Chapter 18

B-Spline

Our aim this chapter is to master the theory underpinning B-spline primitives, the dominant class of primitives used in freeform design nowadays. As in the preceding chapter on Bézier theory, we'll restrict ourselves here to the polynomial version, reserving the more general rational class of NURBS (Non-Uniform Rational B-Spline) primitives for Chapter 20, as an application of projective spaces, which are the natural setting for these primitives.

Almost all 3D modelers support NURBS primitives – and so, of course, their polynomial subclass as well – in a WYSIWYG design environment. In such a setting, the user can get by merely pushing control points around, with little understanding of theory. OpenGL, on the other hand, provides an interface at a much lower level. In fact, there is almost a one-to-one correspondence between NURBS theory and OpenGL syntax. Consequently, some knowledge at least of the former is required in order to use the latter.

Unfortunately, as NURBS theory is more complex than Bézier, there really is no use-now-learn-later approach. This is the reason we did not introduce NURBS, or even its polynomial subclass, in the earlier chapter on drawing, as we did polynomial Bézier primitives. True, the lack of shortcuts and a fancy interface will be seen as drawbacks by those who care only about design and not so much about what is under the hood. On the other hand, OpenGL's minimalist setting is ideal for the purpose of grasping the underlying theory.

Our account of B-spline theory begins in Section 18.1 with an analysis of the weakness of Bézier primitives, motivating the progression to B-splines as a search, in fact, for better blending functions. The investigation of the B-spline primitives themselves begins with curves in Section 18.2, setting the stage with so-called knot vectors in anticipation of new blending functions that are polynomial in knot intervals. In subsections 18.2.1-18.2.3, we go from (uniform) first-order to quadratic B-spline curves, applying an intuitive "break-and-make" procedure to increase the degree of the spline functions. The reader is asked to apply this procedure herself in 18.2.4 to fill in the details for cubic B-splines. A significant generalization is made in 18.2.5, not only by extending the theory to B-splines of arbitrary order, but by allowing the knot vector to be non-uniform as well. We'll see the utility of non-uniform knot vectors, particularly of repeated knots which empower the designer with the best of both worlds, Bézier and B-spline. From B-spline curves to surfaces in Section 18.3 is exactly the same process as from Bézier curves to surfaces.

Finally, we come to code in Section 18.4, particularly, the syntax of OpenGL NURBS drawing primitives, though in this chapter we go only as far as their polynomial functionality. Subsections 18.4.1 and 18.4.2 discuss drawing B-spline curves and surfaces, respectively. We describe how to light and texture a B-spline surface in 18.4.3. The useful technique of trimming a B-spline surface is described in 18.4.4.

Section 18.5, with notes and suggestions for future reading, concludes the chapter.

18.1 Problems with Bézier Primitives: Motivating B-Splines

Bézier curves and surfaces, the topics of the previous chapter, are easy to use and powerful enough to create complex designs. However, they suffer from two weaknesses:

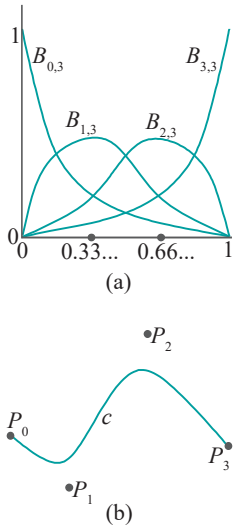


Figure 18.1: (a) Bernstein polynomials of degree 3: $B_{0,3}(u) = (1-u)^3$, $B_{1,3}(u) = 3(1-u)^2u$, $B_{2,3}(u) = 3(1-u)u^2$, $B_{3,3}(u) = u^3$ (b) A cubic Bézier curve.

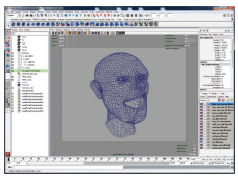


Figure 18.2: Mesh of Boris's head (courtesy of Sateesh Malla at `www.sateeshmalla.com`).

1. *Lack of local control.*

   Observe that the blending function of every control point of a Bézier curve is non-zero over the entire open parameter interval $(0, 1)$; in other words, every control point has non-zero weight (or attraction, or pull) at every point of the curve, except, possibly, the endpoints. For example, Figure 18.1(a) shows the blending functions of a cubic Bézier curve, which are, of course, the Bernstein polynomials of degree three.

   This makes modifying a Bézier curve difficult: moving any one control point alters the *entire* curve, not just near the control point. Albeit points on the curve far from the relocated control point move little because its weight is small at distant points, nevertheless, there is change. Moving control point P_1 in Figure 18.1(b), for example, from a reading of its blending function $B_{1,3}(u)$ in Figure 18.1(a) maximally affects the curve in the vicinity of $c(0.33)$, but all points on the curve, except for the endpoints, are altered to some extent. E.g., moving P_1 would cause the curve to twitch even near P_3, which is far from P_1.

   The situation for Bézier surfaces is similar, as each control point has non-zero weight at every point of the surface, except, possibly, the corners.

   Typically, in designing a complex object with numerous control points a designer would prefer to be able to modify parts of the object independently, in other words, have local control, which in turn would necessitate restricting each control point to its own limited "region of influence". For example, in arranging Boris's smirk – see Figure 18.2 – the designer may want to leave his nose and eyes exactly as they are.

2. *The degree increases with the number of control points.*

   The Bézier curve $c(u)$ approximating $n+1$ control points is polynomial of degree n in u. Evaluating a high-degree polynomial is expensive and repeated products lead to numerical instability. Complex curves, therefore, with multiple control points present a computational problem. And ditto for surfaces.

What to do about these problems? First, let's step back a bit to take the following abstract view of Bézier curves: a Bézier curve is the sum

$$c(u) = f_0(u)P_0 + f_1(u)P_1 + \ldots + f_n(u)P_n \qquad (0 \le u \le 1) \tag{18.1}$$

of its control points P_i weighted by blending functions f_i which *happen to be* Bernstein polynomials. There's no reason they *have to be* Bernstein polynomials, provided that the resulting curve c – maybe no longer Bézier – does a satisfactory job of approximating the control points. The plan then is to try and invent new blending functions which, hopefully, alleviate the Bézier difficulties.

Before proceeding, here's a bit of useful terminology: if a function f, defined on the interval domain $[a, b]$, is non-zero everywhere inside the subinterval $[a', b']$, excepting possibly its endpoints a' and b', and zero on the rest of $[a, b]$, then it is said to have *support* in $[a', b']$. Figure 18.3(a) depicts a function $f_i(u)$ defined on $[0, 1]$ with support in the subinterval $[a', b']$.

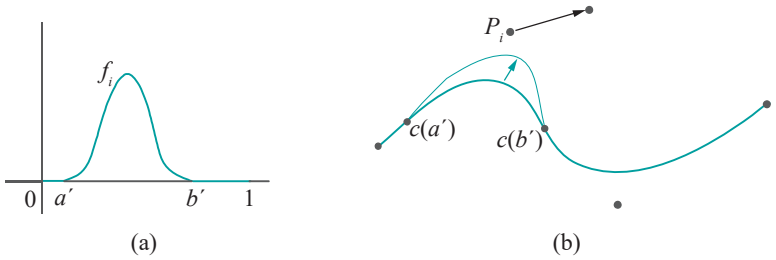


Figure 18.3: (a) Function f_i defined on $[0,1]$ has support in $[a', b']$ (b) Moving P_i, with associated blending function f_i, changes c only between $c(a')$ and $c(b')$.

Exercise 18.1. If the blending function f_i of control point P_i in expression (18.1) has support in the proper subinterval $[a', b']$ of the parameter interval $[0,1]$, then show that moving P_i changes the arc of the approximating curve c only between $c(a')$ and $c(b')$. See Figure 18.3(b).

Exercise 18.2. Prove that the ith Bernstein polynomial of degree n for every i, $0 \le i \le n$, has support in the entire parameter interval $[0,1]$ (keep in mind that the behavior of the polynomial outside of $[0,1]$ is of no interest).

From the preceding two exercises, it seems, then, that the first problem with Bézier curves mentioned above arises because the blending function of every control point has support in the entire parameter interval $[0,1]$. A solution, therefore, would be to find blending functions each having support in only part of that interval.

Moreover, the second problem would be solved if the degree of the blending functions could be *decoupled* from the number of control points, so that increasing the latter did not necessarily raise the former.

So, now that we have an idea of what we want, let's see what we can find. Suppose, to begin with, that we ask for blending functions all quadratic, *no matter* the number of control points. Further, to obtain local control, then, we seek quadratics with limited support – whose graphs resemble that of f_i in Figure 18.3(a). Sadly, this is a hopeless task because a quadratic is zero only at its at most two roots, not on any interval stretch like that between 0 and a', or b' and 1. But, look again at f_i. Except for the two straight zero parts at either end, the graph of f_i does resemble somewhat an upside-down parabola – see the graph of the parabola $f(u) = u^2$ in Figure 18.4(a).

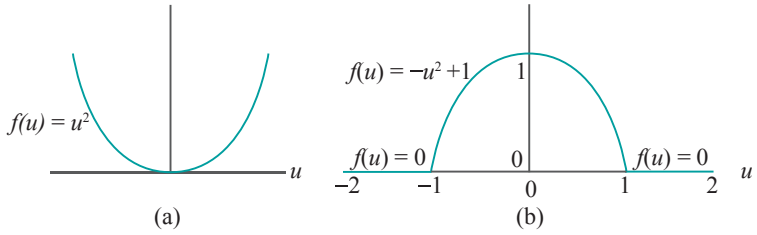


Figure 18.4: (a) Parabola (b) Three-part function: one upside-down parabola and two straight.

Note: Curves drawn in this chapter are fairly accurate sketches, but not necessarily exact plots of their equations.

Here, then, is a drastic solution. Let's make a blending function f like f_i by assembling it from three parts – one quadratic (an upside-down parabola) and two straight zero – as follows:

$$f(u) = \begin{cases} 0, & -2 \le u \le -1 \\ -u^2 + 1, & -1 \le u \le 1 \\ 0, & 1 \le u \le 2 \end{cases}$$

There's no law that says that a formula has to be one line! So the specification of f is fine. Figure 18.4(b) shows its graph. We seem to be headed in the right direction. We have a blending function which is at most quadratic and which has support in $[-1, 1]$, just half of its whole domain $[-2, 2]$.

Note: If the reader is wondering about the new parameter interval $[-2, 2]$, keep in mind that there's nothing special about the parameter interval $[0, 1]$ we use most often, other than that it's convenient to write. Parameter intervals can be any $[a, b]$, with $a < b$. In the case above, $[-2, 2]$ helps avoid fractions in the formula for f.

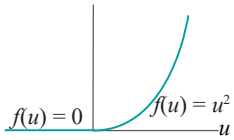


Figure 18.5: The right wing of the parabola $f(u) = u^2$ meeting the straight left half of the x-axis smoothly at the origin.

The corners (C^1-discontinuities, to be precise) at $u = \pm 1$, where the straight parts of f meet the parabolic, are undesirable though, because discontinuities in the blending function will carry over to discontinuities in the approximating curve employing such a function. It'll be nice to be rid of them. How do we get a parabolic part to join a straight part without making a corner? Oddly enough, the parabola $f(u) = u^2$ in Figure 18.4(a) itself suggests a solution. Consider the part of this parabola to the *right* of the y-axis and the (straight) part of the x-axis to the *left* of the y-axis: they meet smoothly at the origin! See Figure 18.5.

So here's the next draft. For $u \leq 0$ and $u \geq 4$, define $f(u)$ to be 0, giving two long straight parts; define $f(u) = u^2$ between 0 and 1; and, $f(u) = (u-4)^2$ between 3 and 4. See the green curves in Figure 18.6. Particularly, $f(u) = u^2$ in $[0, 1]$ is part of the right wing of the parabola of Figure 18.4(a), while $f(u) = (u-4)^2$ in $[3, 4]$ from the left wing of the same parabola (but shifted 4 units to the right). The two quadratics meet the straight parts smoothly, so that's taken care of, but there's a piece missing in between (pretend you don't see the black curve!). Now, if we could only find a quadratic to sit smoothly atop the two side quadratics and cap the gap.

It turns out that a fairly intuitive choice works: drag $f(u) = u^2$ two units to the right, flip it upside down and then raise it two units. The equation is $f(u) = -(u-2)^2 + 2$, which in $[1, 3]$ gives the black curve in Figure 18.6. We leave verification to the reader in the next two exercises.

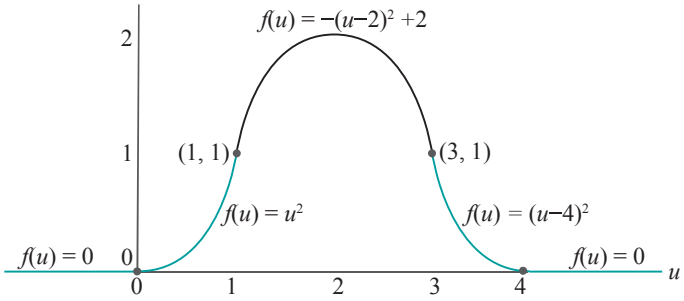


Figure 18.6: Five-part function: three parabolic and two straight parts. Joints are black points.

Exercise 18.3. Show that the curve $f(u) = -(u-2)^2 + 2$ indeed meets $f(u) = u^2$ at $(1, 1)$ and $f(u) = (u-4)^2$ at $(3, 1)$.

Exercise 18.4. Show that at each of the four *joints* $(0, 0)$, $(1, 1)$, $(3, 1)$ and $(4, 0)$ of the five-part function depicted in Figure 18.6 the tangent lines of the curves on either side are equal. Therefore, there is no C^1-discontinuity at a joint and the function is C^1-continuous everywhere.

Part answer: At $(1, 1)$, where $u = 1$, the tangent on the left is from $f(u) = u^2$ and on the right from $f(u) = -(u-2)^2 + 2$. Now, $\frac{\mathrm{d}}{\mathrm{d}u} u^2 = 2u$, which is 2 at $u = 1$, and $\frac{\mathrm{d}}{\mathrm{d}u}(-(u-2)^2 + 2) = -2(u-2)$, which is also 2 at $u = 1$, so, indeed, the tangent lines on either side of the joint $(1, 1)$ are equal.

For the record, here's the 5-line formula specifying f:

$$f(u) = \begin{cases} 0, & u \le 0 \\ u^2, & 0 \le u \le 1 \\ -(u-2)^2 + 2, & 1 \le u \le 3 \\ (u-4)^2, & 3 \le u \le 4 \\ 0, & 4 \le u \end{cases} \tag{18.2}$$

f has support in $[0, 4]$ and, from the preceding exercise, is C^1-continuous throughout. Moreover, if its parameter interval is chosen to be an interval larger than $[0, 4]$, e.g., $[-2, 6]$, then we have indeed a C^1-continuous blending function with limited support.

The moral then is to look for blending functions among the class of piecewise polynomial functions – a function is *piecewise polynomial* if its domain can be split into subintervals in each of which it's polynomial. For example, f above is composed of five polynomial pieces. From a computational point of view, evaluating a piecewise polynomial is not much harder than evaluating a polynomial. If one thinks in terms of C or C++ code, then there is simply an extra `if-else` ladder to determine the appropriate subinterval and corresponding polynomial.

The piecewise polynomials to be used as blending functions must be chosen carefully though. For example, looking back at Propositions 17.1 and 17.3 of the last chapter, it's desirable for the set of blending functions to form a partition of unity over the parameter space. Good things happen then: (a) points on the curve (or surface) are convex combinations of its control points, so the whole lies in the convex hull of its control points and (b) affine invariance.

Writing down all the properties we want, then, we put together a Wish List for blending functions. We ask that they

(a) be at least a C^1-continuous piecewise polynomial,

(b) be of a low degree independent of the number of control points,

(c) each have support in only part of the parameter space, and,

(d) together form a partition of unity over the parameter space.

We're led to B-splines.

18.2 B-Spline Curves

Let's set the stage for the *B-spline blending functions* (or, as they are also called, *B-spline functions*, or *B-splines*, or *spline functions*) that we are going to define. Each will be piecewise polynomial, in other words, polynomial on subintervals. In anticipation, then, let's fix a particular parameter space and chop it up into subintervals. For convenience now, we choose $[0, r]$, where r is some positive integer, and its r subintervals to be the equally sized

$$[0, 1], \quad [1, 2], \quad \ldots, \quad [r-1, r]$$

See Figure 18.7. The sequence

$$\{0, 1, \ldots, r\}$$

Figure 18.7: Parameter space $[0, r]$ with uniformly-spaced knots.

of successive interval endpoints is called the *knot vector* and the endpoints $0, 1, \ldots, r$ themselves, *knots*. Each subinterval $[i, i+1]$, for $0 \le i \le r-1$, is a *knot interval*. We expect to define blending functions which are polynomial in each knot interval with (hopefully, well-behaved) joints at knot values.

Remark 18.1. A knot vector as above with equally spaced knots is called a *uniform* knot vector. Later in this chapter we'll see non-uniform knot vectors as well.

Remark 18.2. The "B" in B-splines, the name given these functions by Schoenberg [129], a pioneer in their use, comes from "basis".

18.2.1 First-Order B-Splines

We'll start at the lowest level possible and define the *B-splines of degree 0* by means of constant functions. There are r B-splines of degree 0, each equal to 1 on one knot interval and 0 outside it. Precisely, the ith B-spline of degree 0, for $0 \leq i \leq r-1$, denoted $N_{i,1}$, is defined as follows.

When $i = 0$:

$$N_{0,1}(u) = \begin{cases} 1, & 0 \leq u \leq 1 \\ 0, & \text{otherwise} \end{cases} \tag{18.3}$$

When $1 \leq i \leq r-1$:

$$N_{i,1}(u) = \begin{cases} 1, & i < u \leq i+1 \\ 0, & \text{otherwise} \end{cases} \tag{18.4}$$

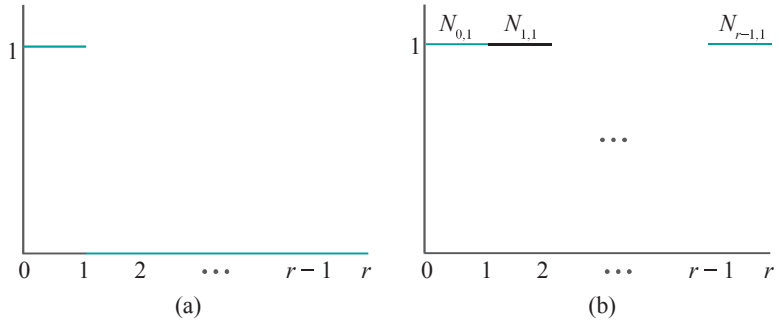


Figure 18.8: First-order B-splines: (a) $N_{0,1}$ (b) Non-zero parts of $N_{i,1}$, $0 \leq i \leq r-1$, distinguished by alternate green and black colors.

In other words, each $N_{i,1}$ is 1 on the knot interval $[i, i+1]$, except, possibly, at the endpoints, and 0 outside it; so, $N_{i,1}$ has support in $[i, i+1]$. Figure 18.8(a) shows the graph of $N_{0,1}$ over the entire parameter space $[0, r]$, while Figure 18.8(b) only the non-zero parts of the graphs of $N_{i,1}$, for $0 \leq i \leq r-1$. The niggling technicality – see the first line of the two equations above – of having to define $N_{0,1}$ to be 1 on a closed interval, while the other $N_{i,1}$ are equal to 1 on a half-open interval, is unavoidable. For, we want the r B-splines of degree 0 to form together a partition of unity over $[0, r]$, so no two are allowed to be 1 at the same point.

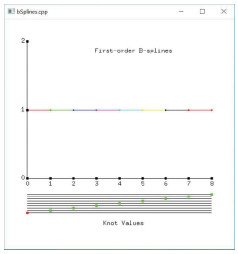


Figure 18.9: Screenshot of `bSplines.cpp` at first order.

Experiment 18.1. Run `bSplines.cpp`, which shows the non-zero parts of the spline functions from first order to cubic over the uniformly spaced knot vector

$$[0, 1, 2, 3, 4, 5, 6, 7, 8]$$

Press the up/down arrow keys to choose the order. Figure 18.9 is a screenshot of the first order. The knot values can be changed as well, but there's no need to now. **End**

B-splines of degree 0 are commonly called *first-order B-splines*. If the knot vector is uniform, as above, they are called *uniform first-order B-splines*.

Interestingly, as the reader may easily verify, all items on the Wish List at the end of Section 18.1 are fulfilled by the first-order B-splines, except for C^1-continuity, where, obviously, they fail badly because the $N_{i,1}$ are not even continuous (i.e., not even C^0-continuous). As we see next, expectedly this deficiency carries over to approximating curves made from first-order B-splines as well.

First-Order B-Spline Curves

A *first-order B-spline approximation* of r control points $P_0, P_1, \ldots, P_{r-1}$ is called a first order B-spline curve. This is the curve c obtained from applying the first-order B-splines as blending functions to these control points, namely,

$$c(u) = \sum_{i=0}^{r-1} N_{i,1}(u) P_i \qquad (0 \le u \le r) \tag{18.5}$$

What sort of a curve is c? Well, one would be hard pressed to call c a curve in the first place! Applying the definitions of $N_{i,1}$ from Equations (18.3)-(18.4) to Equation (18.5) above, one sees that c is *stationary* at P_0 for u from 0 to 1. When u crosses 1, c *jumps* to P_1, staying stationary again till u crosses 2, when c jumps to P_2, and so on. The graph of c is then just the collection of its own control points! See Figure 18.10. Obviously, if there are even two distinct control points then c is not C^0. Clearly, we'll have to move to higher orders of B-splines for satisfaction.

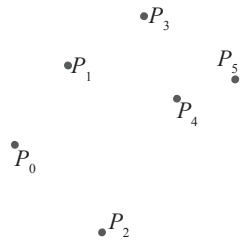


Figure 18.10: First-order B-spline approximation – the "curve" consists of its control points.

First-order B-Spline Properties

However, before leaving the first order, here are a few of their properties for future reference:

1. Each $N_{i,1}$ is piecewise polynomial, consisting of at most three pieces, each of which is constant.
2. $N_{i,1}$ has support in the single knot interval $[i, i+1]$.
3. Each $N_{i,1}$ is not C^0 only at the endpoints of its supporting interval; elsewhere, it's C^∞. In other words, it's smooth – remember from Definition 10.7 that C^∞ is also called smooth – apart from its joints.
4. Together, the $N_{i,1}$ form a partition of unity over the parameter space $[0, r]$.
5. Except for $N_{0,1}$, the $N_{i,1}$ are translates of one another, i.e., the graph of one is a translate of that of another. This is a consequence of the knots being uniformly spaced.
6. A first-order B-spline approximation is, generally, not even C^0.

18.2.2 Linear B-Splines

The clear problem with first-order B-splines is that their polynomial degree 0 is too low, allowing little flexibility in shape. Straight and horizontal is all that they can be. Let's go one higher to degree 1. We'll do this in a particular way which will be easy to generalize down the road.

The trivial formula

$$1 = u + (-u + 1) \tag{18.6}$$

allows one to "break" each B-spline $N_{i,1}$, of degree 0, into two functions $N^0_{i,1}$ and $N^1_{i,1}$ of degree 1. For example, $N_{0,1}$ breaks into $N^0_{0,1}$ and $N^1_{0,1}$, where

$$N^0_{0,1}(u) = \begin{cases} u, & 0 \le u \le 1 \\ 0, & \text{otherwise} \end{cases} \tag{18.7}$$

and

$$N^1_{0,1}(u) = \begin{cases} -u + 1, & 0 \le u \le 1 \\ 0, & \text{otherwise} \end{cases} \tag{18.8}$$

the two obviously adding to give back $N_{0,1}$, viz.,

$$N_{0,1}(u) = N^0_{0,1}(u) + N^1_{0,1}(u) \tag{18.9}$$

$N_{i,1}$, when $i > 0$, can likewise be broken into $N^0_{i,1}$ and $N^1_{i,1}$, where

$$N^0_{i,1}(u) = \begin{cases} u - i, & i < u \le i+1 \\ 0, & \text{otherwise} \end{cases} \tag{18.10}$$

$$\text{and} \qquad N^1_{i,1}(u) = \begin{cases} -u + i + 1, & i < u \le i+1 \\ 0, & \text{otherwise} \end{cases} \tag{18.11}$$

which actually invokes an i-shift of (18.6), namely, $1 = (u - i) + (-u + i + 1)$. And, again

$$N_{i,1}(u) = N^0_{i,1}(u) + N^1_{i,1}(u) \tag{18.12}$$

Figure 18.11 shows the (non-zero pieces of the) two parts of each first-order B-spline. For obvious reasons, we call the $N^0_{i,1}$s "up" and the $N^1_{i,1}$s "down". The up parts are all left or right translates of one another, as are the down parts, except that the technicality that their values at the left end of the knot interval $[0, 1]$ are different from those at the left end of other knot intervals persists from first-order.

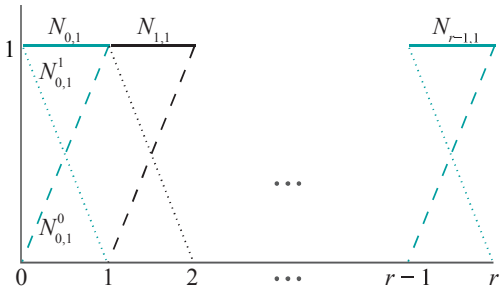


Figure 18.11: First-order B-splines each broken into an up part (dashed) $N^0_{i,1}$ and a down part (dotted) $N^1_{i,1}$. Successive $N_{i,1}$'s are distinguished by color.

Remark 18.3. This is important! For future reference, think of what we have just done as the following: each $N_{i,1}$ is broken into two new functions over its support, one obtained from multiplying $N_{i,1}$ by a *straight-line function increasing from 0 to 1* from the left end of its support to the right (namely, $u - i$), while the other from multiplying it by a *straight-line function decreasing from 1 to 0* over the same interval (namely, $-u + i + 1$). Because $(u - i) + (-u + i + 1) = 1$, the two new functions add to give back the one that was broken. And, of course, multiplication by such degree-1 functions causes the degree of the new functions to rise by 1 (in this case, from the degree 0 of $N_{i,1}$ to degree 1 of $N^0_{i,1}$ and $N^1_{i,1}$).

Equations (18.9) and (18.12) evidently guarantee that $N^0_{i,1}$ and $N^1_{i,1}$, for $0 \le i \le r - 1$, together form a partition of unity, because the $N_{i,1}$, $0 \le i \le r - 1$, do. However, there are $2r$ of the former, which is twice as many as we need to blend r control points. What to do? Figure 18.11, in fact, suggests a way to pair them up nicely – join each up part to the following down part! Accordingly, define the *second-order B-splines* (or *linear B-splines*), for $0 \le i \le r - 2$, as follows:

$$N_{i,2}(u) = N^0_{i,1}(u) + N^1_{i+1,1}(u) = \begin{cases} 0, & u \le i \\ u - i, & i \le u \le i+1 \\ -u + i + 2, & i+1 \le u \le i+2 \\ 0, & i+2 \le u \end{cases} \tag{18.13}$$

Figure 18.12 shows the non-zero parts of the linear B-splines $N_{i,2}$, $0 \le i \le r-2$, on the domain $[0, r]$. *See the magic*: pairing has removed all C^0-discontinuities because each linear B-splines drops to zero at the ends of its support, so is continuous everywhere.

Exercise 18.5. Verify that the multi-part formula above for $N_{i,2}(u)$ indeed follows from joining up and down parts (using the equations for $N^0_{i,1}$ and $N^1_{i,1}$ given earlier).

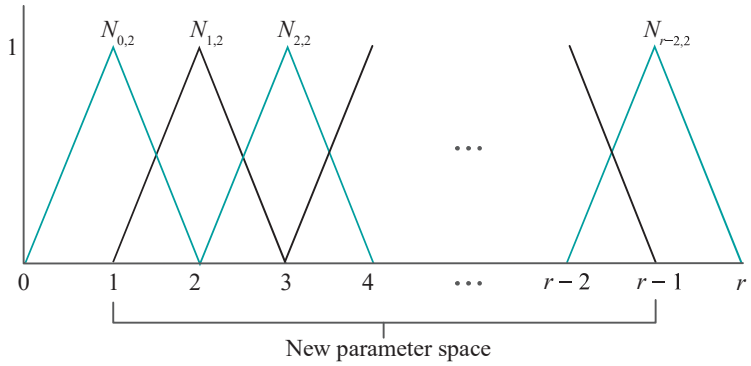


Figure 18.12: Non-zero parts of linear B-splines. Each is an inverted V. Successive ones are distinguished by color. The down part in the first knot interval and the up part in the last are discarded. The new (truncated) parameter space is $[1, r-1]$.

Experiment 18.2. Run again `bSplines.cpp` and select the linear B-splines over the knot vector

$$[0, 1, 2, 3, 4, 5, 6, 7, 8]$$

Figure 18.13 is a screenshot. **End**

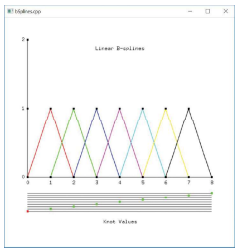


Figure 18.13: Screenshot of `bSplines.cpp` at second-order.

Remark 18.4. The technicality of the $N_{i,1}$ not being all of the same value at the left endpoint of a supporting interval is now gone. The definition of $N_{i,2}$ is the same for all i in $0 \le i \le r-2$.

Remark 18.5. Second-order B-splines as defined above are often called *uniform linear B-splines* to emphasize the use of a uniform knot vector.

Note that the down part $N^1_{0,1}$ of $N_{0,1}$ and the up part $N^0_{r-1,1}$ of $N_{r-1,1}$ have no partners, so have been discarded, which is why we have $r-1$ linear B-splines $N_{i,2}$, for $i = 0$ to $r-2$, versus the r first-order B-splines we started with. It's clear from Figure 18.12 that the parameter space must be truncated from $[0, r]$ to $[1, r-1]$ as well, for, otherwise, there's a problem with the partition-of-unity property in the two end knot intervals $[0, 1]$ and $[r-1, r]$. Once this is done, though, we're in good shape, or at least in significantly better shape than the first-order B-splines. All items in the Wish List at the end of Section 18.1 are now fulfilled except for C^1-continuity, but now the functions are at least C^0, if not quite C^1 (because of corners at the joints).

Linear B-Spline Curves

What sort of curve is the linear B-spline approximation c of $r-1$ control points $P_0, P_1, \ldots, P_{r-2}$, which uses the linear B-splines as blending functions? It's defined by

$$c(u) = \sum_{i=0}^{r-2} N_{i,2}(u) P_i \qquad (1 \le u \le r-1) \tag{18.14}$$

and we'll ask the reader next to see what this gives.

Exercise 18.6. Verify that the linear B-spline approximation c given by Equation (18.14) is the polygonal line through the control points in the sequence they are given. See Figure 18.14, where $r = 7$. This is certainly more respectable a curve than the first-order approximation.

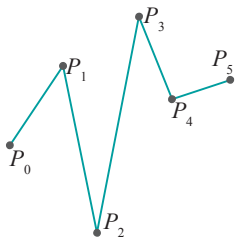


Figure 18.14: Linear B-spline approximation – the curve is a polyline.

Terminology: A B-spline approximation of a sequence of control points is often called a *B-spline curve*, a *spline curve* or, simply, a *spline*. There is ambiguity sometimes, therefore, with the terminology for B-spline blending functions, but it'll be clear from the context if the term refers to a blending function or an approximating curve.

Linear B-Spline Properties

Here's a list of properties of linear B-splines similar to the one made earlier for first-order B-splines:

1. Each $N_{i,2}$ is piecewise polynomial, consisting of at most four pieces, each of which is linear, except for zero end pieces.
2. $N_{i,2}$ has support in $[i, i+2]$, the union of two consecutive knot intervals.
3. Each $N_{i,2}$ is C^0, but not C^1, at its joints. Apart from its joints it's smooth everywhere.
4. Together, the $N_{i,2}$ form a partition of unity over the parameter space $[1, r-1]$.
5. The $N_{i,2}$ are translates of one another.
6. A linear B-spline approximation is C^0, but, generally, not C^1.

18.2.3 Quadratic B-Splines

Linear B-splines are certainly preferable to first-order ones, but we're still shy of C^1-continuity. If we could raise the degree of the polynomial pieces yet again, from 1 to 2, we might do better.

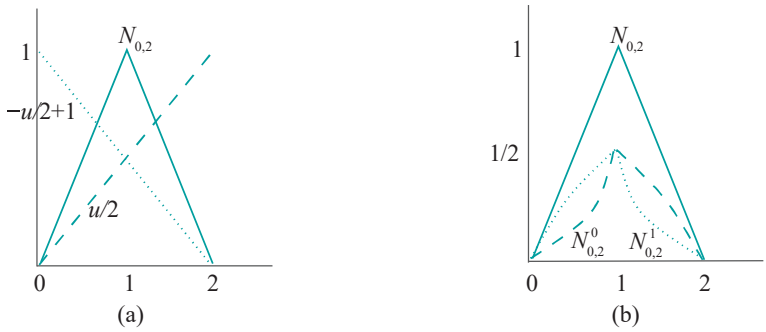


Figure 18.15: (a) The graphs of the two straight-line multiplying functions for $N_{0,2}$, one dashed and one dotted (b) The result of the multiplication: the up part $N^0_{0,2}$ (dashed) and the down part $N^1_{0,2}$ (dotted).

It turns out that the approach introduced in the last section of breaking first-order B-splines into up and down parts of one higher degree, and then pairing them up to make linear B-splines, generalizes. Consider first $N_{0,2}$, whose non-zero part is graphed in Figure 18.15(a). Recalling Remark 18.3, to break $N_{0,2}$ into two we'll multiply it by a straight-line function increasing from 0 at the left end of its support to 1 at the right, as well as by the complementary function decreasing from 1 to 0. Since the supporting interval of $N_{0,2}$ is $[0, 2]$, the two straight-line functions called for are $u/2$ and $-u/2+1$, respectively, which are shown in Figure 18.15(a) as well.

Accordingly, break $N_{0,2}$ as follows:

$$N_{0,2} = \frac{u}{2} N_{0,2} + (-\frac{u}{2} + 1) N_{0,2} \tag{18.15}$$

where the "up part" – it's not really increasing throughout any more but we'll stick with the term – is

$$N^0_{0,2}(u) = \frac{u}{2} N_{0,2} = \begin{cases} 0, & u \le 0 \\ \frac{1}{2}u^2, & 0 \le u \le 1 \\ -\frac{1}{2}u^2 + u, & 1 \le u \le 2 \\ 0, & 2 \le u \end{cases} \tag{18.16}$$

and the down part is

$$N^1_{0,2}(u) = (-\frac{u}{2} + 1)\, N_{0,2} = \begin{cases} 0, & u \le 0 \\ -\frac{1}{2}u^2 + u, & 0 \le u \le 1 \\ \frac{1}{2}u^2 - 2u + 2, & 1 \le u \le 2 \\ 0, & 2 \le u \end{cases} \tag{18.17}$$

The graphs of the two parts, resembling opposing shark fins, are shown in Figure 18.15(b).

Exercise 18.7. Verify the formulae for $N^0_{0,2}$ and $N^1_{0,2}$ by multiplying that for $N_{0,2}$ by $u/2$ and $-u/2+1$, respectively.

The other linear B-splines $N_{i,2}$, for $1 \le i \le r-2$, can similarly be broken. Figure 18.16 shows the graphs of the up and down parts.

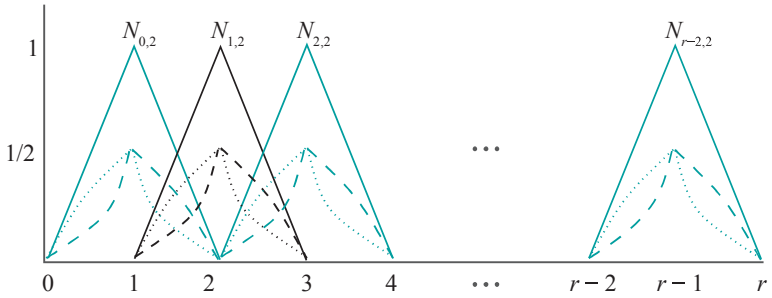


Figure 18.16: Linear B-splines each broken into an up (dashed) part $N^0_{i,2}$ and down (dotted) part $N^1_{i,2}$. Successive ones are distinguished by color.

Next, as in the first-order case, pair them up, adding each up part to the down part of the following linear B-spline. Non-zero pieces of paired up and down parts overlapped only at a point in the first-order case, so adding meant simply splicing graphs end to end. Now, we do actually have to add on interval overlaps.

And again magic! Two adjacent and opposing shark fins, one dashed and the other dotted, both with a sharp corner in the middle, add up to a smooth-looking floppy hat! See Figure 18.17. Precisely, the up part of one linear B-spline adds to the down part of the following one to make a *quadratic B-spline* (or, *third order B-spline*).

Figure 18.17 explains exactly what's happening. The graph of $N^0_{0,2}$ is green dashed, while that of $N^1_{1,2}$ black dotted. The graph $N_{0,3}$ of their sum consists of the outer green dashed arc on $[0,1]$, the outer black dotted arc on $[2,3]$ and the unbroken red arc on $[1,2]$ in the middle, the latter being the sum of the inner green dashed and the inner black dotted. So it's in the middle interval $[1,2]$ that actual summing takes place. We'll see the summed equation itself momentarily.

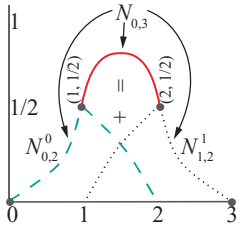


Figure 18.17: Adding $N^0_{0,2}$ and $N^1_{1,2}$ to make $N_{0,3}$. $N_{0,3}$ consists of three parts: on $[0,1]$ it's just $N^0_{0,2}$, on $[2,3]$ it's $N^1_{1,2}$, while in the middle, on $[1,2]$ it is the sum of $N^0_{0,2}$ and $N^1_{1,2}$. Joints are indicated.

Experiment 18.3. Run again `bSplines.cpp` and select the quadratic B-splines over the knot vector

$$[0, 1, 2, 3, 4, 5, 6, 7, 8]$$

Figure 18.18 is a screenshot. Note the joints indicated as black points. **End**

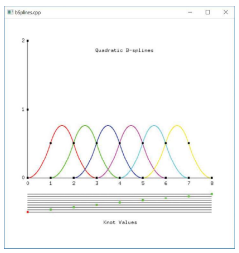


Figure 18.18: Screenshot of `bSplines.cpp` at third order.

$N_{0,3}$ is the first quadratic B-spline. Figure 18.19 depicts the sequence of quadratic B-splines $N_{i,3}$, $0 \le i \le r-3$, on the domain $[0,r]$. Now, for their equations. As they are evidently translates of one another, it's sufficient to write only that of the first one:

$$N_{0,3}(u) = N^0_{0,2}(u) + N^1_{1,2}(u) = \begin{cases} 0, & u \le 0 \\ \frac{1}{2}u^2, & 0 \le u \le 1 \\ \frac{3}{4} - (u - \frac{3}{2})^2, & 1 \le u \le 2 \\ \frac{1}{2}(-u+3)^2, & 2 \le u \le 3 \\ 0, & 3 \le u \end{cases} \tag{18.18}$$

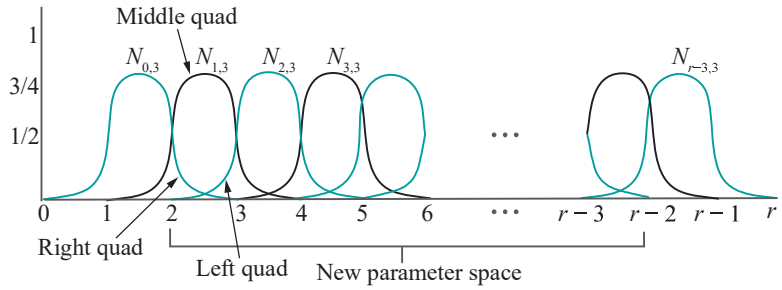


Figure 18.19: Non-zero parts of the quadratic B-splines. Successive splines are distinguished by color. The pieces adding up to 1 on $[2, 3]$ are indicated.

Exercise 18.8. Verify the preceding formula with the help of (18.16) and (18.17). Don't forget to shift the second equation to the right for the formula for $N^1_{1,2}$.

Exercise 18.9. Use Equation (18.18) to determine the equation of $N_{1,3}(u)$ and, generally, $N_{i,3}(u)$.

Exercise 18.10. Verify that the first quadratic B-spline $N_{0,3}$ is C^1 *everywhere* by differentiating the functions on the RHS of (18.18) and comparing the tangents on either side at each joint (which is only where discontinuity might occur). The four joints of $N_{0,3}$, with x-values 0, 1, 2 and 3, are indicated in Figure 18.17.

Differentiating again, verify that $N_{0,3}$ is *not* C^2 at its joints.

As the quadratic B-splines are translates one of one another, it follows from the preceding exercise that they are all C^1 everywhere, though not C^2 at their joints.

Remark 18.6. Compare the 5-line formulas (18.2) and (18.18) to see that we've come now full circle back to almost the same piecewise quadratic blending function which we used to motivate B-splines in the first place!

As in the linear case, the parameter space must be truncated, this time to $[2, r-2]$, to ensure that the partition-of-unity property holds. The key to keep in mind is that partition-of-unity holds in those knot intervals on which there is defined a left, a middle and a right quadratic arc – from successive quadratic B-splines (Figure 18.19 shows these three pieces over the interval $[2, 3]$).

Pop the champagne: we now officially have every item on the Wish List!

Quadratic B-Spline Curves

So what sort of curve is the quadratic B-spline approximation c of $r-2$ control points $P_0, P_1, \ldots, P_{r-3}$, defined by

$$c(u) = \sum_{i=0}^{r-3} N_{i,3}(u) P_i \qquad (2 \le u \le r-2) \tag{18.19}$$

where the quadratic B-splines are used as blending functions?

First, and importantly, since the quadratic B-splines are all C^1, so is a quadratic B-spline approximation. We've gained at least respectable continuity then. However, as we ask the reader to show next, the property of interpolating the first and last control points has been lost (though not on our Wish List, this, nevertheless, is desirable).

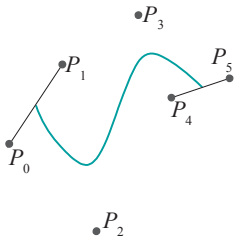


Figure 18.20: Quadratic B-spline approximation.

Exercise 18.11. Prove that the quadratic spline curve c defined by Equation (18.19) begins at the midpoint of P_0P_1, ends at the midpoint of $P_{r-4}P_{r-3}$, and doesn't necessarily interpolate *any* of its control points. See Figure 18.20.

Darn, just when we thought things were going our way, a potentially nasty bug rears its ugly head. Not to worry, as soon as we are able to loosen up the knot vector from being uniform, we'll be happily interpolating first and last control points.

Experiment 18.4. Run `quadraticSplineCurve.cpp`, which shows the quadratic spline approximation of nine control points in 2D space over a uniformly spaced vector of 12 knots. Figure 18.21 is a screenshot.

The control points are green. Press the space bar to select a control point – the selected one turns red – and the arrow keys to move it. The knots are the green points on the black bars at the bottom. At this stage there is no need to change their values. The blue points are the joints of the curve, i.e., images of the knots. Also drawn in light gray is the control polygon.

Ignore the code itself for now. We'll be seeing how to draw spline curves and surfaces using OpenGL soon. **End**

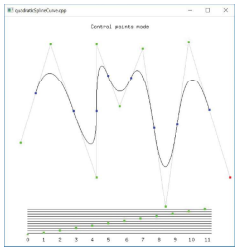


Figure 18.21: Screenshot of `quadraticSplineCurve.cpp`.

Exercise 18.12. What part of the quadratic spline curve c approximating the control points $P_0, P_1, \ldots, P_{r-3}$ is altered by moving only P_i? Your answer should be in terms of an arc of c between a particular pair of its joints. Verify using `quadraticSplineCurve.cpp`.

Quadratic B-Spline Properties

A list of properties for quadratic B-splines:

1. Each $N_{i,3}$ is piecewise polynomial, consisting of at most five pieces, each of which is quadratic, except for zero end pieces.
2. $N_{i,3}$ has support in $[i, i+3]$, the union of three consecutive knot intervals.
3. Each $N_{i,3}$ is C^1, but not C^2, at its joints. Apart from its joints it's smooth everywhere.
4. Together, the $N_{i,3}$ form a partition of unity over the parameter space $[2, r-2]$.
5. The $N_{i,3}$ are translates of one another.
6. A quadratic B-spline approximation is C^1, but, generally, not C^2.

When placing it on our Wish List, we expected to be rewarded for the partition-of-unity property by felicitous behavior of the B-spline approximating curves. The reader is asked to show next that indeed we are.

Exercise 18.13.

(a) Prove that the quadratic spline curve approximating a sequence of control points lies in the convex hull of the latter.

(b) Affine invariance: prove that an affine transformation of a quadratic spline curve is the same as the quadratic spline curve approximating the transformed control points.

18.2.4 Cubic B-Splines

We're going to ask you to do most of the lifting in this section.

To start with, break the first quadratic B-spline $N_{0,3}$ into two parts: an "up" part obtained from multiplying it by a straight-line function increasing from 0 at the left end of its support to 1 at the right end and a "down" part from multiplying it by the complementary function decreasing from 1 to 0 over its support. Here's the equation showing the split:

$$N_{0,3} = \frac{u}{3}\, N_{0,3} + (-\frac{u}{3} + 1)\, N_{0,3} \qquad (18.20)$$

Exercise 18.14. Write equations for the up part

$$N^0_{0,3}(u) = \frac{u}{3} N_{0,3}$$

and the down part

$$N^1_{0,3}(u) = (-\frac{u}{3} + 1) N_{0,3}$$

in a manner analogous to Equations (18.16) and (18.17) for the quadratic B-splines. Both up and down parts are piecewise cubic.

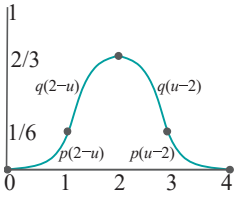


Figure 18.22: The first cubic B-spline function $N_{0,4}$. Joints are indicated.

Exercise 18.15. Verify by adding $N^0_{0,3}(u)$ and $N^1_{1,3}(u)$ that the equation of the first cubic B-spline is:

$$N_{0,4}(u) = \begin{cases} 0, & u \le 0 \\ p(2-u), & 0 \le u \le 1 \\ q(2-u), & 1 \le u \le 2 \\ q(u-2), & 2 \le u \le 3 \\ p(u-2), & 3 \le u \le 4 \\ 0, & 4 \le u \end{cases} \tag{18.21}$$

where the functions p and q are given by:

$$p(u) = \frac{1}{6}(2-u)^3$$

and

$$q(u) = \frac{1}{6}(3u^3 - 6u^2 + 4)$$

See Figure 18.22.

Exercise 18.16. Verify that cubic B-splines are C^2, but not C^3, at their joints.

Exercise 18.17. Sketch the sequence of cubic B-splines $N_{i,4}$, for $0 \le i \le r-4$, over $[0, r]$ similarly to Figure 18.19 for quadratic B-splines. What should be the new parameter range?

Experiment 18.5. Run `bSplines.cpp` and change the order to see a sequence of cubic B-splines. **End**

Cubic B-Spline Curves

The cubic spline curve c approximating $r-3$ control points $P_0, P_1, \ldots, P_{r-4}$ is obtained as

$$c(u) = \sum_{i=0}^{r-4} N_{i,4}(u) P_i \qquad (3 \le u \le r-3) \tag{18.22}$$

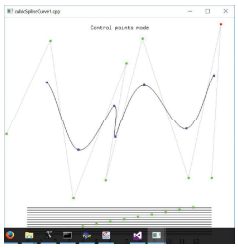


Figure 18.23: Screenshot of `cubicSplineCurve1.cpp`.

Experiment 18.6. Run `cubicSplineCurve1.cpp`, which shows the cubic spline approximation of nine control points in 2D space over a uniformly-spaced vector of 13 knots. The program's functionality is similar to that of `quadraticSplineCurve.cpp`. See Figure 18.23 for a screenshot.

The control points are green. Press the space bar to select a control point – the selected one is colored red – then the arrow keys to move it. The knots are the green points on the black bars at the bottom. The blue points are the joints of the curve. The control polygon is a light gray. **End**

Exercise 18.18. Prove that a cubic spline curve doesn't necessarily interpolate any of its control points. See again Exercise 18.11 and say now where a cubic spline curve starts w.r.t. its control points and where it ends.

Cubic B-Spline Properties

A list of properties for cubic B-splines:

1. Each $N_{i,4}$ is piecewise polynomial, consisting of at most six pieces, each of which is cubic, except for zero end pieces.
2. $N_{i,4}$ has support in $[i, i+4]$, the union of four consecutive knot intervals.
3. Each $N_{i,4}$ is C^2, but not C^3, at its joints. Apart from its joints it's smooth everywhere.
4. Together, the $N_{i,4}$ form a partition of unity over the parameter space $[3, r-3]$.
5. The $N_{i,4}$ are translates of one another.
6. A cubic B-spline approximation is C^2, but, generally, not C^3.

Remark 18.7. Cubic B-splines are the most commonly used in design applications, because they offer the best trade-off between continuity and computational efficiency.

18.2.5 General B-Splines and Non-uniform Knot Vectors

It's probably evident now how to manufacture B-splines of arbitrary order over the uniform knot vector $\{0, 1, \ldots, r\}$. One would apply the *break-and-make* procedure to B-splines of each order to derive ones of one higher order. We formalize the derivation of B-splines of arbitrary order over $\{0, 1, \ldots, r\}$ recursively as follows:

Definition 18.1. The first-order B-splines $N_{i,1}$, $0 \le i \le r-1$, are as defined in Section 18.2.1:

$$N_{0,1}(u) = \begin{cases} 1, & 0 \le u \le 1 \\ 0, & \text{otherwise} \end{cases} \tag{18.23}$$

and for $1 \le i \le r-1$

$$N_{i,1}(u) = \begin{cases} 1, & i < u \le i+1 \\ 0, & \text{otherwise} \end{cases} \tag{18.24}$$

Suppose, recursively, that the B-splines $N_{i,m-1}$, for $0 \le i \le r-m+1$, have been defined for some order $m-1 \ge 1$. Then define the ith B-spline $N_{i,m}$ of order m, for $0 \le i \le r-m$, by the equation:

$$N_{i,m}(u) = \left(\frac{u-i}{m-1}\right) N_{i,m-1}(u) \; + \; \left(\frac{i+m-u}{m-1}\right) N_{i+1,m-1}(u) \tag{18.25}$$

It's not hard to see that the inductive formula (18.25) comes from a straightforward application of break-and-make. The summand

$$\left(\frac{u-i}{m-1}\right) N_{i,m-1}(u)$$

is the up part of $N_{i,m-1}(u)$ obtained from multiplying it by the straight-line function $(u-i)/(m-1)$ increasing from 0 at i, the left end of its support, to 1 at $i+m-1$, the right end.

Likewise, the summand

$$\left(\frac{i+m-u}{m-1}\right) N_{i+1,m-1}(u)$$

is the down part of $N_{i+1,m-1}(u)$ obtained from multiplying it by the straight-line function $(i+m-u)/(m-1)$ decreasing from 1 to 0 from the left end $i+1$ to the right $i+m$ of its support.

Terminology: The reader will have noted the convention that the *degree* of a B-spline is that of its polynomial pieces, while its *order* is its degree plus one.

Exercise 18.19. A tacit assumption in the discussion above was that the support of $N_{i,m-1}$ is in the interval $[i, i+m-1]$. Prove this is true by induction.

Exercise 18.20. Make a six-point list of properties for uniform B-splines of the mth order like the ones earlier for uniform lower-order splines.

Before proceeding further, though, we are going to loosen restrictions on the knot vector, which till now had been the uniform sequence

$$\{0, 1, \ldots, r\}$$

Keep in mind that the operative word is *uniform*, in particular, that knots are equally spaced; it does not matter that they are integers. For instance, if a knot vector were of the form

$$\{a, a+\delta, a+2\delta, \ldots, a+r\delta\}$$

for some a, and some $\delta > 0$, e.g.,

$$\{1.3, 2.8, 4.3, \ldots, 1.3+1.5r\}$$

all calculations made so far would clearly go through again, though with different (and awkward) number values, and all properties of B-splines deduced previously would hold, too.

The restriction of uniformity is removed by allowing the knot vector to be any sequence of knots of the form

$$T = \{t_0, t_1, \ldots, t_r\}$$

where the t_i are *non-decreasing*, i.e.,

$$t_0 \le t_1 \le \ldots \le t_r \tag{18.26}$$

Such knot vectors are called *non-uniform*. Yes, successive knots can even be equal and such so-called multiple knots have important applications, as we'll see.

Remark 18.8. The term non-uniform knot vector is a little unfortunate in that it actually means *not necessarily* uniform, because a uniform knot vector evidently satisfies (18.26) as well.

Hmm, do we start afresh working our way up from first-order splines, this time around over non-uniform knot vectors? Not at all. Pretty much all our earlier discussions go through again, including break-and-make. Without further ado then, here's the recursive definition of B-splines over non-uniform knot vectors.

Definition 18.2. Let

$$T = \{t_0, t_1, \ldots, t_r\} \tag{18.27}$$

be a non-uniform knot vector, where $r \ge 1$.

The (non-uniform) first-order B-spline functions $N_{i,1}$, for $0 \le i \le r-1$, are defined as follows:

$$N_{0,1}(u) = \begin{cases} 1, & t_0 \le u \le t_1 \\ 0, & \text{otherwise} \end{cases} \tag{18.28}$$

and for $1 \le i \le r-1$

$$N_{i,1}(u) = \begin{cases} 1, & t_i < u \le t_{i+1} \\ 0, & \text{otherwise} \end{cases} \tag{18.29}$$

The (non-uniform) mth order B-spline functions $N_{i,m}$, where the order m lies within $1 < m \le r$, and the index i in $0 \le i \le r-m$, are recursively defined by:

$$N_{i,m}(u) \;=\; \left(\frac{u-t_i}{t_{i+m-1}-t_i}\right) N_{i,m-1}(u) \;+\; \left(\frac{t_{i+m}-u}{t_{i+m}-t_{i+1}}\right) N_{i+1,m-1}(u) \tag{18.30}$$

Note: The convention to follow in case the denominator of either of the two fractional terms is 0 – which may occur if there are equal knots – is the following: if the term is of the form $\frac{0}{0}$, then declare its value to be 1; if it is of the form $\frac{a}{0}$, where a is not 0, then declare its value to be 0.

This recursive formula (18.30), discovered by Cox, de Boor and Mansfield independently in 1972, known accordingly as the Cox-de Boor-Mansfield (CdM) formula or recurrence, was an important milestone in B-spline theory. However, it's really straightforward for us to understand now, given our development of the topic so far.

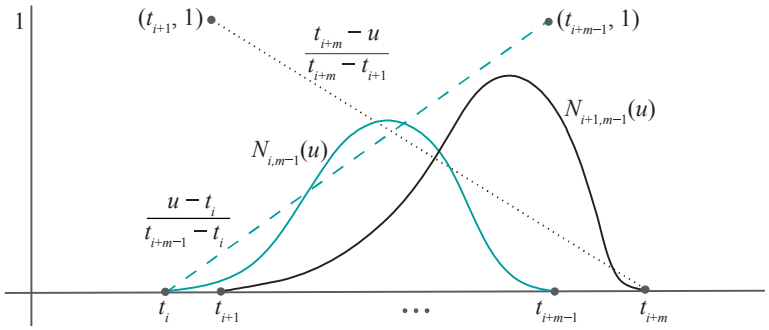


Figure 18.24: Graphs of the functions on the RHS of Equation (18.30): $N_{i,m-1}$ and $N_{i+1,m-1}$ and their respective linear multipliers $\frac{u-t_i}{t_{i+m-1}-t_i}$ and $\frac{t_{i+m}-u}{t_{i+m}-t_{i+1}}$.

Equations (18.28) and (18.29), respectively, replicate, with obvious changes, (18.23) and (18.24) for first-order B-splines over a uniform knot vector. Equation (18.30) imitates (18.25). It formalizes break-and-make – the summands are the up and down parts, respectively, of two successive spline functions of one lower order. Figure 18.24 shows graphs of all four functions on the RHS of Equation (18.30).

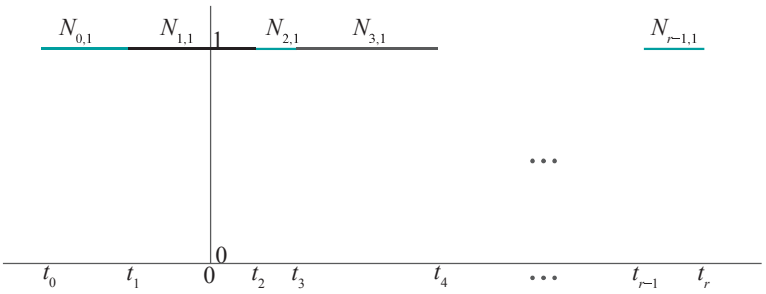


Figure 18.25: Non-zero parts of the first-order B-splines over a non-uniform knot vector.

Figure 18.25 shows the graphs of the first-order B-splines over a non-uniform knot vector, while Figure 18.26 those of linear B-splines over the same knot vector.

The equations of the spline functions themselves are a little more complicated than in the case of integer knots for the simple reason that they now involve variables for knot values. For example, here's the equation, analogous to (18.18), for the first quadratic B-spline over a non-uniform knot vector:

$$N_{0,3}(u) = \begin{cases} 0, & u \le t_0 \\ \frac{u-t_0}{t_2-t_0}\,\frac{u-t_0}{t_1-t_0}, & t_0 \le u \le t_1 \\ \frac{u-t_0}{t_2-t_0}\,\frac{t_2-u}{t_2-t_1} + \frac{t_3-u}{t_3-t_1}\,\frac{u-t_1}{t_2-t_1}, & t_1 \le u \le t_2 \\ \frac{t_3-u}{t_3-t_1}\,\frac{t_3-u}{t_3-t_2}, & t_2 \le u \le t_3 \\ 0, & t_3 \le u \end{cases} \tag{18.31}$$

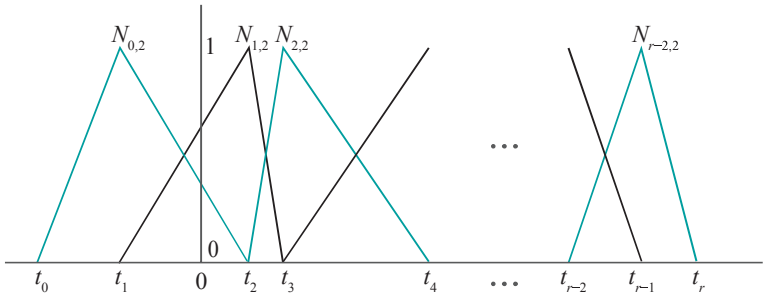


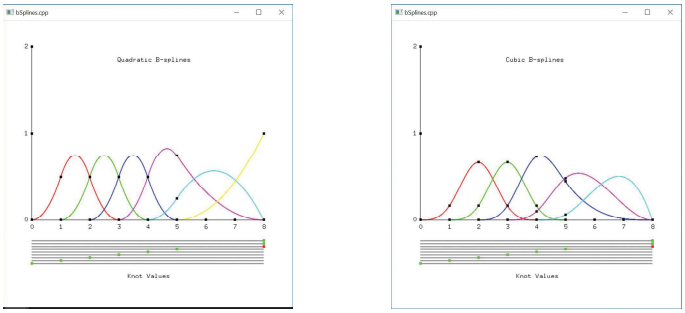
Figure 18.26: Non-zero parts of the linear B-splines over a non-uniform knot vector.

Not pretty, but B-spline computations are almost invariably done recursively in code, so a formula like this rarely needs to be written explicitly.

Experiment 18.7. Run again `bSplines.cpp`. Change the knot values by selecting one with the space bar and then pressing the left/right arrow keys. Press delete to reset knot values. Note that the routine `Bspline()` implements the CdM formula (and its convention for 0 denominators).

In particular, observe the quadratic and cubic spline functions. Note how they lose their symmetry about a vertical axis through the center, and that no longer are they translates of one another.

Play around with making knot values equal – we'll soon be discussing the utility of multiple knots. Figures 18.27(a) and (b) are screenshots of the quadratic and cubic functions, respectively, both over the same non-uniform knot vector with a triple knot at the right end. **End**



(a) (b)

Figure 18.27: Screenshots of `bSplines.cpp` over a non-uniform knot vector with a triple knot at the right end: (a) Quadratic (b) Cubic.

Example 18.1. Find the values of (a) $N_{3,3}(5)$ and (b) $N_{4,3}(5)$, if the knot vector is $\{0, 1, 2, 3, 4, 5, 5, 5, 6, 7, \ldots\}$, the non-negative integers, except that 5 has multiplicity three.

Answer: The successive knot values are

$$t_0 = 0,\ t_1 = 1,\ t_2 = 2,\ t_3 = 3,\ t_4 = 4,\ t_5 = 5,\ t_6 = 5,\ t_7 = 5,\ t_8 = 6,\ t_9 = 7,\ \ldots$$

(a) Instantiating the CdM formula (18.30):

$$N_{3,3}(u) \quad = \quad \frac{u - t_3}{t_5 - t_3}\, N_{3,2}(u) \ + \ \frac{t_6 - u}{t_6 - t_4}\, N_{4,2}(u)$$

Plugging in $u = 5$ and the given knot values:

$$N_{3,3}(5) = \frac{5-3}{5-3} N_{3,2}(5) + \frac{5-5}{5-4} N_{4,2}(5) = N_{3,2}(5) \tag{18.32}$$

Using CdM again,

$$N_{3,2}(u) = \frac{u-t_3}{t_4-t_3} N_{3,1}(u) + \frac{t_5-u}{t_5-t_4} N_{4,1}(u)$$

so that

$$\begin{aligned} N_{3,2}(5) &= \frac{5-3}{4-3} N_{3,1}(5) + \frac{5-5}{5-4} N_{4,1}(5) \\ &= 2*0 + 0*1 \quad \text{(from Equations (18.28) and (18.29))} \\ &= 0 \end{aligned}$$

Taking the above back to (18.32) we have

$$N_{3,3}(5) = 0$$

(b)

$$N_{4,3}(u) = \frac{u-t_4}{t_6-t_4} N_{4,2}(u) + \frac{t_7-u}{t_7-t_5} N_{5,2}(u)$$

giving

$$\begin{aligned} N_{4,3}(5) &= \frac{5-4}{5-4} N_{4,2}(5) + \frac{5-5}{5-5} N_{5,2}(5) \\ &= N_{4,2}(5) + \frac{0}{0} N_{5,2}(5) \\ &= N_{4,2}(5) + N_{5,2}(5) \quad \text{(using convention } \tfrac{0}{0} = 1) \end{aligned} \tag{18.33}$$

Using CdM again,

$$N_{4,2}(u) = \frac{u-t_4}{t_5-t_4} N_{4,1}(u) + \frac{t_6-u}{t_6-t_5} N_{5,1}(u)$$

so that

$$\begin{aligned} N_{4,2}(5) &= \frac{5-4}{5-4} N_{4,1}(5) + \frac{5-5}{5-5} N_{5,1}(5) \\ &= 1*1 + 1*0 \quad \text{(note by (18.29) that } N_{5,1} \text{ is zero everywhere)} \\ &= 1 \end{aligned} \tag{18.34}$$

CdM again gives

$$N_{5,2}(u) = \frac{u-t_5}{t_6-t_5} N_{5,1}(u) + \frac{t_7-u}{t_7-t_6} N_{6,1}(u)$$

implying

$$\begin{aligned} N_{5,2}(5) &= \frac{5-5}{5-5} N_{5,1}(5) + \frac{5-5}{5-5} N_{6,1}(5) \\ &= 1*0 + 1*0 \\ &= 0 \end{aligned} \tag{18.35}$$

Using (18.34) and (18.35) in (18.33) we have

$$N_{4,3}(5) = 1$$

Exercise 18.21. Find the values of $N_{5,3}(5)$ and $N_{6,3}(5)$ for the same knot vector as in the preceding example.

Exercise 18.22. Compute $N_{4,3}(7)$ again over the knot vector of the preceding example. You will have to invoke the convention that $\frac{a}{0} = 0$, if a is not 0.

General B-Spline Curves

The mth order B-spline approximation c of $r-m+1$ control points $P_0, P_1, \ldots, P_{r-m}$ is the curve obtained by applying the mth order B-splines as blending functions. Its equation is:

$$c(u) = \sum_{i=0}^{r-m} N_{i,m}(u) P_i \qquad (t_{m-1} \leq u \leq t_{r-m+1}) \tag{18.36}$$

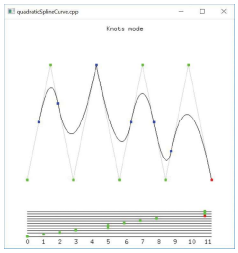


Figure 18.28: Screenshot of `quadraticSplineCurve.cpp` with a double knot at 5 and a triple knot at 11.

Experiment 18.8. Run again `quadraticSplineCurve.cpp`. Press 'k' to enter knots mode and alter knot values using the left/right arrow keys and 'c' to return to control points mode. Press delete in either mode to reset.

Try to understand what happens if knots are repeated. Do you notice a loss of C^1-continuity when knots in the interior of the knot vector coincide? What if knots at the ends coincide? Figure 18.28 is a screenshot of `quadraticSplineCurve.cpp` with a double knot at 5 and a triple at the end at 11. **End**

Exercise 18.23. Can you find an arrangement of the knots for the quadratic spline curve to interpolate its first and last control points?

Exercise 18.24. Why does changing the value of only the first, or only the last knot, not affect the quadratic spline curve?

Exercise 18.25. (Programming) Run again `cubicSplineCurve1.cpp`. Press 'k' to enter knots mode and alter knot values using the left/right arrow keys and 'c' to return to control points mode. Press delete in either mode to reset.

Can you find an arrangement of the knots so that the cubic spline curve interpolates its first and last control points?

Exercise 18.26. What part of the mth order spline curve c approximating the control points $P_0, P_1, \ldots, P_{r-m}$ is altered by moving only P_i? Your answer should be in terms of an arc of c between a particular pair of its joints.

We collect information about mth order B-spline functions and their corresponding approximating spline curves in the following proposition.

Proposition 18.1. *Let*

$$T = \{t_0, t_1, \ldots, t_r\}$$

be a non-uniform knot vector, where $r \geq 1$.

The m*th order B-spline functions* $N_{i,m}$*, for some order* m *lying within* $1 \leq m \leq r$*, and, where* $0 \leq i \leq r-m$*, satisfy the following properties:*

(a) Each $N_{i,m}$ *is piecewise polynomial, consisting of at most* $m+2$ *pieces, each of which is a degree* $m-1$ *polynomial, except possibly for zero end pieces.*

(b) $N_{i,m}$ *has support in* $[t_i, t_{i+m}]$*, the union of* m *consecutive knot intervals.*

(c) If the knots in T *are distinct, each* $N_{i,m}$ *is* C^{m-2}*, but not* C^{m-1}*, at its joints. In this case, apart from its joints, each* $N_{i,m}$ *is smooth everywhere.*

(d) The $N_{i,m}$ *together form a partition of unity over the parameter space* $[t_{m-1}, t_{r-m+1}]$.

(e) Every point of the m*th order B-spline approximation* c *of* $r-m+1$ *control points* $P_0, P_1, \ldots, P_{r-m}$*, defined by Equation (18.36), over the parameter space* $[t_{m-1}, t_{r-m+1}]$*, is a convex combination of the control points and lies inside their convex hull.*

(f) (Affine Invariance) If $g : \mathbb{R}^3 \to \mathbb{R}^3$ *is an affine transformation, and* c *is the* m*th order B-spline approximation of* $r-m+1$ *control points* $P_0, P_1, \ldots, P_{r-m}$ *in* $\mathbb{R}^3$*, then the image curve* $g(c)$ *is the* m*th order B-spline approximation of the images* $g(P_0), g(P_1), \ldots, g(P_{r-m})$ *of the control points.*

(g) *If the knots in T are distinct, the mth order B-spline approximation c of $r-m+1$ control points $P_0, P_1, \ldots, P_{r-m}$ defined by Equation (18.36) is C^{m-2}, but, generally, not C^{m-1}.*

Proof. The proofs are a straightforward technical slog and we'll not write them out. □

The following relation for a B-spline curve is useful to remember:

$$\textit{number of knots} = \textit{number of control points} + \textit{order} \tag{18.37}$$

Exercise 18.27. Deduce (18.37).
Hint: Count the number of knots and control points in (18.36).

Non-uniform Knot Vectors

So, of what use are non-uniform knot vectors?

One is to be able to control the influence that a control point has over an approximating curve. For example, consider the cubic spline curve c approximating control points $P_0, P_1, \ldots$ over the knot vector $\{t_0, t_1, \ldots\}$, as in Figure 18.29(a), which shows a few intermediate control points. Moving control point, say, P_5 alters only the arc of c between $a = c(t_5)$ and $b = c(t_9)$, as $N_{5,4}$ has support in $[t_5, t_9]$. Consequently, the closer or farther apart are the knots from t_5 to t_9, the more concentrated or diffuse the influence of P_5. This generalizes, of course, to all P_i, allowing the designer to vary the domain of influence of control points by rearranging knots.

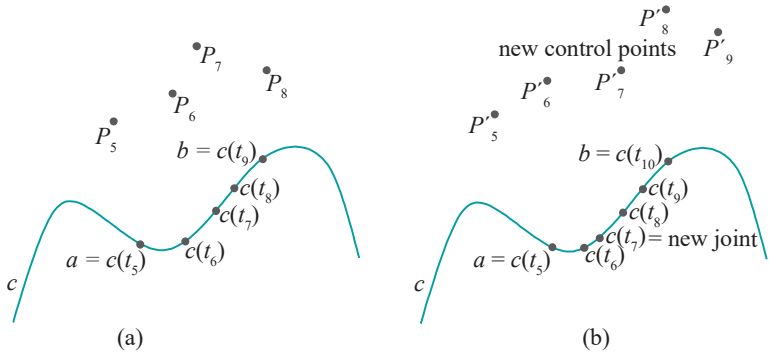


Figure 18.29: (a) Part of a cubic spline curve (b) With a new knot inserted.

Another practical consequence of non-uniform knot vectors is the technique of *knot insertion*, implemented in many commercial modelers, to allow the designer increasingly fine control over part of a spline curve. Clearly, the more knot images (joints, that is) there are in an arc of a curve, the more control points have influence over it and, therefore, the more finely it can be edited. Refer again to Figure 18.29(a). Currently, the shape of the arc between a and b is determined by the four control points P_5, P_6, P_7 and P_8. If one could insert a new knot, say, between t_6 and t_7 *without* changing the shape of the curve, there would then be five control points, instead of four, acting upon the same arc, affording the designer an added level of control.

Knots can, in fact, be inserted without changing either the shape of a spline curve or its degree, though, with a newly computed set of control points. See Figure 18.29(b), where a new knot has been inserted between t_6 and t_7, giving rise to a corresponding new joint. The joints have been re-labeled in sequence and a (hypothetical) new set of control points shown; now, in fact, the arc of the spline curve between a and b is shaped by the five control points $P'_5, P'_6, \ldots, P'_9$, not four as before. We'll not go into the theory of knot insertion ourselves, referring the reader instead to more mathematical texts such as Buss [21], Farin [45] and Piegl & Tiller [113].

Multiple Knots

Coincident knots – *multiple knots* and *repeated knots* are the terms most commonly used – have a particularly useful application.

We'll motivate our discussion with a running example using the knot vector

$$T = \{t_0{=}0,\ t_1{=}1,\ t_2{=}2,\ t_3{=}3,\ t_4{=}3,\ t_5{=}4,\ t_6{=}5,\ t_7{=}6,\ \ldots\}$$

which has a double knot at $t_3 = t_4 = 3$. Generally, the *multiplicity* of a knot is the number of times it repeats.

The graphs of some of the B-spline functions over T are shown in Figure 18.30.

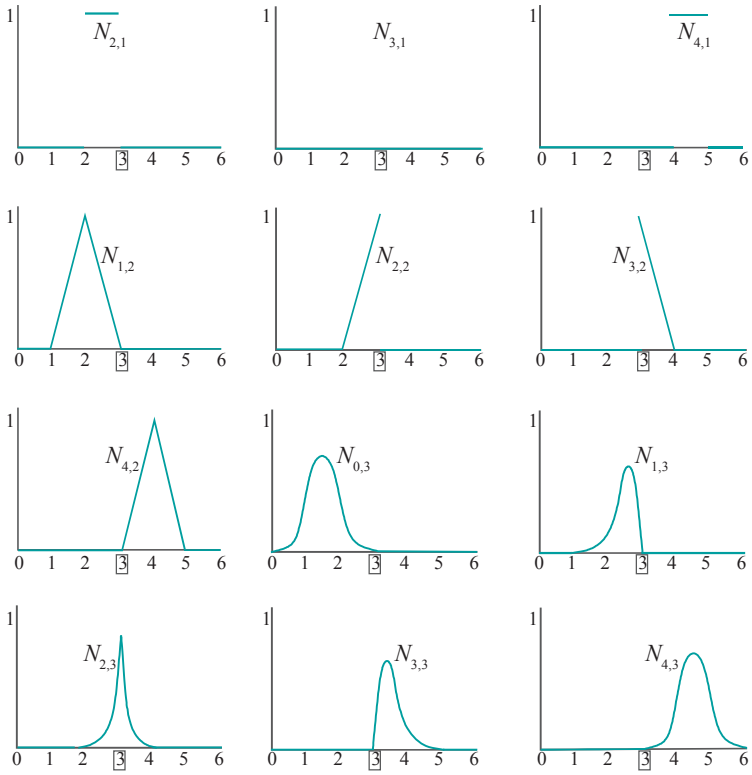


Figure 18.30: B-spline functions over the knot vector $T = \{0, 1, 2, 3, 3, 4, 5, \ldots\}$ with a double knot at 3 (distinguished inside a box).

Exercise 18.28. Verify that the graphs of the first-order B-splines over T are correctly depicted in the top row of Figure 18.30 by applying the defining Equations (18.28) and (18.29). In particular, the first-order B-splines are all 1 on their supporting intervals, excluding possibly endpoints, and 0 elsewhere, *except* for $N_{3,1}$, which is 0 throughout.

Exercise 18.29. Derive the equations of the linear B-splines from the first-order ones – by plugging $m = 2$ into the recursive Equation (18.30) – to verify their graphs in the second row of Figure 18.30, as well as at the leftmost in the third. In particular, the linear B-splines over T are all C^0 and translates of one another, *except* for $N_{2,2}$ and $N_{3,2}$, neither of which is C^0.

Unfortunately, the artifact of vertical edges in the display when knots coincide makes it tricky to use `bSplines.cpp` to visually verify the linear B-spline graphs in

Figure 18.30. However, there is no such issue with quadratic B-splines, so we ask the reader to do the following.

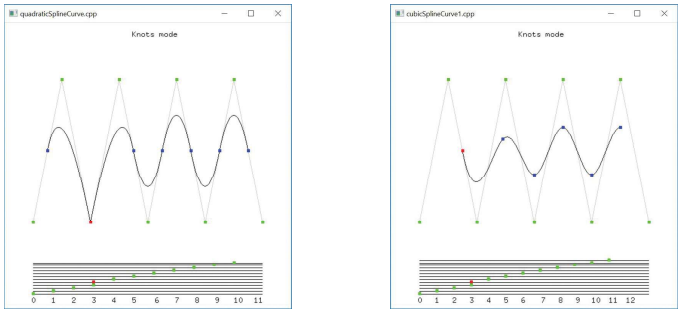
Exercise 18.30. (Programming) Arrange the knots of `bSplines.cpp` to make their nine successive values 0, 1, 2, 3, 3, 4, 5, 6 and 7, which are the first few knots of T. Then verify visually the graphs of the five quadratic B-splines in Figure 18.30. In fact, all the quadratic B-splines over T are C^1 and translates of one another, *except* for $N_{1,3}$, $N_{2,3}$ and $N_{3,3}$, which are C^0 but not C^1.

Next, we investigate the behavior of the approximating B-spline curve in the presence of repeated knot values.

Exercise 18.31. Use Equation (18.36) and the graphs already drawn of the first-order and linear spline functions over T to verify that the first-order and linear spline curves approximating nine control points – arranged, alternately, in two horizontal rows – are correctly drawn in Figures 18.31(a) and (b), respectively.

In particular, the first-order approximation loses the control point P_3 (drawn hollow) altogether, while the linear approximation loses the segment P_2P_3 and, therefore, is no longer C^0.

Experiment 18.9. Use the programs `quadraticSplineCurve.cpp` and `cubicSplineCurve1.cpp` to make the quadratic and cubic B-spline approximations over the knot vector $T = \{0, 1, 2, 3, 3, 4, 5, 6, 7, \ldots\}$ of nine control points placed as in Figure 18.31(a) (or (b)). See Figure 18.32(a) and (b) for screenshots of the quadratic and cubic curves, respectively.



(a) (b)

Figure 18.32: Screenshots of (a) `quadraticSplineCurve.cpp` and (b) `cubicSplineCurve1.cpp` over the knot vector $T = \{0, 1, 2, 3, 3, 4, 5, 6, 7, \ldots\}$ and approximating nine control points arranged in two horizontal rows.

The quadratic approximation loses C^1-continuity precisely at the control point P_2, which it now *interpolates* as the curve point $c(3)$. It's still C^0 everywhere.

It's not easy to discern visually, but the cubic spline drops from C^2 to C^1-continuous at $c(3)$. **End**

Let's see next what happens with even higher multiplicity.

Experiment 18.10. Continuing with `cubicSplineCurve1.cpp` with control points as in the preceding experiment, press delete to reset and then make equal t_4, t_5 and t_6, creating a triple knot at 4. Figure 18.33 is a screenshot of this configuration. Evidently, the control point P_3 is now interpolated at the cost of a drop in continuity there to mere C^0. Elsewhere, the curve is still C^2. **End**

It seems, generally, that repeating a knot increases the influence of a particular control point, to the extent that if the repetition is sufficient then that control point itself is interpolated, though at the cost of continuity at the control point itself. This

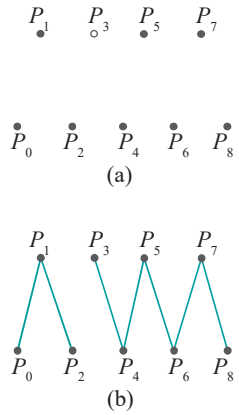


Figure 18.31: (a) First-order and (b) linear spline curves over the knot vector $T = \{0, 1, 2, 3, 3, 4, 5, 6, 7, \ldots\}$, approximating nine control points arranged alternately in two horizontal rows. The (hollow) control point P_3 is the only one missing from the first-order "curve", which consists of the remaining eight points. The second-order curve is the polyline $P_0P_1 \ldots P_8$ *minus* P_2P_3.

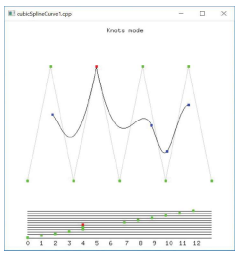
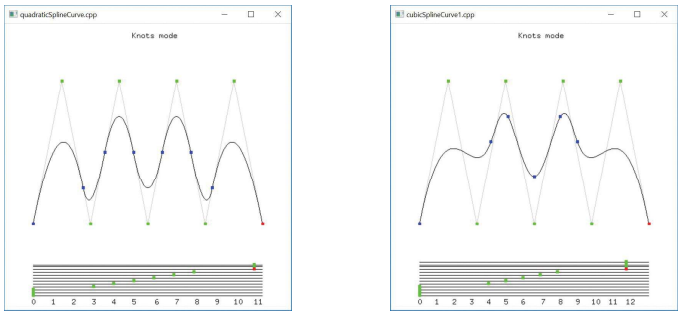


Figure 18.33: Screenshot of `cubicSplineCurve1.cpp` with a triple knot at 4.

does not appear to be a particularly appealing trade-off unless a low-continuity artifact, e.g., a corner, is itself a design goal.

Let's examine more closely how the loss arises – evidently, because of the difference in the value of the derivative (of some order) of c *on either side* of a control point P. For example, the tangents to the arcs on either side of the interpolated control point P_2 of the quadratic spline curve in Figure 18.32(a) are different.

Consider now if P were an *endpoint* of c. Then continuity cannot be lost by derivatives differing on the two sides of P, for the simple reason that the curve is only to one side! And, yet, there is no reason why the influence of P cannot still be increased by repeating knots. We are on our way to recovering the property of interpolating end control points that was lost at first by quadratic spline curves.



(a) (b)

Figure 18.34: Screenshots of (a) `quadraticSplineCurve.cpp` and (b) `cubicSplineCurve1.cpp`, both with knots repeated at the end to interpolate the first and last control points.

Experiment 18.11. Make the first three and last three knots separately equal in `quadraticSplineCurve.cpp` (Figure 18.34(a)). Make the first four and last four knots separately equal in `cubicSplineCurve1.cpp` (Figure 18.34(b)). Are the first and last control points interpolated in both. *Yes*. Do you notice any impairment in continuity? *No*. **End**

Generally, if the first m and last m knots of an mth order spline curve are coincident, and there are no other multiple knots, then the curve interpolates its first and last control points without losing C^{m-2}-continuity anywhere. In fact, a knot vector which starts and ends with a multiplicity of m and whose intermediate knots are uniformly spaced is called a *standard knot vector*. From formula (18.37), the size of a standard knot vector is the sum of the number of control points and the order of the spline curve. E.g., a standard knot vector for a quadratic spline with nine control points is

$$\{0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 7, 7\}$$

Exercise 18.32. Jot down a standard knot vector for a quadratic spline over 10 control points and for a cubic spline over 9 control points.

Exercise 18.33. Use the CdM formula to show that $N_{0,3}(t_2) = 1$ over the standard knot vector

$$T = \{0,\ 0,\ 0,\ 1,\ 2,\ \ldots,\ r-6,\ r-5,\ r-5,\ r-5\}$$

of size r for a quadratic spline. Use this to prove that the quadratic spline

$$c(u) = \sum_{i=0}^{r-3} N_{i,3}(u) P_i \qquad (t_2 = 0 \le u \le r-5 = t_{r-2})$$

approximating the $r-m+1$ control points P_i, $0 \le i \le r-m$, over T indeed interpolates the first one, in particular, $c(t_2) = P_0$.

For the record here's a proposition:

Proposition 18.2. *A spline curve over a standard knot vector interpolates its first and last control points.*

Proof. The proof is a generalization of the preceding exercise to establish that the first control point is always interpolated. We'll leave the reader to do this by an induction. That the last control point is interpolated as well follows by symmetry. □

The use of a standard knot vector for splines bequeaths yet another Bézier-like property – recall Proposition 17.1(f) – in addition to the interpolation of the end control points:

Proposition 18.3. *The tangent lines at the endpoints of a spline curve over a standard knot vector each pass through the adjacent control point.*

Proof. We'll prove this for quadratic splines in the next example. The general proof is not difficult but tedious, and we'll leave it to the motivated reader to do on her own. □

Example 18.2. Prove that the tangent lines at the endpoints of a quadratic spline curve over a standard knot vector each pass through the adjacent control point.

Answer: We'll show that the tangent vector at the first control point passes through the second. The result at the other end follows by symmetry.

For quadratic splines, the standard knot vector is

$$T = \{0,\, 0,\, 0,\, 1,\, 2,\, \ldots\}$$

The quadratic spline curve approximating the control points $P_0, P_1, P_2, \ldots$ is

$$c(u) = N_{0,3}(u)P_0 + N_{1,3}(u)P_1 + N_{2,3}(u)P_2 + N_{3,3}(u)P_3 + \ldots$$

Now, the blending functions $N_{i,3}$, for $i \geq 3$, all vanish in $[t_2, t_3] = [0, 1]$. Consequently, in $[0, 1]$:

$$c(u) = N_{0,3}(u)P_0 + N_{1,3}(u)P_1 + N_{2,3}(u)P_2$$

Plugging the standard knot vector values into formula (18.31) for $N_{0,3}$ we get

$$N_{0,3}(u) = 1 - 2u + u^2, \quad u \in [0, 1]$$

One can use (18.31) to determine $N_{1,3}(u)$ as well by incrementing the subscripts on its RHS by 1. This gives

$$N_{1,3}(u) = 2u - \frac{3}{2}u^2, \quad u \in [0, 1]$$

Likewise, it's found that

$$N_{2,3}(u) = \frac{1}{2}u^2, \quad u \in [0, 1]$$

Therefore,

$$c(u) = (1 - 2u + u^2)\, P_0 + (2u - \frac{3}{2}u^2)\, P_1 + (\frac{1}{2}u^2)\, P_2, \quad u \in [0, 1]$$

Differentiating,

$$c'(u) = (-2 + 2u)\, P_0 + (2 - 3u)\, P_1 + u\, P_2, \quad u \in [0, 1]$$

Plugging in $u = 0$, one sees that

$$c'(0) = 2(P_1 - P_0)$$

which is indeed in the direction from P_0 to P_1.

We see it's for good reason, therefore, that standard knot vectors are most often used in B-spline design.

Exercise 18.34. Proposition 18.1(e) says that a spline curve is contained in the convex hull of (all) its control points. Prove the stronger statement that a spline curve of order m can be divided into successive stretches that each lie in the convex hull of only some m of its control points.

Bézier Curves and Spline Curves

It turns out that Bézier curves are special cases of spline curves:

Proposition 18.4. *The $(n+1)$th order Bézier curve approximating the $n+1$ control points*

$$P_0, P_1, \ldots, P_n$$

coincides with the $(n+1)$th order spline curve approximating the same control points over the particular standard knot vector

$$\{0, 0, \ldots, 0, 1, 1, \ldots, 1\}$$

consisting of $n+1$ 0's followed by $n+1$ 1's.

Proof. In the following example we'll restrict ourselves to establishing the quadratic case, leaving the general proof by induction to the mathematically inclined reader. □

Example 18.3. Show that the quadratic Bézier curve approximating the three control points P_0, P_1 and P_2 coincides with the quadratic spline curve approximating the same control points over the particular standard knot vector $\{0, 0, 0, 1, 1, 1\}$.

Answer: Recall from the previous chapter that the Bézier curve approximating P_0, P_1 and P_2 is

$$c_B(u) = (1-u)^2 P_0 + 2(1-u)uP_1 + u^2 P_2, \quad u \in [0,1]$$

The quadratic spline approximating the same three points over the knot vector $T = \{t_0{=}0,\ t_1{=}0,\ t_2{=}0,\ t_3{=}1,\ t_4{=}1,\ t_5{=}1\}$ is

$$c_S(u) = N_{0,3}(u)P_0 + N_{1,3}(u)P_1 + N_{2,3}(u)P_2, \quad u \in [t_2, t_3] = [0,1]$$

Therefore, we must show that spline blending functions of the preceding equation match the Bernstein polynomial blending functions of the one before it, over the knot interval $[0,1]$. Refer to formula (18.31) for $N_{0,3}$. The fourth line on the RHS gives

$$\begin{aligned} N_{0,3}(u) &= \frac{t_3 - u}{t_3 - t_1}\,\frac{t_3 - u}{t_3 - t_2} \\ &= (1-u)^2 \end{aligned}$$

after plugging in the knot values $t_0 = t_1 = t_2 = 0$ and $t_3 = 1$ in the interval $t_2 = 0 \le u \le 1 = t_3$, confirming a match with the first Bernstein polynomial.

We can use (18.31) for $N_{1,3}$ as well, making sure to increment the subscripts on the RHS by 1. This gives

$$N_{1,3}(u) = \frac{u - t_1}{t_3 - t_1}\,\frac{t_3 - u}{t_3 - t_2} + \frac{t_4 - u}{t_4 - t_2}\,\frac{u - t_2}{t_3 - t_2}$$

in $u \in [0,1]$, using $t_1 = t_2 = 0$ and $t_3 = t_4 = 1$, so matching the second Bernstein polynomial. We'll leave the reader to verify that $N_{2,3}(u) = u^2$, $u \in [0,1]$, completing the proof.

In the opposite direction, the following is true because spline curves are piecewise polynomial (from the way they are constructed) and polynomial curves are Bézier (from Proposition 17.2).

Proposition 18.5. *A spline curve is piecewise Bézier.* □

Exercise 18.35. Why is it not possible that the preceding proposition can somehow be strengthened to say that spline curves are, in fact, Bézier entirely, not just piecewise? *Hint*: Bézier curves are smooth throughout.

18.3 B-Spline Surfaces

The construction of B-spline surfaces as a continuum of B-spline curves parallels exactly the construction of Bézier surfaces from Bézier curves described in Section 17.2. See Figure 18.35 for the following.

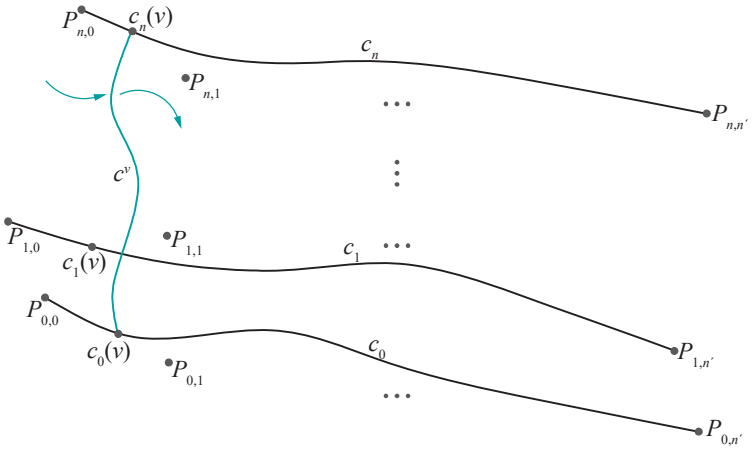


Figure 18.35: Constructing the B-spline surface approximating an array of control points by sweeping a B-spline curve. The B-spline curves depicted all interpolate both end control points, which need not always be the case in practice.

Suppose that we are given an $(n+1) \times (n'+1)$ array of control points

$$P_{i,j}, \quad \text{for} \quad 0 \le i \le n, \ 0 \le j \le n'$$

and two spline orders m and m', and a knot vector

$$T = \{t_0, t_1, \ldots, t_r\}, \quad \text{whose size satisfies} \quad |T| = r + 1 = n + 1 + m$$

(to ensure that *number of knots* = *number of control points* + *order*) and another knot vector

$$T' = \{t_0, t_1, \ldots, t'_r\}, \quad \text{whose size satisfies} \quad |T'| = r' + 1 = n' + 1 + m'$$

Think of the control points array as $n+1$ different sequences, each of $n'+1$ control points. In particular, the ith sequence, for $0 \le i \le n$, consists of $P_{i,0}, P_{i,1}, \ldots, P_{i,n'}$, lying along the ith row of the control points array. Construct the m'th order B-spline curve c_i, for $0 \le i \le n$, approximating the control points sequence $P_{i,0}, P_{i,1}, \ldots, P_{i,n'}$, each using the knot vector T' over the parameter space $[t_{m'-1}, t_{r'-m'+1}]$.

For each v in $t_{m'-1} \le v \le t_{r'-m'+1}$, generate the mth order B-spline curve c^v approximating the control points sequence $c_0(v), c_1(v), \ldots, c_n(v)$, using the knot vector T over the parameter space $[t_{m-1}, t_{r-m+1}]$. The union of all these B-spline curves c^v, for $t_{m'-1} \le v \le t_{r'-m'+1}$, then, is the B-spline surface s approximating the control points array $P_{i,j}$, $0 \le i \le n$, $0 \le j \le n'$. One may imagine s as being swept by c^v, as v varies from $t_{m'-1}$ to $t_{r'-m'+1}$.

Exercise 18.36. Prove that the parametric equation of the B-spline surface s constructed as above is

$$s(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{n'} N_{i,m}^{T}(u) N_{j,m'}^{T'}(v) P_{i,j} \tag{18.38}$$

for $t_{m-1} \leq u \leq t_{r-m+1}$ and $t_{m'-1} \leq v \leq t_{r'-m'+1}$, and where $N^T_{i,m}$ (respectively, $N^{T'}_{j,m'}$) denotes the B-spline function $N_{i,m}$ over the knot vector T (respectively, $N_{j,m'}$ over T').

In other words, the surface is obtained from applying the blending function $N^T_{i,m}(u)N^{T'}_{j,m'}(v)$ to the control point $P_{i,j}$, over the parameter domain $t_{m-1} \leq u \leq t_{r-m+1}$, $t_{m'-1} \leq v \leq t_{r'-m'+1}$.
Hint: Mimic the proof of (17.16) for a Bézier surface.

Exercise 18.37. Formulate an analogue for B-spline surfaces of Proposition 18.1 for curves.

We've given thus far an account of NURBS (non-uniform rational B-spline) theory, *except* for the 'R', or rational, part. Instead of generally rational, being a ratio of two polynomials, our functions have been all just polynomial. You could say that we have covered NUPBS, or simply NUBS, as the default for B-splines is polynomial. We'll put the 'R' into NURBS in Chapter 20 with the help of projective spaces.

18.4 Drawing B-Spline Curves and Surfaces

NURBS – the full-blown rational version of B-splines – curves and surfaces are implemented in the GLU library of OpenGL. Now that we have a fair amount of the theory, the GLU NURBS interface will turn out to be simple to use, as the mapping between theory and syntax is almost one-to-one. We'll, of course, restrict ourselves to polynomial B-spline primitives for now, leaving the rational ones to a later chapter.

18.4.1 B-Spline Curves

We had already used OpenGL to draw polynomial B-spline curves in the programs `quadraticSplineCurve.cpp` and `cubicSplineCurve1.cpp` earlier this chapter, without caring then about the drawing syntax itself. Let's look at this now.

The command

```
gluNurbsCurve(*nurbsObject, knotCount, *knots, stride, *controlPoints,
              order, type)
```

defines a B-spline curve which is pointed by *nurbsObject*. The parameter *knotCount* is the number of knots in the knot vector – a one-dimensional array – pointed by *knots*. The parameter *order* is the order of the spline curve, *controlPoints* points to the one-dimensional array of control points, and *stride* is the number of floating point values between the start of the data set for one control point and that of the next in the control points array. The number of control points is not explicitly specified, but computed by OpenGL with the help of (18.37):

$$number\ of\ control\ points = number\ of\ knots - order$$

The parameter *type* is `GL_MAP1_VERTEX_3` or `GL_MAP1_VERTEX_4`, according as the spline curve is polynomial or rational.

A `gluNurbsCurve()` command must be bracketed between a `gluBeginCurve()`-`gluEndCurve()` pair of statements. The following statements from the drawing routine of `quadraticSplineCurve.cpp`, defining a quadratic B-spline curve approximating nine control points, should now be clear:

```
gluBeginCurve(nurbsObject);
gluNurbsCurve(nurbsObject, 12, knots, 3, ctrlpoints[0], 3,
              GL_MAP1_VERTEX_3);
gluEndCurve(nurbsObject);
```

Exercise 18.38. Refer to Section 10.3.1 for the syntax of the call `glMap1f()` defining a Bézier curve and compare it with that of `gluNurbsCurve()`.

There are certain initialization steps to be completed prior to a `gluNurbsCurve()` call. First, `gluNewNurbsRenderer()` returns the pointer to a NURBS object, which is passed to the subsequent `gluNurbsCurve()` call. Then optional `gluNurbsProperty()` calls control the quality of the rendering, as well as other related attributes of the curve. We refer the reader to the red book for a complete listing of possible parameter values for `gluNurbsProperty()`. Our own usage is kept to a simple minimum – the relevant statements from the `setup()` routine of `quadraticSplineCurve.cpp` are the following:

```
nurbsObject = gluNewNurbsRenderer();
gluNurbsProperty(nurbsObject, GLU_SAMPLING_METHOD, GLU_PATH_LENGTH);
gluNurbsProperty(nurbsObject, GLU_SAMPLING_TOLERANCE, 10.0);
```

The last two statements specify that the longest length of a line segment in a strip approximating a NURBS curve (or that of a quad edge, in the case of a mesh approximating a NURBS surface) is at most 10.0 pixels.

Experiment 18.12. Change the last parameter of the statement

```
gluNurbsProperty(nurbsObject, GLU_SAMPLING_TOLERANCE, 10.0);
```

in the initialization routine of `quadraticSplineCurve.cpp` from 10.0 to 100.0. The fall in resolution is noticeable as one sees in Figure 18.36. End

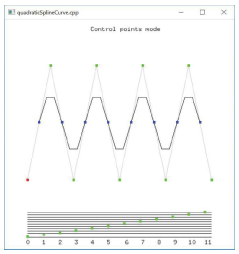


Figure 18.36: Screenshot of `quadraticSplineCurve.cpp` with sampling tolerance increased to 100.

If you are wondering whether a B-spline curve can be drawn in a manner similar to that using `glMapGrid1f()` followed by `glEvalMesh1()` for a Bézier curve – sampling the curve uniformly through the parameter domain – the answer is yes. Though we shall not use them ourselves the two requisite calls for this purpose are `gluNurbsProperty(`*`nurbsObject, GLU_SAMPLING_METHOD, GLU_DOMAIN_DISTANCE)` and `gluNurbsProperty(`*`nurbsObject, GLU_U_STEP, value)`. The reader is referred to the red book for implementation details.

Experiment 18.13. Run `cubicSplineCurve2.cpp`, which draws the cubic spline approximation of 30 movable control points, initially laid out on a circle, over a fixed standard knot vector. Press space and backspace to cycle through the control points and the arrow keys to move the selected control point. The delete key resets the control points. Figure 18.37 is a screenshot of the initial configuration. The number of control points being much larger than the order, the user has good local control.

Incidentally, note how managing large numbers of control points has been made efficient with B-splines. Together 30 control points would have led to a 29th degree polynomial Bézier curve, a computational nightmare; alternatively we could split the control points into smaller sets, e.g., of size 4 for cubic curves, but then would come the issue of smoothly joining the successive sub-curves. End

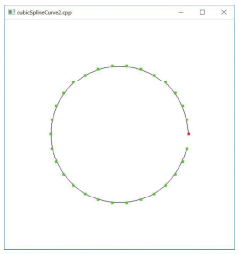


Figure 18.37: Screenshot of `cubicSplineCurve2-.cpp`.

Exercise 18.39. (Programming) Use `cubicSplineCurve2.cpp` to draw a closed loop like the one in Figure 18.38.

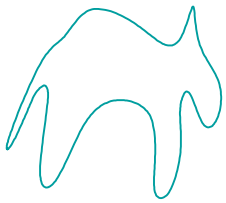


Figure 18.38: A cat or whatever.

18.4.2 B-Spline Surfaces

The OpenGL syntax for a B-spline surface is a straightforward extension of that for a B-spline curve. The `gluNurbsSurface()` command, which must be bracketed between a `gluBeginSurface()-gluEndSurface()` pair of statements, has the following form:

`gluNurbsSurface(`*`*nurbsObject, uknotCount, *uknots, vknotCount, *vknots, ustride, vstride, *controlPoints, uorder, vorder, type`)

`*vknots` points to the knot vector used with the control point row, in other words, to make the parameter curves c_i in the discussion in Section 18.3 of a B-spline curve sweeping a surface; *`*uknots`* points to the knot vector used with the control point columns, i.e., to make the curves c^v in that discussion.

The parameter *vknotCount* is the number of knots in the vector pointed by **vknots*, *vorder* is the order of the B-spline curves c_i and *vstride* is the number of floating point values between the data set for one control point and the next in a row of the control points array. The parameters *uknotCount*, *uorder* and *ustride* represent similar values for the control point columns.

The parameter *type* is `GL_MAP2_VERTEX_3` or `GL_MAP2_VERTEX_4` for polynomial or rational surfaces, respectively; it can have other values as well to specify surface normals and texture coordinates.

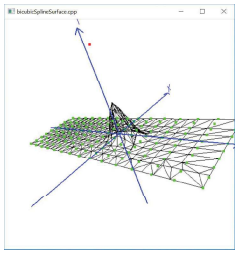


Figure 18.39: Screenshot of `bicubicSplineSurface.cpp`.

Experiment 18.14. Run `bicubicSplineSurface.cpp`, which draws a spline surface approximation to a 15×10 array of control points, each of which the user can move in 3-space. The spline is cubic in both parameter directions and a standard knot vector is specified in each as well.

Press the space, backspace, tab and enter keys to select a control point. Move the selected control point using the arrow and page up and down keys. The delete key resets the control points. Press 'x/X', 'y/Y' and 'z/Z' to turn the surface. Figure 18.39 is a screenshot. End

Exercise 18.40. (Programming) Use `bicubicSplineSurface.cpp` to draw separately a hilly terrain and a boat.

18.4.3 Lighting and Texturing a B-Spline Surface

Lighting and texturing a B-spline surface is similar to doing likewise for a Bézier surface. Normals are required for lighting and the quickest way to create normals for a B-spline surface is to generate them automatically with a call, as for Bézier surfaces, to `glEnable(GL_AUTO_NORMAL)`.

And, again as for Bézier surfaces, determining texture coordinates for a B-spline surface requires, first, the creation of a "fake" B-spline surface in texture space on the same parameter rectangle as the real one – the reader should review if need be the discussion in Section 12.5 on specifying texture coordinates for a Bézier surfaces. OpenGL, subsequently, assigns as texture coordinates to the image on the real surface of a particular parameter point the image of that same point on the fake surface in texture space. Code will clarify.

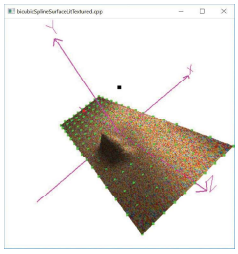


Figure 18.40: Screenshot of `bicubicSplineSurfaceLitTextured.cpp`.

Experiment 18.15. Run `bicubicSplineSurfaceLitTextured.cpp`, which sugar-coats the spline surface of `bicubicSplineSurface.cpp`. Figure 18.40 is a screenshot. The surface is illuminated by a single positional light source whose location is indicated by a large black point. User interaction remains as in `bicubicSplineSurface.cpp`. Note that pressing the 'x'-'Z' keys turns only the surface, not the light source.

The bicubic B-spline surface, as well as the fake bilinear one in texture space, are created by the following statements in the drawing routine:

```
gluBeginSurface(nurbsObject);
gluNurbsSurface(nurbsObject, 19, uknots, 14, vknots,
       30, 3, controlPoints[0][0], 4, 4, GL_MAP2_VERTEX_3);
gluNurbsSurface(nurbsObject, 4, uTextureknots, 4, vTextureknots,
       4, 2, texturePoints[0][0], 2, 2, GL_MAP2_TEXTURE_COORD_2);
gluEndSurface(nurbsObject);
```

We'll leave the reader to parse in particular the third statement and verify that it creates a "pseudo-surface" – a 10×10 rectangle – in texture space on the same parameter domain $[0, 12] \times [0, 7]$ as the real one. End

Exercise 18.41. (Programming) Light and texture the B-spline surfaces you created for Exercise 18.40.

18.4.4 Trimmed B-Spline Surface

A powerful design tool is to *trim* (i.e., excise or remove) part of a B-spline surface. Here, first, is what happens theoretically.

Say the parametric specification of a surface s is given to be

$$x = f(u, v),\ y = g(u, v),\ z = h(u, v), \quad \text{where} \quad (u, v) \in W = [u_1, u_2] \times [v_1, v_2]$$

The parametric equations map the rectangle W from uv-space onto the surface s in xyz-space. Moreover, a loop (closed curve) c on W maps to a loop c' on s. See Figure 18.41.

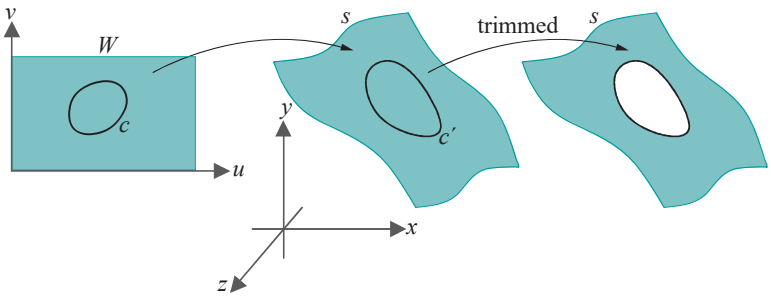


Figure 18.41: The loop c on the parameter space W is mapped to the loop c' on the surface s by the parametric equations for s. Then s is trimmed by c.

If the part of s inside, or outside, the loop c' is excised, then s is said to be trimmed by the loop c (probably, more accurate would be to say that it is trimmed by c', but the given usage is common). Figure 18.41 shows the inside trimmed. Loop c itself is called the *trimming loop*.

OpenGL allows B-spline surfaces to be trimmed. We use the program `bicubicBsplineSurfaceTrimmed.cpp`, as a running example to explain OpenGL syntax for trimming.

Experiment 18.16. Run `bicubicBsplineSurfaceTrimmed.cpp`, which shows the surface of `bicubicBsplineSurface.cpp` trimmed by multiple loops. The code is modified from the latter program, functionality remaining same. Figure 18.42(a) is a screenshot. **End**

All the code relevant to trimming is in the drawing routine:

```
gluBeginSurface(nurbsObject);
gluNurbsSurface(nurbsObject, 19, uknots, 14, vknots,
        30, 3, controlPoints[0][0], 4, 4, GL_MAP2_VERTEX_3);

gluBeginTrim(nurbsObject);
   gluPwlCurve(nurbsObject, 5, boundaryPoints[0], 2,
               GLU_MAP1_TRIM_2);
gluEndTrim(nurbsObject);

gluBeginTrim(nurbsObject);
   gluPwlCurve(nurbsObject, 11, circlePoints[0], 2,
               GLU_MAP1_TRIM_2);
gluEndTrim(nurbsObject);

gluBeginTrim(nurbsObject);
   gluNurbsCurve(nurbsObject, 10, curveKnots, 2, curvePoints[0], 4,
                 GLU_MAP1_TRIM_2);
gluEndTrim(nurbsObject);

gluEndSurface(nurbsObject);
```

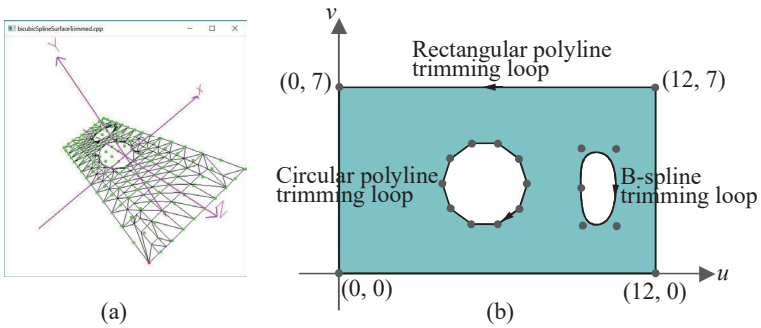


Figure 18.42: (a) Screenshot of `bicubicBsplineSurfaceTrimmed.cpp` (b) The three trimming loops – two polygonal and one B-spline.

Points to note:

1. Each trimming loop is defined within `gluBeginTrim()-gluEndTrim()` pair of statements, which itself must lie within the `gluBeginSurface()-gluEndSurface()` pair. The trimming loop definitions are located after the `gluNurbsSurface()` definition.

2. Each trimming loop must be a closed curve in the parameter space.

3. There are two ways to define a trimming loop:

   (a) As a polygonal line loop defined by a

   `glPwlCurve(`*`*nurbsObject, pointsCount, *pointsArray, stride, type`*`)`

   statement, where *pointsCount* is the number of vertices in an array of the form $\{v_0, v_1, \ldots, v_n\}$ pointed by *pointsArray* (it is required that $v_0 = v_n$). There are two such polyline trimming loops in the program (see Figure 18.42(b)):

   (i) The five vertices (first and last equal) of one are in the array `boundaryPoints`, describing the rectangular boundary of the parameter space itself, oriented *counter-clockwise*. We'll soon see why this particular bounding trimming loop is required.

   (ii) The eleven vertices (again, first and last equal) of the other are in the array `circlePoints`, equally spaced along a circle, oriented *clockwise*.

   (b) As a B-spline loop defined by

   `gluNurbsCurve(`*`nurbsObject, knotCount, *knots, stride, *controlPoints, order, type`*`)`

   In the program there is a single such B-spline trimming loop, whose six control points (first and last equal) are in the array `curvePoints` oriented *clockwise* (Figure 18.42(b)).

4. The part outside a trimming loop oriented counter-clockwise is trimmed, while that inside a trimming loop oriented clockwise is trimmed.

   Accordingly, the first trimming polyline loop of the program, which bounds the parameter space going counter-clockwise, trims off the *exterior* of the drawn surface, not trimming the surface itself per se. The other two trimming loops actually create holes in the surface.

Exercise 18.42. (Programming) Draw the forbidding terrain of an uninhabited planet with volcanoes, craters, lakes of lava, and such.

18.5 Summary, Notes and More Reading

We have learned a fair amount of the theory underlying the widely-used class of 3D design primitives – B-splines, both curves and surfaces. Emphasis was on motivating each new concept. We did *not* want to pull stuff out of a hat. A test if we were successful is for the reader to deduce some formula, e.g., (18.18) for the first quadratic B-spline $N_{0,3}$ over a uniform knot vector or the Cox-de Boor-Mansfield recurrence (18.30), using just pencil and paper, and not referring again to the text. This chapter prepares the reader, as well, for the rational version of the theory – NURBS – coming up in Chapter 20.

As for OpenGL, we learned not only how to draw B-spline curves and surfaces, but to illuminate, texture and trim the latter as well.

While B-spline theory is extensive, material we covered in this chapter of the polynomial B-spline primitives, together with what is covered in Chapter 20 of NURBS, is ample for an applications programmer to function knowledgeably. However, the reader is well-advised to expand her knowledge, particularly, of such practical topics as "knot insertion", "degree elevation", etc. It's easy enough given the number of excellent books available – Bartels et al. [9], Farin [45], Mortenson [97], Piegl & Tiller [113] and Rogers & Adams [120] are a few that come to mind. The mathematically inclined reader, in particular, will find much to fascinate her in the more specialized nooks and crannies. Advanced 3D CG books, e.g., Akenine-Möller, Haines & Hoffman [1], Buss [21], Slater et al. [137] and Watt [150], each have a presentation of B-spline theory as well.

B-spline functions were first studied in the 1800s by the Russian mathematician Nicolai Lobachevsky. However, the modern theory began with Schoenberg's [128] application of spline functions to data smoothing and received particular impetus with the discovery in 1972 of the recursive formula (18.30) for B-spline functions by Cox [29], de Boor [33] and Mansfield. It has since seen explosive growth and B-spline (and NURBS) primitives are *de rigueur* in modern-day CG design.