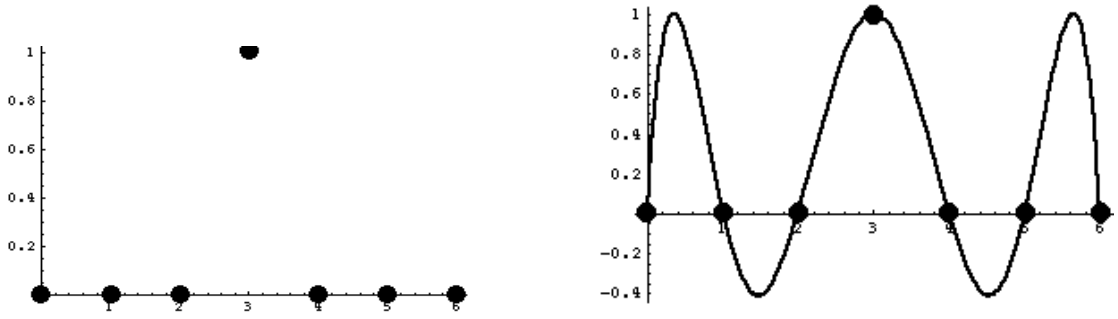
Lecture 21: Bezier Approximation and de Casteljau's Algorithm

and thou shalt be near unto me Genesis 45:10

1. Introduction

In Lecture 20, we used interpolation to specify shape. But interpolation is not always a good way to describe the contour of a curve or surface. To accurately reproduce complicated shapes, we may need to interpolate lots of data. Polynomial interpolation for many points is impractical because the degree of the interpolant can get extremely high leading to slow and numerically unstable computations. Also polynomial interpolants may oscillate unnecessarily and fail to reproduce the desired shapes (see Figure 1). Thus, even if we were to specify more and more points, there is no guarantee that the polynomial interpolants would converge to the curves or surfaces we wish to represent.



Data *Polynomial Interpolant*

Figure 1: Lagrange interpolation. Notice the oscillations in the interpolating polynomial curve, even though there is no oscillation in the original data points.

Spline interpolation -- that is, interpolation by piecewise polynomial functions -- is better computationally because splines allow us to keep the degree low. But interpolating splines may still oscillate unnecessarily and fail to reproduce the desired shapes. Our approach here is rather to abandon interpolation altogether and to take a very different approach to describing the shape of a curve or surface.

Given a relatively small collection of points in affine space, we are going to investigate methods for generating polynomial curves and surfaces that approximate the shape described by these points. We shall not insist that our curves and surfaces go through these points, but we shall insist that these curves and surfaces come near to the points and capture in some mathematically precise way the shape defined by these points. As usual we begin with schemes for curves and later extend our techniques to surfaces.

2. De Casteljau's Algorithm

Let us return for a moment to where we began our investigation of polynomial curves and surfaces: Lagrange interpolation and Neville's algorithm. Recall that Neville's algorithm (Figure 2) is a dynamic programming procedure for computing points along a polynomial interpolant. We are going to start our investigation of approximation schemes by using the same basic triangular structure but simplifying the computations along the edges.

The simplest thing -- one might almost say the only thing -- we know how to do is linear interpolation. All our interpolation procedures, and especially Neville's algorithm, are based on this simple idea or some variant thereof. What makes Neville's algorithm the least bit complicated is that we perform a different linear interpolation at each node of the diagram. To take the same triangular structure and make the evaluation algorithm as easy as possible, we will perform the same linear interpolation at each node. This idea generates the algorithm represented in Figure 3.

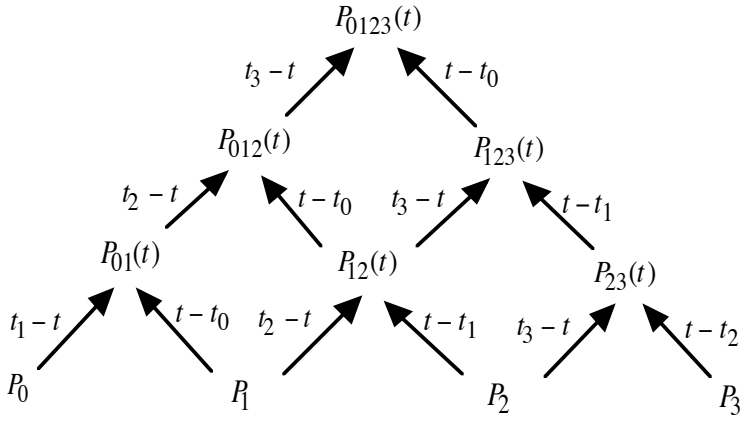


Figure 2: Neville's algorithm (unnormalized) for cubic polynomial interpolation.

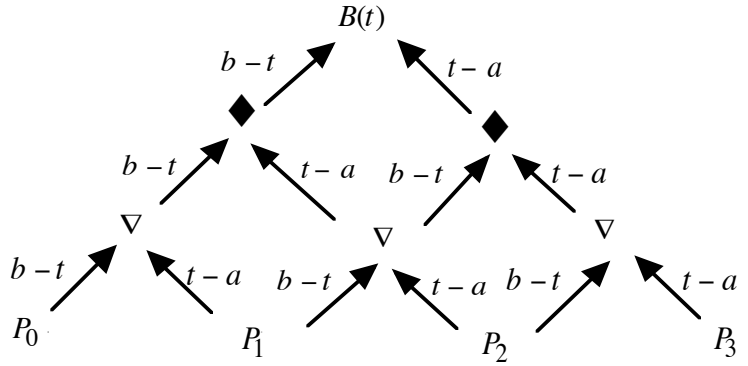


Figure 3: The de Casteljau algorithm for a cubic Bezier curve $B(t)$ in the interval $[a, b]$. The label on every edge must be normalized by dividing by $b - a$, so that the labels along arrows entering each node sum to one.

This algorithm represented in Figure 3 is called *de Casteljau's evaluation algorithm*, and the curves that emerge at the apex of this diagram are called *Bezier curves*. Intermediate nodes marked ∇ and ♦ also represent Bezier curves, but of lower degree. Thus the de Casteljau algorithm is a dynamic programming algorithm for computing points on a Bezier curve. Typically, for reasons that will become clear in the next section, Bezier curves are restricted to the interval $[a, b]$. Usually, for simplicity, we take $a = 0$ and $b = 1$, but there are cases, as we shall see later on, where it is useful to allow a and b to be arbitrary as long as $b > a$. Notice that when $a = 0$ and $b = 1$, no normalization is required.

The de Casteljau algorithm has the following elegant geometric interpretation. Since each node represents a linear interpolation, each node symbolizes a point on the line segment joining the two points whose arrows point into the node. Drawing all these line segments generates the trellis in Figure 4.

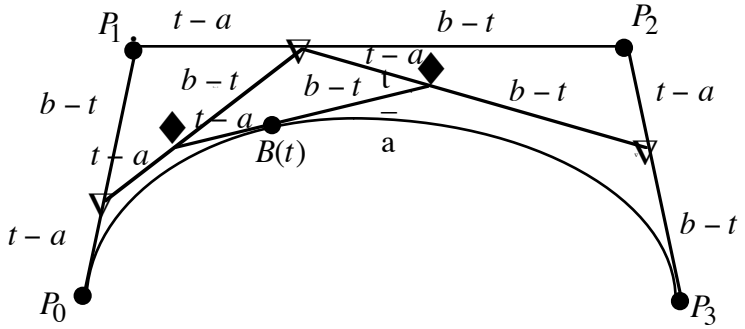


Figure 4: Geometric construction algorithm for a point on a cubic Bezier curve based on a geometric interpretation of the de Casteljau evaluation algorithm. At the parameter t, each line segment in the trellis is split in the ratio $(t - a)/(b - t)$.

We are going to study the geometric characteristics of curves generated by de Casteljau's algorithm. We begin with some simple features; in subsequent lectures we will develop some more advanced mathematical properties of this approximation scheme.

3. Elementary Properties of Bezier Curves

Bezier curves have the following elementary properties:

1. Polynomial Parametrization
2. Affine Invariance
3. Convex Hull
4. Symmetry
5. Interpolation of End Points

Below we briefly discuss and derive each of these properties in turn, and we explain as well why these features are important for computer graphics.

Most of these properties can be proved by direct observation or by easy inductive arguments using the de Casteljau algorithm. To set up these inductive arguments, let $B[P_0,...,P_n](t)$ denote the Bezier curve over the interval $[a,b]$ with affine control points $P_0,...,P_n$. Then the last stage of the de Casteljau algorithm can be written as

$$B[P_0,...,P_n](t) = \frac{b-t}{b-a}B[P_0,...,P_{n-1}](t) + \frac{t-a}{b-a}B[P_1,...,P_n](t). \qquad (3.1)$$

We are now ready to proceed with our derivations.

1. Polynomial Parametrization

In the de Casteljau algorithm, the only operations we perform involving the functions along the edges are addition and multiplication (see Figure 3). Since the functions along the edges are linear polynomials, it follows that a Bezier curve with $n+1$ control points is a polynomial curve of degree n because there are n levels from the control points at the base to the curve at the apex of the triangle. (This result also follows by an easy induction from Equation (3.1)). Since Bezier curves are polynomial curves, all the tools we know for polynomials apply.

2. Affine Invariance

A curve is said to be *affinely invariant* if it consists of a collection of points in affine space. Equivalently, a curve scheme is said to be affinely invariant if applying an affine transformation to the control points transforms every point on the curve by the same affine transformation. Affine invariance is an easy consequence of the de Casteljau algorithm. since by linear interpolation every node in the algorithm is affinely invariant (see Figures 5,6). (This result also follows by an easy induction from Equation (3.1)).

Affine invariance is a crucial feature for any curve scheme because it asserts that the curve is independent of the choice of the coordinate system. This property is essential for a good approximation scheme, since in a typical geometric model many different coordinate systems are present. Affine invariance guarantees that the curve will be the same no matter which coordinate system is invoked.

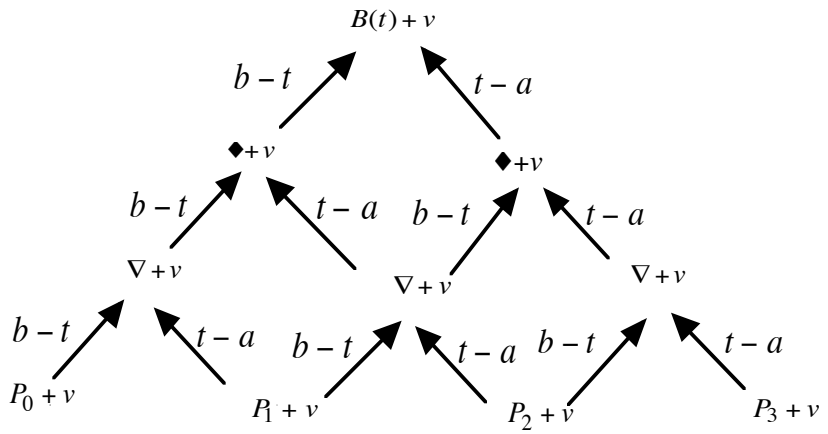


Figure 5: Translation invariance. Translating each control point by a vector v translates every point on the curve by the same vector v.

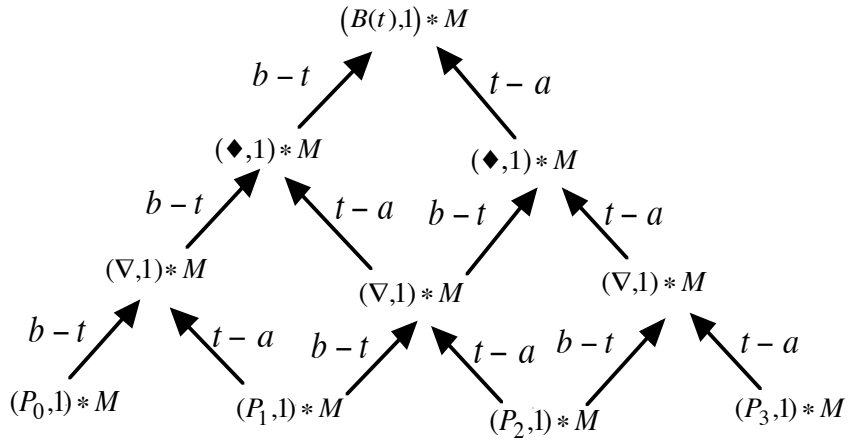


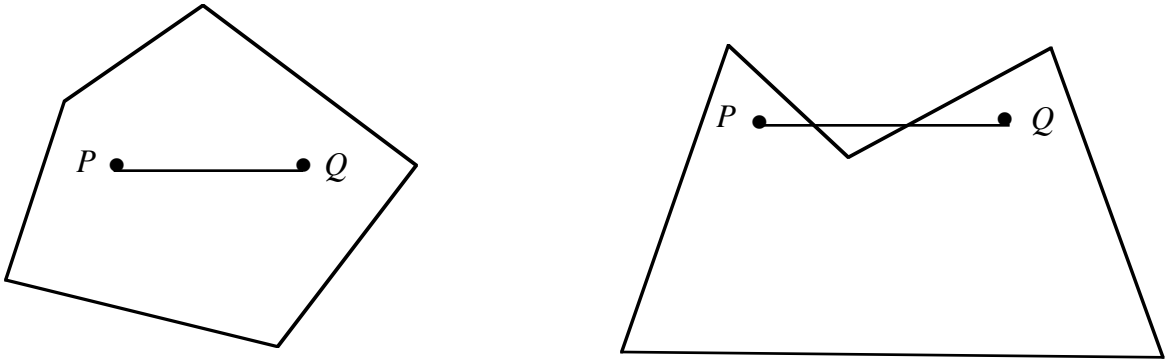
Figure 6: Affine invariance. Transforming each control point by an affine map M transforms every point on the curve by the same affine map M.

3. Convex Hull Property

A set S of points in affine space is said to be convex if whenever P and Q are points in S the entire line segment from P to Q lies in S (see Figure 7). The intersection S of a collection of convex sets $\{S_i\}$ is a convex set because if P and Q are points in S, they must also be points in each of the sets S_i. Since, by assumption, the sets S_i are convex, the entire line segment from P to Q lies in each set S_i. Hence the entire line segment from P to Q lies in the intersection S, so S too is convex.

The *convex hull* of a collection of points in affine space is the intersection of all the convex sets containing the points. Since the intersection of convex sets is a convex set, the convex hull is the smallest convex set containing the points. For two points, the convex hull is the line segment

joining the points. For three non-collinear points, the convex hull is the triangle whose vertices are the three points. The convex hull of a finite collections of points in the plane can be found mechanically by placing a nail at each point, stretching a rubber band so that its interior contains all the nails, and then releasing the rubber band. When the rubber band comes to rest on the nails, the interior of the rubber band is the convex hull of the points.



(a) Convex Set (b) Non-Convex Set

Figure 7: In a convex set (a) the line segment joining any two points in the set lies entirely within the set. In a non-convex set (b) part of the line segment joining two points in the set may lie outside the set.

Since the convex hull of two points is the line segment joining the two points,

$$ConvexHull\{P_0, P_1\} = \{c_0P_0 + c_1P_1 \mid c_0 + c_1 = 1 \ \ and \ \ c_0, c_1 \geq 0\} .$$

More generally it follows by a simple inductive argument (see Exercise 1) that

$$ConvexHull\{P_0, ..., P_n\} = \{ \sum_{k=0}^{n} c_k P_k \mid \sum_{k=0}^{n} c_k = 1 \ \ and \ \ c_k \geq 0\} .$$

Bezier curves always lie in the convex hull of their control points. That is,

$$B[P_0, ..., P_n](t) \subseteq ConvexHull\{P_0, ..., P_n\} .$$

Again we can prove this assertion by a simple inductive argument. First recall that, by convention, we always restrict the Bezier curve $B[P_0, ..., P_n](t)$ in (3.1) to the parameter interval $a \leq t \leq b$. With this restriction, the convex hull property is certainly true for a Bezier curve with only two control points since, by construction, this curve is the line segment joining the two control points. More generally suppose that this result is valid for Bezier curves with n control points. By (3.1), $B[P_0, ..., P_n](t)$ lies on the line segment joining the points $B[P_0, ..., P_{n-1}](t)$ and $B[P_1, ..., P_n](t)$, and by the inductive hypothesis $B[P_0, ..., P_{n-1}](t)$ and $B[P_1, ..., P_n](t)$ both lie in the convex hull of the points $P_0, ..., P_n$. But if two points lie in a convex set, the entire line segment joining them also lies in the set; thus the entire Bezier curve $B[P_0, ..., P_n](t)$, $a \leq t \leq b$, must lie in $ConvexHull\{P_0, ..., P_n\}$

. The convex hull property is important because it constrains Bezier curves to lie in the proximity of their control points. This property is a vital feature for an approximation scheme. Scientists and engineers not only require curves that approximate the shape defined by their control points, they also demand curves that lie in the same region of space as their control points. To be useful in computer graphics, the curves must be visible on the graphics terminal. The convex hull property guarantees that if all the control points are visible on the graphics terminal, then the entire curve is visible as well. The restriction $a \le t \le b$ on the parameter t is there precisely to guarantee the convex hull property.

4. Symmetry

Replacing t by $a + b - t$ reverses the order of the parameter domain. As the parameter t varies from a to b, the curve $B[P_0,...,P_n](a + b - t)$ traverses the same points as $B[P_0,...,P_n](t)$ but in the direction from b to a rather than from a to b. Thus $B[P_0,...,P_n](a + b - t)$ is essentially the same curve as $B[P_0,...,P_n](t)$ but with opposite orientation. Similarly, reversing the order of the control points of a Bezier curve generates the same Bezier curve but with opposite orientation. Analytically this means that,

$$B[P_n,...,P_0](t) = B[P_0,...,P_n](a + b - t) \qquad a \le t \le b. \tag{3.2}$$

This symmetry property is what professional engineers and most naive users would naturally expect of a simple approximation scheme, so it is gratifying to see that it holds for all Bezier curves.

To prove (3.2), simply replace t by $a + b - t$ in the de Casteljau diagram and observe that the new diagram is the mirror image of the de Casteljau diagram for $B[P_n,...,P_0](t)$ (see Figure 8).

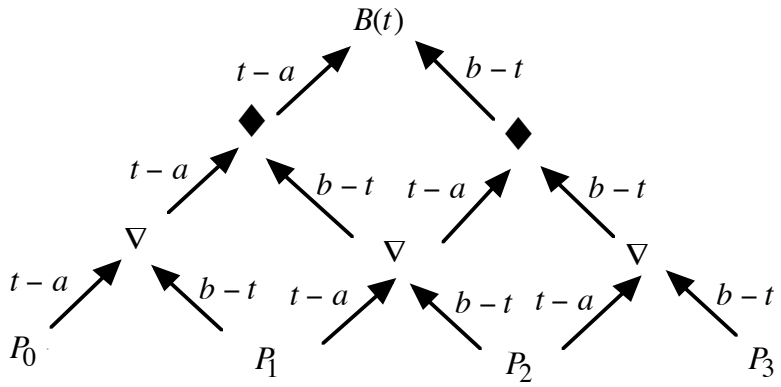


Figure 8: The de Casteljau algorithm for $B[P_0,...,P_n](a + b - t)$. Compare to Figure 3 with the control points in reverse order.

5. Interpolation of End Points

Unlike Lagrange polynomials, Bezier curves generally do not interpolate all their control points. But Bezier curves always interpolate their first and last control points. In fact,

$$B[P_0,...,P_n](a) = P_0$$
$$B[P_0,...,P_n](b) = P_n.$$

The first result follow easily from setting $t = a$ in de Casteljau's algorithm and observing that all the labels on left pointing arrows become zero while all the labels on right pointing arrows become one. If $k \neq 0$, then any path from P_k to the apex of the triangle must traverse at least one left pointing arrow, so there is no contribution from P_k to the value of the curve at $t = a$. On the other hand, when $t = a$ all the labels on the single path from P_0 to the apex of the triangle are one. Hence $B[P_0,...,P_n](a) = P_0$. A similar argument for $t = b$ shows that $B[P_0,...,P_n](b) = P_n$. Again an easy inductive argument based on (3.1) yields the same results.

Interpolating end points is important because we often want to connect two curves. To assure that two Bezier curves join at their end points, all we need to do is to make sure that the first control point of the second curve is the same as the last control point of the first curve. This device insures continuity. In the next section, we shall develop techniques for guaranteeing higher order smoothness between adjacent Bezier curves, but before we can perform this analysis we need to know how to differentiate Bezier curves.

4. Differentiation Algorithm for Bezier Curves

To compute the derivative of a Bezier curve, we need to differentiate the de Casteljau algorithm. Differentiating the de Casteljau algorithm for a degree *n* Bezier curve is actually quite easy: simply differentiate the labels -- that is, replace $1 - t \rightarrow -1$ and $t \rightarrow 1$ -- on any level of the de Casteljau algorithm and multiply the output by *n* (see Figure 9).

We shall defer the proof of this differentiation algorithm for Bezier curves till Lecture 23. In this section we will study the consequences of this differentiation algorithm.

There are several things to notice about Figure 9. First, by running only the lowest level of the differentiation algorithm, we see that, up to a constant multiple, we can think of the derivative of a cubic Bezier curve with control points $\{P_k\}$ as a quadratic Bezier polynomial with control vectors $\{P_{k+1} - P_k\}$ (see Figure 10).

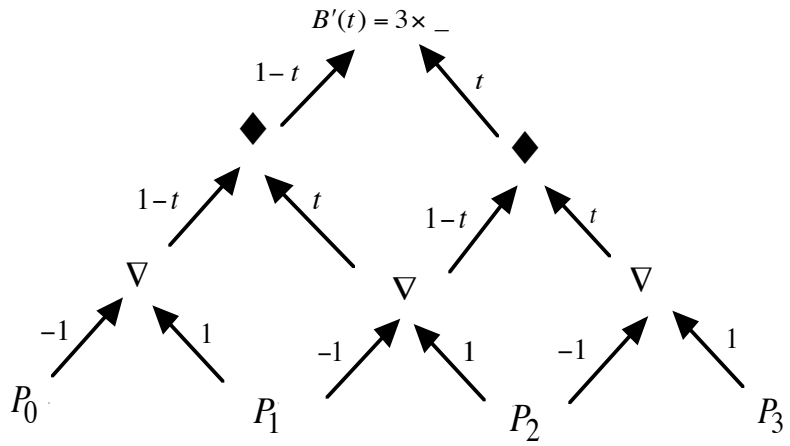


Figure 9: Computing the first derivative of a cubic Bezier curve with control points P_0, P_1, P_2, P_3 by differentiating the labels on the bottom level of the de Casteljau algorithm. To get the correct derivative $B'(t)$, we must multiply the output of this algorithm by $n = 3$.

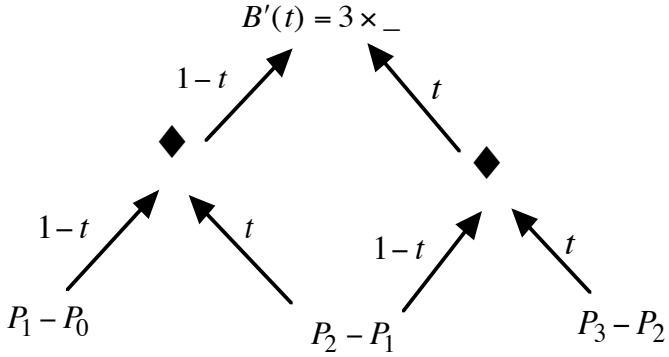


Figure 10: The derivative of a cubic Bezier curve with control points $\{P_k\}$ is, up to a constant multiple, a quadratic Bezier polynomial with control vectors $\{P_{k+1} - P_k\}$.

In general, up to a constant multiple, the derivative of a Bezier curve with control points $\{P_k\}$ is a Bezier polynomial of one lower degree with control vectors $\{P_{k+1} - P_k\}$. Therefore, by induction, we can express the *kth* derivative of a degree n Bezier curve as a Bezier polynomial of degree $n-k$. Moreover, to compute the *kth* derivative of a Bezier curve, we can differentiate the labels on any k levels the de Casteljau algorithm and multiply the output by $\frac{n!}{(n-k)!}$ (see Figure 11).

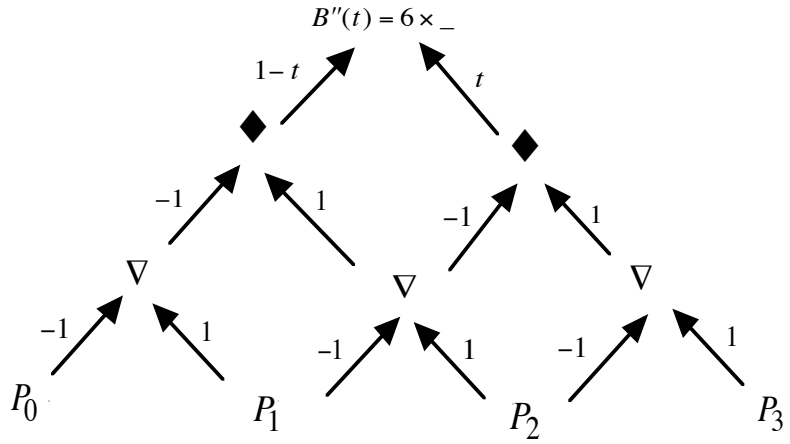


Figure 11: Computing the second derivative of a cubic Bezier curve with control points P_0, P_1, P_2, P_3 by differentiating the labels on the bottom two levels of the de Casteljau algorithm. Here to get the correct second derivative $B'(t)$, we must multiply the output of this algorithm by 6.

We prefaced our discussion of differentiation by saying that we wanted to be able to join two Bezier curves together smoothly. To do so, we need to calculate derivatives at their end points. Let $B(t)$ be a Bezier curve with control points $P_0,\ldots,P_n$. Substituting $t=0$ or $t=1$ into the differentiation algorithm, we see that:

$$B(0) = P_0 \qquad B(1) = P_n$$
$$B'(0) = n(P_1 - P_0) \qquad B'(1) = n(P_n - P_{n-1})$$
$$B''(0) = n(n-1)(P_2 - 2P_1 + P_0) \qquad B''(1) = n(n-1)(P_n - 2P_{n-1} + P_{n-2})$$

In general, it follows by induction on *k* that the *kth* derivative at $t=0$ depends only on the first $k+1$ control points, and the *kth* derivative at $t=1$ depends only on the last $k+1$ control points.

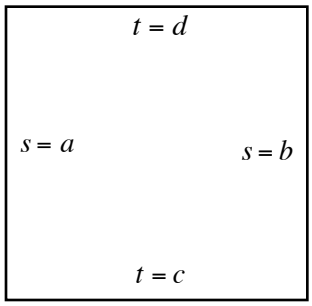
Suppose then that we are given a Bezier curve with control points $P_0,\ldots,P_n$ and we want to construct another Bezier curve with control points $Q_0,\ldots,Q_n$ that meets the first curve and matches its first *r* derivatives at its end point. From the results in the previous paragraph, we find that the control points $Q_0,\ldots,Q_n$ must satisfy the following constraints:

$$r = 0: \quad Q_0 = P_n$$
$$r = 1: \quad Q_1 - Q_0 = P_n - P_{n-1} \;\Rightarrow\; Q_1 = P_n + (P_n - P_{n-1})$$
$$r = 2: \quad Q_2 - 2Q_1 + Q_0 = P_n - 2P_{n-1} + P_{n-2} \;\Rightarrow\; Q_2 = P_{n-2} + 4(P_n - P_{n-1})$$

and so on for higher values of *r*. Each additional derivative allows us to solve for one additional control point. Clearly we could go on in this manner solving for one point at a time. An alternative approach that avoids all this tedious computation will be presented in Lecture 22.

5. Tensor Product Bezier Surfaces

A tensor product Bezier patch is a rectangular surface patch generated by a rectangular array of control points. Connecting control points with adjacent indices by straight lines generates a control polyhedron that controls the shape of the Bezier patch in much the same way that the Bezier control polygon controls the shape of a Bezier curve (see Figures 12 and 13). In particular, dragging a control point pulls the surface patch in the same general direction as the control point.



(a) Domain -- Rectangle

$$\begin{matrix} P_{03} & P_{13} & P_{23} & P_{33} \\ P_{02} & P_{12} & P_{22} & P_{32} \\ P_{01} & P_{11} & P_{21} & P_{31} \\ P_{00} & P_{10} & P_{20} & P_{30} \end{matrix}$$

(b) Range -- Rectangular Array of Points

Figure 12: Data for a bicubic tensor product Bezier patch. Notice that the domain is simply a rectangle and that, unlike Lagrange interpolation, the domain has no nodes and no grid. Compare to Figure 8 in Lecture 20.

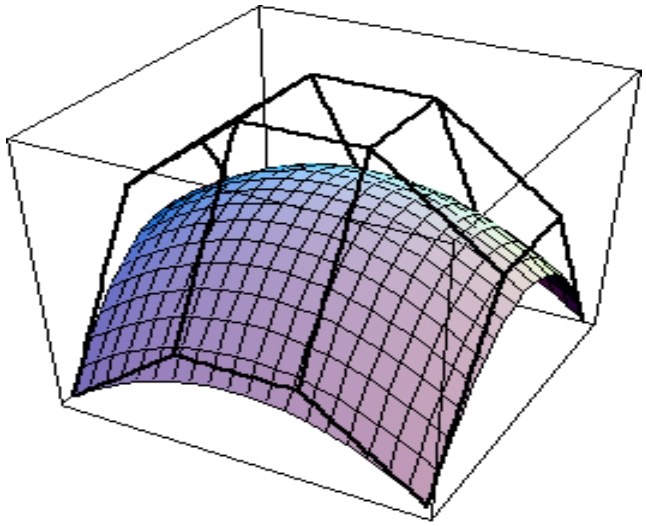


Figure 13: A bicubic tensor product Bezier surface with its control polyhedron, formed by connecting control points with adjacent indices. The control points are the same as those for the Lagrange surface in Figure 10 of Lecture 20.

To construct a tensor product Bezier patch $B(s,t)$ from a rectangular array of control points $\{P_{ij}\}$, we proceed in a manner similar to the construction of a tensor product Lagrange surface,

except that Lagrange curves and Neville's algorithm are replaced by Bezier curves and de Casteljau's algorithm. Let

$$P_i(t) = B[P_{i,0},\ldots,P_{i,n}](t) \qquad i = 0,\ldots,m, \tag{4.1}$$

be the Bezier curve for the control points $P_{i,0},\ldots,P_{i,n}$. For each fixed value of t, let

$$B(s,t) = B[P_0(t),\ldots,P_m(t)](s) \tag{4.2}$$

be the Bezier curve for the control points $P_0(t),\ldots,P_m(t)$. Then as s varies from a to b and t varies from c to d, the curves $B(s,t)$ sweep out a surface (see Figures 14,15). This surface is called a *tensor product Bezier patch*.

Equations (4.1), (4.2) and Figures 14,15 suggest the following evaluation algorithm for tensor product Bezier patches: first use de Casteljau's algorithm $m+1$ times to compute the points at the parameter t along the degree n Bezier curves $P_0(t),...,P_m(t)$; then use de Casteljau's algorithm one more time to compute the point at the parameter s along the degree m Bezier curve with control points $P_0(t),...,P_m(t)$ (see Figure 16).

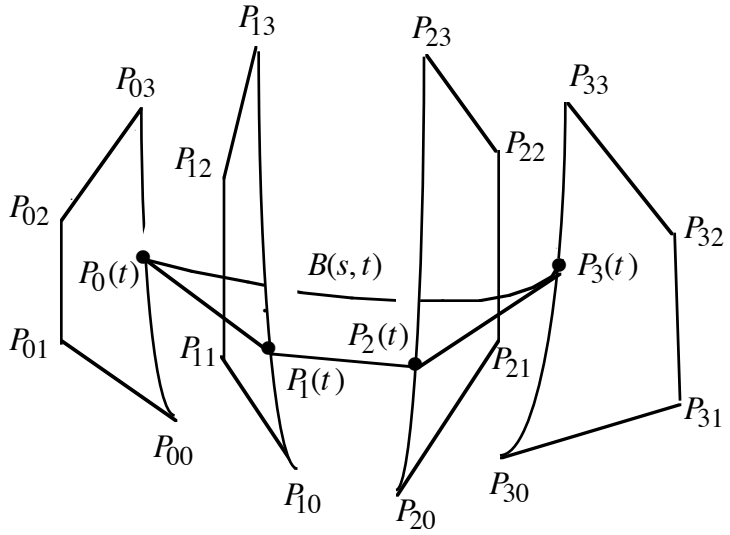


Figure 14: Construction of points on a bicubic tensor product Bezier surface $B(s,t)$. First the Bezier curves $P_i(t)$, $i = 0,...,3$ are constructed from the control points $P_{i0}, P_{i1}, P_{i2}, P_{i3}$. Then for a fixed value of t, the Bezier curve $B(s,t)$ is constructed using the points $P_0(t), P_1(t), P_2(t), P_3(t)$ as control points. In general, the Bezier surface $B(s,t)$ does not interpolate its control points. Compare to Figure 9 of Lecture 20 for bicubic Lagrange interpolation.

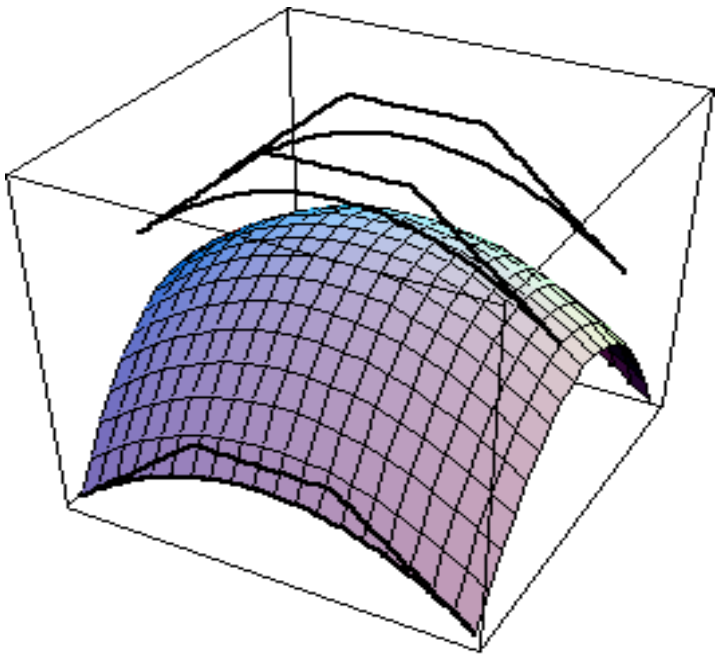


Figure 15: The bicubic Bezier patch in Figure 13 along with its cubic Bezier control curves. Notice that only the boundary control curves are interpolated by the surface. Compare to the tensor product Lagrange surface in Figure 10 of Lecture 20.

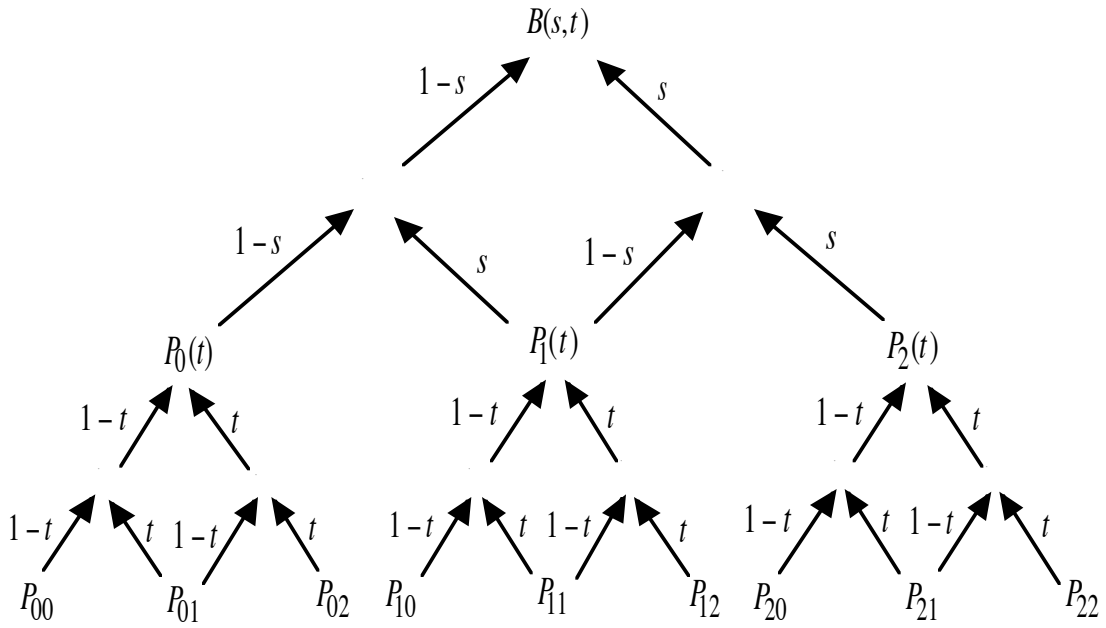


Figure 16: De Casteljau's evaluation algorithm for a biquadratic Bezier patch. The three lower triangles represent Bezier curves in the t direction, and the upper triangle blends these curves in the s direction. Compare to Figure 11 of Lecture 20 -- Neville's algorithm for a biquadratic Lagrange interpolating patch.

Alternatively, instead of starting with the Bezier curves

$$P_i(t) = B[P_{i,0},\ldots,P_{i,n}](t) \qquad i = 0,\ldots,m,$$

we could begin with the Bezier curves

$$P_j^*(s) = B[P_{0,j},\ldots,P_{m,j}](s) \qquad j = 0,\ldots,n,$$

and for each fixed value of s let

$$B(s,t) = B[P_0^*(s),\ldots,P_n^*(s)](t)$$

be the Bezier curve for the control points $P_0^*(s),\ldots,P_n^*(s)$. Again as s varies from a to b and t varies from c to d, the curves $B(s,t)$ sweep out a surface. We shall show in a later lecture that this surface is identical to the tensor product Bezier patch in our previous construction. Thus we have the following alternative evaluation algorithm for tensor product Bezier patches: first use de Casteljau's algorithm $n+1$ times to compute the points at the parameter s along the degree m Bezier curves $P_0^*(s),...,P_n^*(s)$; then use de Casteljau's algorithm one more time to compute the point at the parameter t along the degree n Bezier curve with control points $P_0^*(t),...,P_n^*(t)$ (see Figure 17). When $m < n$, then this algorithm is more efficient than the previous algorithm.

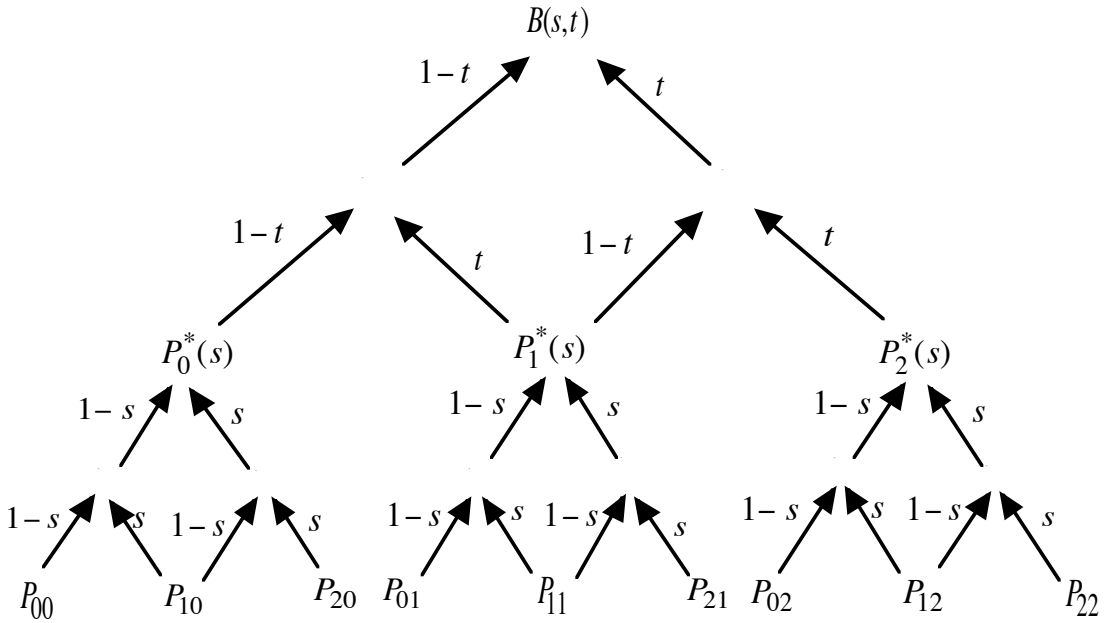


Figure 17: An alternative version of the de Casteljau evaluation algorithm for a biquadratic Bezier patch. The three lower triangles represent Bezier curves in the s direction, and the upper triangle blends these curves in the t direction. Compare to Figure 16.

Tensor product Bezier patches inherit many of the characteristic properties of Bezier curves: they are affine invariant, lie in the convex hull of their control points, and interpolate their corner control points (see Exercises 6-8). These properties follow easily from Figures 14,16 and the corresponding properties of Bezier curves. Moreover the boundaries of a tensor product Bezier patch are the Bezier curves determined by their boundary control points, since

$$B(0,t) = P_0(t) = B[P_{0,0},\ldots,P_{0,n}](t)$$

$$B(1,t) = P_m(t) = B[P_{m,0},\ldots,P_{m,n}](t)$$

and

$$B(s,0) = P_0^*(s) = B[P_{0,0},\ldots,P_{m,0}](t)$$

$$B(s,1) = P_n^*(s) = B[P_{0,n},\ldots,P_{m,n}](t).$$

It follows that although tensor product Bezier patches do not generally interpolate their control points, they always interpolate the four corner points $P_{00}, P_{m0}, P_{0n}, P_{mn}$.

To compute the partial derivatives of a Bezier patch, we can apply our procedure for differentiating the de Casteljau algorithm for Bezier curves. Consider Figure 16. We can compute $\partial B / \partial t$ simply by differentiating the de Casteljau algorithm for each of the Bezier curves $P_0(t),\ldots,P_m(t)$ at the base of the diagram. Similarly, we can compute $\partial B / \partial s$ by differentiating any one of the upper s levels of the diagram and multiplying the result by the degree in s (see Figure 18). Symmetric results hold for differentiating the algorithm in Figure 17: simply reverse the roles of s and t. The normal vector N to a Bezier patch is given by $N = \dfrac{\partial B}{\partial s} \times \dfrac{\partial B}{\partial t}$.

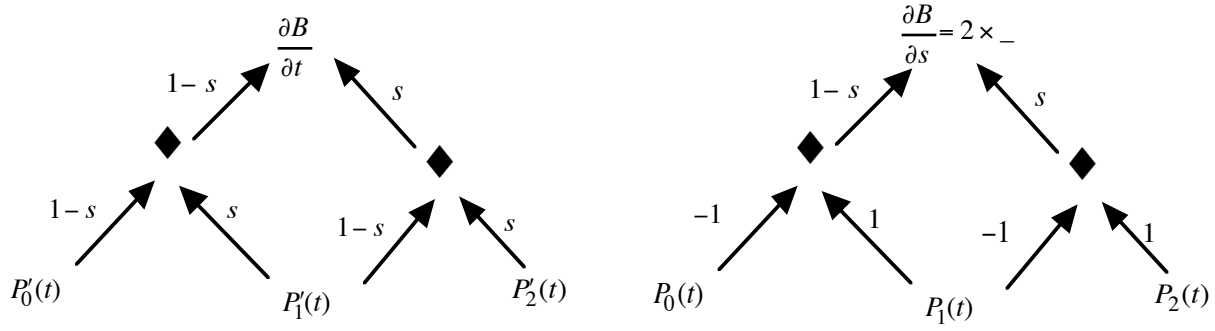


Figure 18: Computing the partial derivatives of a biquadratic Bezier patch by applying the procedure for differentiating the de Casteljau algorithm for Bezier curves. To find $\partial B / \partial t$ (left) simply differentiate the de Casteljau algorithm for each of the Bezier curves $P_0(t), P_1(t), P_2(t)$. To find $\partial B / \partial s$ (right) simply differentiate one level of the de Casteljau algorithm is s and multiply the result by the degree in s.

Exercises:

1. Let $P_0,...,P_n$ be a collection of points in an affine space. Prove that

$$ConvexHull\{P_0,...,P_n\} = \{\sum_{k=0}^{n} c_k P_k \mid \sum_{k=0}^{n} c_k = 1 \ \ and \ \ c_k \geq 0\} .$$

2. Prove that $B[P_0,...,P_n](b) = P_n$.

3. Show that Lagrange interpolating polynomial curves are affine invariant.

4. Give an example to show that a degree n Lagrange interpolating polynomial with nodes $t_0 < t_1 < \cdots < t_n$ does not necessarily satisfy the convex hull property on the interval $[t_0, t_n]$.

5. Give an example to show that Bezier curves do not necessarily reproduce polynomial curves. That is, give an example to show that if there is a degree n polynomial $P(t)$ such that $P_k = P(t_k)$, then it does not necessarily follow that $B[P_0,\ldots,P_n](t) = P(t)$.

6. Show that tensor product Bezier patches are affine invariant.

7. Show that every tensor product Bezier patch lies in the convex hull of its control points.

8. Show that every tensor product Bezier patch interpolates its corner control points.

9. Let $P_{i,j}$, $0 \leq i \leq m$, $0 \leq j \leq n$, be a rectangular array of control points, and let $B[P_{i,j}](s,t)$, $0 \leq s,t \leq 1$, denote the corresponding tensor product Bezier patch of bidegree (m,n). Show that the surface patch $B[P_{i,j}](s,t)$ has the following symmetry properties:

   a. $B[P_{j,i}](t,s) = B[P_{i,j}](s,t)$;

   b. $B[P_{i,n-j}](s,t) = B[P_{i,j}](s,1-t)$;

   c. $B[P_{m-i,j}](s,t) = B[P_{i,j}](1-s,t)$.

10. Let $P_{i,j}$, $0 \leq i \leq m$, $0 \leq j \leq n$, be a rectangular array of control points, and let

$$P_i(t) = B[P_{i,0},\ldots,P_{i,n}](t) \qquad i = 0,\ldots,m,$$

be a sequence of Bezier curves. Show how to combine Neville's algorithm and de Casteljau's algorithm to generate a surface $C(s,t)$ that interpolates the curves $P_0(t),...,P_m(t)$ at the parameter values $s_0,...,s_m$.

11. Given point and derivative data $(R_0, v_0), \ldots, (R_n, v_n)$, explain how to place Bezier control points to generate a C^1 piecewise cubic curve to interpolate this data.

12. The formulas for the unit tangent $U(t)$, the curvature $K(t)$, and the torsion $T(t)$ of a parametric curve $P(t)$ are given by

i. $$U(t) = \frac{P'(t)}{|P'(t)|}$$

ii. $$K(t) = \frac{|P'(t) \times P''(t)|}{|P'(t)|^3}$$

iii. $$T(t) = \frac{P'(t) \bullet (P''(t) \times P'''(t))}{|P'(t) \times P''(t)|^2}$$

a. Compute the unit tangent, curvature, and torsion of a Bezier curve at $t = 0, 1$.

b. Find conditions on the control points of a Bezier curve $C(t)$ so that it matches a given Bezier curve $B(t)$ with continuous unit tangent, curvature, and torsion.