2.4

Nonuniform Splines

Thomas Lowe, Krome Studios

tomlowe@kromestudios.com

Splines are curved paths, usually defined by a series of points in 3D space called *nodes*. Sometimes more information is required per node to further describe the shape of the curve. Usually splines are evaluated by calling a simple function like `GetPosition(float time)`, where time varies between zero and one.

This article describes three types of nonuniform cubic splines. Nonuniform splines have the useful property that their velocity is not affected by the distance between nodes, making them particularly useful in game development. The three types of splines are:

- **Rounded nonuniform spline:** Approximately constant velocity, useful for trains running along tracks.
- **Smooth nonuniform spline:** C^2 or continuous acceleration, useful for particle effect paths.
- **Timed nonuniform spline:** A variation of the previous spline with specifiable time intervals; for example, for cut-scene cameras.

Types of Splines

The following is a list of commonly used spline types, of which the basic types are compared in Figure 2.4.1 (see [Demidov03] for more information). The notation C^n means that the n^{th} derivative of the spline is continuous.

- **Bezier curves:** These pass through only the start and end nodes and are continuous in all derivatives. Unlike the other spline types, their complexity increases with each extra node added.
- **Catmull-Rom splines:** Pass through each node. C^1 continuous.
- **Kochanek-Bartels splines:** Extension of Catmull-Rom requiring extra parameters per node.
- **Natural cubic splines:** Pass through each node. C^2 continuous.

- **Cubic b-spline:** Do not pass through each node. C^2 continuous.
- **NURBS:** Extension of b-splines, requiring extra parameters per node. Capable of defining exact circles, hyperbolas, and ellipses.

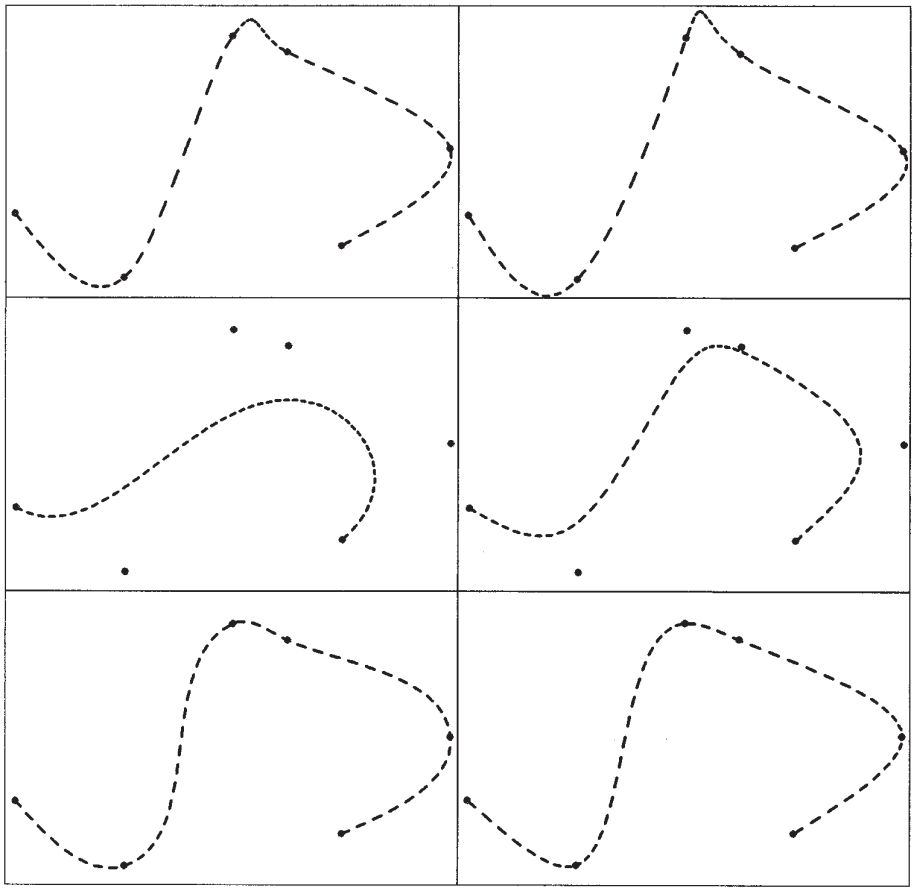

FIGURE 2.4.1 *Types of splines, from left to right, top to bottom. Catmull-Rom, natural, bezier, b-spline, rounded nonuniform, smooth nonuniform.*

Basic Cubic Spline Theory

The nonuniform splines described in this article are very similar to Catmull-Rom and natural cubic splines. They all pass through each node (a useful property giving a high level of control). They are all piecewise; each curve segment between two nodes is a separate function of time. Lastly, they are all cubic, meaning the function is of the form $p(t) = at^3 + bt^2 + ct + d$ where $p(t)$ is position at time t.

Three-dimensional splines are cubic in each dimension, so we can write (in vector notation):

$$\mathbf{p}(t) = \mathbf{a}t^3 + \mathbf{b}t^2 + \mathbf{c}t + \mathbf{d} \text{ or}$$
$$\mathbf{p}(t) = [t^3 \quad t^2 \quad t \quad 1]\mathbf{A} \text{ for some matrix A.} \tag{2.4.1}$$

Looking at a single segment, the curve can be fully described by a start and end position, and a start and end velocity (or by tangents) (see Figure 2.4.2).

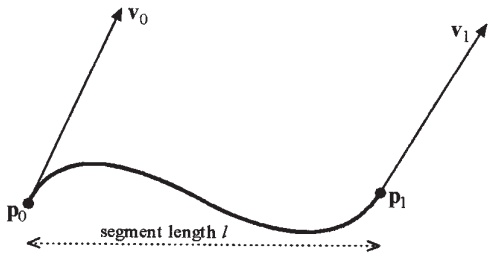

FIGURE 2.4.2 *A cubic curve described by four vectors.*

If we scale t to go from 0 to 1 in this segment, then we can describe these node velocity vectors as the change in position after one time unit if continuing in a straight line. Further, we can fully generate the matrix **A**:

$$\mathbf{A} = \mathbf{HG} \text{ so}$$
$$\mathbf{p}(t) = \mathbf{tHG} \text{ where}$$

$$\mathbf{t} = [t^3 \quad t^2 \quad t \quad 1] \quad \mathbf{H} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \text{ and } \mathbf{G} = \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{v}_0 \\ \mathbf{v}_1 \end{bmatrix} \tag{2.4.2}$$

Here, **t** is the time vector, **H** is a Hermite interpolation matrix, and **G** is the geometry matrix. The Hermite interpolation matrix is the unique matrix that satisfies our description. See [Hermite99] for its derivation. So now we can obtain the position along the segment curve at any time $0 \le t \le 1$. Equation 2.4.2 has been implemented in the function `GetPositionOnCubic()` on the companion CD-ROM.

Catmull-Rom splines and natural cubic splines are both built up from these cubic segments, just with different methods of choosing the node velocity vectors. Since they are uniform, their nodes are equally spaced along the timeline. Returning a position along the spline at some time (between 0 and 1) is simply a case of finding the appropriate segment and its t value, and then applying Equation 2.4.2.

Rounded Nonuniform Spline

Figure 2.4.3 shows a racing path created by a Catmull-Rom spline on the left, and a rounded nonuniform spline on the right. Notice how the unevenly spaced nodes greatly distort the velocity and shape of the left spline, whereas the right spline has a very regular velocity and shape.

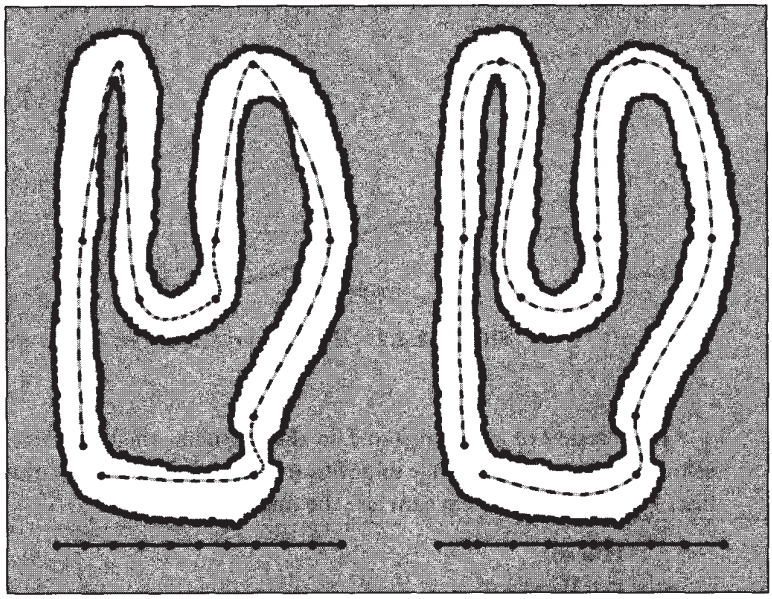

FIGURE 2.4.3 *Comparison of Catmull-Rom spline (left) with rounded nonuniform spline (right). Below each track, we see the nodes placed on timelines.*

The rounded nonuniform spline (RNS) is therefore very useful for describing spatial curves (such as train tracks or rope geometry) or for paths that should be traversed at a constant rate (such as enemy patrol paths or objects on rails). In addition, these splines can be used where rounded corners are required, the traversal speed being controlled externally. The racetrack in Figure 2.4.3 is a good example of this.

Implementation

So, how does one implement an RNS? First, divide the spline's timeline up in proportion to the segment lengths (this is the nonuniform part). One can use the linear distance between nodes here as an approximation to the segment length. This division means that an object traversing the spline evenly (i.e., with a constant time increment) will travel at the same average velocity between each pair of nodes. A simple data structure and lookup function for this is shown in the following pseudocode:

```
struct Node {
    Vector position, velocity;
    float distance; // distance to next node in array
} node[10];

Vector GetPosition(float time)
{
    float distance = time * maxDistance;
    float currentDistance = 0.f;
    int i = 0;
    while (currentDistance + node[i].distance <
           distance && i < 10)
    {
        currentDistance += node[i].distance;
        i++;
    }
    float t = distance - currentDistance;
    // i is segment number, t is time along segment

    // Note: Remainder of function listed below
}
```

Now each segment is a cubic function with a different time range. If we can rescale the cubic to the range $0 \le t \le 1$, then we can continue to use Equation 2.4.2 on each segment. This means scaling the node velocities $\mathbf{v}^N$ to get the segment time start and end velocities $\mathbf{v}^S$. First, the time conversion:

$$\Delta t^s = \Delta t^w * \Omega / l \tag{2.4.3}$$

where Ω = traversal speed along spline
= length of spline / time to traverse spline

In English, the segment t value (0 to 1) is scaled from world time by multiplying by the speed that the spline is being traversed, and dividing by the segment length. One can see that a 10-meter segment will take 10 times longer to traverse ($\Delta t^s = 1$) than a 1-meter segment.

From Equation 2.4.3, we get $\mathbf{v}^S = \mathbf{v}^W * l/\Omega$ (2.4.4)

The node velocities are stored as world velocities assuming a traversal speed of one.

So: $\mathbf{v}^S = \mathbf{v}^N * l$ (2.4.5)

In other words, the velocities passed into `GetPositionOnCubic()` must first be multiplied by the segment length so as to be in the correct timescale. We complete the lookup function by appending:

```
t /= node[i].distance; // scale t in range 0 - 1
Vector startVel = node[i].velocity * node[i].distance;
Vector endVel = node[i+1].velocity * node[i].distance;
return GetPositionOnCubic(node[i].position, startVel,
                          node[i+1].position, endVel, t);
```

The second part to implementing an RNS is obtaining the node velocities (here is the rounded part of RNS). We choose them to be unit length (so the velocity at the nodes is equal to the average velocity between nodes), and we split the angle between the previous and the next node in half (see Figure 2.4.4).

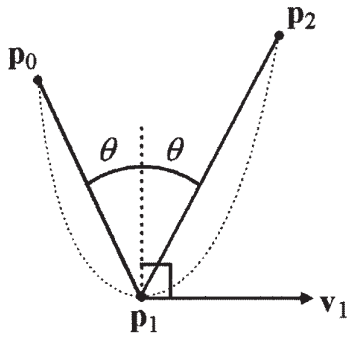

FIGURE 2.4.4 *Choice of node's velocity vector.*

The complete code for creating and accessing the RNS is on the companion CD-ROM.

Smooth Nonuniform Splines

Figure 2.4.5 compares a smooth nonuniform spline (SNS) with an RNS and a natural cubic spline (smooth uniform spline). Notice how, like the RNS, the velocity is not affected by node spacing, but like the natural spline, the motion appears smooth.

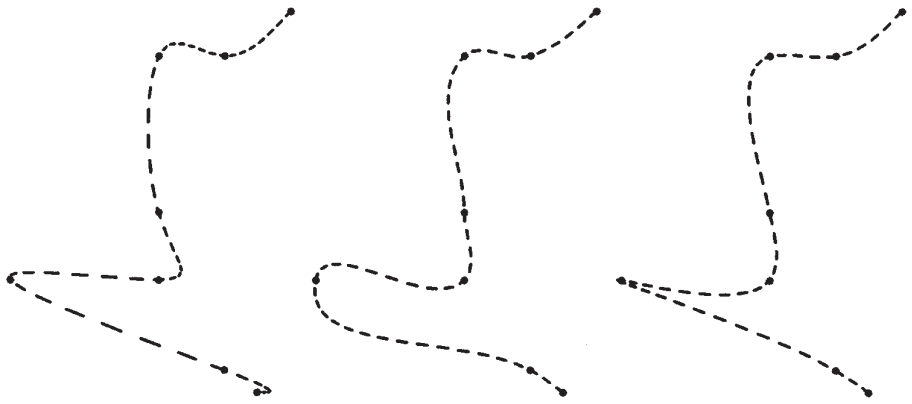

FIGURE 2.4.5 *Natural spline, RNS and SNS.*

The SNS is C^2 continuous, meaning it has a continuous (nonjerky) acceleration as one traverses it, unlike the C^1 RNS. It is a smooth temporal curve and so ideally suited to describing the motion of objects that accelerate and decelerate smoothly and have no turning circle. For example, the motion of a camera, the path of a flying saucer, approximating handwriting, or creating particle effect trails. It's also worth noting that this spline is the "point curve" that 3ds max uses.

Implementation

The SNS works the same way as the RNS; it is just the node velocities that are different, making it C^2 continuous. The complexity here is all in generating these node velocities based on the node positions.

To make this spline C^2 continuous, a velocity vector for each node must be chosen such that the end acceleration of the previous cubic is equal to the start acceleration of the next cubic (see Figure 2.4.6).

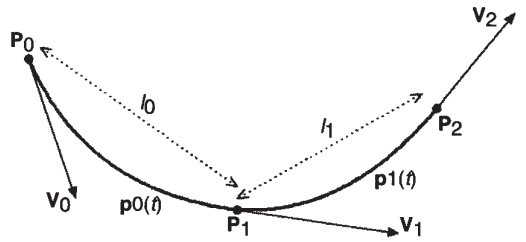

FIGURE 2.4.6 *The SNS is C^2 continuous when* $\boldsymbol{a0}^W(1) = \boldsymbol{a1}^W(0)$. (2.4.6)

So, what is the formula for acceleration? Examining a single segment and taking derivatives of Equation 2.4.2:

$$\mathbf{p}(t) = (2\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{v}_0^S + \mathbf{v}_1^S)t^3 + (-3\mathbf{p}_0 + 3\mathbf{p}_1 - 2\mathbf{v}_0^S - \mathbf{v}_1^S)t^2 + \mathbf{v}_0^S t + \mathbf{p}_0$$

so (2.4.7)

$$\mathbf{v}^S(t) = 3(2\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{v}_0^S + \mathbf{v}_1^S)t^2 + 2(-3\mathbf{p}_0 + 3\mathbf{p}_1 - 2\mathbf{v}_0^S - \mathbf{v}_1^S)t + \mathbf{v}_0^S \quad (2.4.8)$$

$$\mathbf{a}^S(t) = 6(2\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{v}_0^S + \mathbf{v}_1^S)t + 2(-3\mathbf{p}_0 + 3\mathbf{p}_1 - 2\mathbf{v}_0^S - \mathbf{v}_1^S) \quad (2.4.9)$$

However, we are looking for world acceleration. We can derive from Equation 2.4.3 that:

$$\mathbf{a}^W = \mathbf{a}^S * \Omega / l^2 \quad (2.4.10)$$

Converting Equation 2.4.9 and applying to Equation 2.4.6, we get:

$$\frac{6(2\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{v}_0^S + \mathbf{v}_1^S) + 2(-3\mathbf{p}_0 + 3\mathbf{p}_1 - 2\mathbf{v}_0^S - \mathbf{v}_1^S)}{l_0^2} = \frac{2(-3\mathbf{p}_1 + 3\mathbf{p}_2 - 2\mathbf{v}_1^S - \mathbf{v}_2^S)}{l_1^2} \tag{2.4.11}$$

Then, applying Equation 2.4.5 and rearranging in terms of $\mathbf{v}_1$, we get:

$$\mathbf{v}_1^N = \frac{l_1(3(\mathbf{p}_1 - \mathbf{p}_0) / l_0 - \mathbf{v}_0^N) + l_0(3(\mathbf{p}_2 - \mathbf{p}_1) / l_1 - \mathbf{v}_2^N)}{2(l_0 + l_1)} \tag{2.4.12}$$

We now have each node velocity specified in terms of the velocity of its two neighboring nodes. Given that node velocities can be assigned to the first and last nodes (defined later), there are two methods of solving this system.

First, Equation 2.4.12 forms a tridiagonal system of equations that can be solved in O(n) time. See [RecipesC93] for solving tridiagonal systems.

The second method, and the one implemented in the sample code, is to iteratively apply Equation 2.4.12 to each node. One can think of it as a smoothing filter applied to the spline that reduces the acceleration discontinuities with each pass. In practice, only three or four passes are required to reach an almost exact solution. See the function `Smooth()` on the companion CD-ROM.

Now that we have this filter, an SNS is simply an RNS that has `Smooth()` called on it several times after all nodes are added (see Figure 2.4.7).

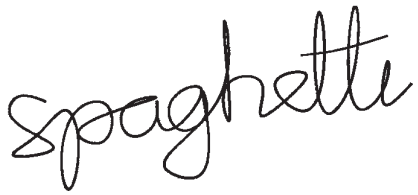

FIGURE 2.4.7 *Example of an SNS.*

Timed Nonuniform Spline

The timed nonuniform spline (TNS) is an extension of the SNS with an additional parameter per node: the time interval to the next node. It is also C^2 continuous.

Figure 2.4.8 shows three versions of the TNS generated from the same set of points, but with different specified time intervals between each node (shown on the timeline bar at the bottom).

The ability to specify a time schedule for the spline path makes the TNS ideally suited for generating cut-scene camera paths or flythroughs. You can place down points at any location in space and time and the camera will pass through them exactly and smoothly.

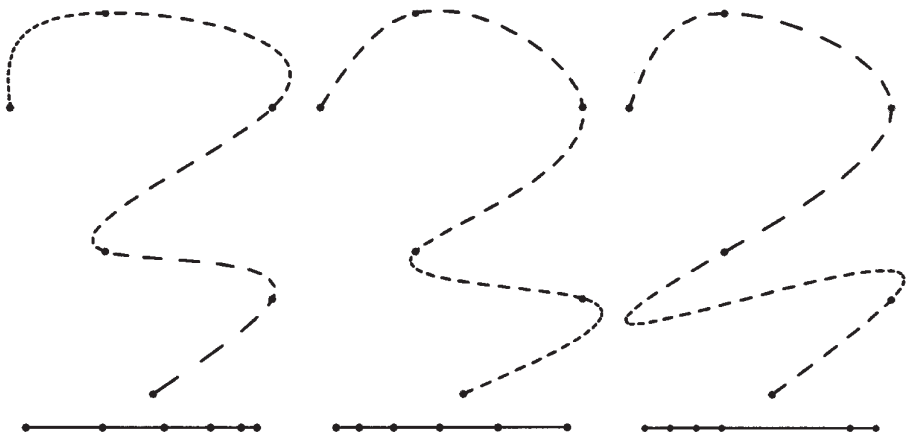

FIGURE 2.4.8 *Timed nonuniform splines with different time intervals specified on the nodes.*

The implementation of this spline is a simple extension of the SNS. One just uses the specified time intervals for each segment in place of the segment length.

In the far right spline in Figure 2.4.8, we see that the large time span allocated to one segment leads to it veering off track quite a bit. This is often undesirable and calls for a modification. A good method of countering the effect is to scale down the node velocity vectors in the more extreme situations after each smooth. The following scaling does this job well and is shown in Figure 2.4.9.

$$\mathbf{v}_1^N * = 4r_0r_1/(r_0 + r_1)^2 \tag{2.1.13}$$

where r = actual segment length / time interval.

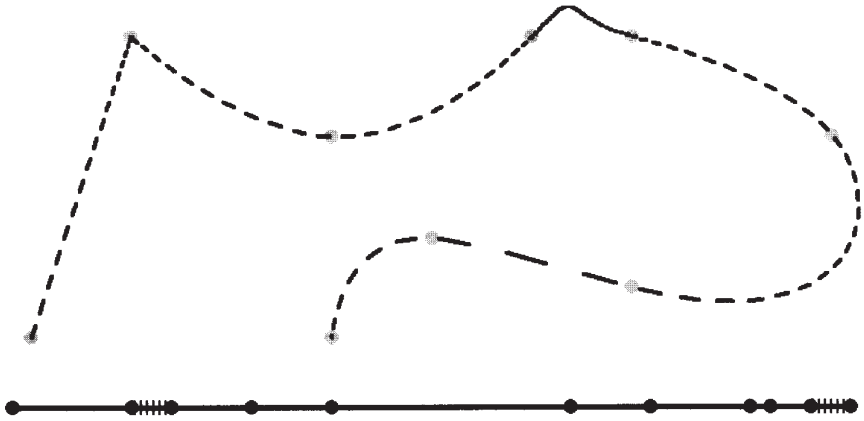

FIGURE 2.4.9 *Constrained TNS. Note how the doubled-up nodes allow the camera to stop and start smoothly.*

Calculating Start and End Node Velocities

This is the last remaining undefined aspect to the splines described previously. Fortunately, there is a fairly standard solution, which is to choose the velocity such that the acceleration is zero at the end (implying that there is no further force on the curve). Visually this means the curve becomes straight at the two end positions.

Taking Equation 2.4.9 at time $t = 0$, we have:

$$\mathbf{a}^S(0) = 2(-3\mathbf{p}_0 + 3\mathbf{p}_1 - 2\mathbf{v}_0^S - \mathbf{v}_1^S) = 0 \text{ So (with Equation 2.4.5):} \quad (2.4.14a)$$

$$\mathbf{v}_0^N = (3(\mathbf{p}_1 - \mathbf{p}_0) / l_0 - \mathbf{v}_1^N) / 2 \quad (2.4.14b)$$

This is our solution for the spline's start node. The end node velocity is much the same.

$$\mathbf{v}_n^N = (3(\mathbf{p}_n - \mathbf{p}_{n-1}) / l_{n-1} - \mathbf{v}_{n-1}^N) / 2 \quad (2.4.14c)$$

Obtaining the Velocity and Acceleration on the Spline

Examining a single segment, Equations 2.4.8 and 2.4.9 can be written in matrix form and are thus equivalent to Equation 2.4.2 with a different matrix **H**.

$$\mathbf{H}_{velocity} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 6 & -6 & 3 & 3 \\ -6 & 6 & -4 & -2 \\ 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{H}_{acceleration} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 12 & -12 & 6 & 6 \\ -6 & 6 & -4 & -2 \end{bmatrix} \quad (2.4.15)$$

The standard lookup function can be used, but it returns a value in segment space. To get a world velocity, apply Equation 2.4.4; to get a world acceleration, apply Equation 2.4.10.

These additional quantities can be very useful. The velocity vector can provide an orientation for an object following the spline path. The acceleration is used in measures such as the tilt of the object, the current curvature, or the direction of thrust.

Optimizations

There are opportunities for speeding up the spline access function `GetPosition()`. The function does a linear search of the segments, which, despite being a fast loop, is still O(n). If the segment lengths are stored cumulatively, then one can do a binary search of the segments, which is $O(\log_2 n)$.

In situations where the spline is being traversed, rather than accessed randomly, the current segment and distance from start can be cached, leading to a constant lookup time. Additionally, in such situations the 4 x 4 matrix multiplication **HG** from Equation 2.4.2 can be calculated once and cached, reducing the access code to little more than a vector transform. On architectures with a vector unit, this can be done very quickly in parallel. If the spline is to be traversed with a constant Δt, then we can optimize the evaluation of each cubic even further. The function can be reduced to just three vector additions:

```
acc += jerk;
vel += acc;
pos += vel;
```

where jerk is the third derivative of the cubic (a constant), and `acc`, `vel`, and `pos` are given appropriate initial values.

Conclusion

This article described three types of nonuniform splines that give game developers control over the timing and velocity of a point traversing the spline. This makes them especially useful for objects traveling at constant speed, such as vehicles, or paths with ancillary timing information, like camera paths. Nonuniform splines are more appropriate for these applications than traditional splines, which instead aim for higher order continuity or other goals.

References

[Demidov03] Demidov, Evgeny, "An Interactive Introduction to Splines," available online at *www.people.nnov.ru/fractal/Splines/Intro.htm.*

[Hermite99] "Hermite Splines," available online at *www.siggraph.org/education/materials/HyperGraph/modeling/splines/hermite.htm.*

[RecipesC93] Press, William H., et al., *Numerical Recipes in C, Second Edition*, Cambridge University Press, 1993, available online at *www.library.cornell.edu/nr/bookcpdf.html.*