TechNote TN-06-006

Composite Bezier Curves

Jim Armstrong
Singularity
November 2006

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

Introduction

The previous TechNote on recursive subdivision discussed the concept of fitting a composite curve (or spline) to a set of knots. The knots are to be animated an a manner that roughly mimics the behavior of an organic item such as rope, cloth, or hair acting under forces such as wind or gravity. The goal is not a physically realistic simulation. The animation should be 'believable' and the curve redraws fast enough to keep up with a reasonable frame rate.

A composite, cubic Bezier curve was suggested. Each knot is connected with a single, cubic Bezier curve. The control cages for each segment are constructed in a manner that meets some pre-specified continuity criteria (to be discussed). Recursive subdivision is used to rapidly draw each cubic segment with a small number of quadratic curves. The recursive subdivision TechNote illustrated one approach to a single subdivision for very rapid redraws.

This composite curve trades physical accuracy for fast drawing. The curve is not to be used to interpolate values in between knots or animate a sprite along a path. This implies that some of the criteria used to construct the curve are subjective. Different users may have different opinions as to what constitutes a suitable curve for animation.

It is desirable to provide the user with as much control as possible over the visual appearance of the composite curve, commensurate with meeting certain required criteria at the join points. Every additional constraint on the curve's construction reduces the amount of user control.

Although G^0 continuity at the joins is a minimum requirement, such a curve is hardly useable in practice. Organic items sometimes have kinks, but it is better to seek at least G^1 or even C^1 continuity. Kinks can be loosely modeled by moving knots close together. A more interesting effect is to simulate 'ripples' or waves moving along sections of the curve. Such ripples are often seen when observing water. They can be seen at the end of a whip or in hair or cloth being blown in the wind.

Since the composite curve is not to be used for static drawing or animating sprites, the approach presented in this TechNote is a bit off the beaten path. The composite Bezier curve discussed in this TechNote is G^1 continuous at the joins and provides the user with a certain amount of tension control. Lower tension settings have the potential to induce occasional artificial ripples in the curve. While this is not a desirable effect for interpolation or static drawing, it can be very cool when used in animation.

After all, life isn't much fun if we can't do something different from time to time ☺

The construction of individual cubic Bezier curves has already been discussed and the previous TechNote discussed recursive subdivision in detail. The remaining details are setting tangents at the joins and constructing control cages for each segment to provide the desired curve qualities.

Before discussing these items, an exercise from the previous TechNote is reviewed.

Intersection Review

A quadratic control cage is constructed from an existing cubic control cage in the process of recursive subdivision. The fundamental diagram is shown below.

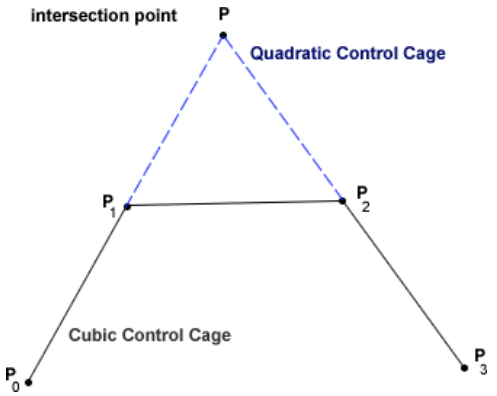

Diagram 1 - Construction of quadratic control cage

The point, P, is the intersection of the P_0P_1 and P_3P_2 line segments. The intersection formula had been discussed in a prior TechNote and the code was used in the modified *Bezier3* class from the recursive subdivision TechNote. In that TechNote, it was noted that the intersection formula did not handle the case where one (or both) of the line segments was vertical or near vertical. The intersection formula also breaks down when the line segments are nearly or exactly collinear. Adjusting the formula to handle these cases was left as an exercise. If you have already completed that exercise, then move onto the next section.

The first case to consider is where one of the segments is vertical.

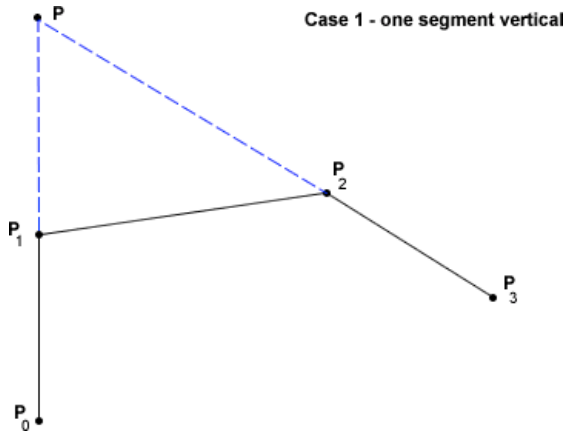

Diagram 2 - Intersection with one vertical segment

Review the basic intersection code,

```
var m1:Number = (_points[3] - _points[1])/(_points[2] - _points[0]);
var m2:Number = (_points[5] - _points[7])/(_points[4] - _points[6]);
var b1:Number = _points[1] - m1*_points[0];
var b2:Number = _points[7] - m2*_points[6];

__pX = (b2-b1)/(m1-m2);
__pY = m1*__pX + b1;
```

where the x- and y-coordinates of the cubic control points are stored sequentially in the ***_points*** array. The first slope, m_1, is nearly infinite when $|P_{1x} - P_{0x}|$ is nearly zero. If the x-coordinates are within some prespecified tolerance, an alternate formula is used. The equation of the P_3P line is used to obtain

$$\begin{aligned} P_x &= P_{0x} \\ P_y &= P_{3y} + m_2(P_{3x} - P_{0x}) \end{aligned} \qquad [1]$$

A similar equation is used for the case where $|P_{2x} - P_{3x}|$ is less than the zero tolerance. In the case where both line segments are nearly or exactly vertical, there is no intersection point. Since this intersection computation is used in recursive subdivision, it is only necessary to construct a quadratic control cage that forces a subdivision. In this case, a modified version of eq. [1] is used.

In the case where the two line segments are nearly or exactly collinear, there is no well-defined intersection. There is a loss of significance in the subtraction of the two very small slopes in the line of code

```
__pX = (b2-b1)/(m1-m2);
```

or the denominator is exactly zero if the segments are collinear. The x-coordinate computation is numerically unstable and this instability is propagated to the y-coordinate computation. For this case, the midpoint of the P_2P_3 segment is returned as the intersection point.

Since the quad control cage in this case produces a curve that should not require subdivision, it is possible to set a flag to exit the subdivision loop immediately. This is one of the reasons why termination tests based on the geometry of the control cage are used more often in practice than methods such as midpoint differencing.

A more robust intersection code is provided in the *FastBezier* class, provided with the code distribution in this TechNote. Contrast this code to that used in the *Bezier3* class, distributed with the recursive subdivision TechNote. Note the use of a zero tolerance in the code. It is poor practice to test for exactly zero because of propagation of rounding errors. Sufficiently near zero can be just as dangerous as exactly zero.

Tangent Construction

This TechNote illustrates the construction of a curve that is G^1 continuous at the knots. The in- and out-tangents at each knot have a common direction, but generally different magnitudes. We are at liberty to choose the direction of each tangent.

One approach that comes to mind is to use the same algorithm as a Catmull-Rom spline. The tangent at each knot is parallel to the chord between the prior and successive knot. As with Catmull-Rom splines, artificial points must be placed at both ends of the knot set so that tangent information is available for all knots.

There are about half a dozen methods that are used in practice for tangent construction in composite curves. For reasons of time and space, this TechNote only covers one in detail -- the normal to the angle bisector of the chord between knots. This is illustrated in the following diagram. The angle bisectors are shown in blue and the tangents in red.

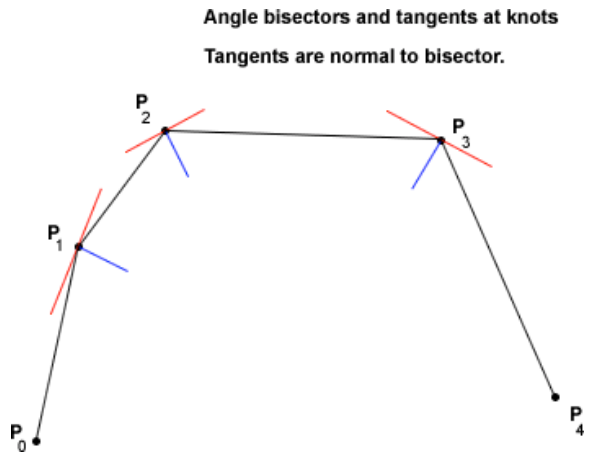

Diagram 3 - Angle bisectors and tangents at interior knots

This approach still raises the question of how to compute tangents at the beginning and end of the knot sequence, i.e. P_0 and P_4 in diagram 3. The typical approach is to match the 'in' and 'out' angles as shown below.

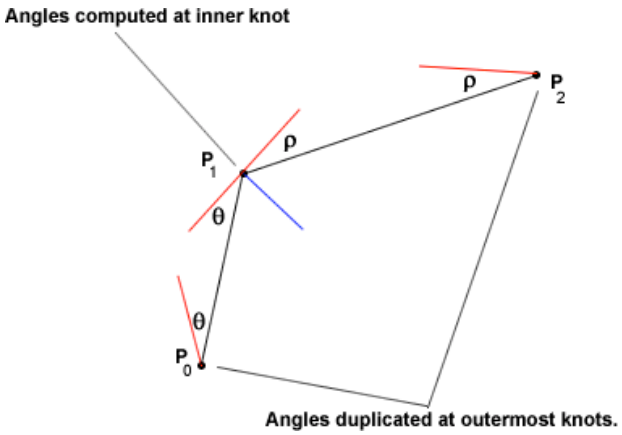

Diagram 4: Tangents at outermost knots

At first glance, this approach looks as if it requires like a lot of trig computations. A certain amount of computation is required no matter what tangent approach is employed. As it happens, the control cages for every segment can be constructed without any trig computations. The process is a useful exercise in analytic geometry.

Angle Bisector and Tangent Magnitudes

Consider how to compute the angle bisector at point P_1 in diagram 4. The first thought that comes to mind is using the geometric interpretation of the inner product of P_1P_0 and P_1P_2 to compute the cosine of the angle between these two vectors. After an inverse cosine computation, divide the angle by two. The slope of the bisector segment can be computed from the angle. Given a point and a slope, the normal through that point can be computed.

The inner product computation requires normalizing P_1P_0 and P_1P_2 to unit length. Suppose the normalized vectors are given by u_1 and u_2, as illustrated in the following diagram.

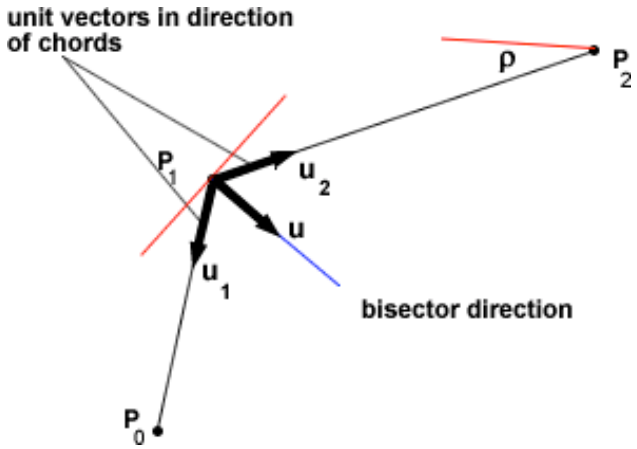

Diagram 5: Construction of angle bisector

There is no need for the trig computations to compute the angle bisector. The vector $u = u_1 + u_2$ is in the direction of the angle bisector. Define

$$
\begin{aligned}
d_1 &= \|P_1P_0\|_2 \\
d_2 &= \|P_2P_0\|_2 \\
\Delta x_1 &= P_{0x} - P_{1x} \\
\Delta y_1 &= P_{0y} - P_{1y} \\
\Delta x_2 &= P_{2x} - P_{1x} \\
\Delta y_2 &= P_{2y} - P_{1y}
\end{aligned}
$$

from which

$$u_x = (\Delta x_1 + \Delta x_2)/d_1$$
$$u_y = (\Delta y_1 + \Delta y_2)/d_2$$

Suppose u is normalized. Points along the normal to the bisector are given by

$$d = \|u\|_2$$
$$n_x = P_{1x} - tu_y \qquad [2]$$
$$n_y = P_{1y} + tu_x$$

for $t > 0$. Referring to diagram 5, points along the normal in the opposite direction are given by

$$n^*_x = P_{1x} + tu_x$$
$$n^*_y = P_{1y} - tu_y \qquad [3]$$

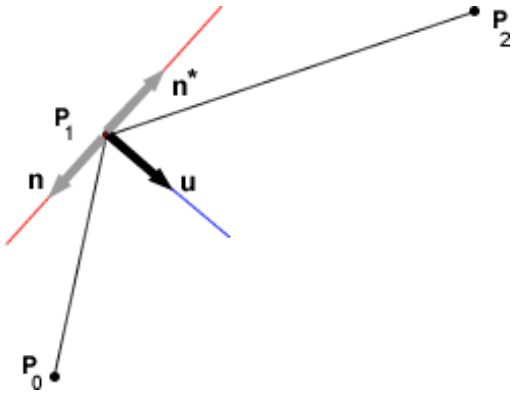

Diagram 6: Normal to angle bisector

Not only is this computation relatively simple, a parameter, t, is available to control the geometry of the control cage. Since u is normalized to unit length, t can be chosen based on the chord lengths that have already been computed. Smaller values of t result in a control cage that is closer to the chord, resulting in a curve that is closer to a line segment. Higher values of t induce more curvature. Thus, the t-parameter serves as a tension control.

Now, it can't be that easy, can it? Astute readers may have already noted that the equations for the coordinates along the normal directions depend on knot ordering. Both equations [2] and [3] are correct if the knot sequence $P_0 \rightarrow P_1 \rightarrow P_2$ is in clockwise order. Signs need to be altered if the knot sequence is in counter-clockwise order. An ordering test based on cross-products is easily computed. The algorithm can be deconstructed from the sample code.

Once the parameter value is assigned, equations [2] and [3] provide the inner control points for the 'right' side of one control cage and the 'left' side of the next control cage. One way to choose specific parameter values is based on the chord distances, which have already been computed.

Define a range $[t_1, t_2]$, representing fractions of chord distance. A multiple in the range of 0.1 to 0.4 is a pretty good starting point.

The user is provided with a simple tension indicator, say a range of 1-5, where a value of 1 is interpreted as very low tension and 5 is interpreted as very high tension. It is the programmer's responsibility to determine how to map the user-specified tension indicator into a parameter value, t.

This approach breaks down if the two segments are nearly or exactly collinear as the vector, u, approaches the zero vector. The code supplied with this TechNote illustrates a simple compensation for this case.

Local Oscillations or Ripples

If there are substantial differences in the control cage angles moving from one chord to the next, the local cubic curve may have an inflection point, appearing to 'hump' at one end. As knots animate, this results in the appearance of a ripple along part of the composite curve. The effect is illustrated in the following diagram.

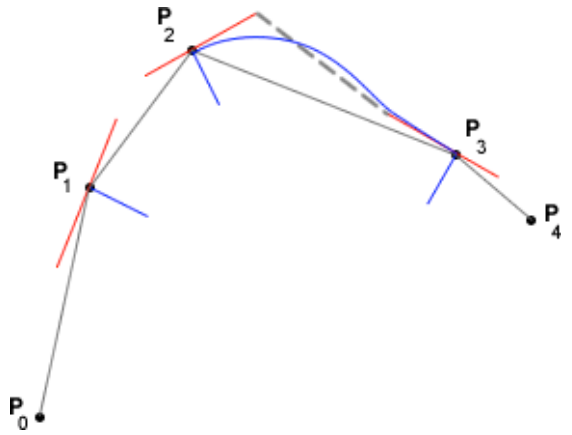

Diagram 7: Local Oscillation

There is a substantial difference between the angle formed by the P_1P_2 and P_2P_3 chords vs. the angle formed by the P_2P_3 and P_3P_4 chords. The hump near the P_2 knot is **heavily** exaggerated for purposes of illustration. If the latter knots in the sequence were rapidly animated up and down (with P_3 having greater amplitude), the result mimics the appearance of a 'ripple' moving along the composite curve. In practice, these ripples occur only occasionally and are of much less magnitude than the exaggeration in diagram 7.

The probability of such a hump appearing in one of the cubic segments increases as the tension decreases. Whether or not this effect is desirable is subjective. It is probably not desirable for static drawings. For certain types of animation (surface of water, hair, cloth), it can produce a very cool effect. Organic objects moving under complex forces such as wind tend to exhibit local oscillations.

The effect can be completely eliminated by altering the computation of one of the control points so that the line segment between inner control points is parallel to the chord, as illustrated in the following diagram.

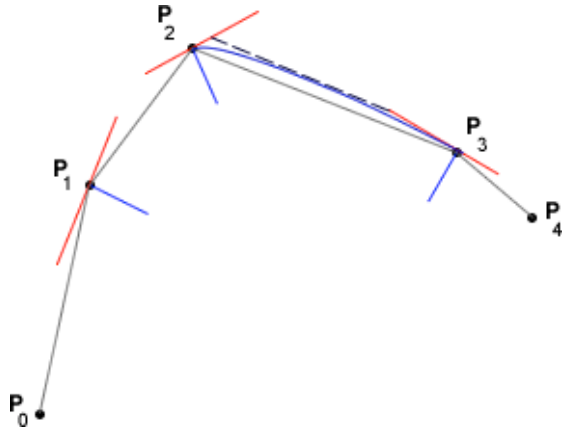

Diagram 8: Modifying control cage to eliminate local oscillation

Compare the orientation of the dotted line between inner control points in diagrams 7 and 8. The additional computations for the modified cage include determining the smaller angle and the line intersection. The code distribution in this TechNote allows the rippling effect. Modifying the code to construct control cages as illustrated in the diagram 8 is left as an exercise.

Initial and Final Segments

The first and last chords require matching angle computations to compute the control points, as illustrated in diagram 4. The necessary control point can once again be generated without any trig computations. Consider the illustration of the leftmost control cage as shown in diagram 9. The control points are P_0PRP_1. The point, R, is the 'right' inner control point computed using equation [2], or the appropriate modification if the first three knots are in counterclockwise order.

The point, M, is the midpoint of the P_0P_1 segment. The point, P, is to be computed. The vector, n, is normal to the P_0P_1 segment and passes through the midpoint of the segment.

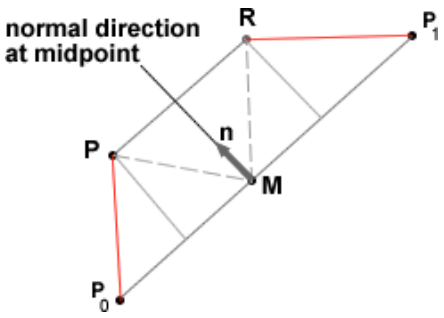

Diagram 9: Reflection used to compute inner control point

Once n is normalized to unit length, the point P is simple reflection of R about n. The vector, n, is easily computed from $P_0 - M$ and $P_1 - M$ with a normalization multiplier of $2/\|d_1\|_2$. The norm, $\|d_1\|_2$ is already available from the computation of R.

The reflection matrix used to transform R is

$$\begin{bmatrix} n_x^2 - n_y^2 & 2n_x n_y \\ 2n_x n_y & n_y^2 - n_x^2 \end{bmatrix}$$

which is symmetric, so only the upper triangle is computed. This reflection is only used for computing control points at the initial and final chords.

It is possible to preprocess the knot sequence and compute an artificial point to insert before the first real knot. The insertion is done in a manner so that the normal control point computations produce a 'left' inner point for the first user-specified segment that matches the result obtained by the above approach. There is no decrease in the total amount of computation. The pre-processing code is offset by not having to test for initial and final segments, allowing control cages to be generated by a single algorithm.

Implementation and Examples

The *BezierSpline* class (Flash code no longer online) implements the composite cubic Bezier curve as described in this TechNote. The *BezierSpline* class uses the *Composite* class to generate control points for each segment, given a knot sequence. The *Composite* class contains an array of *CubicBezier* instances, which is a small helper class used to store the control points.

The *BezierSpline* class also contains an array of *FastBezier* instances; one for each segment. The general usage of the *BezierSpline* class is very simple – add knots and draw.

The first call to the *draw()* method causes the internal *__assignControlPoints()* method to be called. This method is called when the knot sequence is invalid, either because a knot has been moved or the knots are entered for the first time. This method first constructs the control points for each segment and then applies the points to each cubic Bezier segment.

After control points are assigned, the local *draw()* method of each *FastBezier* instance is called, resulting in a single subdivision. Each cubic segment is drawn with exactly two quadratic curves.

The current plot is invalidated on any knot move. Since the anticipated use is that most, if not all, knots are moved each frame of an animation, the entire set of control points is reconstructed on invalidation.

Note that this code is only useful for fast animation – not static drawing or animating a sprite along the composite curve. The arc-length parameterized Catmull-Rom spline is better suited for the latter task. Although it is possible to query x- and y-coordinates along the composite curve, it is not to be used for interpolation between knots.

The following screen shot illustrates a sample plot. The control cages are drawn in red and the approximate curve is drawn in blue.

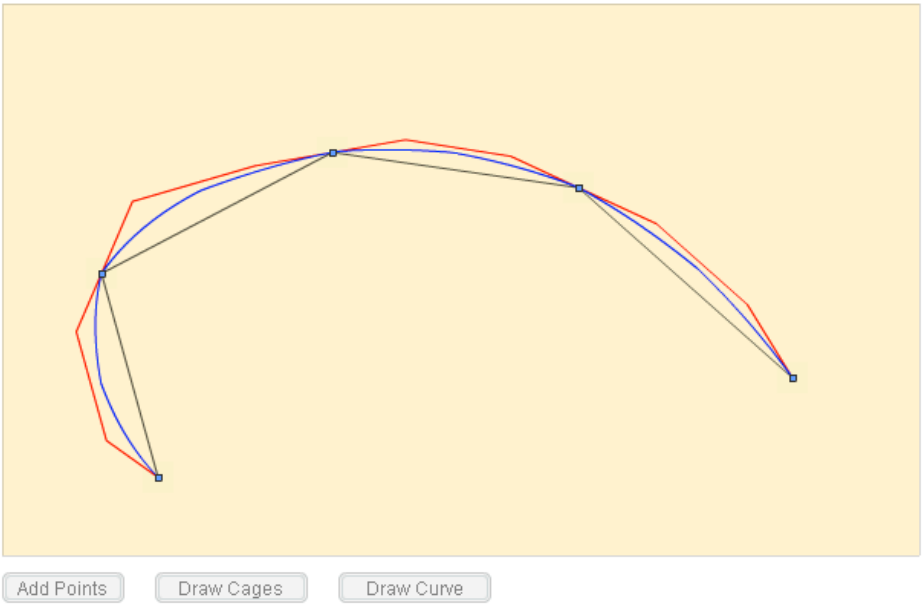

Diagram 10: Example composite cubic curve

The curve in diagram 10 is drawn with relatively low tension. The following example illustrates the highest tension setting.

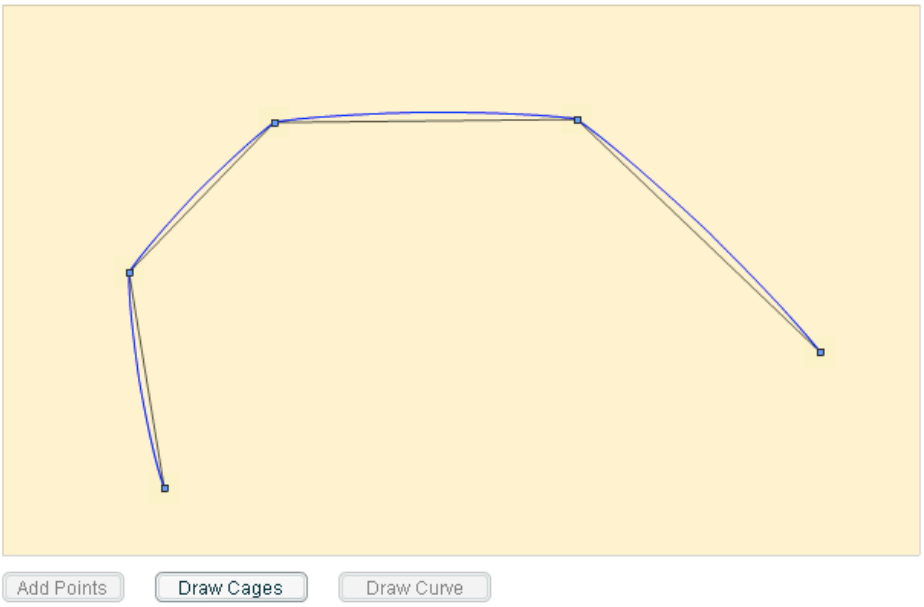

Diagram 11: Maximum Tension

The following example illustrates local oscillation. In this case, a small 'hump' appears at one of the knots. The lowest tension setting was used. Note that the effect is very small. It disappears by moving the tension factor up by one.

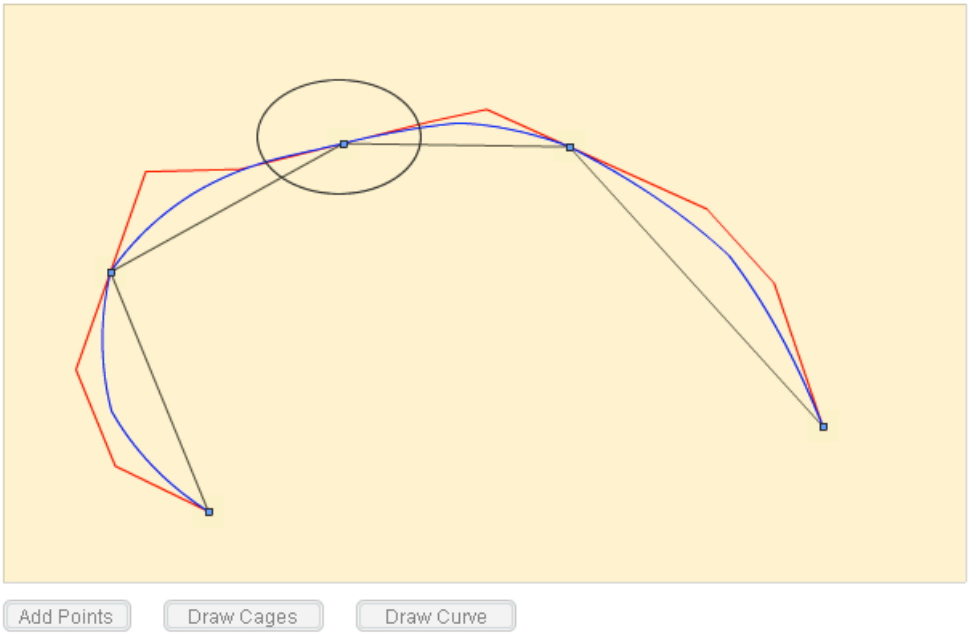

Diagram 12: Appearance of a small local oscillation

At the risk of too much repetition, it bears mentioning again that this approach is not suited for static drawing. For better static drawings, it is advisable to perform an extra subdivision. An algorithm I generally use in practice for slower animations where the more subtle visual characteristics of a curve are more noticeable is to first perform a subdivision on inflection. If no inflection point exists, a midpoint subdivision is applied. A second midpoint subdivision is applied for a total of four control cages per cubic curve. The data structures represented in the Bezier3 class from the recursive subdivision TechNote can be compacted in this case.

This approach is a more reasonable compromise between drawing accuracy and performance if greater drawing accuracy is desired.

Closed Loops

The composite Bezier spline may be used for closed loops. An extra knot is automatically added that duplicates the first knot. The same tangent strategy is employed, except that the angle bisector for knots n-2, 0, and 1 is used for the leftmost part of the control cage of the first composite segment. Knot n-1 is the same as knot 0. The desired continuity is achieved by reflection when setting the rightmost part of the control cage of the last segment. The process is divided into separate method calls for open vs. closed splines in the BezierSplineControl class to make the approach easier to follow.

Closure is illustrated below.

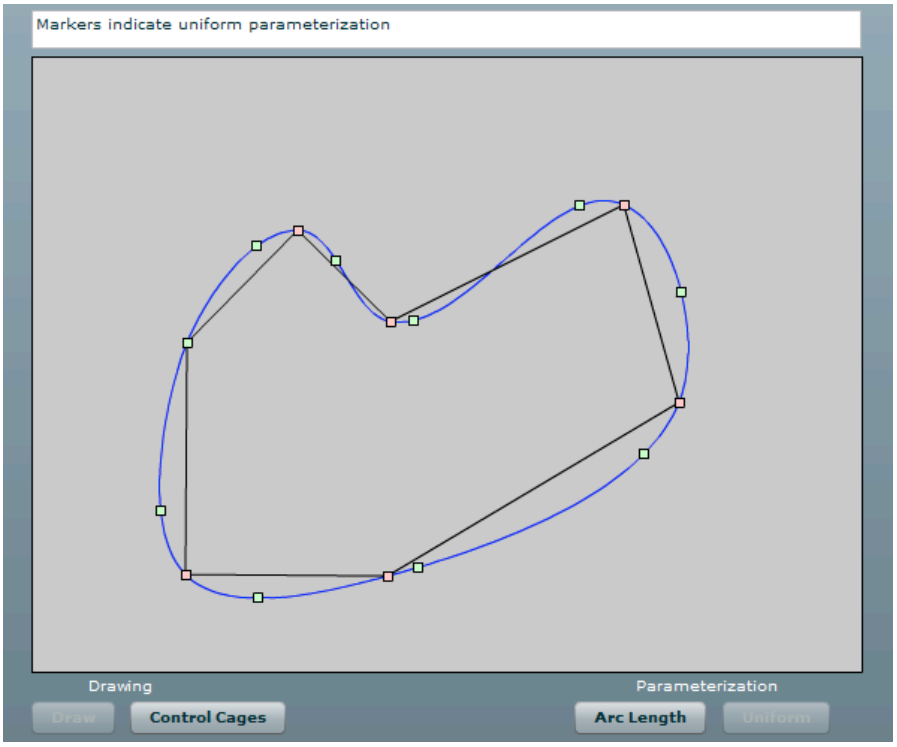

Diagram 13: Automatic Spline Closure

The orange markers are the original knots (the final knot is automatically inserted when a closed spline is selected). The green markers illustrate uniform parameterization at t=0, 0.1, 0.2, 0.3 … 1.0 .

The code supports approximate arc-length parameterization, so the demo provides the ability to switch between the parameterizations, observing the marker distribution. The availability of arc-length parameterization allows the composite curve to be used in path animation with velocity control.

Exercises

Since Flash draws each cubic segment in two parts, there may be some slight inconsistencies in the two curve segments. This is an artifact of the algorithm used to scanline render the quadratic segment. For static drawings, another subdivision pass may be desirable. For animations, especially where many curves are involved in the animation, the effect is not noticeable.

Some suggested exercises for those wishing to code this approach include

1 - Modify the control-point algorithm to implement the method illustrated in diagram 8. Notice the tendency of the composite curve to move more closely to the chords.

2 - Control points for each segment are regenerated if any knot is moved. This is easier if the anticipated usage is that most (or all) knots are moved every frame of an animation. Consider an alternate usage where only a single knot (or a small number of knots) is moved every frame. How would the code be altered to take this situation into account?

3 – Each control cage is constructed based on a multiple of the chord distance. The multiple corresponding to minimum tension is 0.4. What happens if this multiple is 0.5 or greater? To find out, adjust the tension mapping and make two of the chords near or exactly collinear.

4 - Place markers along the composite curve at regular intervals. Notice the distribution based on uniform parameterization and the difference between actual points on the composite curve and the approximation. You should see that the approximation tends to be below the actual curve. This emphasizes that the best use of the fast draw() method is in animation, not static drawings.

References

[1] Kochaneck, D. and Bartels, R., "Interpolating Splines with Local Tension, Continuity, and Bias control," Proceedings of the 11th International Conference on Computer Graphics and Interactive Techniques, ACM Press, pp. 33-41, 1984.

[2] De Boor, C. "A Practical Guide to Splines", Applied Mathematical Sciences 27, Springer.