Lecture #31 – April 26, 2004

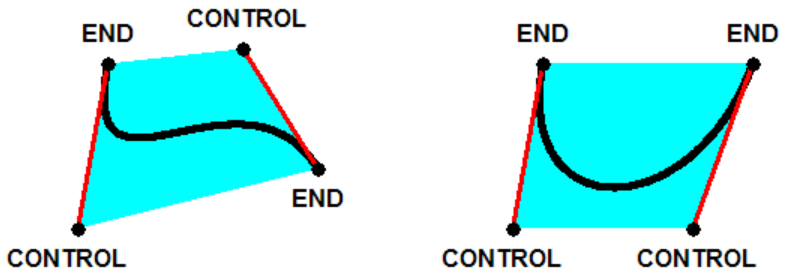
Bézier Curves and Quadratic Splines

The topics of this lecture may seem very remote from assembly language. We will be discussing a special set of mathematical curves with some very interesting properties. While the equations for these curves can be created and evaluated using traditional mathematical techniques, the curves can be generated quickly and efficiently for plotting on a video screen using nothing more than integer additions and division by two. Both of these operations are very fast in assembly language; remember that division by two is the same as a right shift by one bit.

Bézier Curves

A ***Bézier Curve*** is a piecewise, parametric, cubic polynomial. *Polynomial* means that the equations are nothing more than adds, subtracts, multiplies, and divides of simple real numbers. *Cubic* means that the equations are of the form $at^3 + bt^2 + ct + d$, where t is the independent variable. It also means that the curve will never contain more than two ***inflection points*** (changes in direction). *Parametric* means that the curve is not a function in the traditional sense of $y = f(x)$, but instead both x and y are functions of an independent variable called the ***parameter***, thus allowing the curve to be placed anywhere in the plane. There is a (different) cubic function for each dimension: $x = f_x(t)$, and $y = f_y(t)$. If the Bézier curve is in space instead of in the plane then there will be a third function of the parameter: $z = f_z(t)$. (This same process extends to any number of dimensions.) *Piecewise* means that creating a complicated curve requires several simple curves be joined end-to-end.

It takes four points to define a Bézier curve: two end points and two ***control points***. Each control point is associated with one of the end points. The interesting part of the curve starts at one end point, approaches but *does not go through* its corresponding control point, approaches but does not go through the second control point, and ends at the other end point. The curve is ***tangent*** at each end to the line between the end points and their corresponding control points. (Joining two curves end-to-end smoothly requires that the associated control points be collinear with the common end points.) The entire curve fits inside the smallest convex polygon that encloses the four points, called the ***convex hull*** of the points. Here are a couple of pictures of Bézier curves:



Given that we know the positions of the four points, p_1, p_2, p_3, and p_4, (where p_2 is the control point for end point p_1, and p_3 is the control point for end point p_4), we can generate a set of cubic equations, one for each dimension, which directly represents the curve. For example, if p_{1x}, p_{2x}, p_{3x}, and p_{4x} represent the x values for the four points, then the coefficients for the $f_x(t)$ function are as follows:

$$a_x = p_{3x} - 3\,p_{2x} + 3\,p_{1x} - p_{0x}$$
$$b_x = 3\,p_{2x} - 6\,p_{1x} + 3\,p_{0x}$$
$$c_x = 3\,p_{1x} - 3\,p_{0x}$$
$$d_x = p_{0x}$$

The final function for x is $f_x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$, and the corresponding function for y is $f_y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$. Thus, we can plot the curve by computing the value of $f_x(t)$ and $f_y(t)$ for "enough" successive values of t. The problems with this approach are manifest: the computational load is very heavy, requiring extensive use of the floating point portion of the processor, and the selection of the proper step value for t is critical. If the step value for t is too large then the Bézier curve will appear as discrete line segments, and if the step value is too small the curve will take too long to compute. Depending on the locations of the control points, the step value for t may need to be very small in some regions of the curve, but larger elsewhere. This approach requires a measure of adaptability difficult to implement.

DeCasteljau Algorithm

The traditional approach is fine as long as you need to evaluate the functions at precise values of t to get precise coordinates along the Bézier curve. For plotting curves on screen this is overkill and a completely different approach is needed. Enter DeCasteljau's algorithm. This technique can use either floating point or integer pixel coordinates for the end points and control points. If integer coordinates are used all computations are integer as well. The downside is that the routine is highly recursive.

The basic idea is to subdivide a large Bézier curve into two smaller curves, and then repeat recursively until each curve is small enough that it can be plotted on screen as a simple line segment (or as a single pixel if the segment collapses far enough). If the recursion is set up properly the curve will be plotted continuously from one end point to the other.

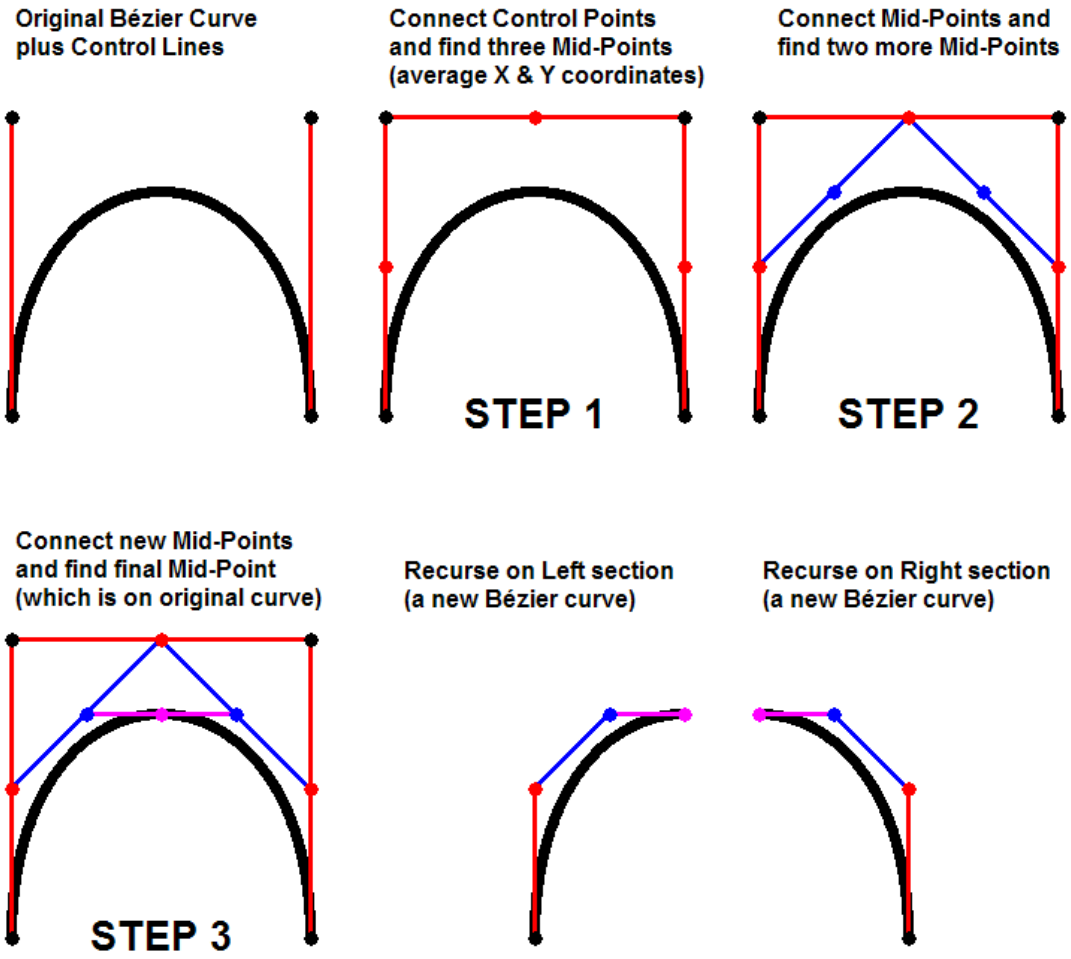
The termination condition of the recursion can be detected in a number of ways. The most obvious is when the distance between the end points of any curve fall below some threshold. (It is possible, however, for the end point distance to be below threshold and the curve still loop out away a great distance from those end points.) The distance metric can be computed as either a ***Euclidean distance*** or as ***Manhattan distance***. For points $<x_1,y_1>$ and $<x_2,y_2>$ the Euclidean distance is the square root of the sum of the squares of the differences in x and in y, and the Manhattan distance is the sum of absolute values of the differences in x and in y.

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad \textit{Euclidean Distance}$$

$$|x_2 - x_1| + |y_2 - y_1| \quad \textit{Manhattan Distance}$$

Euclidean distance gets a "true distance" and usually works well with floating point computations, but the square root is expensive in its execution time. The Manhattan distance (the distance between any two street corners in Manhattan, where you cannot go diagonally through buildings) is easy to compute in both integer and floating point.

When the distance has been computed and found to be above threshold, the DeCasteljau algorithm must divide the Bézier curve into two smaller Bézier curves. This process is shown in the following diagram:



Averaging two points to find the mid point requires averaging the *x* values and the *y* values independently. For points $\langle x_1,y_1 \rangle$ and $\langle x_2,y_2 \rangle$ the mid point is computed to be at $\langle (x_2+x_1)\div 2,\ (y_2+y_1)\div 2 \rangle$.

The following ARM assembly code shows how to average two variables in both single-precision floating point format and in integer format:

```
Floating Point          Integer
LDFS F0,X2              LDR R0,X2
LDFS F1,X1              LDR R1,X1
ADFS F0,F0,F1           ADD R0,R0,R1
DVFS F0,F0,#2           MOV R0,R0,LSR #1
```

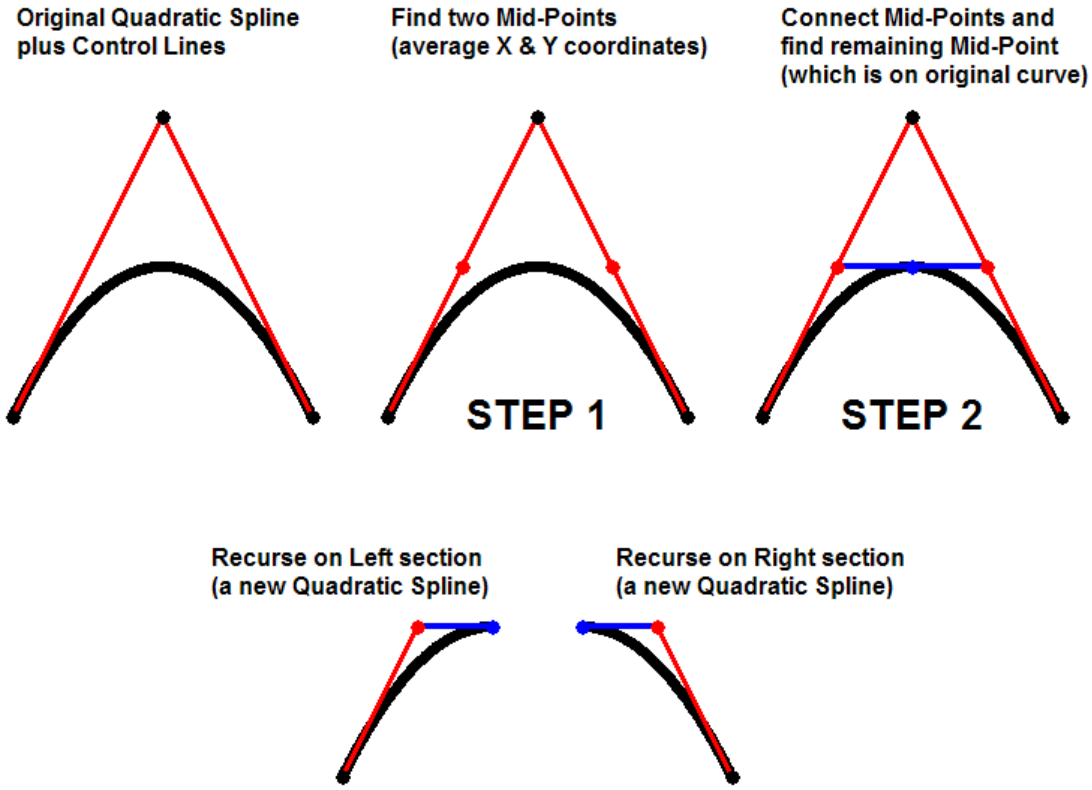
For each recursive call the activation record needs to contain local variable space for the six computed mid points, as well as for the parameters. In a high-level language format, the general form of the DeCasteljau algorithm can be expressed as follows:

```
Procedure DeCasteljau (P0,P1,P2,P3:Point)

    Var P01   : Point    { Mid point from Step 1 }
        P12   : Point    { Mid point from Step 1 }
        P23   : Point    { Mid point from Step 1 }
        P012  : Point    { Mid point from Step 2 }
        P123  : Point    { Mid point from Step 2 }
        P0123 : Point    { Mid point from Step 3 }

Begin
     If Distance(P0,P3) < Threshold Then
          Plot_Line (P0, P3)
     Else
          Begin
               P01   := (P0   + P1  ) / 2
               P12   := (P1   + P2  ) / 2
               P23   := (P2   + P3  ) / 2
               P012  := (P01  + P12 ) / 2
               P123  := (P12  + P23 ) / 2
               P0123 := (P012 + P123) / 2
               DeCasteljau (P0, P01, P012, P0123)
               DeCasteljau (P0123, P123, P23, P3)
          End
End
```

If Manhattan distance calculations are used, you can see from the pseudocode above that the entire subroutine can be performed using nothing but integer instructions. The trickiest part of coding this subroutine in assembly language is managing the stack; each activation record will have four points passed in as value parameters and six points as local variables. In the plane each point has an *x* and a *y* coordinate, so there are twenty integers (80 bytes) on the stack in addition to any registers that need to be saved for purposes of transparency.

Quadratic Splines

A similar but simpler curve is the ***quadratic spline***, which uses quadratic parametric polynomials instead of cubic polynomials. As with the Bézier curve the quadratic spline curve is tangent to control lines between each end points and a control point, but in this case there is only a single shared control point. A recursive form of the DeCasteljau algorithm exists for the spline, as shown below:



Why These Curves are Important

Bézier curves were developed (by Pierre Bézier) to create smoothly changing shapes for the automotive industry. Bézier curves and quadratic splines are used in describing the outlines of typefaces in Microsoft Windows, which can be mathematically scaled to any size without changing shape. In plotting circles on a graphics screen it is curiously often faster to plot two cubic Bézier curves in complementary positions than it is to plot one quadratic circle, although the result is not quite circular. (There are also some very fast circle plotting routines that don't use Bézier curves; there are a very large number of alternative drawing routines!) Of course, the most important use of Bézier curves is in the "mystify" screen saver in Windows!