline is widely available, including *Numerical Recipes In C* [Press97]. The sample code provided is modeled after this.

A natural cubic spline is an interpolating curve that is a mathematical representation of the original drafting spline. One important characteristic of this spline is its lack of local control. This means that if any single control point is moved, the entire spline is affected. This isn't necessarily a disadvantage; in fact, this functionality may be desirable. As you begin to use this spline, you'll see the advantages it has in smoothing out the camera movement when you sample the spline at a higher frequency.

It is important to differentiate between open and closed splines. In the case of a closed spline, the spline is specified such that the last point is the same as the first point. This is done to treat the camera path as a closed loop. To work around any possible discontinuities in the spline at the loop point, simply replicate the last four points of the spline to the beginning of the array, and the first four sample points to the end of the array. In practice, we've found that using four points was sufficient to eliminate any visual artifacts.

This replication eliminates the need for modulus arithmetic and also simplifies our preprocessing of the camera path. This is even more important when dealing with orientations using the selective negation method as described later (Figure 2.6.1).

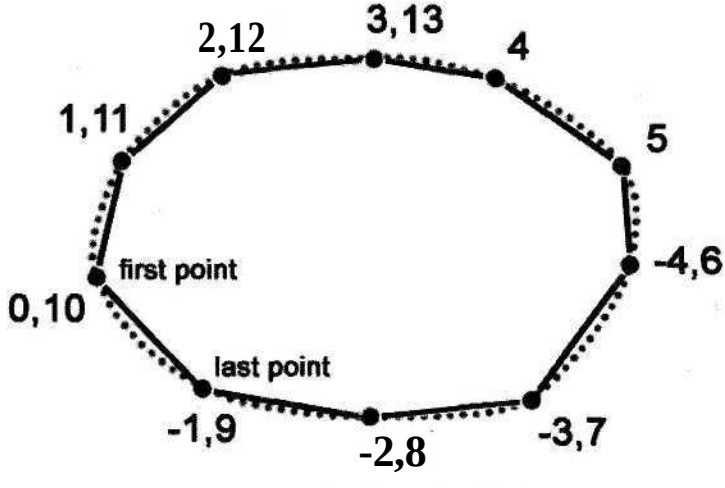

FIGURE 2.6.1 *Replicating points for a closed spline.*

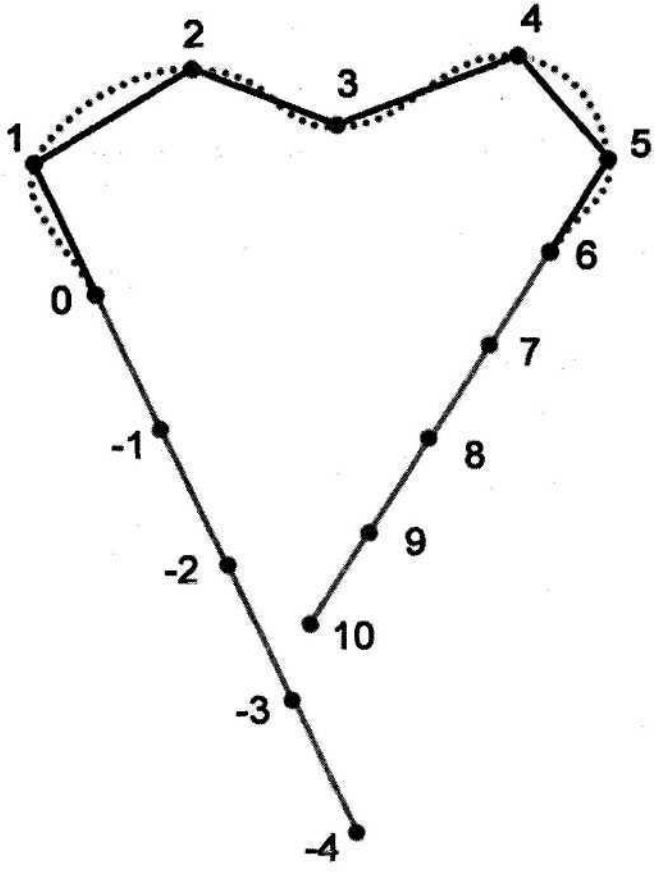

FIGURE 2.6.2 *Creating phantom points for an open spline.*

In contrast, an open spline has a different beginning and end point. In order to sample the spline, you need to pad the spline with several "phantom" points at both the beginning and end of the open spline (Figure 2.6.2). A constant velocity is assumed for the phantom points before and after the open spline path. At the beginning of the spline in Figure 2.6.2, the vector V^Pj-Po is subtracted from P_0 to get the resulting point P_j. Similarly, V_0 is subtracted from P_j to create P_2, and so on. The trailing phantom points are calculated in a similar way.

Orientation Interpolation

Sample Points

Unit quaternions are used as the orientation data at the sample points. Quaternions can be very useful for numerous applications. The beauty of quaternions is that, for rotations, they take the form of a normalized 4-element vector (later referred to as a 3-element vector and a scalar component). This is exactly enough information to repre-

sent an axis of rotation and an angle of rotation around that axis [GPG1]. Quaternions give us everything we need to represent a rotation and nothing more.

For orientation, however, there is an ambiguity in using quaternions. Orientation can be thought of as a rotation from a base orientation. When using quaternions, there are two possible rotations that will bring you to the same orientation. Suppose there is a counterclockwise rotation *9* about an axis *w* that gives you the desired orientation. A rotation by 360°-0 about the axis —*w* also results in the same orientation. When converted into a quaternion representation, the second quaternion is simply the negation of the first one.

Direction of Rotation and Selective Negation

When performing quaternion interpolation, there is one small nuance that needs to be considered. When representing an orientation, either a quaternion or its negation will suffice. However, when interpolating orientations (for example, performing a rotation), the positive and negative quaternions result in vastly different rotations and consequently different camera paths. If the desired result is to perform the smallest possible rotation between each pair of two orientations, you can preprocess the quaternions to achieve this.

Taking the dot product of two quaternions gives the cosine of half the angle of rotation between them. If this quantity is negative, the angle of rotation between the two quaternions is greater than 180 degrees. In this case, negating one of the orientation quaternions makes the angle between the two quaternions less than 180 degrees. In terms of interpolation, this makes the orientation spline path always perform the shortest rotation between the orientation key frames. We call this process *selective negation.*

The technique of selectively negating orientation quaternions can be incorporated as a preprocessing step for a camera flythrough path. For the preprocessing step, traverse the flythrough path from start to end, and for each quaternion *q,on* the path, negate it if the dot product between it and its predecessor is negative (in other words, if $(q_t \cdot q_{,.j}) < 0$). Using selective negation as a preprocessing step makes spline interpolation much more efficient by not requiring the selective negation math for every sample.

To preprocess a closed spline path, it is necessary to replicate the first four points of the spline path and append them to the end of the path prior to the selective negation. Note that the replicated points may have different signs than the original points. When dealing with an open spline, you need to create phantom quaternions (corresponding to the phantom control points) to pad the spline. The concept is similar in that you want to linearly interpolate the difference between the two quaternions closest to the beginning or end of the path. However, linearly interpolating quaternions doesn't suffice. Instead, we use the spherical linear interpolation (slerp) algorithm. Given quaternions q_0 and q_t, we need to generate four phantom quaternions—$q.j$, $q_{\_2}$, and so on— to pad the beginning of an open spline. We use the

slerp function (spherical linear interpolation) to slerp from q_i to q_0 with a slerp value of 2.0. This effectively gives us a linear change in rotation at our phantom points.

Once we have preprocessed our entire list of orientation quaternions for interpolation, it is straightforward to perform smooth spline-based quaternion interpolation techniques.

Spline Interpolation for Quaternions

As seen for positional interpolation, splines can be used to give us much smoother interpolation than linear interpolation can. However, spline interpolation for quaternions is not so straightforward, and there are several techniques that can be used. One technique simply interpolates the raw quaternion values, and then renormalizes the resulting quaternion. However, this technique does not result in a smooth path and produces bizarre changes in angular velocity. Another idea is to use techniques based on the logarithms of quaternions. SQUAD (spherical quadrangle interpolation) [Shoemake91] is an example of this. A performance limitation is incurred when using these techniques because they require transcendental functions (sin, cos, log, pow, and so on). Other techniques involve blending between great 2-spheres laying on the unit quaternion hypersphere [Kim95], or involve some sort of iterative numeric technique [Neilson92]. While many of these techniques provide decent results, most of them do not provide C^2 continuity or are computationally prohibitive to use, especially when many flythrough paths are used (for game characters or projectiles, as an example).

However, there is a technique for quaternion spline interpolation that gives very good results and obeys derivative continuity requirements. This uses an invertible rational mapping Qohnstone99] M between the unit quaternion 4-sphere (S^3) and another four-dimensional space (R^4). In the following equations, *a, b,* and *c* are the components of the vector portion of the quaternion, and *s* is the scalar portion.

The transformation $M^{-/}$ from S^3 — >R^4 is:

$$x = a/sqrt(2(l-s))$$
$$y = b/sqrt(2(l-s))$$
$$z = c/sqrt(2(l-s))$$
$$w=(l-s) /sqrt(2(l-s))$$

The transformation M from R^4 — > S^3 is:

$$s = (x^2 + y^2 + z^2 - w^2) / (x^2 + y^2 + z^2 + w^2)$$
$$a = 2xu> I (y? + y^2 + z? + iv^2)$$
$$b = 2yw / (x^2 + y^2 + z^2 + ui^2)$$
$$c = 2zw I (x^2 + y^2 + z^2 + iv^2)$$

To use this for quaternion spline interpolation is straightforward. First, selective negation should be applied to the control quaternions to assure the shortest possible rotation between control points. After this, apply $M^{'}$ to all the control quaternions to get their resulting value in R^4. This can be done as a preprocessing step and can be

done in the flythrough-path building or loading stage of a program. This way, die square root does not have to be calculated when die flythrough path is being evaluated.

Next, the resulting 4-vectors can be interpolated using the spline of your choice. Because this is a continuous rational mapping, the continuity of the interpolated S^3 quaternion path has the same continuity as die spline used for interpolation in R^4 space.

In our application, we use natural cubic splines [Hearn94] [Press97] for the interpolation in R^4 space.

This gives us C^2 continuous orientation interpolation as well. The qualitative effect of this is that die camera path does not have any sharp changes in angular acceleration.

After the desired point on the spline in R^4 is found, it can be converted back into a quaternion using M.

However, there is one mathematical nuance in using this technique that needs to be addressed.

Singularity in the Rational Mapping

If your flythrough path contains orientation quaternions close or equal to (1,0,0,0), it will cause numerical instability when interpolated through in R^4 space, as $M\sim^{i}(1,0,0,0) = C^{00},^{00},^{00},^{00}$). There are a few ways to handle diis singularity. One possible option is to ignore it, and surprisingly, this is feasible in many cases. For example, if the z-axis of the world is pointing up, and you know the camera will never point straight up with the camera's up-vector pointing up the y-axis, the orientation quaternion (1,0,0,0) will never occur in the flythrough path, and die problem is solved.

If this is not the case, another option is to find a quaternion ^that is not within 30 degrees of any of the orientation quaternions, and use gy-to rotate all of the quaternions into "safe" orientations that are not near the singularity Qohnstone 99]. The basic idea behind this is to perform the spline interpolation on a rotated version of your flythrough path, and then rotate die interpolated orientations back into their original coordinate frame. All that has to be done is to multiply all your orientation quaternions by $q^{\wedge}$ after the selective negation step. Following, you transform the quaternions from S^3 space into R^4 space, apply the natural cubic spline, and transform the resulting R^4 values back into S^3 space. After this, we add the additional step of rotating each resulting quaternion by qj^{1} before using it.

An easy way to find gy-is to randomly generate unit length quaternions, until you find one that is not within 30 degrees of any of the selectively negated orientation quaternions.

Camera Cuts

A *camera cut* is defined as moving the camera from one point in your scene to another. You can't just introduce a cut in the middle of a spline, and you can't simply step over

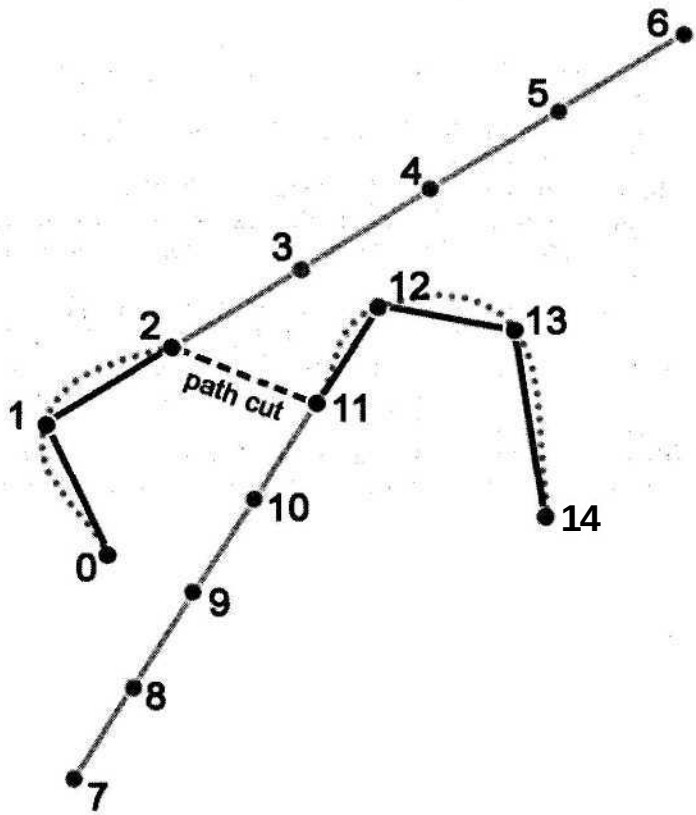

FIGURE **2.6.3** *Creating phantom points for a path cut segment.*

a segment of the spline. Instead, you segment your spline into two separate splines at a cut point, and process these splines as separate open splines. This is done simultaneously for both the position- and orientation-based splines. The code we supply deals with camera cuts in such a way that you don't need to explicitly create a separate path (see Figure 2.6.3).

Code

The code accompanying this gem is a C++ class that implements most of the techniques explained in this article. It has member functions for creating and editing the control points manually, reading and writing path files, dealing with cut points, sampling the spline at a given time, and setting up vertex and index buffers for drawing the spline. The class assumes there is a constant time between control points as opposed to a constant velocity. In addition, the code does not solve the singularity problem, since we never saw the singularity in our project. Please see the source files for more information.

References

[Press97] Press, William H., et al, *Numerical Recipes in C,* Cambridge University Press, 1997.

[Hearn94] Hearn, Donald, Baker, M. Pauline, *Computer Graphics Second Edition,* Prentice Hall, Inc. 1994.

[Johnstone99] Johnstone, J. K., Williams, J. P., "A Rational Quaternion Spline of Arbitrary Continuity," Tech Report: *www.cis.uab.edu/info/faculty/jj/cos.html,* 1999.

[GPG1] Edited by Mark DeLoura, *Game Programming Gems,* Charles River Media, 2000.

[Shoemake91] Shoemake, K., *Quaternion Calculus for Animation,* Math for SIGGRAPH (ACM SIGGRAPH '91 Course Notes #2), 1991.

[Neilson92] Neilson, G., and Heiland, R., "Animating Rotations Using Quaternions and Splines on a 4D Sphere," English Edition, *Programming and Computer Software,* Plenum Pub., New York. 1992.

[Kim95] Kim, M.S. and Nam, K.W., *Interpolating Solid Orientations with Circular Blending Quaternion Curves,* Computer-Aided Design, Vol. 27, No. 5, pp. 385-398,1995.