TechNote TN-06-004

Arc-Length Parameterization Part I

Jim Armstrong
May 2006

This is the eighth in a series of TechNotes on the subject of applied curve mathematics in Adobe Flash[TM]. Each TechNote provides the mathematical foundation for a set of Actionscript examples.

Curve Parameterization

Parametric curves discussed in previous TechNotes were defined in terms of a parameter, t, that varied from 0 to 1, with Cartesian coordinates $x(t)$ and $y(t)$. As t varies from 0 to 1 in sufficiently small increments, the resulting (x,y) plot traces the shape of the parametric curve.

Programmers often associate curve parameterization with shape control. In previous TechNotes, it was shown that a curve's shape was a function of geometric constraints and basis matrix. Intuitively, curve parameterization can be described as the specification of parameter type and parameter values at interpolation points.

Consider piecewise curves that interpolate at each knot. In between knots, a local parametric curve is defined with a parameter, u, that varies from 0 to 1. The entire curve is defined by another parameter, t, that also varies from 0 to 1. The t-parameter for any point on the piecewise curve is mapped to a local u for each segment. The curve's parameterization determines the t-values at each knot. A specific t is mapped to a specific segment and specific u for the parametric curve corresponding to that segment.

The TechNote on Catmull-Rom splines discussed a uniform parameterization that assigned parameter values at the knots based only on the total number of knots. Consider a Catmull-Rom spline with four user-specified knots. Recall that the curve interpolates these knots. The additional knots on each end affect the curve's shape. Uniform parameterization specifies the following values,

Knot 1: t = 0
Knot 2: t = 1/3
Knot 3: t = 2/3
Knot 4: t = 1

Knot spacing makes no difference in the parameterization. The chord-length parameterization assigned parameter values at each knot based on the fraction of cumulative chord length at each knot to the total chord length between all knots. Suppose in the same four-knot example that the chord length from knot 1 to knot 2 was 42% of the total chord length. Suppose that the chord length from knot 2 to knot 3 was 18% of the total chord length, leaving the chord length from knot 3 to knot 4 at 40% of total chord length. The chord-length parameterization specifies the following values.

Knot 1: t = 0.0

Knot 2: t = 0.42
Knot 3: t = 0.6
Knot 4: t = 1.0

Think about a series of markers or visual indicators plotted on top of the entire curve at uniform parameter increments, say t=0, t=0.1, t=0.2 … t=1. While the curve's shape may remain the same between two parameterizations, each parameterization affects the distribution of markers along the curve.

In between knots, the curve is specified by local geometric constraints, a local basis, and a local parameter. The value of t in between knots is linearly mapped into a local parameter in the range [0,1]. Suppose $u^{(i)}$ represents the local parameter for the i-th segment. In the above example for chord-length parameterization, t in the interval [0.42, 0.6] maps $u^{(2)}$ into [0,1], so t = 0.42 corresponds to $u^{(2)}$ = 0 and t = 0.6 corresponds to $u^{(2)}$ = 1. Specifics of local mapping for both uniform and chord-length parameterizations were illustrated in the code base for the previous TechNote on Catmull-Rom splines.

If changing the parameterization does not affect the shape of the curve, why should programmers be concerned with parameterization in the first place?

Animation and Velocity

Although mathematically imprecise, the previous section provided some intuition on the mechanics of parameterizing piecewise curves. If the goal is to plot the curve, then it is still not clear why we should care about parameterization. Even the curve-constrained scroller TechNote illustrated an application of both single-segment and piecewise (Hermite) curves whose function did not depend on parameterization.

Consider the problem of path animation. A typical Flash implementation involves an *onEnterFrame()* handler that varies a parameter, t, from 0 to 1. At each frame, the current parameter value is used to query the corresponding x- and y-coordinate on the curve. A sprite (MovieClip) is moved from its current position to the next position along the curve. For suitably small parameter increments, the sprite appears to move smoothly along the curved path.

Most applications require the sprite to move along the curve with uniform velocity. Recall that instantaneous velocity is the derivative of position with respect to time. In this case, position is measured along the curved path. Uniform velocity along the curve does not translate to uniform velocity with respect to horizontal or vertical position.

Recall the concept of distributing markers along the curve at uniform parameter values. Typical parameterizations result in non-uniform marker distribution. If the *onEnterFrame()* handler increments the parameter by a fixed delta at every iteration, the result is often erratic motion along the curve. Since the markers are unevenly spaced, the sprite moves faster along some sections of the curve than others. There is the same number of frames between markers, but the distance along the curve in between markers is not the same.

Now, we care about curve parameterization.

Arc-Length Parameterization

In order to provide the programmer with motion control along the curve, parameterization in terms of arc length is very desirable. Unfortunately, such a parameterization is problematic since arc

length, s, along the curve and the curve's 'natural' parameter, t, are generally not linearly related. The relationship for piecewise curves is particularly complex.

Since the total arc length of a parametric curve can vary greatly with knot location and screen units, it is often more convenient to work with normalized arc length, s^* in [0,1] . s^* = 0 corresponds to zero arc-length and s^* = 1 corresponds to total arc length at the final knot.

Since s^* is a strictly increasing function of t, there is a 1-1 relationship between s^* and t. Suppose this relationship is expressed as $s^* = L(t)$. Since $L : [0,1] \rightarrow [0,1]$ is 1-1, $t = L^{-1}(s^*)$.

The curve is naturally parameterized (and plotted) on t, i.e. $Q(t) = (x(t), y(t))$, so we may consider an arc-length parameterization as describing the trace

$$P(s^*) = [x(L^{-1}(s^*)), y(L^{-1}(s^*))] \qquad [1]$$

Given the function, L^{-1}, uniform motion along the curve is programmed by varying s^* from 0 to 1 in small, fixed increments. The exact same *onEnterFrame()* handler described above could be used 'as is', by changing the curve's parameterization.

There are several methods for achieving an approximate arc-length parameterization. The method presented in this TechNote is a compromise between performance and simplicity and it takes advantage of already existing tools from previous TechNotes. A future TechNote will consider more advanced approaches. The goal of the current work is to provide an intuitive foundation for the mathematics that are to follow in a subsequent paper.

Bisection and Newton's Method

A prior TechNote illustrated that the arc length of a parametric curve is given by

$$s = \int_0^1 \sqrt{f'(t)^2 + g'(t)^2}\, dt$$

where $x = f(t)$ and $y = g(t)$. In general, this integral must be evaluated numerically. This also implies that the function L^{-1} must be approximated numerically.

One way to approximate L^{-1} is to construct a table of s^* vs. t values. At the i-th iteration, a table search is used to isolate the interval, $L(t_i) \le s_i^* < L(t_{i+1})$. The interval $[t_i, t_{i+1}]$ is used as a start interval for bisection. When the bisection tolerance is met, the iteration stops and the midpoint of the final interval is taken as an approximation of the t-value corresponding to the normalized arc length. Although the number of iterations is finite, an arc-length computation is required at each step of the iteration, so this method is very expensive.

An alternative approach is Newton's method, searching for zeros of the equation $s^* - L(t)$. A table search is used to isolate an initial interval and set a starting point for the iteration. Each Newton iteration still requires an arc-length computation. Newton's method has better theoretical convergence properties than bisection, although these properties depend greatly on choice of

initial starting point. Convergence is not guaranteed and it is theoretically possible for an iteration to produce an estimate outside the original interval, forcing termination of the procedure.

Cubic Spline Interpolation

Although the relationship between t and s^* is generally nonlinear, it is also just as generally smooth and well-behaved. Since we already have a cubic spline method available from a previous TechNote, another possible approach is to pre-compute a relatively small initial table and use the cubic spline to interpolate in between table values.

There is no iteration or additional arc-length computation required to evaluate $L^{-1}(s^*)$. The computation per spline evaluation is modest, although it is possible to do better. The advantage of this approach is its conceptual simplicity and the ability to use already packaged library methods.

The first question is where to place spline knots. Although the method of approach is conceptually simple, the piecewise structure of the spline makes it difficult to perform any type of error analysis on knot placement. We are left with ‘back of the envelope' approaches. One approach I have used in many occasions in the past is to place a spline knot at each of the Catmull-Rom knots and two spline knots in between. For many curves used in games, user-interfaces, and other applications, this approach has worked rather well. Keep in mind that it's just a back-of-the-envelope heuristic with no error analysis behind the method.

This method of approach is best suited for situations where the Catmull-Rom spline is constructed once and sprites are animated across the path one or more times. If an animation calls for simultaneously adjusting the knots during animation, I would recommend a completely different approach.

The cubic spline method also serves as a good baseline against which more sophisticated methods will be compared in at later TechNote. So, keep in mind that this is just the first step, not the end of the journey ☺

Modifications to Catmull-Rom Class

The Catmull-Rom class has been modified from previous versions for the current problem. The previous code is still separately distributed so that demo codes from prior TechNotes continue to work. The method to compute arc length by segment summation has been removed as it was previously supplied only to compare against the Gauss-Legendre method. Chord-length parameterization has been removed. The *CatmullRom* class now offers two parameterizations; uniform and arc-length.

The *CatmullRom* class now contains a reference to the Spline3 class. The class contains a new method, *arcLengthAt()* that returns the arc length of the curve for a given *t*-parameter.

Example

The first example is a four-knot curve was constructed and markers placed at uniform increments of t. The non-uniform placement was noted as a prelude to arc-length parameterization.

In the diagram below, this curve is re-created and plotted in blue. The knots are shifted horizontally to the right, and the curve is re-parameterized on arc length. The new curve is

plotted in green and markers are plotted on top of the curve at uniform increments in normalized arc length.

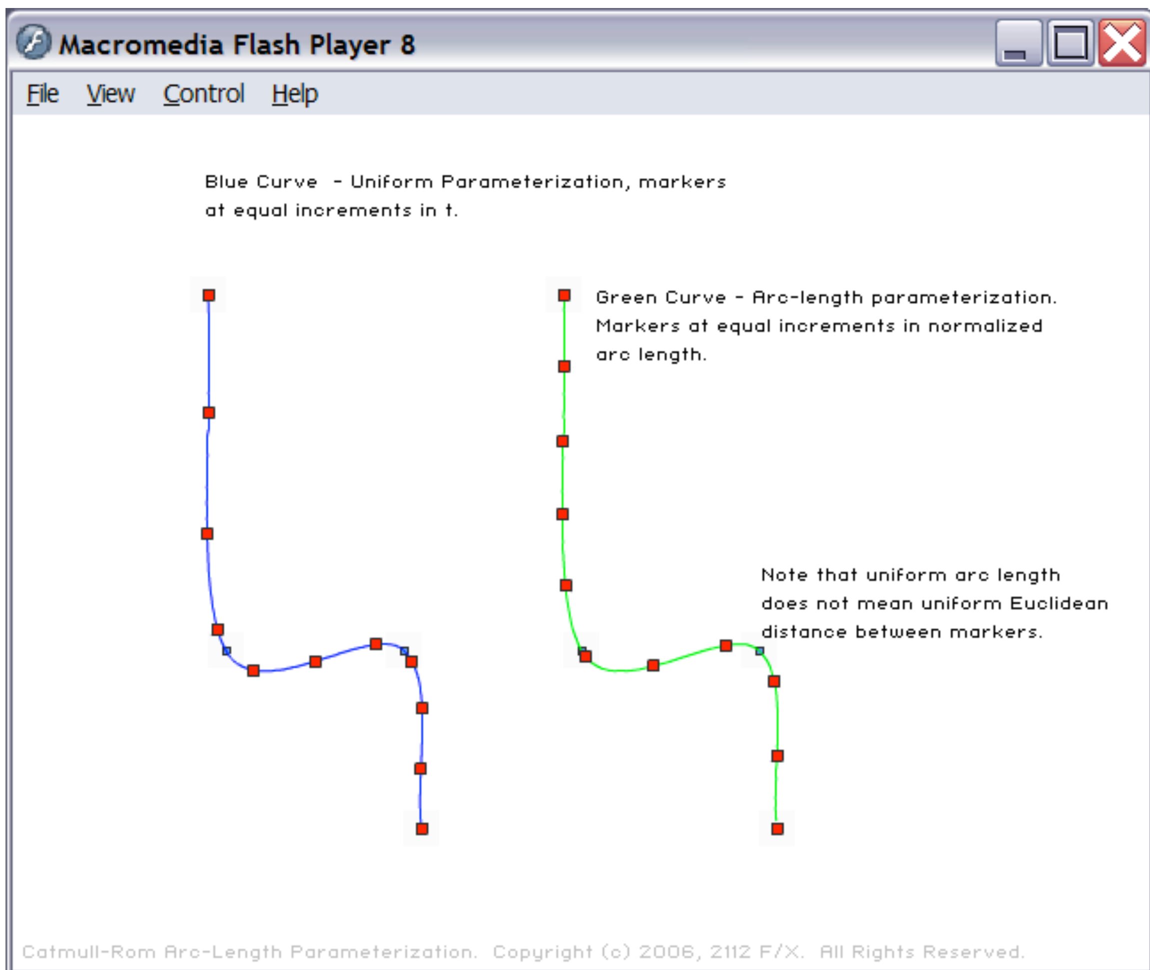

Diagram 2: Distribution of markers for different parameterizations.

As it happens, we can use parametric curves to approximate conic sections. Cubic Bezier's are well-suited for this task. If you don't mind overkill, the Catmull-Rom spline can be used for tasks such as distributing MovieClips evenly across an ellipse.

The file **ellipse.fla** illustrates such an example, although it is far from a computationally efficient means for this specific task. In fact, the endpoint tangent specifications are incorrect if the goal is to draw an ellipse. As an exercise, try plotting the curve to see for yourself. Move the y-coordinate of each tangent up and down and notice the effect on the plotted curve. A future TechNote will explore how to assign control points of cubic Bezier segments to approximate conic sections. In the mean time, sample output from **ellipse.fla** is shown below.

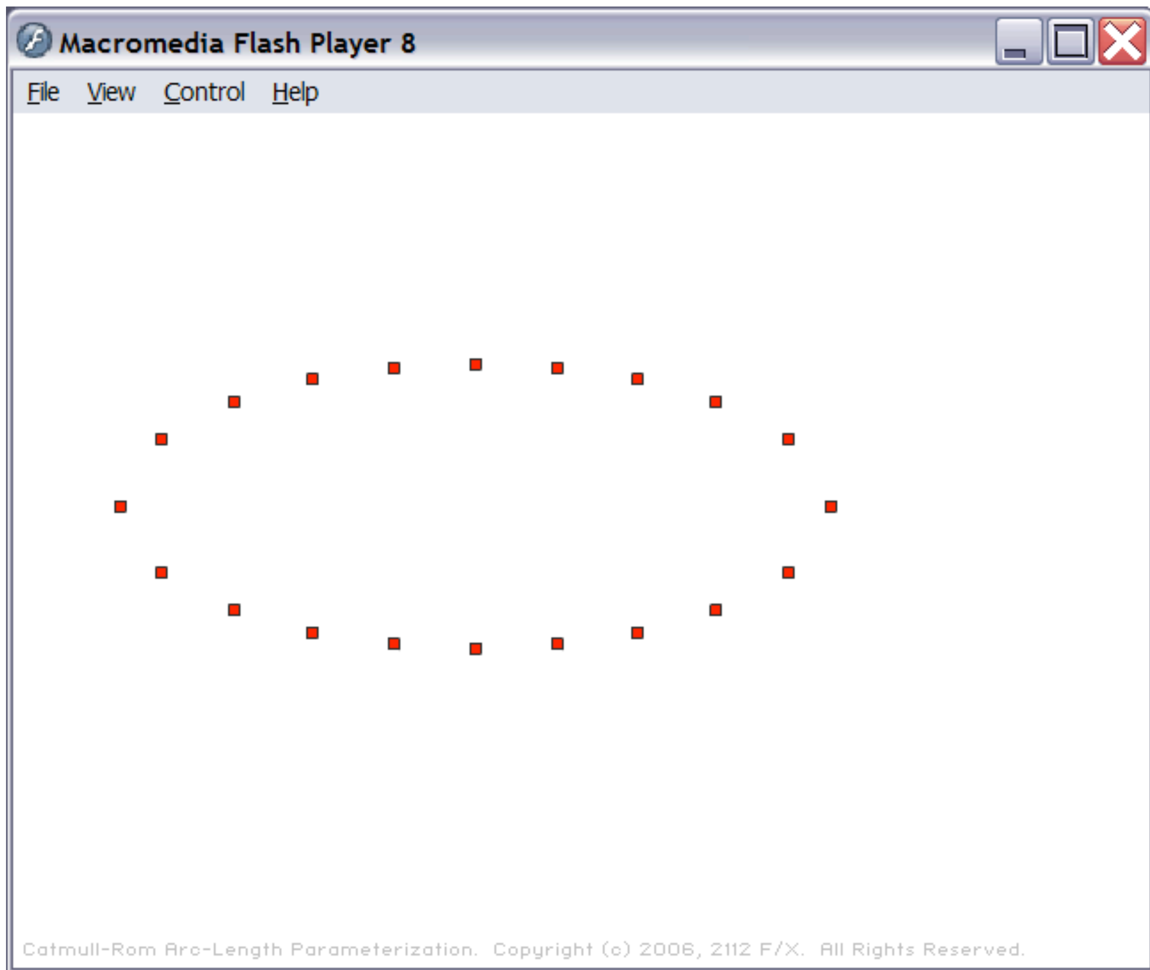

Diagram 3: Distribution of markers along an approximately elliptical boundary using a Catmull-Rom spline.

I would never recommend this as a method of approach if you are truly interested in distributing points evenly around an ellipse. This is more of an interesting example than a practical approach. On the topic of interesting examples, arc-length parameterization can be directly applied to the parametric form of an ellipse. This process reparameterizes the ellipse from t in $[0,2\pi]$ to normalized arc length in [0,1].

When distributing the markers, it is important to exploit symmetry. The file **ellipse2.fla** illustrates this approach. After reparameterization, normalized arc length is sampled in the interval [0,1/4], which corresponds to the natural parameter in $[0,\pi/2]$. Reflections about semimajor and semiminor axes are used to position markers in the other quadrants.

Path Following

In practice, it is necessary to both move a sprite along a path and orient the sprite in the direction of the path. This is normally accomplished by rotating the sprite to align with the curve's tangent vector at a particular point.

Astute readers may recall from calculus that the relationship between the curve's tangent line and the horizontal axis is given by $\tan\theta = dy/dx$. Since the curve is parameterized on t, the chain rule is applied to compute $dy/dx = (dy/dt)/(dx/dt)$. The CatmullRom class contains methods to compute the necessary derivatives and these were used to plot tangent lines in a prior TechNote. So, it seems all the necessary tools are available.

There are two problems with this approach. In addition to computing t from s^*, this approach requires evaluation of both the x- and y-coordinates and two derivatives. The second problem is with the tangent function itself. Consider a roller coaster or airplane performing a loop. The tangent function is undefined at a vertical angle of attack and numerically unstable for angles close to vertical. A sprite might 'flip' its orientation passing through a 90-degree angle.

Related to stability is the issue of initial orientation. Consider a knot sequence whose x-coordinates are strictly increasing. Suppose the curve is oriented upwards from the first to second knot and downward going into the final knot. We would expect the sprite to be oriented down and to the right as it completes the animation.

Now, suppose the knot sequence was reversed so that each knot is in exactly the same position, but the order is from right-to-left. In this case, the initial orientation of the sprite is expected to be up and to the left. The same formula used to animate the sprite in the previous case now orients the sprite in the wrong direction.

Fortunately, there are easy solutions to both cases. As long as the curve is traced in small increments of arc-length, Δx and Δy should be very small, meaning that $\Delta y/\Delta x$ is a good approximation to dy/dx. We get a double bonus from the coordinate deltas as the tangent issues are alleviated by using the *atan2()* function. Now, it may seem that there is a problem applying *atan2()* as it measures angles from the origin to a specific point.

The required angle is between the horizontal axis and the curve's tangent vector (in the direction the sprite is moving). Given a point (x,y) on the curve, any point on the tangent vector in the correct direction gives the line segment necessary to compute the angle. When the point (x,y) is translated to the origin, (Δx , Δy) is a good approximation to the appropriate point required for the *atan2()* function. To initially orient the sprite, it is necessary to pre-compute the initial Δx and Δy. This ensures the sprite is pointing in the correct direction for left-to-right and right-to-left motion.

Proper orientation of the sprite depends on the curve normal and tangent vector in the direction of motion along the path. Unit vectors in these directions form a basis for a very interesting coordinate space. Further discussion is beyond the scope of this TechNote. Readers are encouraged to Google 'Frenet Coordinates or 'Frenet-Serret formulas.'

Path Following Example Code

Observations from the previous section are implemented in the **pathtween.fla** example. This driver allows interactive creation of a Catmull-Roms spline path and illustrates animating a sprite along that path. After placing the knots, click on the 'Draw Curve' button to plot the spline Click the 'Markers' button to overlay markers at constant increments of normalized arc length. Click the 'Animate' button to move a sprite (top view of an SR-71) to the first knot and begin the animation.

The sprite is first positioned and oriented in the *__initPlane()* function,

```
var __initPlane:Function = function():Void
{
  __prevX   = __myCurve.getX(0);
  __prevY   = __myCurve.getY(0);
  __plane._x = __prevX;
  __plane._y = __prevY;

  __arc = 0;
  __plane._visible = true;

  var deltaX:Number = __myCurve.getX(__delta);
  var deltaY:Number = __myCurve.getY(__delta);
  __plane._rotation = Math.atan2(deltaY, deltaX)*DEG_TO_RAD;
};
```

The sprite is moved to the first spline knot and the normalized arc-length is set to zero. The delta value is used to compute the first Δx and Δy. These values are used to assign the initial orientation.

Sprite motion is controlled with a simple onEnterFrame() handler,

```
var __planeAnimation:Function = function():Void
{
  __arc += __delta;
  if( __arc > 1 )
    delete this.onEnterFrame;

  var planeX:Number = __myCurve.getX(__arc);
  var planeY:Number = __myCurve.getY(__arc);
  __plane._x        = planeX;
  __playe._y        = planeY;

  var deltaX:Number = planeX - __prevX;
  var deltaY:Number = planeY - __prevY;
  __prevX           = planeX;
  __prevY           = planeY;

  this._rotation = Math.atan2(deltaY, deltaX)*DEG_TO_RAD;
};
```

The handler is run in the scope of the airplane sprite (MovieClip). Normalized arc length is incremented by a constant delta. The *getX()* and *getY()* methods use the internal cubic spline to compute *t* as a function of s^*. The resulting value of *t* is used internally to evaluate the Catmull-Rom spline as in previous examples. The programmer is unaware of this internal conversion. A request is made for a coordinate on the curve based on either natural parameter or normalized arc-length, depending on the current parameterization.

The values for Δx and Δy are updated and used to orient the sprite just as in the initialization method. A screen shot is illustrated below.

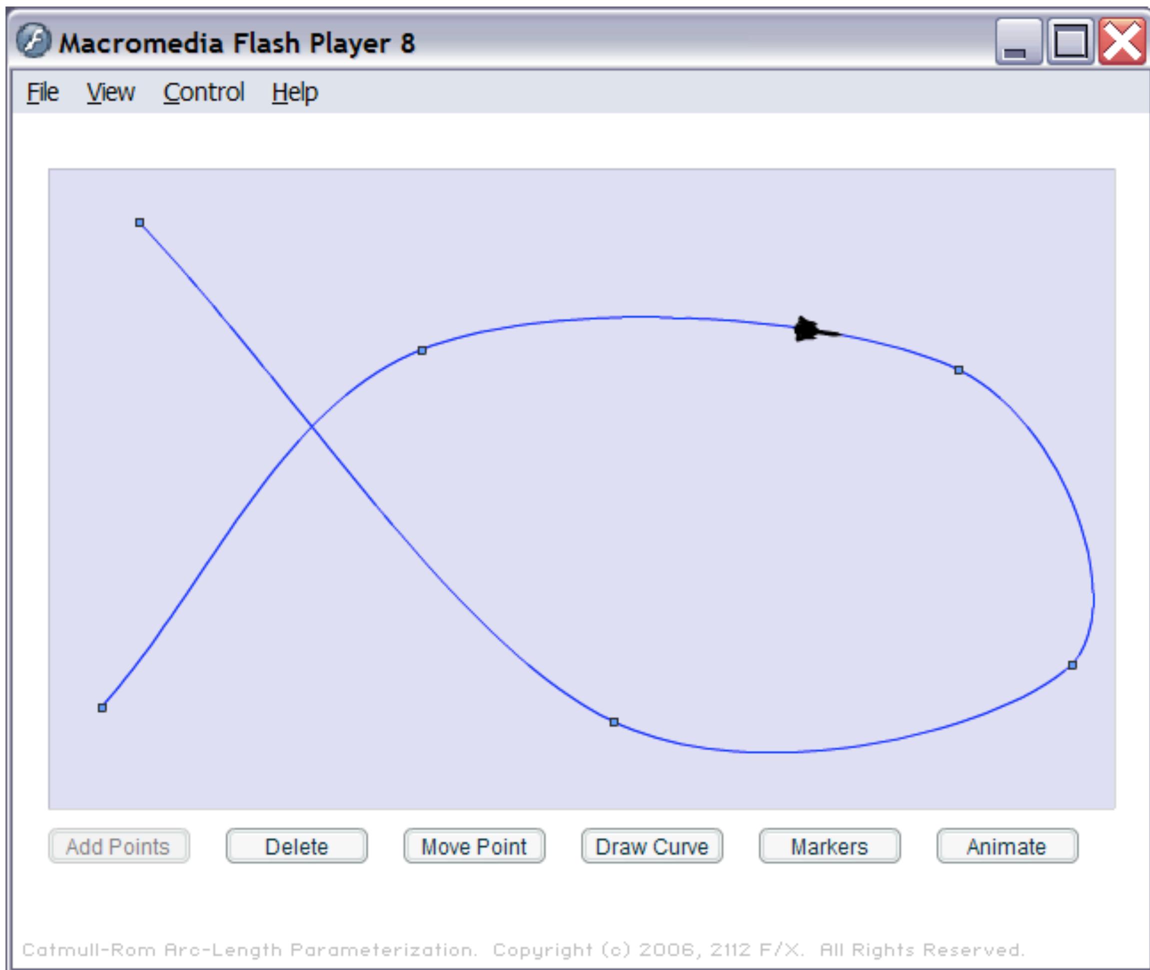

Diagram 3: Orienting a sprite along a path

The code is currently organized to be easily understood, so it has yet to be optimized. Optimization will be performed at a later time when the algorithm is updated.

Preview of Other Methods and Sample Code

The method for arc-length parameterization presented in this paper is intended to lay a foundation for more accurate and faster implementations in the future. This section provides a preview of some work in the literature that will be useful in future versions of this code.

The ideal approach is one that produces a naturally arc-length parameterized curve. DeRose et al. [1] shows how to compose two curves into a single, higher-order curve. Peterson [2] illustrates how to apply this method to compose the original curve and a reparameterization curve into a single higher-order curve that traces the original curve and is approximately parameterized by arc length.

Walter et al. [3] demonstrates application of a single- or double-span cubic Bezier to relate the curve parameter-t to arc-length. Wang et al. [4] illustrate the generation of a new curve that both approximates the original curve and is naturally parameterized on arc length.

Another consideration is fast redraw of the path in the event that some knots are animated while the sprite is in motion. The path may need to be drawn as part of a shape outline that is filled and rendered on Stage. One example is a boat animated along the surface of a moving ocean. Since Flash contains a highly optimized quadratic Bezier, a piecewise quadratic Bezier that interpolates a set of knots could be redrawn very quickly. Such a curve is better suited for real-time knot modification and simultaneous sprite animation.

A future TechNote will discuss the creation of such a curve and a composite curve for arc-length parameterization.

(March 2007) The arc-length parameterized Catmull-Rom spline is part of the AS 3 parametric curve library. The ellipse demo is now in Flex and uses the Ellipse Class. The path tweening example is also provided in Flex. The code distribution requires a development environment capable of compiling Actionscript 3. FlexBuilder 2 is used for all the demos.

References

[1] DeRose, T.D., Goldman, R.N., Hagan, H. and Mann, S. “Functional Composition Algorithms Via Blossoming, “ACM Transactions on Graphics 1993 12 (2), pp. 113-135.

[2] Peterson, J.W. “Arc-Length Parameterization of Spline Curves,” http://www.saccade.com/writing/graphics/RE-PARAM.PDF .

[3] Walter, M. and Fournier, A., “Approximate Arc-Length Parameterization,” Proceedings of the 9th Brazilian Conference on Graphics and Image Processing, Oct. 1996, pp. 143-150.

[4] Wang, H., Kearney, J., and Atkinson, K, “Arc-Length Parameterized Spline Curves,” presented at 5th International Conference on Curves and Surfaces, June 2002.