

Recursive Subdivision

Jim Armstrong
Singularity
October 2006

This is the ninth installment 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.

Cubic Bezier Interpolation

Some animations require a smooth curve to be fit between a number of points. The fit is not for purposes of interpolation, but purely for visual representation. The knots are to be animated, so fast redraw of the curve segments is paramount. Performance is far more important than physical accuracy.

Such curve animations could be used to simulate cartoon drawings of fabrics, ocean waves, terrain, strings and ropes, or even strands of hair. It is not necessary to produce physically accurate simulations of these items in order to create a believable animation. It is just necessary to look good and draw fast.

Since cubic polynomials have inflection control, a cubic Bezier is a reasonable starting choice for each segment. That handles the 'looks good' part of the desired animation. The subject of this TechNote is the 'draws fast' part of the animation. The actual cubic interpolation is discussed in the next TechNote.

One Bezier Curve Into Many

A geometric construction technique for producing what are commonly referred to as Bezier curves was first published in a technical report by de Casteljau in 1959 [1].

A natural consequence of the de Casteljau geometric procedure is that it can be continued in a manner that divides the original curve into two separate cubic segments. Each subdivided cubic segment precisely duplicates one portion of the original curve.

The process can be continued to further subdivide a cubic curve into a larger number of equivalent cubic curves. As the subdivision continues, the cubic segments become increasingly flat as a consequence of the convex hull and variation-diminishing properties of Bezier curves. Originally, this method was used to draw cubic curves with a (hopefully minimal) number of line segments. Since the procedure is recursive, it can be combined with a flatness metric to produce a stopping criterion for the subdivision. At the end of the procedure, the cubic Bezier segment is approximated by a number of line segments. As line drawing is a guaranteed primitive in any plotting software, this process was widely used in the 1960s to draw cubic Bezier curves.

Starting in the 1970s, fast algorithms were devised to scanline render a quadratic Bezier and other smooth curves on a raster display ([2] – [6]). In some drawing environments, a quadratic Bezier is considered a primitive. For this reason, it is useful to consider a process that recursively subdivides a cubic Bezier curve into multiple segments that can be quadratically approximated. The Flash player provides fast display of quadratic Bezier curves through the *curveTo()* function of the drawing API, which is modeled after the PostScript equivalent [7]. The recursive

For the animation outlined in the introduction, the ‘looks good’ criterion is achieved by piecewise cubic Bezier interpolation. The ‘draws fast’ criterion is achieved by recursive subdivision.

Recursive Subdivision Algorithm

The classical deCasteljau geometric construction for a cubic Bezier curve is shown below. As it happens, the diagram was generated for $t = 0.4$, however, the algorithm is valid for any t in $[0, 1]$.

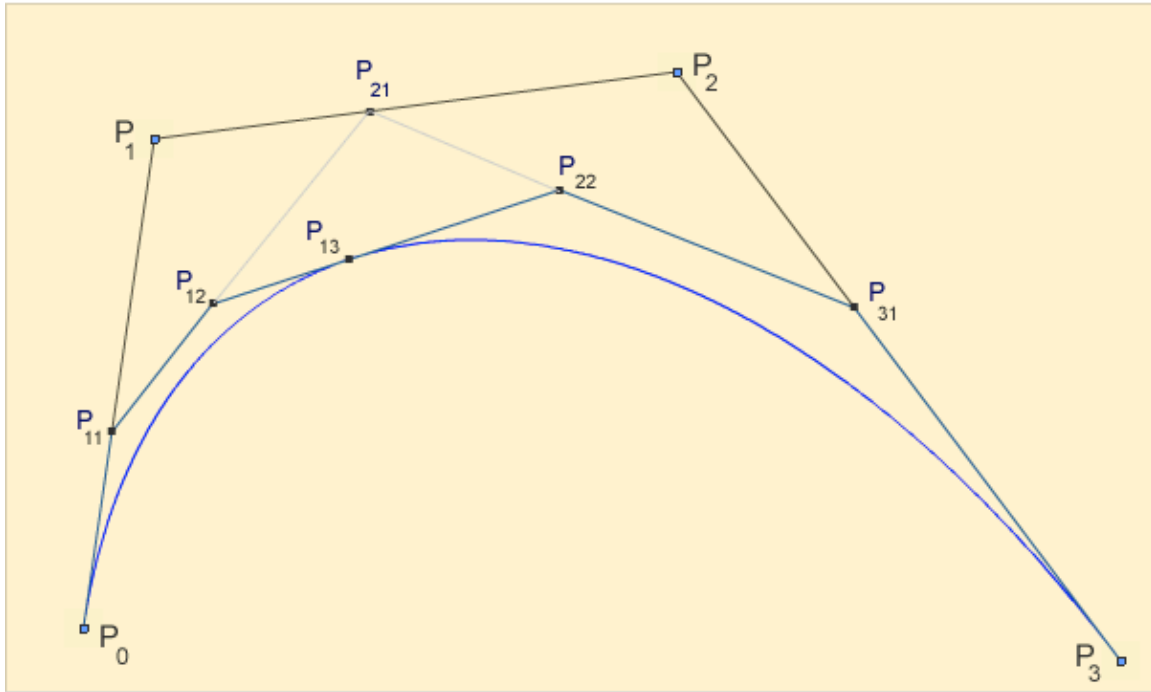


Diagram 1: Illustration of deCasteljau Algorithm

The line segment P_0P_1 is divided in the ratio $t : 1 - t$, at point P_{11} . The segments P_1P_2 and P_2P_3 are divided in the same ratio, producing the points P_{21} and P_{31} . The process is repeated until the point P_{13} is produced, which is the point on the Bezier curve corresponding to parameter value t and the control cage $P_0P_1P_2P_3$. The general formulas for the interim points in the construction are as follows.

$$\begin{aligned}
P_{11} &= (1-t)P_0 + tP_1 \\
P_{21} &= (1-t)P_1 + tP_2 \\
P_{31} &= (1-t)P_2 + tP_3 \\
P_{12} &= (1-t)P_{11} + tP_{21} \\
P_{22} &= (1-t)P_{21} + tP_{31} \\
P_{13} &= (1-t)P_{12} + tP_{22}
\end{aligned}$$

which expanded yields

$$\begin{aligned}
P_{12} &= (1-t)((1-t)P_0 + tP_1) + t((1-t)P_1 + tP_2) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \\
P_{22} &= (1-t)((1-t)P_1 + tP_2) + (1-t)((1-t)P_2 + tP_3) = (1-t)^2 P_1 + 2t(1-t)P_2 + t^2 P_3 \\
P_{13} &= (1-t)[(1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2] + t[(1-t)^2 P_1 + 2t(1-t)P_2 + t^2 P_3] \\
&= (1-t)^3 P_0 + 2t(1-t)^2 P_1 + (1-t)t^2 P_2 + t(1-t)^2 P_1 + 2t^2(1-t)P_2 + t^3 P_3 \\
&= (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3
\end{aligned}$$

This process results in the familiar formula for the point on the cubic Bezier curve at the specified value of t . The third-order Bernstein polynomials in the final formula were noted in a prior TechNote. Notice the second-order Bernstein polynomials that appear in the interim steps.

In the process of generating the point on the cubic curve, a new control cage is created, $P_0 P_{11} P_{12} P_{13}$. Parameterized on u in $[0,1]$, $u = 0$ corresponds to $t = 0$ and $u = 1$ corresponds to $t = t^*$, the desired point on the Bezier curve. The original and inner control cages have G^1 continuity at the endpoints. The subdivision matrix and transformation for the 'left' control cage is given by

$$\begin{bmatrix} P_0 \\ P_{11} \\ P_{12} \\ P_{13} \end{bmatrix} = D_t \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ (1-t) & t & 0 & 0 \\ (1-t)^2 & 2t(1-t) & t^2 & 0 \\ (1-t)^3 & 3t(1-t)^2 & 3t^2(1-t) & t^3 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \quad [1]$$

In a previous TechNote, it was noted that the sum of Bernstein polynomials is unity, thus the 'left' control cage is contained entirely inside the convex hull of the original control cage. At $t = 0.5$, both the left- and right-division matrices have a very compact form.

$$\begin{bmatrix} P_0 \\ P_{11} \\ P_{12} \\ P_{13} \end{bmatrix} = D_t(0.5) \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = 1/8 \begin{bmatrix} 8 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 \\ 2 & 4 & 2 & 0 \\ 1 & 3 & 3 & 1 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

and

$$\begin{bmatrix} P_{13} \\ P_{22} \\ P_{31} \\ P_3 \end{bmatrix} = D_r(0.5) \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = 1/8 \begin{bmatrix} 1 & 3 & 3 & 1 \\ 0 & 2 & 4 & 2 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Any point on the cubic Bezier curve generated using the left, inner control cage can be related to a point on the original Bezier curve using the basis matrix for Bezier curves and the relation $t = ut^*$,

$$B_l(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}^t \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ (1-ut^*) & ut^* & 0 & 0 \\ (1-ut^*)^2 & 2ut^*(1-ut^*) & ut^{*2} & 0 \\ (1-ut^*)^3 & 3(ut^*)(1-ut^*)^2 & 3(ut^*)^2(1-ut^*) & (ut^*)^3 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

If you are inclined to work through the substantial amount of algebra and simplification (which is far too long for this TechNote), it can be shown that $B_l(u) = B(ut^*)$, indicating that the 'left' Bezier segment with control cage $P_0P_{11}P_{12}P_{13}$ exactly duplicates the original Bezier curve for t in $[0, t^*]$. A similar derivation exists for the 'right' segment, so the original Bezier curve is exactly split into two smaller, but equivalent cubic curves at $t = t^*$. The process can be continued, dividing the 'left' Bezier segment at some parameter in $(0, t^*)$ and dividing the 'right' Bezier segment at some parameter in $(t^*, 1)$.

As new Bezier segments are recursively created, the smaller control cages quickly become convex. Because of the convex hull property, each subdivision control cage is contained entirely within the convex hull of its (convex) parent.

The variation-diminishing property of Bezier curves informally states that the Bezier curve 'wiggles' no more than its control cage. One way of stating the property is that if a straight line intersects the curve at c points and the control cage at p points, then there exists a positive integer, j , such that $c = p - 2j$. This is illustrated in the following diagram.

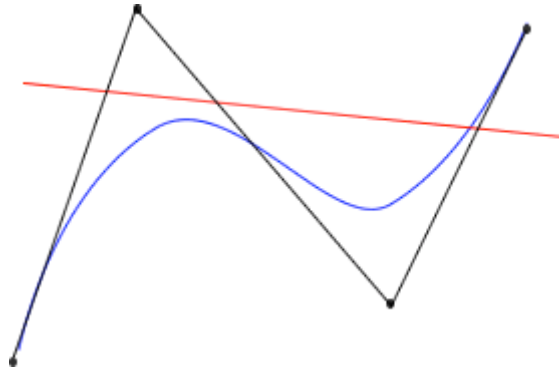


Diagram 2: Variation-diminishing property. $c=1, p=3, j=2$.

As the subdivision continues, the convex control cages begin to flatten. Thus, the cubic segments begin to flatten and are eventually well-approximated by polynomials of degree one or two. Quadratic approximation is desired for purposes of the animation specified at the beginning of this TechNote. For purposes of using the quadratic approximation in interpolation,

G^1 continuity at endpoints is desirable. This is accomplished by finding the intersection point of the initial and final line segments of the cubic control cage, as illustrated in the following diagram from the cubic Bezier TechNote.

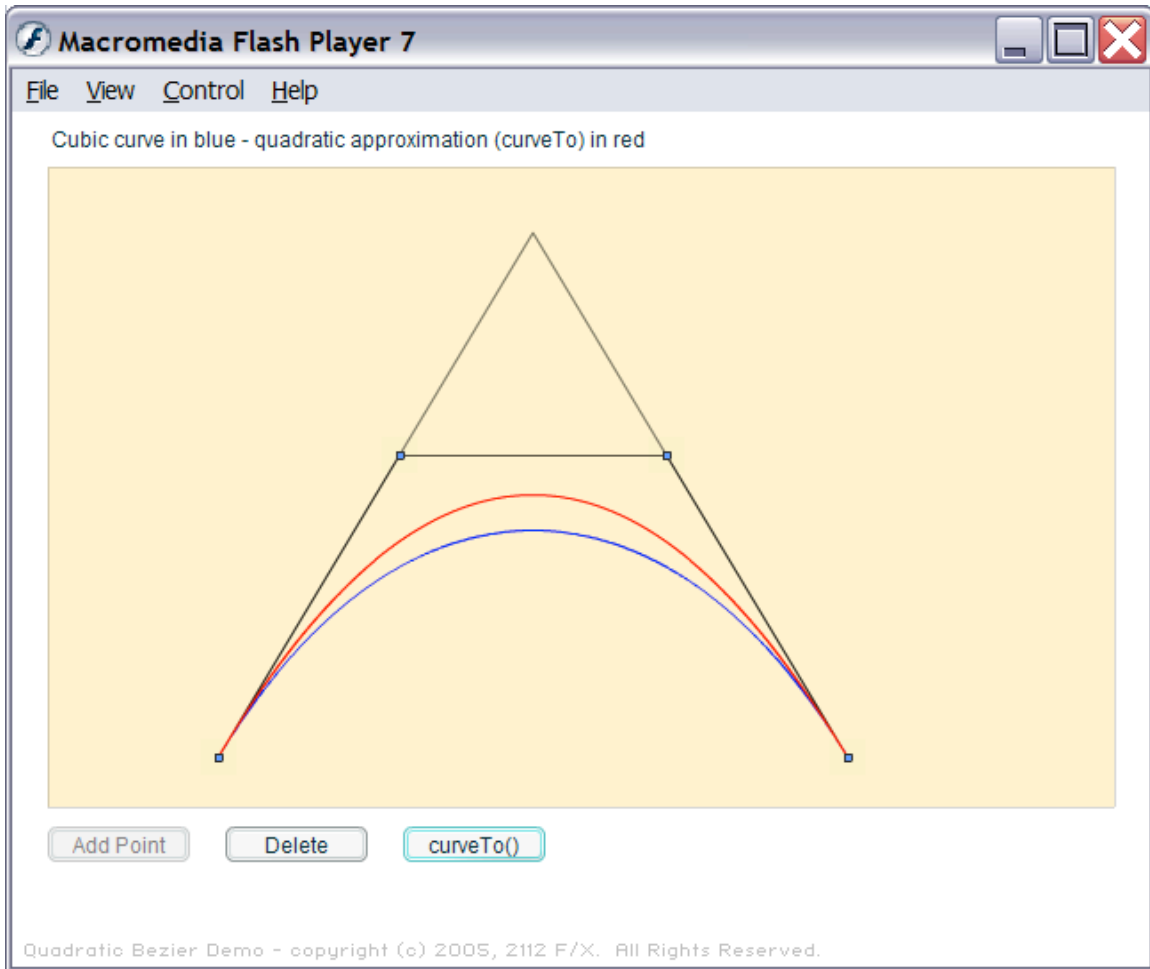


Diagram 3: Quadratic approximation (in red) to a cubic Bezier segment (in blue)

Since the desired use of this process is for piecewise cubic interpolation, it is expected that none of the cubic segments are self-intersecting and there are no cusps. Thus, the cubic control cages should become convex after the first subdivision and there should be an intersection point. This allows quick construction of a quadratic control cage for the approximation.

The next issue is naturally when to stop the subdivision process; i.e. when is the quadratic approximation 'close enough'?

Stopping Criterion

Deciding when to terminate subdivision is naturally subjective. Most termination procedures for recursive subdivision combine a flatness metric with a user-specified tolerance. For the

application discussed in this TechNote, the metric is one of 'closeness' of a cubic and quadratic Bezier curve. The choice of 'closeness' metric is a tradeoff between computational ease and accuracy.

One popular metric is the so-called 'midpoint delta,' i.e. the distance between the quadratic and cubic Bezier segments at $t = 0.5$. Suppose B_c represents a cubic Bezier curve and B_q represents the quadratic approximation to that curve. Let P denote the control point for the quadratic Bezier segment. For an arbitrary cubic control cage $P_0P_1P_2P_3$, the difference between the two points at the same parameter value $t = 0.5$ is

$$\begin{aligned} B_q(0.5) &= P_0/4 + P/8 + P_3/4 = (P_0 + 2P + P_3)/4 = (2P_0 + 4P + 2P_3)/8 \\ B_c(0.5) &= P_0/8 + 3P_1/8 + 3P_2/8 + P_3/8 = (P_0 + 3(P_1 + P_2) + P_3)/8 \\ \Rightarrow B_q(0.5) - B_c(0.5) &= (P_0 + 4P - 3(P_1 + P_2) + P_3)/8 \end{aligned}$$

which is very easy to compute. There are some issues with the accuracy and consistency of this metric. Consider the following diagram, illustrating a midpoint subdivision ($t = 0.5$) of a cubic Bezier curve. The midpoints of the subdivided cubic segments and their quadratic approximations are marked.

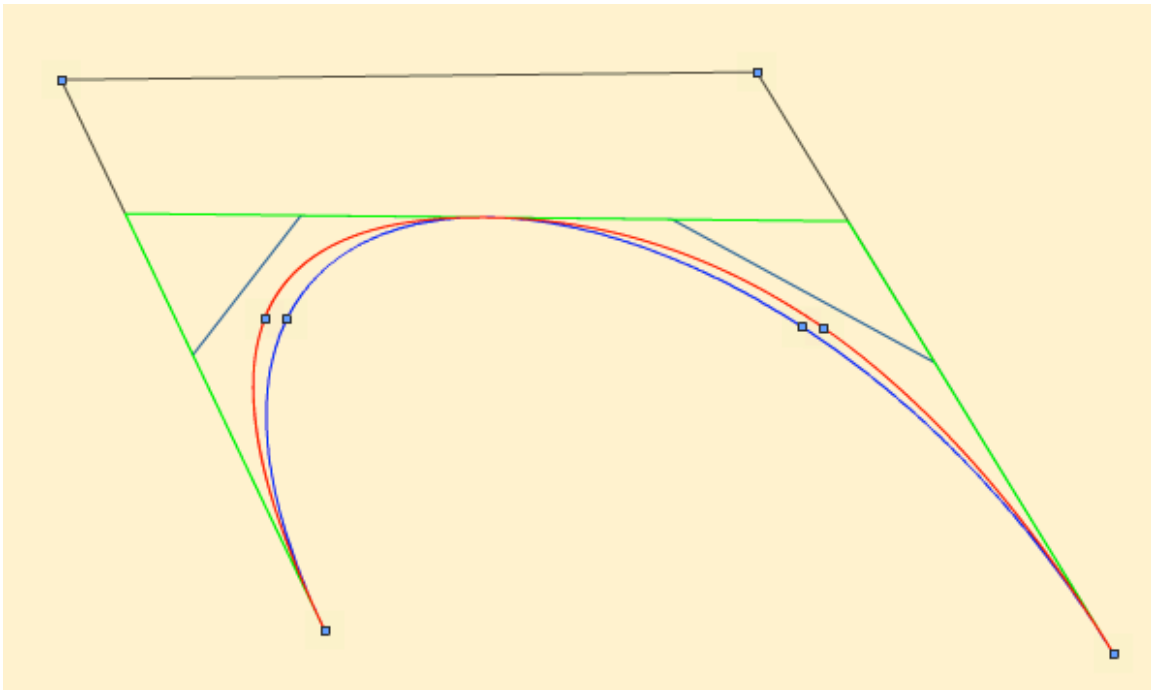


Diagram 4: 'Midpoint' of subdivided cubic Bezier segments and their quadratic approximations

Notice that the distance between the two sets of markers is almost exactly the same for both the left and right subdivisions. If an analyst requested a tolerance requiring the left segment to be further subdivided, the right segment would also be subdivided, even though the same analyst might decide it was suitably close to the cubic segment.

The midpoint split does not necessarily occur at the halfway point along the length of the curve. Since neither the cubic or quadratic segments are naturally arc-length parameterized, there is no reason to expect the midpoint of either segment to be comparable. Even if the two curves were identical, a sprite moving along both curves would not necessarily move at the same velocity.

This does not prohibit the midpoint stopping criterion from being used. It is generally better to use a generous tolerance to avoid splitting the curve more times than necessary. To avoid excessive splitting due to too tight a tolerance, a counter may be employed. For most control cages encountered in interpolation, a very nice fit is obtained after two full passes through the subdivision process. In most cases, three full passes produces a sequence of quadratic segments that if plotted with the same color would be practically indistinguishable from the original cubic curve.

There are many other stopping criteria than may be employed, several of which are based on the shape (relative flatness) of the control cages. Since performance is paramount, I use an entirely different technique, which is discussed in a later section.

Successive Midpoint Example

In this approach, the tolerance is based on the square of the distance between the subdivided cubic and its quadratic approximation at $t = 0.5$.

We happen to know something about the display resolution for most Flash applications. It is also desirable to use a looser tolerance than most users might normally choose. For these reasons, it is better to expose a 'goodness of fit' variable to the user rather than the square of a distance measurement, especially one for which the average user might not have good intuition. Instead of choosing an actual pixel measurement, the user assigns a fitness number in the 1-10 range. A value of 1 implies a very tight fit and a value of 10 implies a very loose fit. Fitness values are used to index into an array of squared pixel distances, representing the allowable squared distance between quadratic and cubic 'midpoints.' In the spirit of providing areas of code that you should modify as part of the learning process, I will point out that there is room for improvement in the currently assigned values, especially at the lower end of the scale.

The subdivision algorithm could be programmed with recursive method calls in Flash, making for a very elegant implementation. In order to easily control the process and provide a bit better correlation between the algorithm and its coded implementation, recursion was not used in the Actionscript code. A counter is placed on the outermost subdivision loop, serving as a control over the total number of subdivision sweeps. Currently, no more than two full sweeps (four quadratic segments) are allowed. You may experiment with the fitness metric and counter to see how the algorithm behaves.

The *Bezier3* class was rewritten to use a slightly faster method for computing cubic Bezier coefficients, which is the same algorithm used inside PostScript. A public *subdivide()* method is provided to support manual subdivision. Calls to this method may be used to see how a cubic segment is split at an arbitrary parameter value.

Instead of using point-to-point plotting with calls to the *getX()* and *getY()* methods, a public *draw()* method is provided to draw the cubic Bezier curve. The *draw()* method accepts a reference to a MovieClip in which the cubic Bezier curve is drawn. If the original cubic has been manually subdivided, those subdivisions are used in the drawing. Otherwise, the *draw()* method implements the successive midpoint subdivision, stopping the process after either the tolerance has been met for all segments or a few complete passes through the algorithm.

An example subdivision is illustrated in the following diagram.

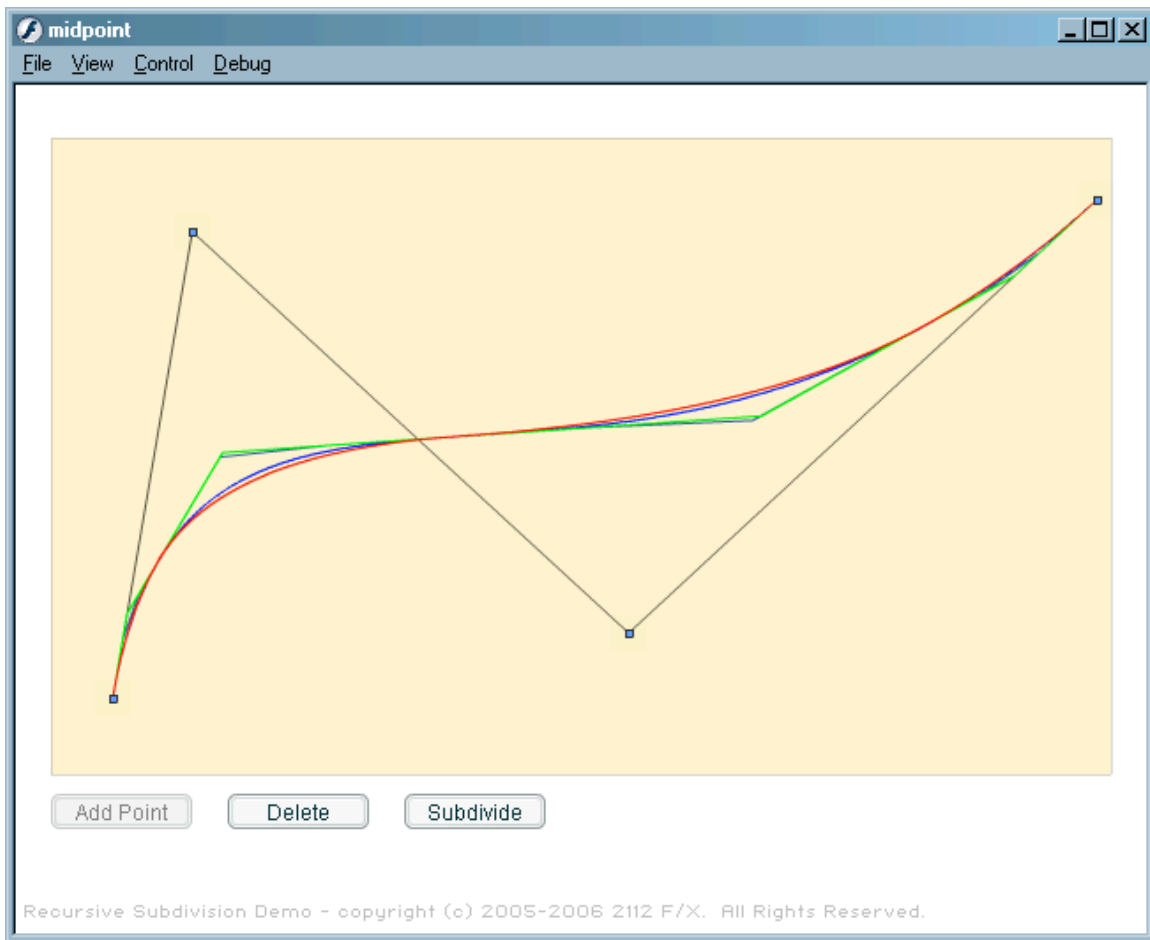


Diagram 5: Example of successive midpoint subdivision

The subdivided cubic control cage for each segment is drawn in a light blue. The control cage for the approximating quadratic is drawn in green. The original cubic curve is drawn in blue. The quadratic approximation is in red.

This code is to be used to illustrate the interim computations in the subdivision process. It **is not** implemented with performance in mind.

Fast Cubic Bezier Plotting

Recall from the introduction that the goal of this effort is to produce piecewise cubic curves that look good and draw fast. In Matrix terms, it really does not matter if you take the red pill or the blue pill (i.e. red curve or blue curve) in any of the above examples. When knots are animated at 20fps, for example, the eye simply can not tell the difference after a single 'good' subdivision. Over a decade ago (in my SGI days), I started experimenting with different methods for producing a reasonable-quality single subdivision.

For many cases, it seems like the $t = 0.5$ subdivision produces a pretty good 'one shot' subdivision. There are (intuitively) a couple areas where we might choose to subdivide at another parameter value. Since quadratics can not represent changes in inflection, dividing precisely at

an inflection point is one possibility. Points of very high curvature, particularly if the curve is 'biased' more towards the left or right portion of the control cage are another area of interest.

The idea behind the fast-draw cubic Bezier is to expend a modest amount of computation to produce a single subdivision. The cubic Bezier is drawn with two quadratic segments. While this approach is not suitable for precise static drawing, it is a reasonable compromise between quality and speed when drawing piecewise segments many times a second. Remember that we are not attempting to deal with original cubic curves that are self-intersecting or other problematic cases. We just want to draw fast!

Curvature

Of the two above-mentioned subdivision criteria, curvature is problematic. The standard definition of curvature of a function $y = f(x)$ from differential geometry is

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

In parametric terms, some authors define curvature as

$$\kappa = \frac{\| P'(t) \times P''(t) \|}{\| P'(t) \|^3}$$

No matter how you slice it, seeking a point of maximum curvature involves setting the derivative of the curvature function to zero. A second differentiation is required to confirm a maximum. Solving for zeros of the derivative is a very complex root-finding operation. In the end, the results are hardly worth the computational effort. In fact, if the control cage is anywhere close to symmetric, the parameter value representing maximum curvature is not very far from the midpoint anyway. This is simply not a fruitful subdivision algorithm in practice.

If you wish to pursue this idea, one heuristic in which I've realized some success is to divide the cage based on the intersection of the perpendicular bisector of $P_1 P_2$ and the cubic curve, as illustrated in the following diagram.

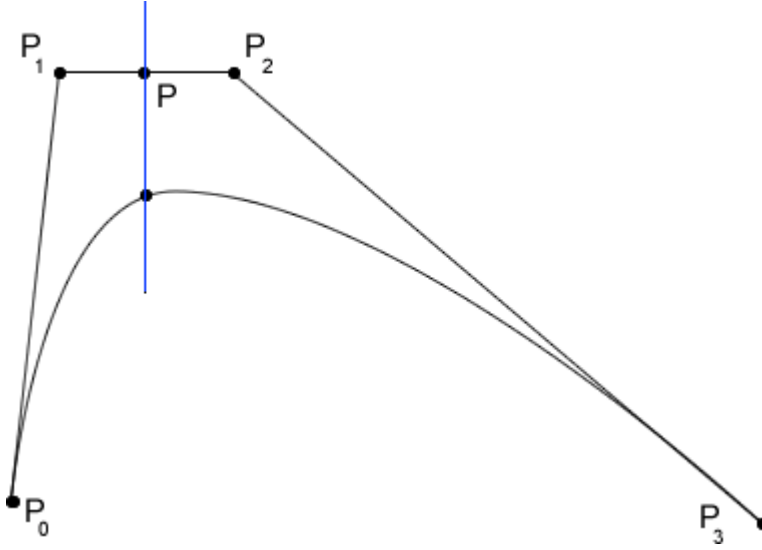


Diagram 6: Intersection between cubic curve and perpendicular bisector of middle control cage segment

The point, P is easily computed as $P = (P_1 + P_2)/2$. The slope of the perpendicular bisector is $m = -(P_{2x} - P_{1x})/(P_{2y} - P_{1y})$. Given a point and a slope, the computation involves finding the parameter corresponding to the intersection of a line and a curve. For convex control cages in which the point of maximum curvature is much closer to one side of the control cage, this algorithm can produce a tighter fit than a simple midpoint subdivision.

This is a pretty simplistic method. Should you wish to study more sophisticated techniques that avoid the optimization process, references [12], [13], and [14] are a good start.

For purposes of very fast animation, I tend use midpoint subdivision unless there is an inflection point on the curve.

Inflection Points

Mathematically, an inflection point corresponds to a point on the curve where the second derivative changes sign, i.e. $f''(x) = 0$ and $\text{sign}(f''(x + \epsilon)) = -\text{sign}(f''(x - \epsilon))$ in a small neighborhood around x . It is the point where the curve transitions from concave upward to concave downward (or vice versa).

Another definition of inflection comes from a physics interpretation. Consider a sprite moving along the curve. The sprite's acceleration vector has a component along the tangent to the curve and a component normal to the tangent. At an inflection point, the acceleration component normal to velocity is zero, meaning that the cross product of the two vectors is zero.

Consider any parametric cubic curve,

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y \end{aligned} \quad [2]$$

At an inflection point,

$$x' \cdot y'' - x'' \cdot y' = 0$$

$$\Rightarrow (3a_x t^2 + 2b_x t + c_x)(6a_y t + 2b_y) - (6a_x t + 2b_x)(3a_y t^2 + 2b_y + c_y) = 0$$

or

$$6(a_y b_x - a_x b_y)t^2 + 6(a_y c_x - a_x c_y)t + 2(b_y c_x - b_x c_y) = 0 \quad [3]$$

This equation is quadratic in t , so there are theoretically two possible inflection points. In practice (taking into account the piecewise segments often encountered in interpolation), there is usually a single inflection point if one exists at all.

To solve equation [3], first normalize the t^2 coefficient to yield

$$\begin{aligned} d &= a_y b_x - a_x b_y \\ \Rightarrow t^2 + d^{-1}(a_y c_x - a_x c_y) + d^{-1}(b_y c_x - b_x c_y)/3 &= 0 \end{aligned} \quad [4]$$

which is of the form

$$At^2 + Bt + C = 0 \quad [5]$$

The solution to [5] is given by the venerable quadratic formula,

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

The reason for normalizing the t^2 coefficient is that the quadratic formula can be simplified when $A = 1$ to

$$t = -(B/2) \pm \sqrt{(B/2)^2 - C} \quad [6]$$

Substituting the equations for the coefficients in [5] into [6] yields,

$$\begin{aligned} t_c &= -d^{-1}(a_y c_x - a_x c_y)/2 \\ t &= t_c \pm \sqrt{t_c^2 - d^{-1}(b_y c_x - b_x c_y)/3} \end{aligned} \quad [7]$$

If no real roots exist to [7] or any root is outside $(0,1)$, there are no inflection points for the cubic curve. If there happen to be two inflection points in $(0,1)$, the one farthest from zero or one is selected.

Given a control cage, $P_0 P_1 P_2 P_3$, the coefficients in eq. [2] for a cubic Bezier curve are

$$a_x = -x_0 + 3(x_1 - x_2) + x_3$$

$$a_y = -y_0 + 3(y_1 - y_2) + y_3$$

$$b_x = 3(x_0 - 2x_1 + x_2)$$

$$b_y = 3(y_0 - 2y_1 + y_2)$$

$$c_x = 3(x_1 - x_0)$$

$$c_y = 3(y_1 - y_0)$$

$$d_x = x_0$$

$$d_y = y_0$$

These computations are implemented inside the *FastBezier* class, with the appropriate test for a negative radical. If an inflection point is found, it is used for the single subdivision. Otherwise, the code uses a midpoint division. This approach is suitable for fast animation, but not static drawing.

Flatness Metrics

Although Flash provides a fast, built-in quadratic Bezier curve in the player, there are times where we might want to consider a piecewise linear approximation to a Bezier curve. Many years ago, I worked on a game prototype where the player shot circular 'photon bombs' at 'globs.' The globs were organic shapes with simple fills. Glob outlines were created with piecewise cubic Bezier curves. They were constructed by randomly modifying one of several pre-defined point templates. As soon as a glob was constructed, recursive subdivision was used to create an approximate piecewise linear representation of the glob outline. The piecewise linear outline was used in collision computations. Some globs were only partially destroyed on a hit, so some of the knots were deleted and the piecewise Bezier was reconstructed. Every time a glob was originally created or partially destroyed, a new linearization was performed to approximate the outline.

There are a number of metrics that can be employed to test for linearity of a subdivided segment. I have found the following to be particularly elegant and rather easy to compute. It is also very robust. Multiple private communications have led me to believe the original process and bound are due to Rodger Willcocks (rops.org).

Consider the typical cubic control cage and a line from P_0P_3 to the farthest point on the curve, as illustrated in the following diagram.

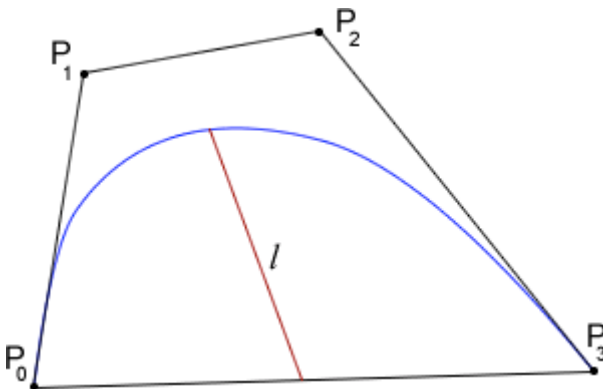


Diagram 7: Flatness measurement of a cubic curve

A bound on $\|l\|_2$ provides a measurement of the flatness of the cubic curve.

The line segment P_0P_3 can be traced as a function of t using either a quadratic or cubic Bezier curve. Let $g(t)$ represent the trace of the P_0P_3 line segment. For a quadratic Bezier, use a cage with endpoints P_0 and P_3 with a control point $(P_0 + P_3)/2$. To see that the quadratic Bezier curve traces the line segment, note that

$$\begin{aligned} g(t) &= t^2 P_0 + 2t(1-t)(P_0 + P_3)/2 + (1-t)^2 P_3 = t^2 P_0 + t(P_0 + P_3) - t^2(P_0 + P_3) + (1-t)^2 P_3 \\ \Rightarrow g(t) &= tP_3 + tP_0 - t^2 P_3 + (1-2t+t^2)P_3 = tP_0 + P_3 + tP_3 - 2tP_3 = tP_0 + (1-t)P_3 \end{aligned}$$

which is exactly the parametric equation of the line segment P_0P_3 .

In a similar manner, $g(t)$ can be represented as a cubic Bezier curve with endpoints P_0 and P_3 and control points $(2P_0 + P_3)/3$ and $(P_0 + 2P_3)/3$. If the cubic Bezier curve is $B(t)$, then

$$B(t) - g(t) = (1-t)^2 t(3P_1 - 2P_0 - P_3) + (1-t)t^2(3P_2 - P_0 - 2P_3) = t(1-t)((1-t)U + tV)$$

where

$$U = 3P_1 - 2P_0 - P_3$$

$$V = 3P_2 - P_0 - 2P_3$$

from which

$$\|B(t) - g(t)\|_2^2 = d^2(t) = (1-t)^2 t^2 [((1-t)U_x + tV_x)^2 + ((1-t)U_y + tV_y)^2] \quad [8]$$

A bound for the squared distance consists of two parts; bounding $(1-t)^2 t^2$ for t in $[0,1]$ and bounding $|(1-t)\alpha + \beta t|$ for arbitrary constants α and β and t in $[0,1]$.

For t in $[0,1]$, $\max((1-t)^2 t^2) = 1/16$

Also for t in $[0,1]$, $\max(|(1-t)\alpha + \beta t|) = \max(\alpha, \beta)$ for any constants α and β .

These observations may be used in [8] to obtain a bound for t in $[0,1]$,

$$d^2(t) \leq (\max(U_x^2, V_x^2) + \max(U_y^2, V_y^2))/16 \quad [9]$$

Given a pixel distance, ε , representing a flatness bound, compute $\tau = 16\varepsilon^2$. Let this value be represented by the variable **tol**. The following pseudocode shows how the bound in [9] is applied.

```
var ux = 3*p1X - 2*p0X - p3X
var uy = 3*p1Y - 2*p0Y - p3Y
```

```

var vx = 3*p2X - p0X - 2*p3X
var vy = 3*p2Y - p0Y - 2*p3Y

ux *= ux; uy *= uy; vx *= vx; vy *= vy

ux = max(ux,vx); uy = max(uy,vy)

return( ux+uy <= tol )

```

As with the quadratic case, several other metrics exist, many of which are based on the shape of the control cage.

To answer the next logical question, it's 'no.' A similar flatness bound exists for the quadratic approximation to a subdivided cubic segment. There is no reason to believe that the difference between flatness bounds represents a useful measure of difference between the cubic curve and its quadratic approximation. You would be better off trying to bound the actual difference between the two curves (hint hint).

Summary

Recursive subdivision is a technique whose practical history goes back many decades. I hope this informal TechNote has provided both a historical overview as well as a sampling of algorithms used in the procedure. The TechNote was much longer than I prefer and I did not even have a chance to discuss blossoming.

As with so many techniques in computational geometry, this one is best learned by looking at code along with the math, so I hope the accompanying Actionscript examples lead to lots of interesting experimentation for you. For the sake of your social life, avoid too much experimentation or risk ending up a hopeless geek like the author ☺

The algorithm behind the intersection code is not documented here. There are some issues regarding extreme cases where one side of the control cage is nearly (or exactly) horizontal or vertical. Try to determine how you would handle these cases before looking at the code. The algorithm is deconstructed in the next TechNote on composite Bezier curves.

Some additional work is required to more effectively handle self-intersecting control cages. This issue was ignored, as this type of control cage is not anticipated in the intended animation. Once you fully understand how the basic algorithm works, extreme cases should not present too much of a challenge.

References

- [1] P. de Casteljau. *Outillages methodes calcul*. Technical report, A. Citroen, Paris, 1959.
- [2] Knuth, Donald E., "The Art of Computer Programming," Addison-Wesley Publishing Co., 1969, pp. 234-238.
- [3] Hersh, R.D. "Digital Typography and Raster Imaging: The Desk Top of the 90's," Eurographics '91 Tutorial Note 11, vol. EG91, No. TN11, (1991), pp. 1-23.

- [4] Kawata, T., et al, "An Outline Font Rendering Processor with an Embedded RISC CPU for High-Speed Hint Processing", IEEE Journal of Solid-State Circuits, vol. 29, No. 3, (1994), pp. 280-289.
- [5] Piegl, L. and Tiller, W., "Software-Engineering Approach to Degree Elevation of B-Spline Curves", Computer Aided Design, vol. 26, No. 1, (1994), pp. 17-28.
- [6] Whitted, T. "A scan line algorithm for computer display of curved surfaces", Proceedings of the 5th annual conference on Computer graphics and interactive techniques, 1978.
- [7] Adobe Systems, Incorporated, PostScript Language Reference, 3rd ed., Addison-Wesley, Reading MA., 1999.
- [8] Foley, Van Dam, Feiner, Hughes, "Computer Graphics: Principles and Practice", 2nd ed., Addison Wesley, Reading, MA. 1996.
- [9] Bartels, Beatty, and Barsky, An Introduction to Splines for Use in Computer Graphics and Geometric Modeling, 1987.
- [10] Birkhoff and de Boor, "Piecewise polynomial interpolation and approximation," Proc. General Motors Symposium of 1964, H. L. Garabedian, ed., Elsevier, New York and Amsterdam, 1965, pp. 164-190.
- [11] G. Farin, *Curves and Surfaces for Computer Aided Geometric Design*, A practical guide, Second edition.
- [12] D. Chetverikov and Z. Szabo. "A simple and efficient algorithm for detection of high curvature points in planar curves". *Proc. 23rd workshop of Australian Pattern Recognition Group*, Steyr, pp.175-184, 1999.
- [13] H. Freeman and L.S. Davis. "A corner finding algorithm for chain-coded curves". *IEEE Trans. Computers*, 26:297-303, 1977.
- [14] A. Rosenfeld and J.S. Weszka. "An improved method of angle detection on digital curves". *IEEE Trans. Computers*, 24:940-941, 1975.