Lecture 9: 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 allows the creation of complex smooth shapes without the need for mainpulating many short line segments or polygons at the cost of a little extra computation time when the objects of a scene are being designed. 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:

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

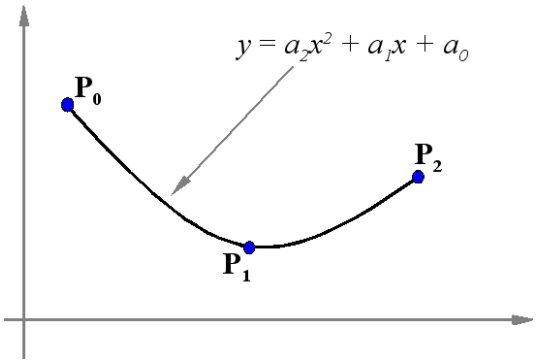


Figure 1: A non-parametric spline

If we now take any three points $[x_0, y_0]$, $[x_1, y_1]$ and $[x_2, y_2]$, 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 figure 1.

Parametric Splines

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:

$$\mathbf{P} = \mathbf{a_2}\mu^2 + \mathbf{a_1}\mu + \mathbf{a_0} \tag{1}$$

where $\mathbf{a_0}, \mathbf{a_1}$ and $\mathbf{a_2}$ are constant vectors whose values determine the shape of the spline. For two dimensional curves we now therefore have six unknowns (rather than the three previously). We can use these extra degrees of freedom to control the shape of the curve.

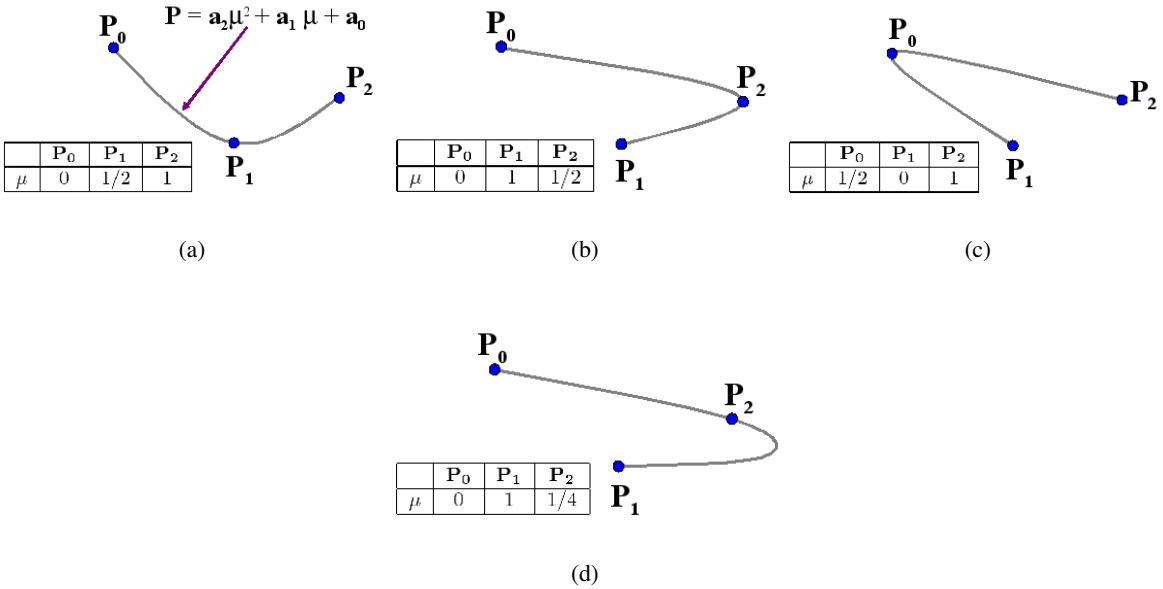


Figure 2: Possibilities using parametric splines

We will use the convention that $0 \leq \mu \leq 1$ over the range of interest. Hence at $\mu = 0$ the curve is at the first point to be interpolated, and at $\mu = 1$ it is at the last. Now consider interpolating the three points as before ($\mathbf{P_0} = [x_0, y_0], \mathbf{P_1} = [x_1, y_1]$ and $\mathbf{P_2} = [x_2, y_2]$). When $\mu = 0$ the curve passes through the first point, say

$\mathbf{P_0}$, and so, substituting $\mu = 0$ into equation 1 we can write $\mathbf{P_0} = \mathbf{a_0}$. Similarly, when $\mu = 1$ the point passes through the last point, say $\mathbf{P_2}$, and this gives us the equation:

$$\mathbf{P_2} = \mathbf{a_2} + \mathbf{a_1} + \mathbf{a_0},$$

substituting for $\mathbf{a_0}$ we get

$$\mathbf{P_2} - \mathbf{P_0} = \mathbf{a_2} + \mathbf{a_1}. \tag{2}$$

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 $0 < \mu < 1$, 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 $\mu = 1/4$ we get

$$\mathbf{P_1} - \mathbf{P_0} = \mathbf{a_2}/16 + \mathbf{a_1}/4. \tag{3}$$

We can now solve equations 2 and 3 for the values of $\mathbf{a_1}$ and $\mathbf{a_2}$ and draw the curve as shown in Figure 2(d). Further possible curves using the same three points and parametric equation are shown in Figures 2(a), 2(b) and 2(c).

SplinePatches

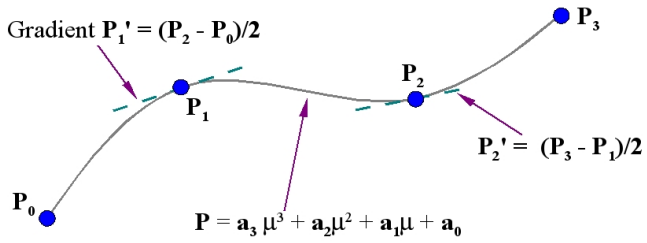


Figure 3: A simple way to join patches

Although we have gained more freedom by using the parametric form, we do not have any intuitive way of using it. That is to say, we have no simple way to choose μ values for each point 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. To overcome these difficulties we introduce a method based on spline patches that allows simple intuitive spline construction. We define a different curve between each pair of adjacent knots, as shown in Figure 3. This is most commonly done by using a cubic spline for each patch, rather than the quadratic splines formulated above. The reason for choosing a cubic form is so that we can join the patches smoothly together. The equation for a parmetric cubic spline patch has four unknowns, $\mathbf{a_0}, \mathbf{a_1}, \mathbf{a_2}$ and $\mathbf{a_3}$:

$$\mathbf{P} = \mathbf{a_3}\mu^3 + \mathbf{a_2}\mu^2 + \mathbf{a_1}\mu + \mathbf{a_0} \tag{4}$$

We use the extra degrees of freedom provided by the cubic equation to set the gradient at either end of the patch, and thus make it join seamlessly to its neighbours. Looking at Figure 3, 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, as we will see later. If we differentiate the curve equation we get:

$$\mathbf{P'} = 3\mathbf{a_3}\mu^2 + 2\mathbf{a_2}\mu + \mathbf{a_1} \tag{5}$$

Now consider the two ends of a spline patch between $\mathbf{P_i}$ and $\mathbf{P_{i+1}}$, where $\mu = 0$ or $\mu = 1$. We can find the positions and gradients at each end by substituting $\mu = 0$ and $\mu = 1$ into equations 4 and 5 which gives:

$\mathbf{P_i} = \mathbf{a_0}$
$\mathbf{P'_i} = \mathbf{a_1}$
$\mathbf{P_{i+1}} = \mathbf{a_3} + \mathbf{a_2} + \mathbf{a_1} + \mathbf{a_0}$
$\mathbf{P'_{i+1}} = \mathbf{3a_3} + \mathbf{2a_2} + \mathbf{a_1}$

We can write this system of equations in matrix form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} \mathbf{a_0} \\ \mathbf{a_1} \\ \mathbf{a_2} \\ \mathbf{a_3} \end{bmatrix} = \begin{bmatrix} \mathbf{P_i} \\ \mathbf{P'_i} \\ \mathbf{P_{i+1}} \\ \mathbf{P'_{i+1}} \end{bmatrix} \tag{6}$$

and since the $\mathbf{a}$ values are unknown and the $\mathbf{P}$ and $\mathbf{P}'$ known, we need to invert the matrix to solve for the parameters of the spline patch.

$$\begin{bmatrix} \mathbf{a_0} \\ \mathbf{a_1} \\ \mathbf{a_2} \\ \mathbf{a_3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{P_i} \\ \mathbf{P'_i} \\ \mathbf{P_{i+1}} \\ \mathbf{P'_{i+1}} \end{bmatrix} \tag{7}$$

Notice that we have vector quantities, so, this formulation represents eight equations for the 2D case, and twelve for the 3D case. 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 the French mathematician Pierre Bezier in the context of car body design. It was based on a mathematical formulation by another French mathematician Paul de Casteljau. A typical Bezier curve is shown in Figure 4. 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, thus: $\mathbf{P'_0} = k(\mathbf{P_1} - \mathbf{P_0})$, where k is 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 Figure 5. For a given value of μ, say 1/2 we first construct the points on the lines [$\mathbf{P_{0,0}}$, $\mathbf{P_{0,1}}$] [$\mathbf{P_{0,1}}$, $\mathbf{P_{0,2}}$] and [$\mathbf{P_{0,2}}$, $\mathbf{P_{0,3}}$] 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 $\mathbf{P_{3,0}}$. This is a point on the Bezier curve. As μ varies from 0 to 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:

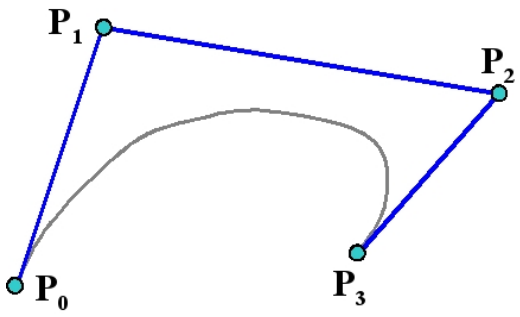


Figure 4: A typical Bezier Curve

```
Point Casteljau (knots P[] ; int N, int r, float μ)
begin
    if (r == 1)
    then Casteljau = μ*P[N+1] + (1-μ)*P[N]
    else Casteljau = μ*Casteljau(P,N+1,r-1,μ) + (1-μ)*Casteljau(P,N,r-1,μ)
end
```

and the curve can be drawn with:

```
for j=1 to L
begin
    locus=Casteljau(Knotarray,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.

Blending

Another way to view a Bezier curve is to think of it as a blend of its knots. In the simplest case, if we apply the Casteljau algorithm to the degenerate case of two knots we get the parametric line equation:

$$\mathbf{P} = \mu\mathbf{P_{0,1}} + (1 - \mu)\mathbf{P_{0,0}}$$

We can think of this as linearly blending the two points 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 spline formulations consist of a blend of the positions of the knots. For more than two points the blend is expressed by the iterative formulation of the Bezier Curve:

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

where $W(N, i, \mu)$ is called the Bernstein blending function:

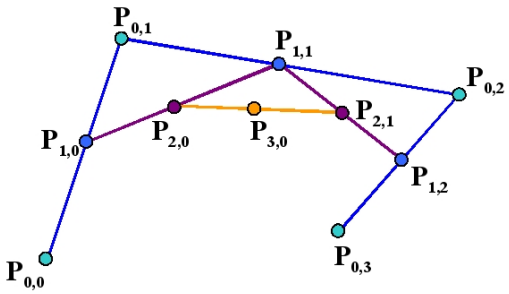


Figure 5: Using Casteljau's construction to draw a Bezier curve

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

$$\binom{N}{i} = \frac{N!}{(N-i)!i!}$$

As before we are using a parameter μ to determine the distance along the curve, and it can be easily verified that when μ is 0 or 1 the spline interpolates the end points, $\mathbf{P(0)} = \mathbf{P_0}$ and $\mathbf{P(1)} = \mathbf{P_N}$, and that when $0 < \mu < 1$ then $\mathbf{P}(\mu)$ is a blend of all the knots $\mathbf{P_i}$.

The iterative equation for the Bezier curve can be computed slightly more efficiently in terms of space than the recursive form, though for most applications 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 6 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 which wiggles with any degree of complexity. This problem can be offset to some degree by pieceing together a number of sections, as we did for the cubic patches.

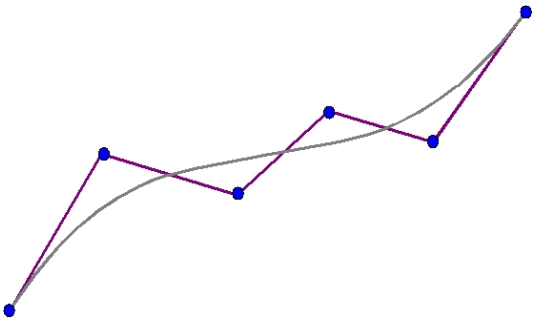


Figure 6: The lack of local detail in Bezier curves

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 verified by expanding the iterative form of the Bezier curve:

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

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

If we multiply out the brackets are collect the terms we get:

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

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 see that the four point Bezier curve is 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.

We can also show that the Casteljau construction results in the creation of a cubic spline patch. To do this we expand the the recursion backwards:

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

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

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

which is the same as 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 $\mathbf{P_2}$ with a mouse and moving them we can control the shape as desired.

The Gradients at the spline ends

To determine the gradient at any point of a parametric spline curve we differentiate it with respect to the parameter. Thus if:

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

we differentiate to get:

$$\mathbf{P}'(\mu) = -3\mathbf{P_0}(1-\mu)^2 + 3\mathbf{P_1}(1-\mu)^2 - 6\mathbf{P_1}\mu(1-\mu) + 6\mathbf{P_2}\mu(1-\mu) - 3\mathbf{P_2}\mu^2 + 3\mathbf{P_3}\mu^2$$

grouping the terms

$$\mathbf{P}'(\mu) = (3\mathbf{P_1} - 3\mathbf{P_0})(1-\mu)^2 + (6\mathbf{P_2} - 6\mathbf{P_1})\mu(1-\mu) + (3\mathbf{P_3} - 3\mathbf{P_2})\mu^2$$

So at the start, where $\mu = 0$, the gradient is $3\mathbf{P_1} - 3\mathbf{P_0}$ and at the end, where $\mu = 1$, the gradient is $3\mathbf{P_3} - 3\mathbf{P_2}$. This confirms that the curve is tangential to $\mathbf{P_1} - \mathbf{P_0}$ at the start and $\mathbf{P_3} - \mathbf{P_2}$ at the end.

Lecture 10: Introduction to Surface Construction

Non-ParametricSurfaces

We now turn to the question of how to represent surfaces, and how to draw them. As was the case with constructing spline curves, one possibility is to adopt the simple solution of non-parametric Cartesian equations. A quadratic surface would have an equation of the form:

$$[x\ y\ z\ 1]\begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

which multiplies out to:

$$ax^2 + ey^2 + hz^2 + 2bxy + 2cxz + 2fyz + 2dx + 2gy + 2iz + 1 = 0$$

and the nine scalar unknowns $(a, b, ..i)$ can be found by specifying nine points $\mathbf{P_i} = [x_i, y_i, z_i]$ through which the surface must pass. This creates a system of nine linear equations to solve. Notice that because of the inherent symmetry of the formulation there are only nine unknowns, not sixteen. The constant term can always be taken as 1 without loss of generality. This method however suffers the same limitations as the analogous method for curves. It is difficult to control the surface shape since there is only one quadratic surface that will fit the points.

Parametric Surfaces

It is a simple generalisation to go to parametric surfaces. In the case of spline curves, the locus of a point was a function of one parameter. However a point on a surface patch is a function of two parameters, which we write as $\mathbf{P}(\mu, \nu)$. We could define a parametric surface using a matrix formulation:

$$\mathbf{P}(\mu, \nu) = [\mu, \nu, 1]\begin{bmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} \\ \mathbf{b} & \mathbf{d} & \mathbf{e} \\ \mathbf{c} & \mathbf{e} & \mathbf{f} \end{bmatrix}\begin{bmatrix} \mu \\ \nu \\ 1 \end{bmatrix} = 0$$

Multiplying out we get:

$$\mathbf{P}(\mu, \nu) = \mathbf{a}\mu^2 + 2\mathbf{b}\mu\nu + 2\mathbf{c}\mu + \mathbf{d}\nu^2 + 2\mathbf{e}\nu + \mathbf{f} \tag{1}$$

The values of the constant vectors (**a**,**b**, . . **f**) determine the shape of the surface. The surface has edges given by the four curves for which one of the parameters is either 0 or 1. These are the quadratics:

$$\mathbf{P}(0, \nu) = \mathbf{d}\nu^2 + 2\mathbf{e}\nu + \mathbf{f}$$

$$\mathbf{P}(1, \nu) = \mathbf{a} + 2(\mathbf{b} + \mathbf{e})\nu + 2\mathbf{c} + \mathbf{d}\nu^2 + \mathbf{f}$$

$$\mathbf{P}(\mu, 0) = \mathbf{a}\mu^2 + 2\mathbf{c}\mu + \mathbf{f}$$

$$\mathbf{P}(\mu, 1) = \mathbf{a}\mu^2 + 2(\mathbf{b} + \mathbf{c})\mu + \mathbf{d} + 2\mathbf{e} + \mathbf{f}$$

The unknown values in the matrix (**a**, **b**, **c** etc) are all vectors whose values can be computed by substituting in six points to be interpolated for given values of μ and ν. Just as we did for the spline curves, we need to specify the values of μ and ν where the knots are located. For example, one possibility is:

	μ	ν
$\mathbf{P_0}$	0	0
$\mathbf{P_1}$	0	1
$\mathbf{P_2}$	1	0
$\mathbf{P_3}$	1	1
$\mathbf{P_4}$	1/2	0
$\mathbf{P_5}$	1/2	1

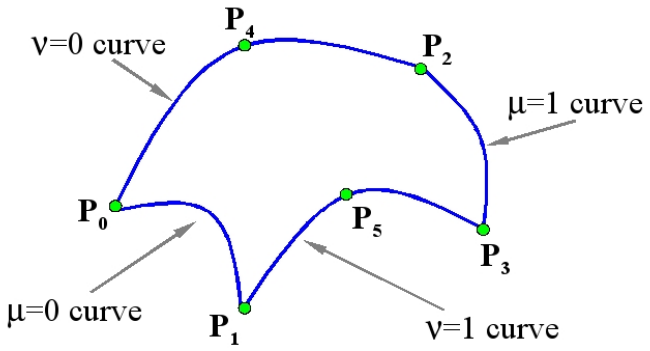


Figure 1: A simple Quadratic Surface

These values will create a surface that is similar to figure 1. Substituting them into equation 1 we obtain a set of linear equations to find the values of the constants (**a,b,c,d,e,f**) that define the patch.

P0 = **f**
P1 = **d** + 2**e** + **f**
P2 = **a** + 2**c** + **f**
P3 = **a** + 2**b** + 2**c** + **d** + 2**e** + **f**
P4 = **a**/4 + **c** + **f**
P5 = **a**/4 + **b** + **c** + **d** + 2**e** + **f**

This is more flexible than the non-parametric formulation, but it still does not provide us with a very useful spline since there are no intuitive ways of using it to create a particular shape. Higher orders can be designed by including μ^2 and ν^2 in the formulation. However little is gained by doing this. Like the equations we developed for the curves, this simple parametric form is only really applicable for solving specific interpolation problems. It is not suitable for use as a general method of surface construction.

Bi-cubic surface patches

The patch method is available, and represents a good general method, but it is more complex than for spline curves. it is quite common to adopt a formulation where the points are distributed on a regular grid, one example is the case of a terrain map. The data in this instance can be writen in a non-parametric formulation in which a function defines the height at each grid point:

$$y = f(x, z)$$

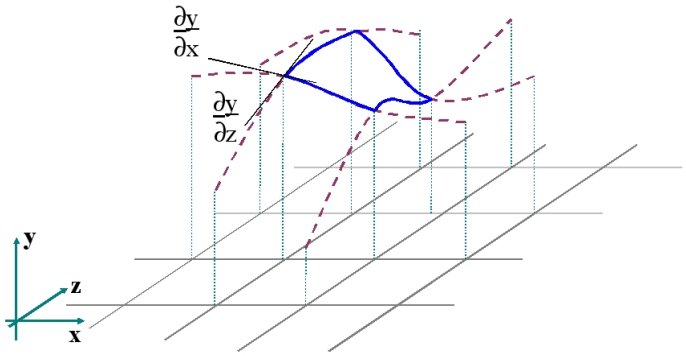


Figure 2: A "Terrain" Surface Patch

A typical patch is shown in figure 2. At every grid point there are two gradients which we can write as $\frac{\partial y}{\partial x}$ and $\frac{\partial y}{\partial z}$. Thus, if we are going to fit a patch to the four points, we need twelve parameters in the equation of the patch, so that we can match the positions and both gradients at the four corners. Furthermore, we need to ensure continuity along the boundaries of the patches. Rather than try to develop one equation which will do this for us, we normally use bi-cubic interpolation. It will be seen from figure 2 that we can easily design spline curves for the four edges. These curves will be continuous with the neighbouring patches. Thus we adopt a solution where we use these curves at the edges of the patch, and we interpolate them in the middle of the patch.

Parametric Surface patches - the Coon's Patch

Following the methods we applied to the spline curves we adopt a parametric formulation for the surface patches which gives more flexibility in design. The parametric formulation can be used for simple terrain maps described above where the Cartesian and parametric axes are the same. in gernerall though, the points which are to be interpolated need not be on a regular grid in the $z-x$ plane though we still need them to form a rectangular array. A typical patch in parametric space is shown in figure 3. We can specify the four curves that bound a patch, using the method of cubic patches described in the lecture on spline curves. In particular, we can derive the gradients from the central differences of the adjoining points such that the surface will fit a rectangular grid of points smoothly.

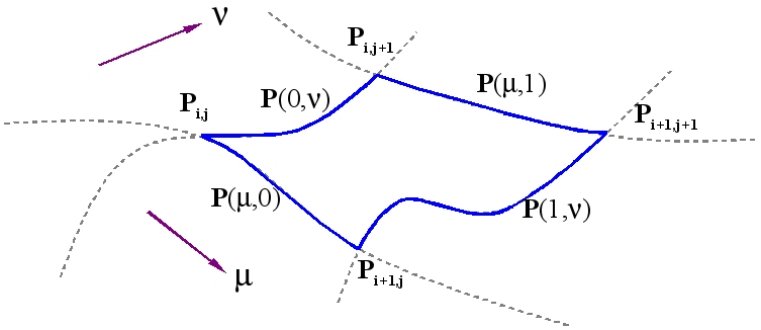


Figure 3: A Parametric Surface Patch

Notice that the contours are orthogonal in the parameter space, and we have two parameters. That is to say, contours joining $\mathbf{P}_{i,j}$ to $\mathbf{P}_{i,j+1}$ and $\mathbf{P}_{i+1,j}$ to $\mathbf{P}_{i+1,j+1}$, are both functions of the ν parameter as it varies from

0 to 1, with constant μ. Similarly the contours joining $\mathbf{P}_{i,j}$ to $\mathbf{P}_{i+1,j}$ and $\mathbf{P}_{i,j+1}$ to $\mathbf{P}_{i+1,j+1}$ are both functions of μ in the range 0 to 1 with constant ν. We can describe the edge contours joining the knots simply by treating one of the parameters as fixed at 1 or 0. Thus we denote the four contours that bound the patch as follows:
$\mathbf{P}(0,\nu)$ joining $\mathbf{P}_{i,j}$ to $\mathbf{P}_{i,j+1}$
$\mathbf{P}(1,\nu)$ joining $\mathbf{P}_{i+1,j}$ to $\mathbf{P}_{i+1,j+1}$
$\mathbf{P}(\mu,0)$ joining $\mathbf{P}_{i,j}$ to $\mathbf{P}_{i+1,j}$
$\mathbf{P}(\mu,1)$ joining $\mathbf{P}_{i,j+1}$ to $\mathbf{P}_{i+1,j+1}$
We need now to define the locus of $\mathbf{P}(\mu,\nu)$ within the patch, in such a way that at the edges it follows the contours, and in the middle it is a reasonable blend of them. This is done by linear interpolation, the equation being:

$$\begin{aligned}\mathbf{P}(\mu,\nu) = \quad & \mathbf{P}(\mu,0)(1-\nu) + \mathbf{P}(\mu,1)\nu + \mathbf{P}(0,\nu)(1-\mu) + \mathbf{P}(1,\nu)\mu - \mathbf{P}(0,0)(1-\mu)(1-\nu) \\ & -\mathbf{P}(0,0)(1-\mu)\nu - \mathbf{P}(0,0)\mu(1-\nu) - \mathbf{P}(1,1)\mu\nu \end{aligned} \tag{2}$$

This is a tricky equation to understand, but you can verify that you get the four edge contours by substituting μ=0, μ=1, ν=0 and ν=1. The first four terms are simply a linear interpolation of both the bounding curves. However it is clear that we cannot just add them together, as this would no longer go through the four points that define the patch. The last four negative terms correct the curve at the corner points without introducing any discontinuity. This formulation is called the Coon's Patch, and is probably the easiest to use surface construction method.

Rendering Surface Patches

A simple way to draw a patch of this kind is called polygonisation. The method is illustrated in figure 4. Using equation 2 we can calculate a value of $\mathbf{P}(\mu,\nu)$ for any given ν and ν. If they are in in the range $[0..1]$ the value is a point on the patch. Thus we can calculate a regular grid of points over the patch and join them into triangles. These are then treated as polygons and fed to a polygon rendrer. Providing we make the grid fine enough we can get an exact representation down to pixel resolution. For faster results we can use coarser polygons, and apply Gouraud or Phong shading to smooth out the discontinuities.

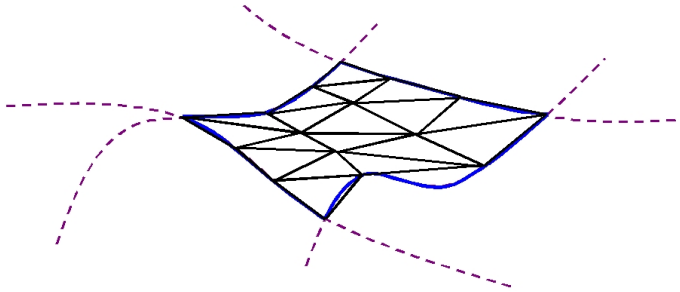


Figure 4: Triangulating a Surface Patch

Rendering the patch using the method of ray tracing is also possible, but it is made difficult by the high order of the patch equation. As we will see in the lecture on ray tracing we need to solve for the intersection of a ray with the patch. The ray in can be written as a stright line $\mathbf{P}(\gamma) = \mathbf{S} + \gamma\mathbf{d}$ then we can equate $\mathbf{P}(\gamma)$ with $\mathbf{P}(\mu,\nu)$) and then attempt to solve for the three parameters, γ, μ and ν. However, the equation of the patch is fourth order (it has terms of $\mu^3\nu$ and $\mu\nu^3$) and so only an iterative solution is possible. There will also be several valid intersections between the ray and the patch, and it is necessary to find the nearest. This can be done by numerical methods, but is computationally very expensive. If the surface is smooth, and relatively well behaved, then a practical solution is to do a coarse polygonistion, using triangles, and intersect the ray with each triangle to find the closest intersection. This area of the patch can then be polygonised further, and the process repeated until sufficient accuracy is reached. Thus the calculation is simply the intersection of the ray and the triangle.

An older way of drawing surfaces is to represent them by contours. This is called lofting (a term originating from the aircraft industry). To draw a contour we need only fix one of the parameters, and draw the curve as the other ranges over the interval 0 to 1. To produce any meaningful representation of a complex surface however, it is necessary to eliminate the hidden lines, and this is done by a method called the floating horizon. Basically we sort the contours into the order of the distance from us, which in the normal configuration is in order of z. Then, we set up a record, for each pixel in the x direction of the largest y (height) found so far: ie the horizon. Before each part of a contour is drawn it is checked against this horizon. If it is above it, it is drawn, and the horizon is updated, if not it is ignored.

Example of Using a Coon's Patch

We will conclude with an example of the construction of a Coons patch. Part of a terrain map defined (for simplicity) on a regular x,y grid is shown in figure 5. We will construct a parametric Coon's spline patch on the four centre points. The corners are defined directly in figure 5 so we can write:

$\mathbf{P}(0,0) = (9,4,12)$
$\mathbf{P}(0,1) = (9,5,11)$
$\mathbf{P}(1,0) = (10,4,13)$
$\mathbf{P}(1,1) = (10,5,14)$

y,ν

x,μ	2	3	4	5	6	7
7	.	.	.	.	.	.
8	.	.	10	9		
9	.	14	12	11	10	.
10	.	15	13	14	10	.
11	.	.	10	11		

Figure 5: An example of the Coons Patch

We need to define the gradients at the corners of the patch. We can use the central difference approximation using the two points adjacent to the corners. For gradients in the μ or x direction:

$\partial\mathbf{P}(0,0)/\partial\mu = ((10,4,13) - (8,4,10))/2 = (1,0,1.5)$
$\partial\mathbf{P}(1,0)/\partial\mu = ((11,4,10) - (9,4,12))/2 = (1,0,-1)$
$\partial\mathbf{P}(0,1)/\partial\mu = ((10,5,14) - (8,5,9))/2 = (1,0,2.5)$
$\partial\mathbf{P}(1,1)/\partial\mu = ((11,5,11) - (9,5,11))/2 = (1,0,0)$

and for the gradients in the ν or y direction:

$\partial\mathbf{P}(0,0)/\partial\nu = ((9,5,11) - (9,3,14))/2 = (0,1,-1.5)$
$\partial\mathbf{P}(1,0)/\partial\nu = ((10,5,14) - (10,3,15))/2 = (0,1,-0.5)$
$\partial\mathbf{P}(0,1)/\partial\nu = ((9,6,10) - (9,4,12))/2 = (0,1,-1)$
$\partial\mathbf{P}(1,1)/\partial\nu = ((10,6,10) - (10,4,13))/2 = (0,1,-1.5)$

To find the bounding contours we use the cubic spline patch equation. Thus:

$$\mathbf{P}(\mu,0) = \mathbf{a}_3\mu^3 + \mathbf{a}_2\mu^2 + \mathbf{a}_1\mu + \mathbf{a}_0$$

We need to solve for the four constant vectors $\mathbf{a}_3$, $\mathbf{a}_2$, $\mathbf{a}_1$ and $\mathbf{a}_0$, and we do this using the matrix formulation introduced in the lecture on spline curves which yields:

$\mathbf{a}_0 = \mathbf{P}_0 = (9,4,12)$
$\mathbf{a}_1 = \mathbf{P}'_0 = (1,0,1.5)$
$\mathbf{a}_2 = -3\mathbf{P}_0 - 2\mathbf{P}'_0 - 3\mathbf{P}_1 - \mathbf{P}'_1 = -3*(9,4,12) - 2*(1,0,1.5) + 3*(10,4,13) - (1,0,1) = (0,0,1)$
$\mathbf{a}_3 = 2\mathbf{P}_0 + \mathbf{P}'_0 - 2\mathbf{P}_1 + \mathbf{P}'_1 = 2*(9,4,12) + (1,0,1.5) - 2*(10,4,13) + (1,0,1) = (0,0,0.5)$

Similarly we can solve for the other bounding curves $\mathbf{P}(\mu,1)$, $\mathbf{P}(0,\nu)$ and $\mathbf{P}(1,\nu)$.

We now have equations for all the individual terms in the Coon's patch:

$\mathbf{P}(\mu,0)$: a cubic polynomial in μ
$\mathbf{P}(\mu,1)$: a cubic polynomial in μ
$\mathbf{P}(0,\nu)$: a cubic polynomial in ν
$\mathbf{P}(1,\nu)$: a cubic polynomial in ν
$\mathbf{P}(0,0), \mathbf{P}(0,1), \mathbf{P}(1,0)$ and $\mathbf{P}(1,1)$: the corner points,

so given μ and ν we can evaluate each of these eight terms and so find the coordinate on the Coon's patch using equation 2.

Tutorial 5: Texture Mapping

This tutorial concerns texture mapping onto polygons. We will consider the case of a terminal in which individual pixels can be set to any intensity in the range $[0, I_{max}]$.

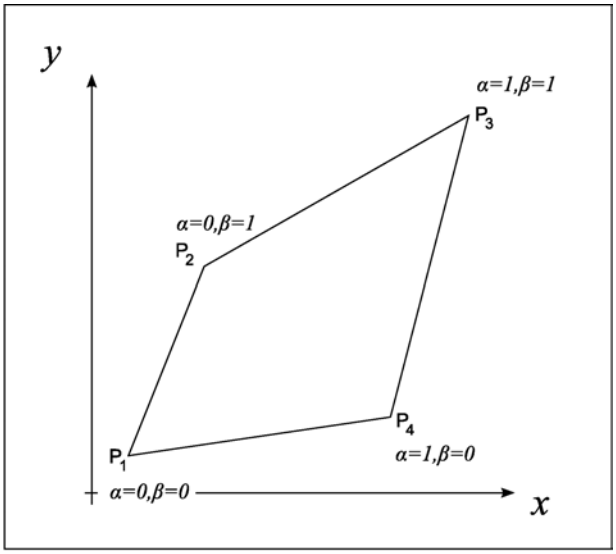
Suppose that the following function is used to transform texture coordinates (α, β) into an intensity value:

$$I = \frac{(\alpha + \beta)}{2} I_{max}$$

I is therefore the intensity given to a pixel which corresponds to position (α, β) in texture space.

A quadrilateral is projected onto the screen. The screen coordinates and texture coordinates of its vertices are as follows:

Vertex	Screen coordinates (x, y)	Texture coordinates (α, β)
$\mathbf{P}_1$	(5,5)	(0, 0)
$\mathbf{P}_2$	(15,30)	(0, 1)
$\mathbf{P}_3$	(50,50)	(1, 1)
$\mathbf{P}_4$	(40,10)	(1, 0)



1. Let $\mathbf{P}_t$ be the point with screen coordinates (30, 30). Calculate the texture coordinates (α, β) corresponding to $\mathbf{P}_t$ and find the intensity to be applied to $\mathbf{P}_t$.

N.B. Use bi-linear interpolation to find the texture coordinates, i.e. solve for (α, β) the equation:

$$\mathbf{p} = \alpha\beta(\mathbf{c} - \mathbf{b}) + \alpha\,\mathbf{a} + \beta\,\mathbf{b}$$

Where

$\mathbf{p} = \mathbf{P}_t - \mathbf{P}_1$ $\quad$ $\mathbf{a} = \mathbf{P}_4 - \mathbf{P}_1$ $\quad$ $\mathbf{b} = \mathbf{P}_2 - \mathbf{P}_1$ $\quad$ $\mathbf{c} = \mathbf{P}_3 - \mathbf{P}_4$

Computing texture coordinates for every pixel by using bi-linear interpolation is very time consuming, since the solution of a quadratic is required. By doing the calculations differentially, much computation time can be saved.

For example, consider the line from $\mathbf{P}_1$ to $\mathbf{P}_2$: At $\mathbf{P}_1$ $\beta = 0$ and at $\mathbf{P}_2$ $\beta = 1$ and there are 26 pixels on the line (the biggest change in the screen coordinates is along the *y*-component, from 5 to 30, which gives 26 pixels including the end-points).

Moving along the line from $\mathbf{P}_1$ to $\mathbf{P}_2$, the differential change in β from one pixel to the next is $1/25$ because 26 pixels (including end-points) have 25 spaces between them.

Thus, we can estimate the values of β at each pixel along the line simply by adding the differential as we move from one pixel to its neighbour.

2. What are the differentials in α and β for the four lines bounding the quadrilateral?

3. Now consider the horizontal line through the point (30, 30). Using the differential method above, find the values of α and β at the points where it intersects the sides of the quadrilateral, and hence find the differentials in α and β along the line.

4. Use these differential values to compute α and β at the point (30, 30). Find also the corresponding intensity to be given to the pixel. Compare your results with the values obtained in part 1.

5. Can you suggest why the methods in parts 1 and 3 do not give exactly the same result?

Notice that with the differential method, calculating α and β for most of the pixels in the quadrilateral requires only two additions, and hence is much faster.