TechNote TN-05-004

Cubic Bezier Curves

Jim Armstrong
Singularity
December 2005

This is the fourth 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.

Cubic Curves

This TechNote extends the introduces cubic Bezier curves and lays the foundation for future discussions on quadratic approximation.

A general matrix equation for parametric curves was discussed in the TechNote on Hermite Curves. While the quadratic Bezier curve equations were derived as an extension of deCasteljau's method, the cubic curve equations are now derived directly from the matrix form of Hermite curves.

Readers who are weak in matrix algebra may find it more instructive to search online references for deCasteljau's method, applied to cubic Bezier curves. The method of approach is the same as that described in the previous paper (just more segments).

Geometric Constraints

One of the concepts missing from many discussions of Bezier curves is how they relate to Hermite curves. In the discussion of Hermite curves, it was noted that direct manipulation of tangent handles made for a cumbersome user experience, especially with piecewise cubic curves. Bezier curves were described as one method for implicitly specifying tangents. It might be tempting to think that the tangent is simply the vector created from segments extending from the first and last control points. If that were the case, Bezier curves would be the same as Hermite curves as that is essentially how the tangent vector was created for a Hermite curve.

Since Bezier curves are supposed to be an extension of Hermite curves, suppose the implicit tangent in a Bezier curve is related to the Hermite tangent by a simple, linear relationship. Consider the case of a cubic curve with control points P_0, P_1, P_2, and P_3. Suppose P_1 and P_2 are tangent handles of a Hermite curve. In the Hermite TechNote, recall the mention of an old 'trick' of artificially weighting the tangent segment to provide more influence in the shape of the curve. Consider implicit Bezier tangents of the form

$$\begin{aligned} R_1 &= \alpha(P_1 - P_0) \\ R_2 &= \alpha(P_2 - P_3) \end{aligned} \qquad [1]$$

Given a value of α, the geometric constraints of the Bezier curve can be specified in a manner similar to the Hermite curve. A reasonable means to determine α in [1] is to consider the

physical interpretation of the parameter, t, as time. Consider four equally spaced control points in the plane (with unit distance),

$$P_0 = (0,0), \quad P_1 = (1,0), \quad P_2 = (2,0), \quad P_3 = (3,0)$$

Since all control points are collinear, the Bezier curve is a straight line, and the magnitude of the derivative at each endpoint is equal to α. A useful feature for the curve is to maintain constant velocity as *t* varies from 0 to 1, satisfied by $\alpha = 3$. If you have not already looked at the *Hermite3* class (included in the code distribution for the Hermite curve paper), load the class into Flash and look at the initial value of the artificial weight ☺

Bezier Basis Matrix

The Bezier constraint vector consists of the control points,

$$g = [P_0 \quad P_1 \quad P_2 \quad P_3]^t$$

It is related to the Hermite geometric constraints as indicated in the previous section, which can be written in the matrix form

$$g_h = [P_0 \quad P_1 \quad R_0 \quad R_1]^t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \qquad [2]$$

The matrix in equation [2] can be considered as a 'transition' matrix, M_{hb} between Hermite and Bezier form. Substituting this matrix into the derivation of the Hermite basis matrix yields the Bezier basis matrix,

$$M_b = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

If M_b is applied to the standard matrix equation for a parametric cubic curve, the cubic Bezier is specified by

$$B(t) = p_3^t M_b g_b = [t^3 \quad t^2 \quad t \quad 1]^t \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \qquad [3]$$

Expanding equation [3] yields,

$$
\begin{aligned}
B(t) &= (-t^3 + 3t^2 - 3t + 1)P_0 + (3t^3 - 6t^2 + 3t)P_1 + (-3t^3 + 3t^2)P_2 + t^3 P_3 \\
&= (1-t)^3 P_0 + 3t(t^2 - 2t + 1)P_1 + 3t^2(1-t)P_2 + t^3 P_3 \qquad [4] \\
&= (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3
\end{aligned}
$$

Notice the appearance of third-order Bernstein polynomials as the blending functions. This observation can be extended to higher-order Bezier curves. Readers familiar with Pascal's triangle (and recursive methods to generate rows while also exploiting symmetry) should find it easy to derive coefficients for any order Bezier curve.

In most discussions regarding cubic Bezier curves, the implicit tangent specification and direct relation to Hermite curves is ignored.

Curve Evaluation

In previous demo codes, the Actionscript code very closely followed the underlying math. While useful for instructional purposes, this is not the most efficient approach to evaluate a curve at a given t. A more efficient approach is to collect terms and evaluate the polynomial with nested multiplication. In previous examples, this was left as an exercise. Nested evaluation is illustrated in the cubic Bezier curve to illustrate the computational gains.

A simple approach to evaluating [4] such as

$$
\begin{aligned}
a &= 1 - t \\
b &= a^2 \\
c &= t^2 \\
c_0 &= ab \\
c_1 &= 3bt \\
c_2 &= 3ac \\
c_3 &= tc \\
B(z) &= c_0 P_0 + c_1 P_1 + c_2 P_2 + c_3 P_3
\end{aligned}
$$

requires twelve multiplications and four additions per curve evaluation! A cubic polynomial can be evaluated using nested multiplication with three multiplies and three additions per step. This is a substantial savings, so it is worthwhile to work through the algebra to collect terms

$$
\begin{aligned}
B(t) &= (1 - 3t + 3t^2 - t^3)P_0 + (3(t - 2t^2 + t^3))P_1 + 3(t^2 - t^3)P_2 + t^3 P_4 \\
&= (P_3 - P_0 + 3P_1 - 3P_2)t^3 + (3P_0 - 6P_1 + 3P_2)t^2 + (3P_1 - 3P_0)t + P_0
\end{aligned}
$$

The polynomial coefficients are computed only when the geometric constraints change,

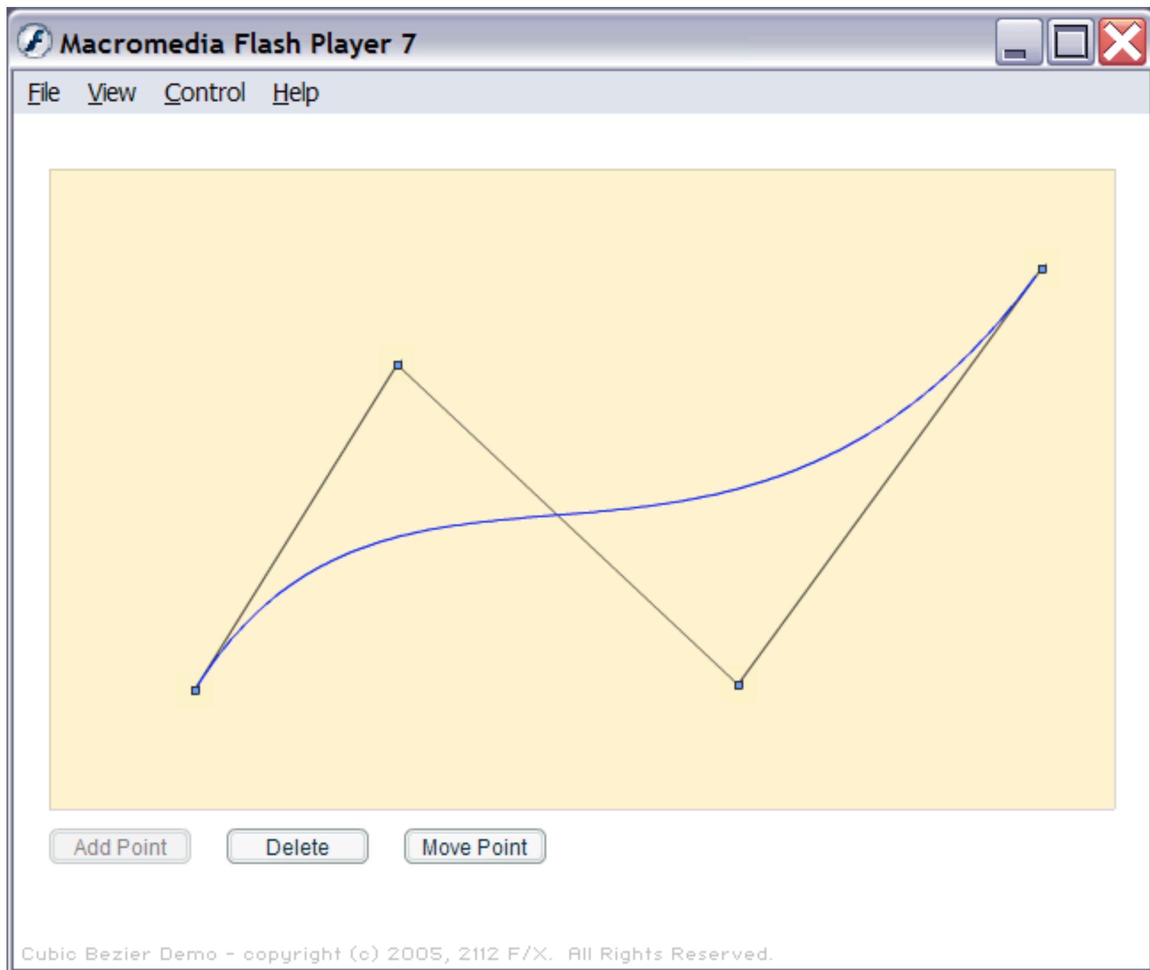
$$a = 3P_0$$
$$b = 3P_1$$
$$c = 3P_2$$
$$c_0 = P_0$$
$$c_1 = b - a$$
$$c_2 = a - 2b + c$$
$$c_3 = P_3 - P_0 + b - c$$

For any value of t, $B(t)$ is evaluated as

$$B(t) = c_0 + t(c_1 + t(c_2 + t(c_3))) \qquad [5]$$

Demo

This illustration is from a demo (Flash source no longer online) that allows four control points to be placed on stage. The cubic Bezier curve is displayed as shown in the following diagram.



By varying the placement of control points, a variety of interesting shapes can be produced. As with the quadratic example, the curve interpolates the first and last control points. Interim control points influence the shape of the curve. You should see that is much easier to create piecewise cubic curves with Beziers than Hermite curves. Eliminating the direct manipulation of tangent handles has some advantages ☺

The *Bezier3* class uses equation [5] to evaluate the cubic polynomial. Contrast this code to the structure in the much simpler *Bezier2* class.

Since the Bernstein polynomials sum to unity, the curve is entirely contained within the convex hull of the control cage.

Approximation

Given the means to draw quadratic and cubic Bezier curves and the fact that Flash comes with a built-in quadratic Bezier, it is only natural to ask how these methods might be used to approximate more complex curves. Detailed discussion of these issues is deferred to later TechNotes. A few examples are shown here to lay the groundwork for future papers.

<u>3-Point Interpolation with a quadratic Bezier</u>

A quadratic Bezier curve interpolates the endpoints of the control cage. The middle control point is used to influence the shape of the curve. As soon as library methods were available for drawing Bezier curves, many people wanted to use the quadratic curve to interpolate three points. It is often useful to ensure the quadratic curve passes through three points, while retaining the general aesthetic qualities of a Bezier curve.

In the case of Flash, such a method could be used to quickly draw a smooth curve through three points. This implies adjusting the control cage to ensure the Bezier curve passes through all three points. The method of approach dates back to the early 70's – this is a very old method! While having been previously coded in Actionscript, the code hides the fact that there is a **family of Bezier curves** passing through three points.

The curve is parameterized, so it is insufficient to specify that the curve must pass through three points. It is also necessary to specify at what value of t the curve passes through the specified second point. Different values of t produce different curve shapes. This is why the method was so popular. Users could not only specify interpolation points, they retained an element of shape control.

Suppose it is desired to fit a quadratic Bezier curve through the three points, (x_1, y_1), (x_1, y_2), and (x_3, y_3). Set

$$P_0 = [x_0 \quad y_0]^t$$

$$P_2 = [x_3 \quad y_3]^t$$

For a given parameter, t, determine the middle point of the control cage, P_1, such that

$$B(t) = [x_2 \quad y_2]^t = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2$$
$$\Rightarrow P_1 = \frac{B(t) - (1-t)^2 P_0 - t^2 P_2}{2t(1-t)} \qquad [6]$$

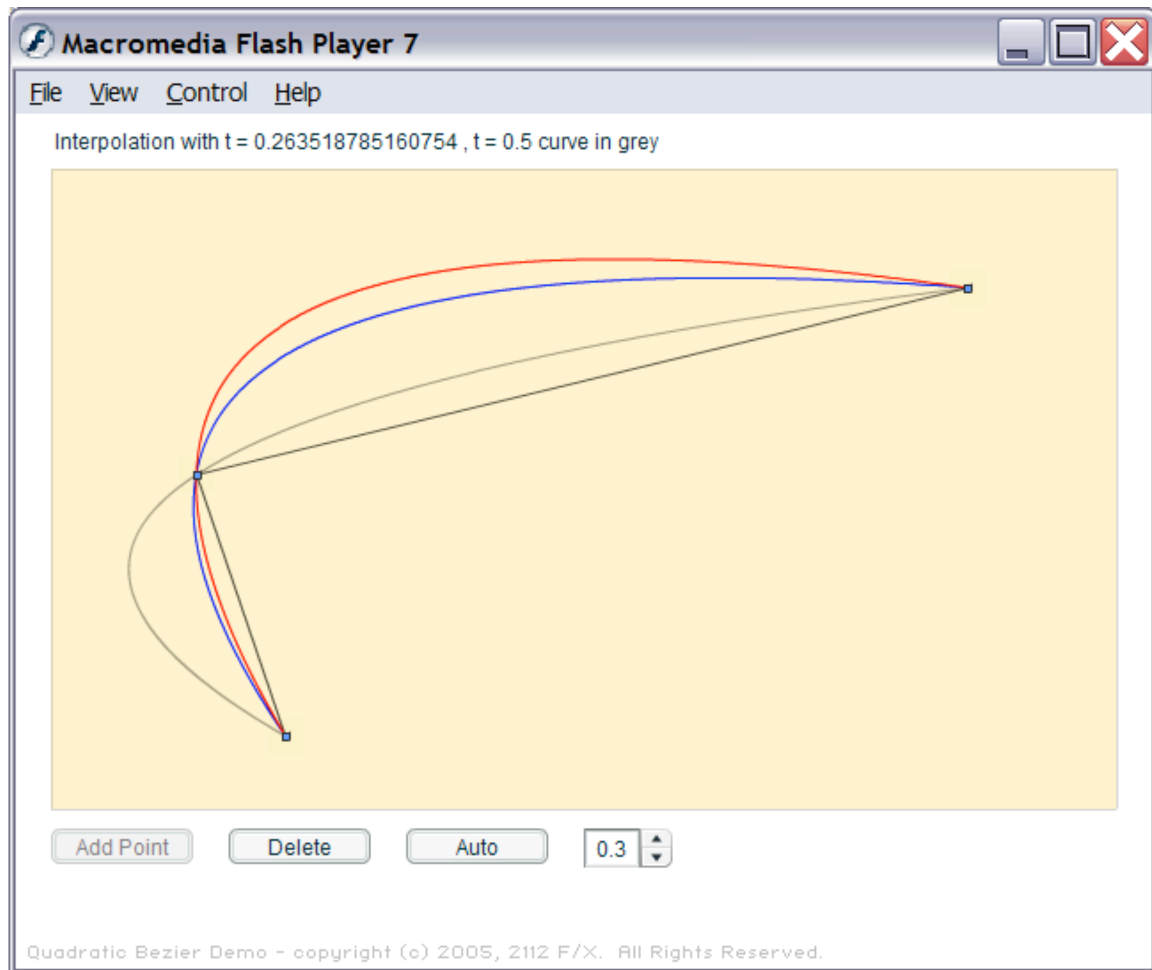
Equation [6] has a particularly simple representation at t = ½,

$$P_1 = \frac{B(0.5) - 0.5^2 P_0 - 0.5^2 P_2}{0.5} = 2B(0.5) - 0.5(P_0 + P_2) \qquad [7]$$

Equation [7] is often called the *midpoint* quadratic (interpolating) Bezier.

Note that equation [6] has poles at $t = 0$ and $t = 1$. The equation can only be applied in $(0,1)$. Since its use is also not recommended for values of t near 0 and 1, it is often applied in the interval $[a,b]$ where $a>0$ and $b<1$. Since the interpolating curve's shape is dependent on t, it is sometimes useful to have an automatic parameterization. One popular parameterization is based on the fraction of the first segment chord-length to the total chord length of the control cage.

The family of curves and chord-length parameterization is illustrated below.



The midpoint quadratic is plotted in light grey. Blue indicates the interpolating curve at $t = 0.3$. The red curve was generated by chord-length parameterization. By manually varying the parameter at the middle interpolating point, the user has the ability to interpolate all points, yet still influence the shape of the curve.

Normally, the user selects geometric constraints (the three control points) and has no specific control over what point the curve passes through at any t-value in the open interval (0,1). This interpolation formula allows the user to select a parameterization and infers the geometric constraints.

Quadratic Approximation of a Cubic

In general, using a single quadratic polynomial to approximate a single cubic is bad practice. There are some instances in which we might want to consider such an approach. One is when the cubic already has a parabolic shape or refinement could reduce the curve to a small number of piecewise cubics, each of which was approximately parabolic. In Flash, the benefit of such an approximation is the ability to directly use the *curveTo()* function for plotting.

Such usage could be helpful, for example, in adaptive degradation. Imagine an application with a number of piecewise cubic curves that allowed the viewer to interactively drag control points. It might be nice to interactively redraw the entire curve while the viewer dragged the control point. Updating and replotting the entire curve inside an onEnterFrame handler could be cumbersome.

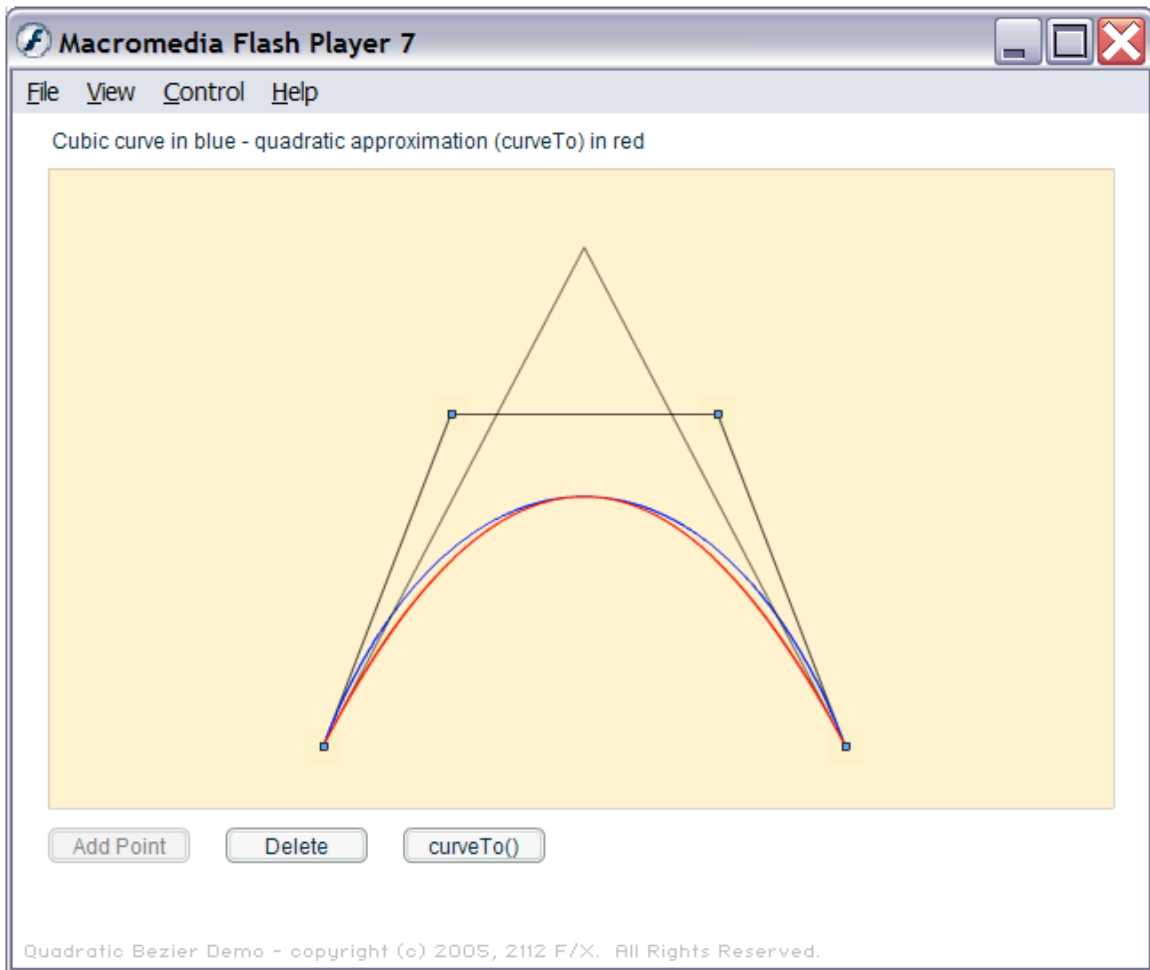
3D packages have the same problem when redrawing viewports containing lots of geometry and complex textures. It is common to adaptively degrade the view by simplifying the rendering of geometry so that viewport operations happen smoothly. Once the operation (zoom, pan, etc) is finished, the viewport is redrawn at normal quality.

In a similar manner, it might be useful to adaptively degrade to a piecewise quadratic approximation using *curveTo()* until an interactive redraw is finished. At that point, the full curve could be redrawn with proper accuracy. This approach would sustain reasonably high frame rates.

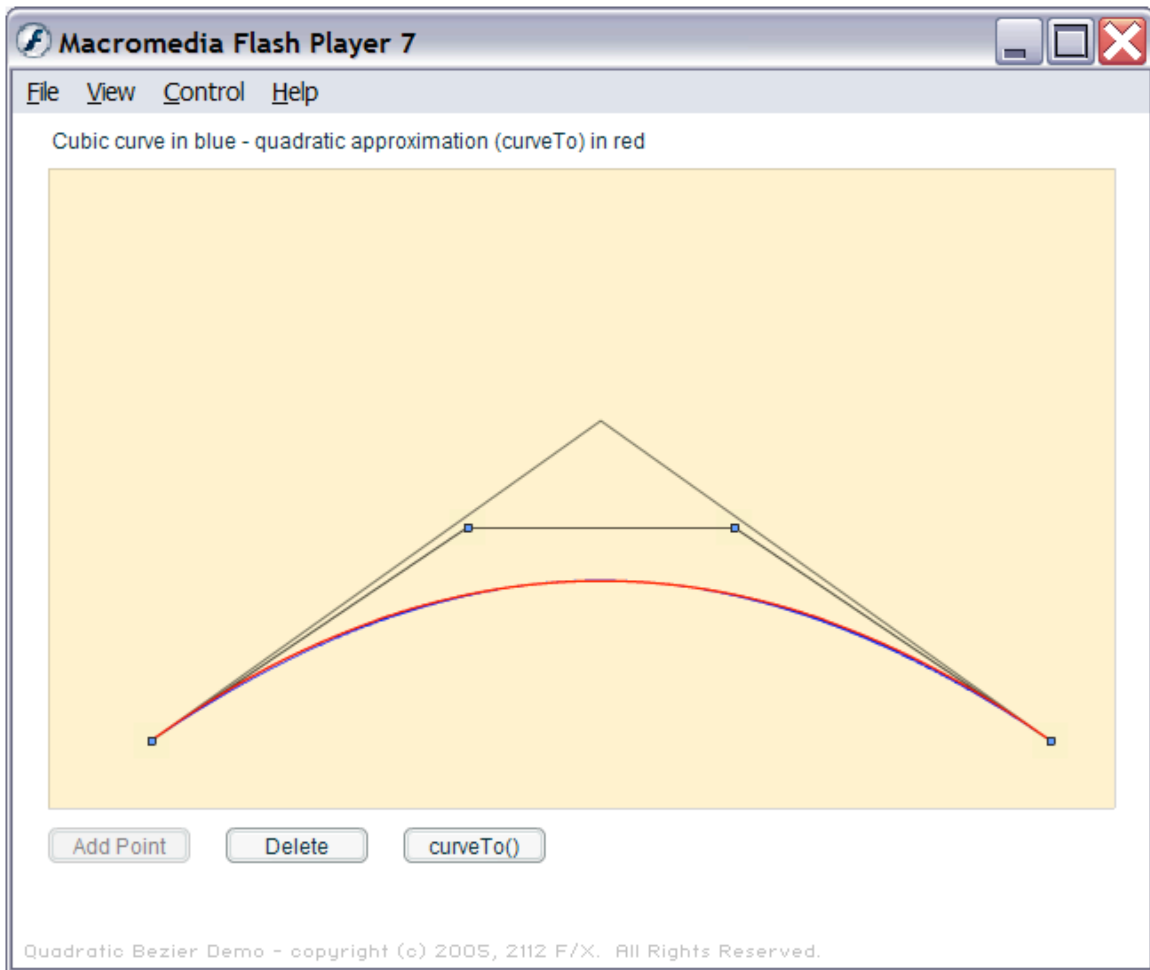
As a prelude to future TechNotes, consider the case of approximating a single cubic Bezier that already has a general parabolic shape with *curveTo()*. One such case is the symmetric control cage generated in the piecewise quadratic example from the prior TechNote.

If G^0 continuity is enforced at the endpoints, there are entire families of approximations. Since we have the ability to quickly construct the midpoint quadratic from the previous example, one approach would be to interpolate the endpoints of the cubic and the value of the cubic at t = 0.5.

This is illustrated below.



The approximation is pretty reasonable, especially given that the original cubic was already parabolic in shape. This method would not be recommended for piecewise cubic curves because of the 'humping' at join points. Notice, however, that for a single segment, the approximation is very good when the original cubic is much flatter, as shown below.



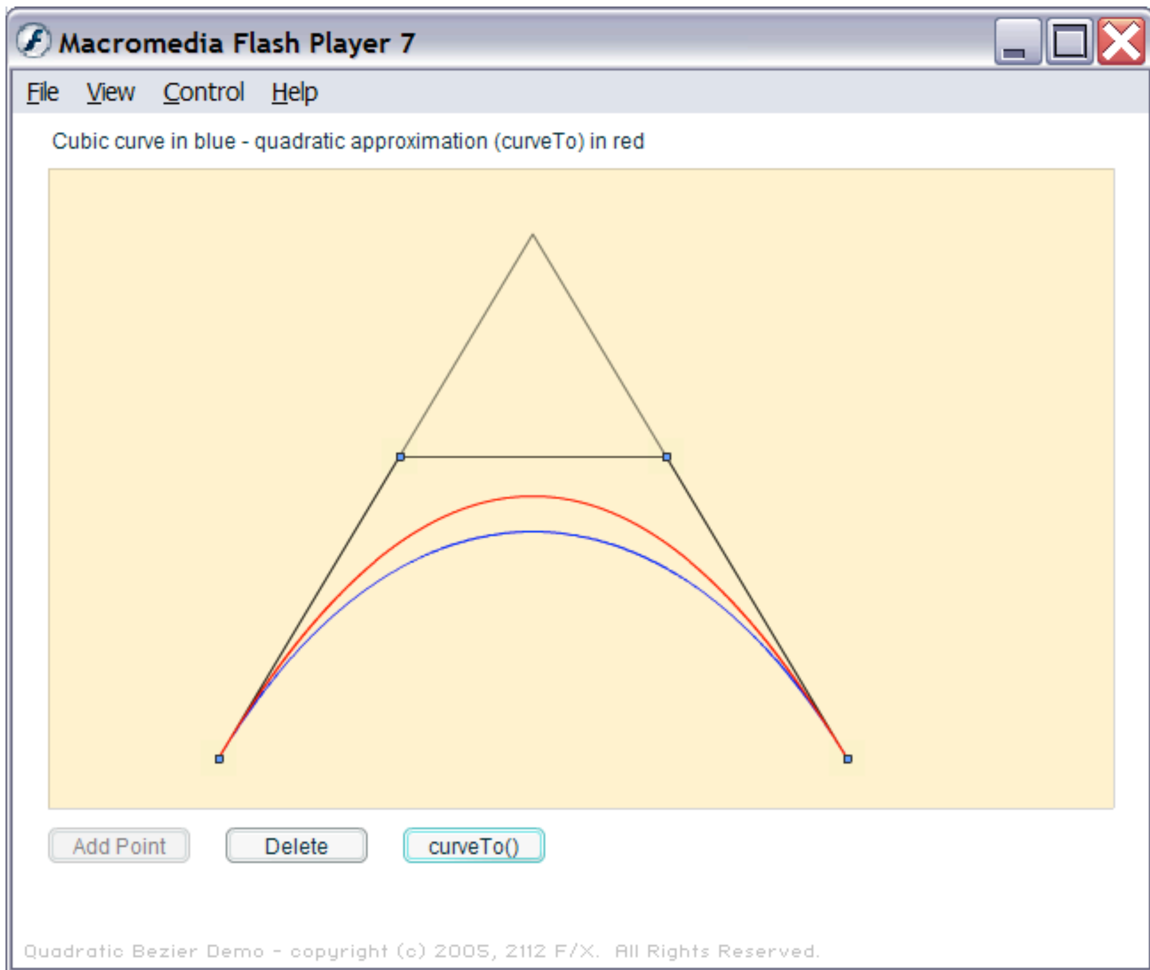
At this resolution and screen dimension, it is very difficult to tell the two curves apart.

Another method that forces G^1 continuity at the endpoints extends the P_0P_1 and P_2P_3 segments until they intersect. This is similar to the plotting approach discussed in the previous TechNote (refer to that diagram). Two lines are defined, one passing through P_0 with slope m_1 and the other passing through P_3 with slope m_2. The intersection equation is simpler with the line equations in slope-intercept form. Since the first line passes through P_0 and P_1 with known slope, its general equation is

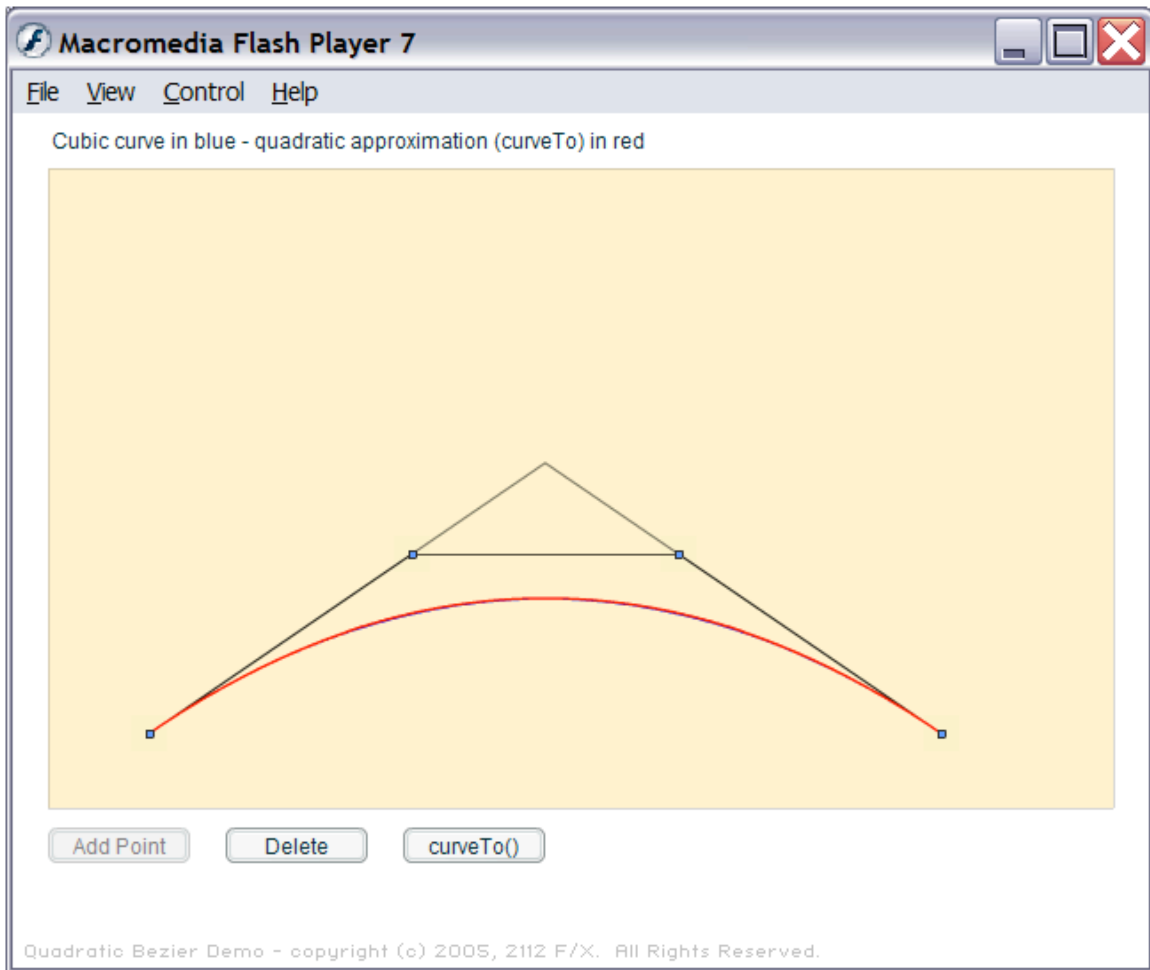
$$y - P_{0y} = m_1(x - P_{0x})$$

from which the y-intercept is $b_1 = P_{0y} - m_1P_{0x}$. Given two equations of the form $y = mx+b$, setting the two equations equal allows the x-coordinate to be solved for at the intersection. The resulting x-coordinate may be substituted into either of the original line equations to compute the y-coordinate at the intersection.

These computations are illustrated in the demo file *cubicapprox1.fla*, the results of which are shown below.



Once again, the approximation is not bad. While this method is useful for smooth curve transition at join points of a piecewise cubic, it generally 'overshoots' the original cubic near the midpoint. You should be able to see the tradeoff between the two approximations. Notice again that the quality of the approximation improves dramatically when the curve is 'flatter,' just as before.



While a quadratic is a poor approximation to a general cubic, for sufficiently small intervals (and thus, sufficiently flat curves), a piecewise quadratic is a suitable approximation to a cubic. Of course, this is just what we would expect as that is why point-to-point line drawing is a suitable method to plot any curve for small enough intervals ☺ . The approximation is a good method for fast drawing, although it would not be used to query the exact value of the cubic curve for any parameter value.

What we will see in the future is the relationship between the approximation and the curvature of the original curve

$$\kappa = \frac{d^2 y / dx^2}{\left[1 + (dy/dx)^2\right]^{3/2}}$$

As it happens, it is possible to refine a cubic Bezier curve into a piecewise cubic curve, with independent control cages for each cubic segment. This idea is exploited in a technique called recursive subdivision, resulting in a very fast method for plotting cubic Beziers. This and other topics such as approximating conic sections with Bezier curves will be discussed in future TechNotes.

References

1) Farin, G, “Curves and Surfaces for Computer Aided Geometric Design,” Academic Press, San Diego, 1993.

2) Bezier, P, “Numerical Control – Mathematics and Applications,” translated by Forrest, A.R. and Pankhurst, A. F., from “*Emploi des Machines a Commande Numerique*,” Wiley London, 1972.