Lecture 13: Introduction to Spline Curves

Splines are used in graphics to represent smooth curves and surfaces. They use a small set of control points (knots) and a function that generates a curve through those points. This saves space in the data base and allows a good deal of flexibility in design, at the cost of extra computation at run time. We will start with a simple, but not very useful spline. Taking the equation y=f(x), we can express f as a polynomial function, say:

$$y = a_2x^2 + a_1x + a_0$$

if we now take any three points [x0,y0],[x1,y1] and [x2,y2], we can substitute then into the equation to get three simultaneous equations which we can solve for the unknowns a_2 a_1 and a_0. We now have the equation of a curve interpolating the three points. It is of course a parabola, or parabolic spline. Notice that we don't have any control over the curve. There is only one parabola that will fit the data as shown in diagram 1. We can improve our choice by the simple expedient of using a parametric spline. Let us consider first a quadratic polynomial spline written in vector notation as:

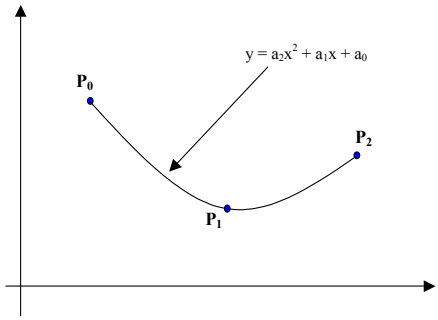
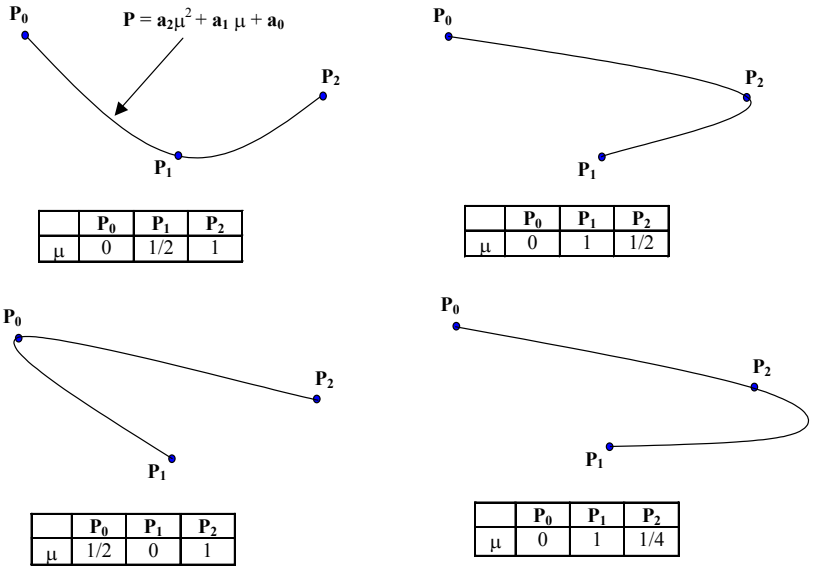


Diagram 1 A non parametric spline

$$\mathbf{P} = \mathbf{a_2}\mu^2 + \mathbf{a_1}\,\mu + \mathbf{a_0}$$

For two dimensional curves we have now six unknowns (rather than the three previously) we can use these extra degrees of freedom to control the shape of the curve. We will use the convention that 0<= μ <=1 over the range of interest. Hence at μ =0 the curve is at the first point to be interpolated, and at μ =1 it is at the last. Now consider interpolating the three points as before ($\mathbf{P_0}$=[x0,y0], $\mathbf{P_1}$=[x1,y1] and $\mathbf{P_2}$=[x2,y2]). We know that when μ=0 the curve passes through $\mathbf{P_0}$ and so we can write $\mathbf{P_0}=\mathbf{a_0}$. We also know that when μ=1 the point passes through $\mathbf{P_2}$ and this gives us the equation $\mathbf{P_2} = \mathbf{a_2} + \mathbf{a_1} + \mathbf{P_0}$ substituting for $\mathbf{a_0}$ we get $\mathbf{P_2}-\mathbf{P_0} = \mathbf{a_2} + \mathbf{a_1}$. We have now met all our conditions except that the curve shall pass through $\mathbf{P_1}$. We can choose μ anywhere in the range 0<μ<1, and get a third equation to solve for the curve parameters. In other words we can now pick one of a family of curves interpolating the three points, by selecting the value of μ at $\mathbf{P_1}$. Choosing μ =1/4 we get $\mathbf{P_1}-\mathbf{P_0} = \mathbf{a_2}/16 + \mathbf{a_1}/4$, and so on. Further possibilities are shown in figure 2. Although we have gained more freedom, we do not have any intuitive way of using it, that is to say, we have no simple way to choose μ to get the type of interpolating spline we want. Moreover, we still face the problem of having to use ever higher degrees of polynomials for higher numbers of points.



	P0	P1	P2
μ	0	1/2	1

	P0	P1	P2
μ	0	1	1/2

	P0	P1	P2
μ	1/2	0	1

	P0	P1	P2
μ	0	1	1/4

Diagram 2: The possibilities offered by parametric splines

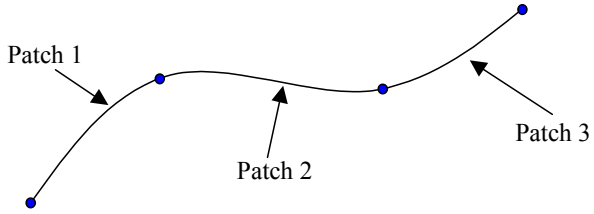


Diagram 3: Spline patches

The most robust solution to this problem is to use spline patches. This means that we define a different curve between each pair of adjacent knots, as shown in Diagram 3. This is most effectively achieved by using cubic spline patches exactly as we used the quadratic patches. The reason for choosing a cubic form is so that we can join the patches smoothly together. Now

our equation has four unknowns:

$$\mathbf{P} = \mathbf{a}_3 \mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$

So we can choose the gradient at either end of the patch. Looking at Diagram 4, we see that this can conveniently be done by taking the difference of the coordinates on either side of the knot in question. This however is not the only way of setting this gradient. If we differentiate the curve equation we get:

$$P' = 3\mathbf{a}_3 \mu^2 + 2\mathbf{a}_2 \mu + \mathbf{a}_1$$

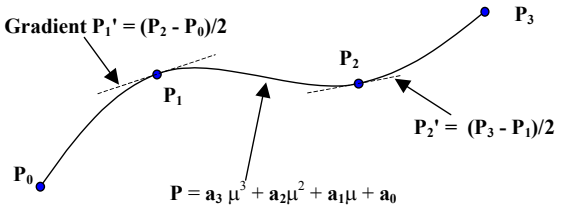


Diagram 4: Joining Spline patches

So, now consider the two ends of a spline patch between Pi and Pi+1, where $\mu = 0$ or $\mu = 1$. The equations are:

$$\mathbf{Pi} = \mathbf{a}_0$$
$$\mathbf{P'i} = \mathbf{a}_1$$
$$\mathbf{Pi+1} = \mathbf{a}_3 + \mathbf{a}_2 + \mathbf{a}_1 + \mathbf{a}_0$$
$$\mathbf{P'i+1} = 3\mathbf{a}_3 + 2\mathbf{a}_2 + \mathbf{a}_1$$

we can write this system of equations in matrix form:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{P}_i \\ \mathbf{P'}_i \\ \mathbf{P}_{i+1} \\ \mathbf{P'}_{i+1} \end{pmatrix}$$

and since the **a** values are unknown and the **P** and **P'** known, we need to invert the matrix to solve for the parameters of the spline patch.

$$\begin{pmatrix} \mathbf{a}_0 \\ \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{P}_i \\ \mathbf{P'}_i \\ \mathbf{P}_{i+1} \\ \mathbf{P'}_{i+1} \end{pmatrix}$$

Notice that we have vector quantities, so, this formulation represents eight equations for the 2D case, and twelve for the 3D case. However, the matrix is the same for each dimension. For any given set of knots, this cubic patch method gives a stable, practical solution.

Bezier Curves

One of the simplest ways of approximating a curve was made popular by Bezier in the context of car body design. It was based on a mathematical formulation by Casteljau. A typical Bezier curve is shown in Diagram 5. Here four knots $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$ are shown. The gradient at each end of the Bezier curve is the same as the gradient of the line joining the first two knots ($\mathbf{P}_0 = k(\mathbf{P}_1-\mathbf{P}_0)$) for some constant k (k=the number of knots - 1) . This is an important property as it allows us to join Bezier patches together smoothly. Computation of the Bezier Curve may be done in two ways. The first uses a recursive algorithm based on a method of Casteljau. The idea is illustrated by Diagram 6. For a given value of μ, say 1/2 we first construct the points on the lines $[\mathbf{P}_0, \mathbf{P}_1]$ $[\mathbf{P}_1, \mathbf{P}_2]$ and $[\mathbf{P}_2, \mathbf{P}_3]$ for the chosen value of μ. These are labelled as the first set of constructed points, $\mathbf{P}_{1,0}$, $\mathbf{P}_{1,1}$ and $\mathbf{P}_{1,2}$. The new points are joined up and the same procedure is followed to construct the second set of points $\mathbf{P}_{2,0}$ and $\mathbf{P}_{2,1}$. The process is repeated to find the point $\mathbf{P}_{3,0}$. As μ varies from 0 - 1 the locus of $\mathbf{P}_{3,0}$ traces out the Bezier Curve. Using a functional pseudocode which allows us such liberties as scalar and vector multiplications and typed functions, this algorithm can be written very simply:

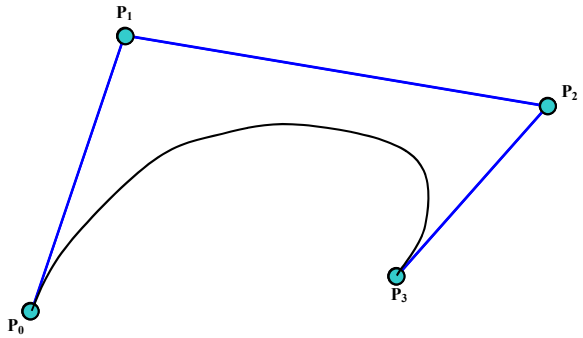


Diagram 5 A typical four point Bezier Curve

```
function   Casteljau  (var P : knots; N,r : integer; μ: real) : point;
begin
     if (r = 1)
     then Casteljau := (1- μ)*P[N] + μ *P[N+1]
     else Casteljau := (1- μ)*Casteljau(P,N,r-1, μ)+ μ*Casteljau(P,N+1,r-1, μ)
end
```

and the curve can be drawn with:

```
for j:=1 to L do
begin
     locus:=casteljau(Knot_array,0,N,j/L)
     drawto(locus[x],locus[y])
end
```

Note that here, and for all subsequent treatment of splines we will use a set of N+1 knots, labelled 0 to N.

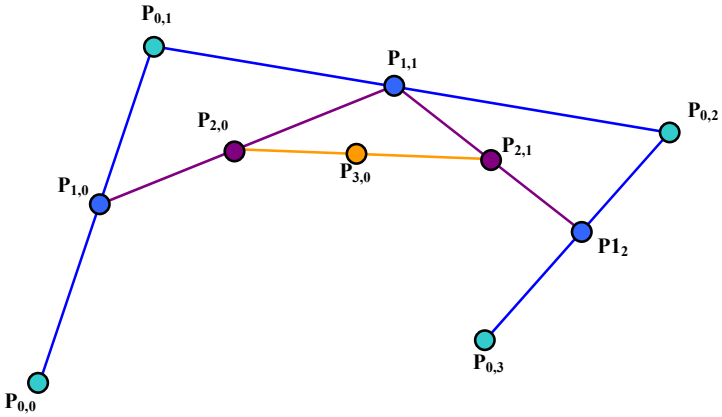


Diagram 6: Casteljau's construction of the Bezier Curve

Blending

Another way to view a Bezier curve is to think of it as a blend of its knots. This can be clearly seen when we consider applying the Casteljau algorithm to the degenerate case of two knots. This gives us the parametric line equation:

$$P = (1- \mu)\mathbf{P_0} + \mu\, \mathbf{P_1}$$

in which we are linearly blending the two position vectors to produce a third. The parameter μ may be thought of as measuring the distance along the line. Like the curve constructed using the Casteljau algorithm, most approximations consist of a blend of the values of the knots. For more than two points more complex blending is required. This leads us to the iterative formulation of the Bezier Curve.

$$\mathbf{P}(\mu) = \sum_{i=0}^{N} \mathbf{Pi}\, W(N,i,\mu)$$

where W(N,i, μ) is called the Bernstein blending function

$$W(N,i,\mu) = {}^{N}C_i\, \mu^{i}(1- \mu)^{N-i}$$

$${}^{N}C_i = N!/((N-i)!\, *\, i!)$$

notice that we are using a parameter as before and for

$\mu = 0$ $\mathbf{P}(0) = \mathbf{P_0}$

$\mu = 1$ $\mathbf{P}(1) = \mathbf{P_N}$

$0 < \mu < 1$ $\mathbf{P}(\mu)$ = some blend of all $\mathbf{P_i}$

The iterative equation for the Bezier curve can be computed slightly more efficiently in terms of space than the recursive form, though in the 2D case this is not likely to be significant, since it is rare to use Bezier curves for more than a few points. The iterative solution, though less elegant, generalises to surface construction more easily, and so tends to be used in preference.

Characteristics of Bezier Curves

As previously mentioned, Bezier curves have their end gradient clamped to the slope of the end line segments, and, beyond the ends they blend the positions of all the points. Since Bezier Curves are a blend of all their control points there is little local control over a part of the curve. Figure 7 shows how a large number of control points tends to be ineffectual with Bezier curves. Moving the intermediate points has little effect, and it is not possible to create a curve

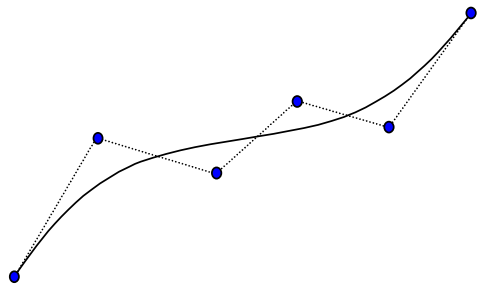


Diagram 7
Lack of local control in Bezier Curves

which wiggles with any degree of complexity. This problem can offset to some degree by piecing together a number of sections, as we did for the cubic patches.

Relation between Bezier Curves and Cubic Patches

We noted previously that the order of the Bezier curve is one less than the number of knots, so for four knots we have a cubic spline. This will be seen by expanding the iterative form of the Bezier curve:

$$\mathbf{P}(\mu) = \sum_{i=0}^{3} \mathbf{Pi}\, W(3,i,\mu)$$

for the case of four knots:

$$\mathbf{P}(\mu) = \mathbf{P_0}(1-\mu)^3 + 3\,\mathbf{P_1}\,\mu\,(1-\mu)^2 + 3\,\mathbf{P_2}\,\mu^2\,(1-\mu) + \mathbf{P_3}\,\mu^3$$

which expands into the form

$$\mathbf{P}(\mu) = \mathbf{a_3}\,\mu^3 + \mathbf{a_2}\,\mu^2 + \mathbf{a_1}\mu + \mathbf{a_0}$$

where $\mathbf{a_3} = \mathbf{P_3} - 3\mathbf{P_2} + 3\mathbf{P_1} - \mathbf{P_0}$

$\mathbf{a_2} = 3\mathbf{P_2} - 6\mathbf{P_1} + 3\mathbf{P_0}$

$\mathbf{a_1} = 3\mathbf{P_1} - 3\mathbf{P_0}$

$\mathbf{a_0} = \mathbf{P_0}$

so, we can look at it as just a cubic spline patch between $\mathbf{P_0}$ and $\mathbf{P_3}$, with two shape control points, $\mathbf{P_1}$ and $\mathbf{P_2}$ which are manipulated by the designer.

The same method can be applied to the Casteljau construction by expanding the recursion backwards:

$$\begin{aligned}
\mathbf{P_{3,0}} &= \mu\,\mathbf{P_{2,1}} + (1-\mu)\,\mathbf{P_{2,0}} \\
&= \mu\,(\mu\,\mathbf{P_{1,2}} + (1-\mu)\,\mathbf{P_{1,1}}) + (1-\mu)\,(\mu\,\mathbf{P_{1,1}} + (1-\mu)\,\mathbf{P_{1,0}}) \\
&= \mu^2\,\mathbf{P_{1,2}} + 2\mu(1-\mu)\,\mathbf{P_{1,1}} + (1-\mu)^2\,\mathbf{P_{1,0}} \\
&= \mu^2\,(\mu\,\mathbf{P_{0,3}} + (1-\mu)\,\mathbf{P_{0,2}}) + 2\mu(1-\mu)\,(\mu\,\mathbf{P_{0,2}} + (1-\mu)\,\mathbf{P_{0,1}}) + (1-\mu)^2\,(\mu\,\mathbf{P_{0,1}} + (1-\mu)\,\mathbf{P_{0,0}})
\end{aligned}$$

if we drop the first subscript, which indicated the construction level of the Casteljau algorithm we get

$$\begin{aligned}
&= \mu^2\,(\mu\,\mathbf{P_3} + (1-\mu)\,\mathbf{P_2}) + 2\mu(1-\mu)\,(\mu\,\mathbf{P_2} + (1-\mu)\,\mathbf{P_1}) + (1-\mu)^2\,(\mu\,\mathbf{P_1} + (1-\mu)\,\mathbf{P_0}) \\
&= \mathbf{P_0}(1-\mu)^3 + 3\,\mathbf{P_1}\,\mu\,(1-\mu)^2 + 3\,\mathbf{P_2}\,\mu^2\,(1-\mu) + \mathbf{P_3}\,\mu^3
\end{aligned}$$

as we had for the blending formulation.

So we can think of a four point Bezier curve as a cubic spline patch with shape control. The curve goes through points $\mathbf{P_0}$ and $\mathbf{P_3}$ and by picking up points $\mathbf{P_1}$ and $\mathbf{P}$ with a mouse and moving them we can control the shape as desired.