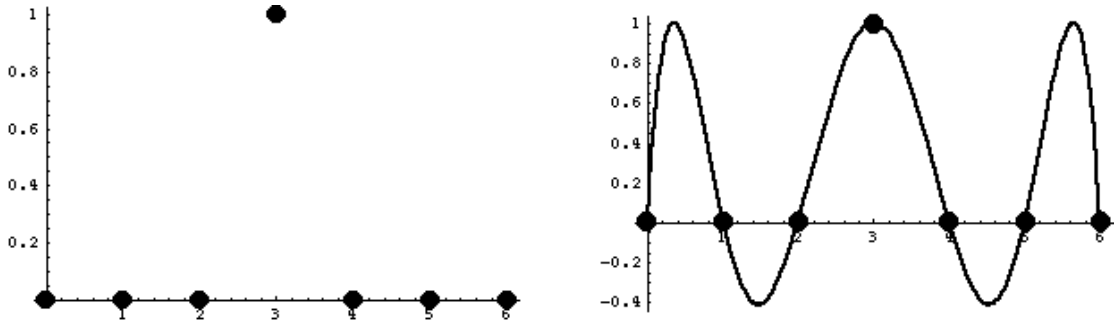
Lecture 24: Bezier Curves and Surfaces

thou shalt be near unto me Genesis 45:10

1. Interpolation and Approximation

Freeform curves and surfaces are smooth shapes often describing man-made objects. The hood of a car, the hull of a ship, the fuselage of an airplane are all examples of freeform shapes. Freeform surfaces differ from the classical surfaces we encountered in earlier lectures such as spheres, cylinders, cones, and tori. Classical surfaces are typically easy to describe with a few simple parameters. A sphere can be represented by a center point and a scalar radius; a cone, by a cone vertex, a cone angle, and a cone axis. The hood of a car or the hull of a ship are not so easy to describe with a few simple parameters. The goal of the next few chapters is to develop mathematical techniques for describing freeform curves and surfaces.

Scientists and engineers use freeform curves and surfaces to interpolate data and to approximate shape. But interpolation and approximation are not always compatible operations. Consider the data in Figure 1(a). If we use a low degree polynomial as in Figure 1(b) to interpolate this data, the polynomial oscillates about the x-axis, even though there are no such oscillations in the data. Thus the shape of the interpolating polynomial does not reflect the shape of the data. Moreover, even providing more and more data points on the desired curve may not eliminate these unwanted oscillations.

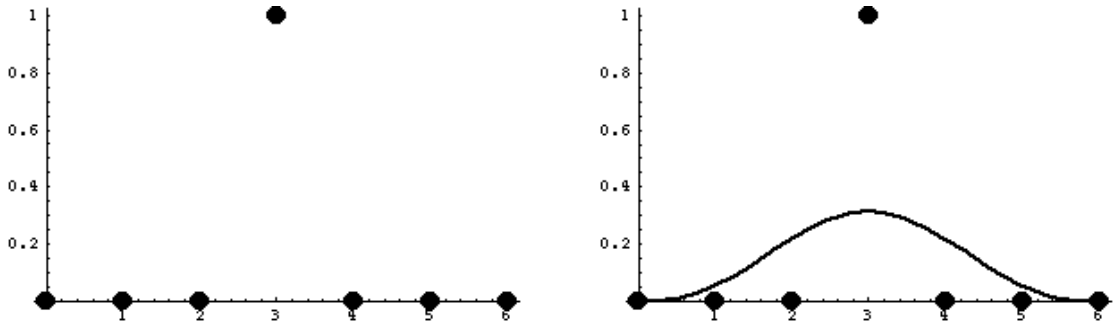


(a) *Data Points* *(b) Polynomial Interpolant*

Figure 1: Polynomial interpolation. Notice that there are oscillations in the interpolating polynomial curve, even though there are no oscillations in the original data points.

The goal of approximation is to capture the shape of a desired curve or surface from a few data points without necessarily interpolating the points. Unlike interpolation, in approximation the data points themselves are not sacred. Rather the data points are *control points*; these points control the shape of the curve or surface and can be adjusted in order to provide a better representation of the desired shape. The curve in Figure 2(b) approximates the shape of the data in Figure 2(a), even

though the curve does not pass through all the data points. The curve in Figure 2(b) is called a *Bezier curve*. Bezier curves have many practical applications, ranging from the design of new fonts to the creation of mechanical components and assemblies for industrial design and manufacture. The goal of this lecture is to develop some of the theory underlying Bezier curves and surfaces.



(a) *Control Points* (b) *Bezier Approximation*

Figure 2: Polynomial approximation. A Bezier curve approximates the shape described by the control points, but the Bezier curve does not interpolate all the control points. The height of the curve can be adjusted by changing the height of the middle control point. Compare to Figure 1(b).

2. The de Casteljau Evaluation Algorithm

The easiest curve to represent with control points is a straight line. Given two points P_0, P_1, the line $P(t)$ joining P_0 and P_1 can be expressed parametrically by setting

$$P(t) = P_0 + t(P_1 - P_0) = (1-t)P_0 + tP_1. \tag{2.1}$$

It is easy to check that $P(t)$ represents a straight line, and that $P(0) = P_0$ and $P(1) = P_1$. Thus the curve $P(t)$ passes through the point P_0 at time $t = 0$ and through the point P_1 at time $t = 1$.

Suppose, however, that you want the straight line to pass through the point P_0 at time $t = a$ and through the point P_1 at time $t = b$. Then mimicking Equation (2.1), you might write

$$P(t) = \big(1 - f(t)\big)P_0 + f(t)P_1 \tag{2.2}$$

with the requirement that

$$f(a) = 0 \quad \text{and} \quad f(b) = 1. \tag{2.3}$$

To find a simple explicit expression for the function $f(t)$, you would need to find the line in the coordinate plane interpolating the data $(a,0)$ and $(b,1)$. Using standard techniques from analytic geometry, you can write the equation of this line as

$$f(t) = \frac{(t-a)}{(b-a)}, \tag{2.4}$$

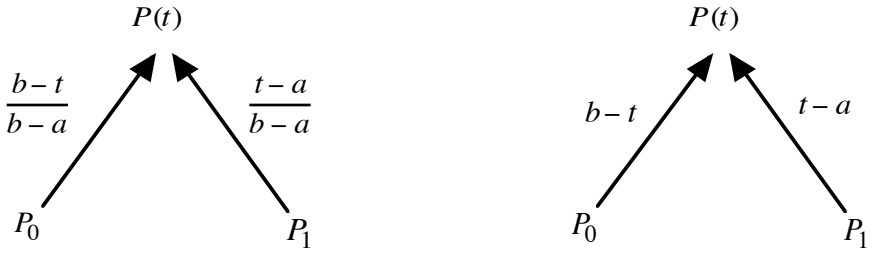
and you can easily verify that $f(a) = 0$ and $f(b) = 1$. Substituting Equation (2.4) into Equation

(2.2) yields

$$P(t) = \frac{b-t}{b-a}P_0 + \frac{t-a}{b-a}P_1. \qquad (2.5)$$

Equation (2.5) is called *linear interpolation* because the curve $P(t)$ interpolates the points P_0, P_1 with a straight line. You have encountered linear interpolation many times before in Computer Graphics; for example, Gouraud and Phong shading are both based on linear interpolation.

Equation (2.5) is so important that we are going to represent this equation by a simple graph. In Figure 3(a) the control points P_0, P_1 are placed in the two nodes at the base of the diagram and the coefficients of the control points P_0, P_1 in Equation (2.5) are placed along the arrows emanating from these nodes. The values in the nodes are multiplied by the values along the arrows; these products are then added and the result is placed in the node at the apex of the diagram. Thus Figure 3(a) is a graphical representation of Equation (2.5). Figure 3(b) is the same as Figure 3(a), except that to avoid cluttering the diagram, we have removed the normalizing constant $b-a$ in the denominator of the functions along the arrows We can easily retrieve this normalizing constant, since the denominator is simply the sum of the numerators: $b-a = (b-t) + (t-a)$. Thus we shall interpret Figure 3(b) to mean Figure 3(a), which in turn is equivalent to Equation (2.5).

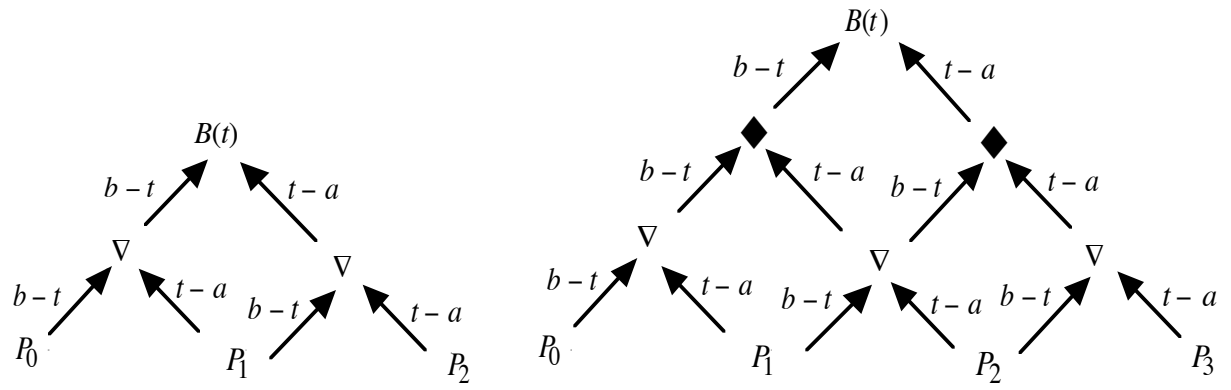


(a) Normalized (b) Unnormalized

Figure 3: Diagrams representing the line in Equation (2.5).

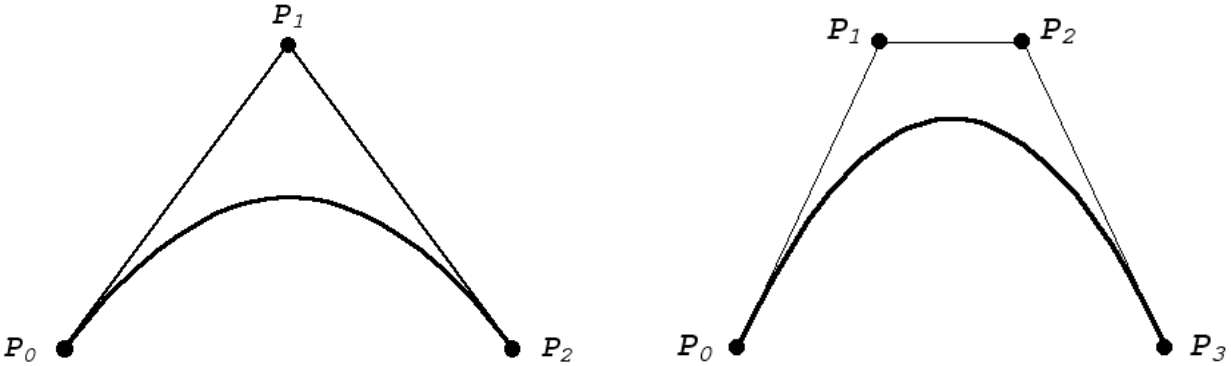
A *Bezier curve* is a curve generated by an algorithm where the steps in Figure 3 are repeated over and over again. Figure 4(a) with three control points at the base represents a quadratic Bezier curve, and Figure 4(b) with four control points at the base represents a cubic Bezier curve. The corresponding curves are illustrated in Figures 5(a) and 5(b). The piecewise linear curve consisting of the lines connecting the control points is called the *control polygon*. Notice how the Bezier curves in Figure 5 mimic the shape of their control polygons.

This algorithm for generating Bezier curves is called the *de Casteljau evaluation algorithm*. A curve generated by the de Casteljau algorithm with $n+1$ control points at the base is called *a degree n Bezier curve*. Notice that in the de Casteljau algorithm all the left pointing arrows are labeled with the function $t-a$ and all the right pointing arrows are labeled with the function $b-t$.



(a) Quadratic de Casteljau Algorithm (b) Cubic de Casteljau Algorithm

Figure 4: The de Casteljau evaluation algorithm for (a) a quadratic Bezier curve, and for (b) a cubic Bezier curve. The label on each edge must be normalized by dividing by $b - a$.



(a) Quadratic Bezier Curve (b) Cubic Bezier Curve

Figure 5: A quadratic Bezier curve (left) and a cubic Bezier curve (right). Notice how the shape of the Bezier curve (dark) mimics the shape of the control polygon (light).

Intermediate nodes of the de Casteljau algorithm represent Bezier curves of lower degree. Thus the de Casteljau algorithm is a dynamic programming procedure for computing points on a Bezier curve. Usually, for reasons that will become clear in the next section, Bezier curves are restricted to the parameter interval $[a, b]$ -- that is, usually we shall insist that $a \le t \le b$. Typically, we shall take $a = 0$ and $b = 1$, though there are cases where we will need to take other values for a,b. Notice that when $a = 0$ and $b = 1$, no normalization is required.

Since each node in the de Casteljau algorithm represents the equation of a straight line joining the points in the nodes immediately below to the left and the right, 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 6.

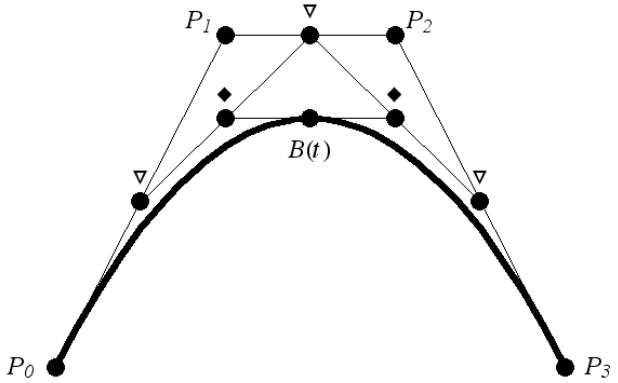


Figure 6: Geometric construction algorithm for a point on a cubic Bezier curve based on a geometric interpretation of the de Casteljau evaluation algorithm. If the labels along the edges in the de Casteljau algorithm are $(b-t)$ and $(t-a)$, then at the parameter t, each line segment in the trellis is split in the ratio $(t-a)/(b-t)$.

3. The Bernstein Representation

Bezier curves are polynomial curves. The de Casteljau algorithm proceeds by adding and multiplying the linear functions $b-t$ and $t-a$ (see Figure 4). But adding and multiplying polynomials generates polynomials of higher degree. Therefore the de Casteljau algorithm generates polynomial curves.

We can find an explicit polynomial representation for Bezier curves. Let $B(t)$ denote the Bezier curve with control points $P_0,\ldots,P_n$. Since $B(t)$ is a degree n polynomial curve, we could try to express $B(t)$ relative to the standard polynomial basis $1,t,t^2,\ldots,t^n$ -- that is, we could ask: what are the constant coefficients of the basis functions $1,t,t^2,\ldots,t^n$ for the polynomial $B(t)$? Unfortunately, these coefficients are numerically unstable, so in practice these values are not very useful. A better, more insightful question to ask is: what are the polynomial coefficients of the control points $P_0,\ldots,P_n$?

Let $B_k^n(t)$ denote the coefficient of the control point P_k in the function $B(t)$. From the de Casteljau algorithm (Figure 4) it is easy to see that

$B_k^n(t)$ = *the sum over all paths from* P_k *at the base to* $B(t)$ *at the apex of the graph*

where

a path = the product of all the labels along the arrows in the path.

Since P_k lies in the *kth* position at the base of the diagram, to reach the apex a path must take exactly k left turns and exactly $n-k$ right turns. But each left pointing arrow carries the label

$(t-a)/(b-a)$ and each right pointing arrow carries the label $(b-t)/(b-a)$. Therefore, all the paths from P_k at the base to $B(t)$ at the apex of the diagram generate the same product, so

$$B_k^n(t) = P(n,k)\frac{(t-a)^k(b-t)^{n-k}}{(b-a)^n},$$

where

$$P(n,k) = \textit{the number of paths from } P_k \textit{ to } B(t).$$

To find a closed formula for $P(n,k)$, observe that $P(n,k)$ satisfies the recurrence

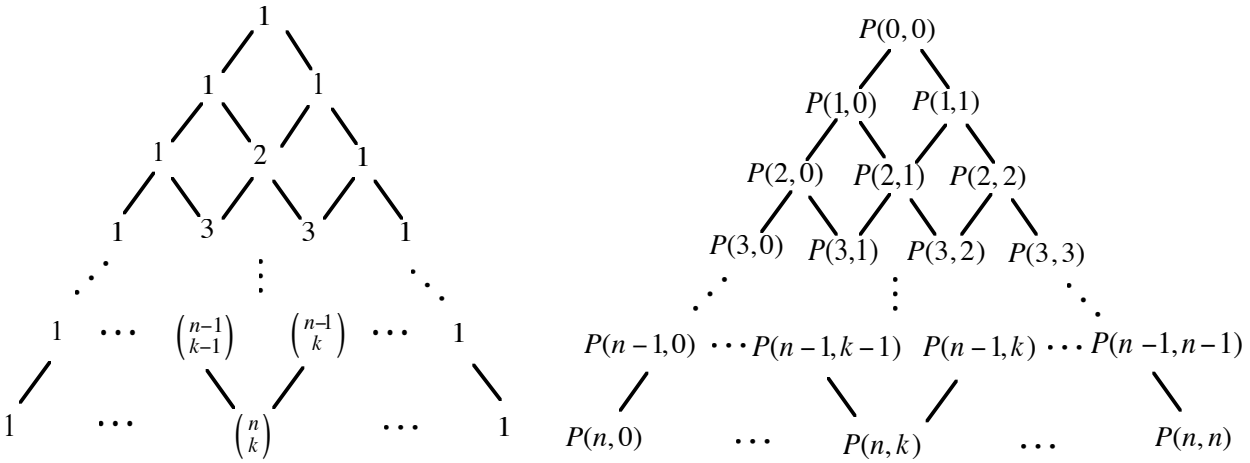
$$P(n,k) = P(n-1,k-1) + P(n-1,k),$$

because the only way to reach the *kth* position on the *nth* level is from either the $(k-1)^{st}$ position on the $(n-1)^{st}$ level or from the *kth* position on the $(n-1)^{st}$ level (see Figure 7). Since $P(0,0)=1$, the values in Pascal's triangle (binomial coefficients) and the values in the path triangle are identical: $\binom{n}{k}$ and $P(n,k)$ start at the same value and satisfy the same recurrence. Therefore

$$P(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!},$$

so

$$B_k^n(t) = \binom{n}{k}\frac{(t-a)^k(b-t)^{n-k}}{(b-a)^n} \qquad k = 0,\ldots,n.$$



(a) Pascal's Triangle (b) Paths Triangle

Figure 7: Pascal's triangle and the paths triangle start at the same value and satisfy the same recurrence. Therefore $P(n,k) = \binom{n}{k}$.

The functions $B_0^n(t),\ldots,B_n^n(t)$ are called the *Bernstein basis functions*. We shall show in Lecture 26 that the Bernstein basis functions form a basis for the polynomials of degree *n*; every

polynomial of degree n can be expressed in terms of the Bernstein basis functions. From this perspective, the control points $P_0,\ldots,P_n$ of a Bezier curve $B(t)$ are simply the coefficients of the Bezier curve relative to the Bernstein basis $B_0^n(t),\ldots,B_n^n(t)$ -- that is,

$$B(t) = \sum_{k=0}^{n} B_k^n(t) P_k$$
$$B_k^n(t) = \binom{n}{k} \frac{(t-a)^k (b-t)^{n-k}}{(b-a)^n} \qquad k = 0,\ldots,n. \tag{3.1}$$

Equation (3.1) is called the *Bernstein representation* of the Bezier curve $B(t)$. The Bernstein basis functions $B_0^n(t),\ldots,B_n^n(t)$ are also called *blending functions*, since these functions blend the discrete control points $P_0,\ldots,P_n$ to form a smooth curve.

4. Geometric Properties of Bezier Curves

Bezier curves have the following key geometric features: they are affine invariant, lie in the convex hull of their control points, satisfy the variation diminishing property (that is, they do not oscillate more that their control polygon), and interpolate their first and last control points.

There are two ways to derive these properties: either we can appeal to the de Casteljau evaluation algorithm or we can invoke the Bernstein representation of a Bezier curve. Here we shall derive most of these properties simply by reading these attributes off the graph representing the de Casteljau algorithm. For simplicity we shall typically illustrate our arguments on cubic Bezier curves, but our proofs are completely general and apply to Bezier curves of arbitrary degree. Alternative proofs based on the Bernstein representation of a Bezier curve are provided in Exercises 3-6 at the end of this lecture.

4.1. Affine Invariance. A curve scheme is said to be *affine invariant* if applying an affine transformation to the control points transforms every point on the curve by the same affine transformation. For example, a curve scheme is *translation invariant* if translating each control point by a vector v translates the entire curve by the vector v.

Translation invariance is an easy consequence of the de Casteljau algorithm. If we translate each control point by the vector v, then each node in the de Casteljau algorithm translates by the same vector v because the coefficients along the two arrows entering each node sum to one (see Figure 8). More generally, an affine transformation can be represented by an affine matrix. If we multiply each control point by an affine transformation matrix M, then each node in the de Casteljau algorithm is multiplied by the same transformation matrix M because matrix multiplication distributes through both addition and scalar multiplication (see Figure 9).

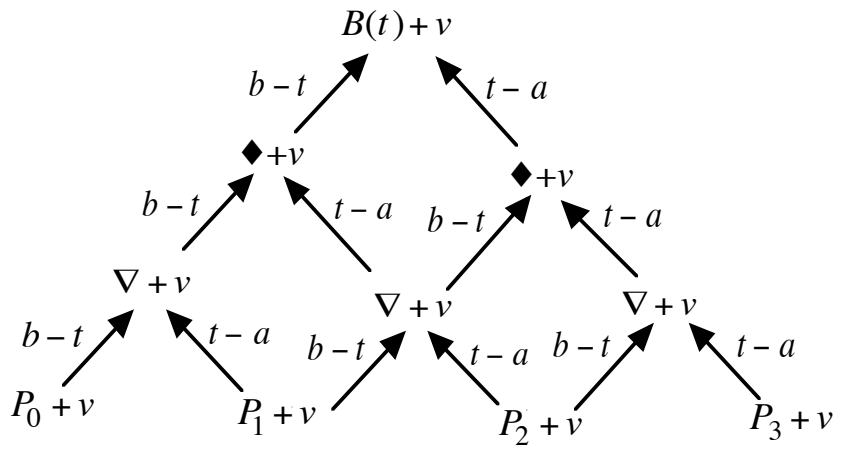


Figure 8: Translation invariance. Translating each control point by a vector v translates each point on the curve by the same vector v.

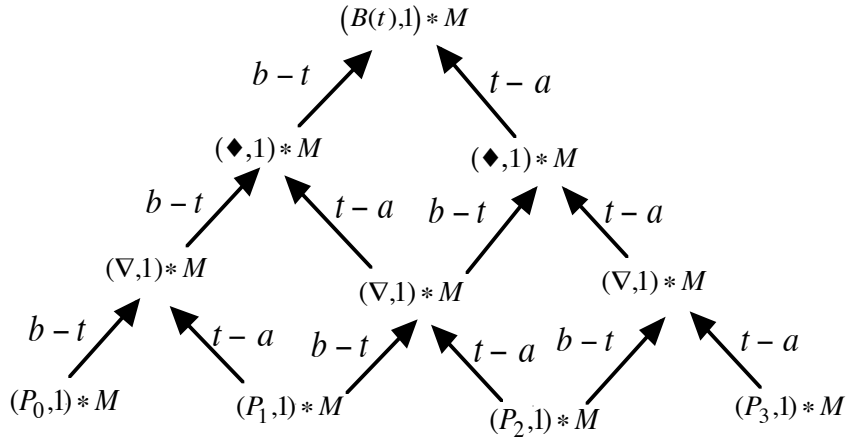
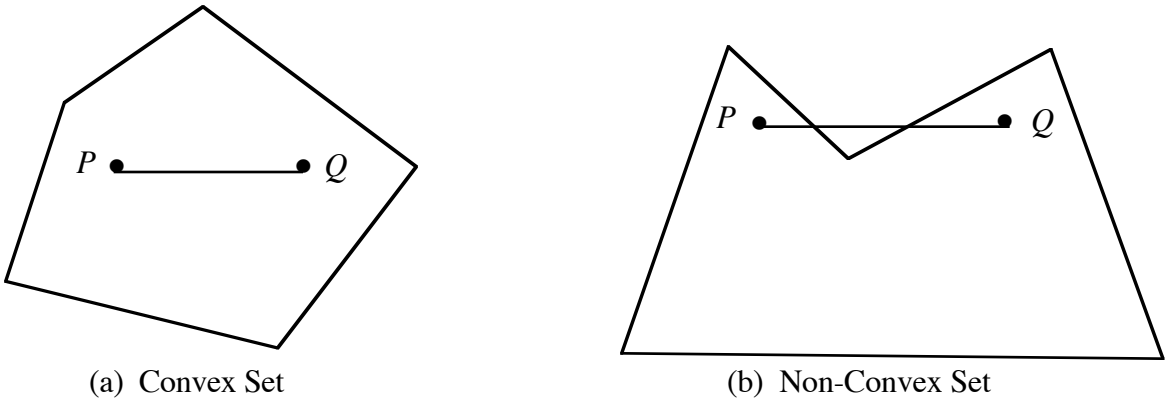


Figure 9: Affine invariance. Transforming each control point by an affine transformation matrix M transforms every point on the curve by the same affine transformation matrix M.

Translation invariance is equivalent to asserting that the curve is independent of the choice of the origin of the coordinate system. Affine invariance insures that the curve is also independent of the choice of the coordinate axes. Both of these properties are essential for a good approximation scheme. In Computer Graphics a curve should be completely determined by its control points; during design we do not want to worry about the location of the coordinate origin or the orientation of the coordinate axes. Bezier curves depend only on their control points; the location of the coordinate origin and the orientation of the coordinate axes affect neither the location nor the shape of a Bezier curve.

4.2 Convex Hull Property. A set of points S is said to be *convex* if whenever P and Q are points in S the entire line segment from P to Q lies within S (see Figure 10). 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.



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

Figure 10: 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.

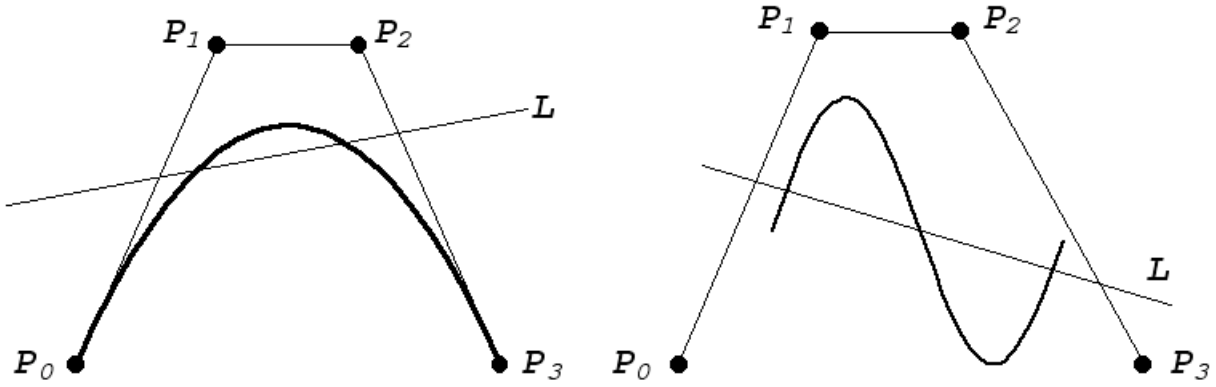
The *convex hull* of a collection of points $\{P_k\}$ is the intersection of all the convex sets containing the points $\{P_k\}$. Since the intersection of convex sets is a convex set, the convex hull is the smallest convex set containing the points $\{P_k\}$. For two points, the convex hull is the line 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.

Bezier curves always lie in the convex hull of their control points. Once again we can prove this assertion directly from the de Casteljau algorithm. First recall that, by convention, we always restrict the Bezier curve to the parameter interval $[a,b]$. With this convention, the points on the first level of the de Casteljau algorithm certainly lie in the convex hull of the control points on the zeroth level because, by construction, the points on the first level lie on the line segments joining adjacent control points (see Figure 6). For the same reason, the points on the $(n+1)^{st}$ level of the de Casteljau algorithm lie in the convex hull of the points on the *nth* level of the de Casteljau algorithm. Hence, by induction, the point at the apex of the de Casteljau algorithm lies in the convex hull of the control points at the base of the de Casteljau algorithm. Thus each point on a Bezier curve lies in the convex hull of the control points.

The convex hull property is important in Computer Graphics because 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.

4.3 Variation Diminishing Property. Intuitively, a curve is said to be *variation diminishing* if the curve does not oscillate more than its control points. In Section 1, we observed that interpolating polynomials may oscillate even if the data does not (see Figure 1). We abandoned interpolation in favor of approximation precisely in order to avoid unnecessary oscillations. Therefore we need to be sure that Bezier curves are variation diminishing.

But how do we measure oscillations? In Figure 1 we considered oscillations with respect to the x-axis. The oscillations of the curve in Figure 1 are linked to the number of times this curve crosses the x-axis. But in affine invariant schemes, there is nothing special about the x-axis; any line will do. Therefore we say that a curve is *variation diminishing* if the number of intersections of the curve with each line in the plane (or each plane in 3-space) is less than or equal to the number of intersections of the line (or the plane) with the control polygon (see Figure 11). In this definition, we ignore lines that coincide with an edge of the control polygon.



(a) Variation Diminishing (b) Not Variation Diminishing

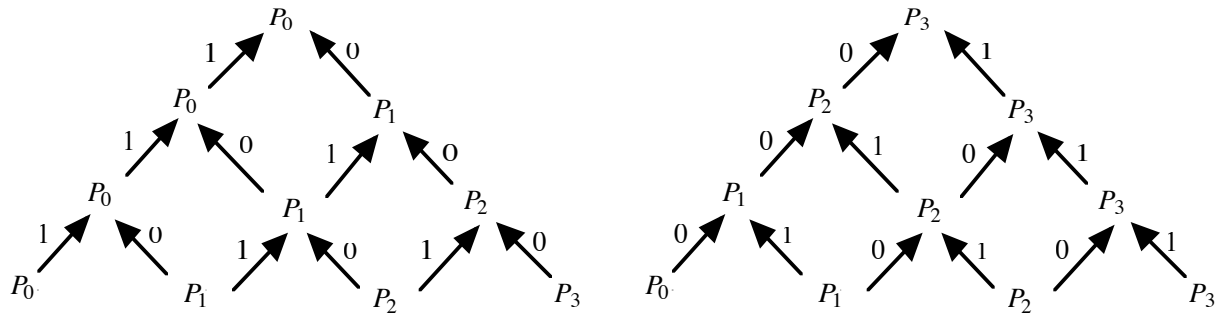
Figure 11. (a) A variation diminishing curve. An arbitrary line L intersects both the curve and the control polygon twice. (b) A curve that is not variation diminishing. The line L intersects the curve three times, but the control polygon only twice.

Bezier curves are indeed variation diminishing, but it is not so clear how to derive this fact from the de Casteljau evaluation algorithm. Therefore, we shall defer our proof of the variation diminishing property for Bezier curves till Lecture 25, where we shall derive the variation diminishing property for Bezier curves from de Casteljau's subdivision algorithm.

4.4 Interpolation of the First and Last Control Points. Bezier curves do not generally interpolate all their control points. But Bezier curves always interpolate their first and last control points. In fact, if $B(t)$ represents the Bezier curve over the interval $[a,b]$ with control points $P_0,...,P_n$, then $B(a) = P_0$ and $B(b) = P_n$. Thus $B(t)$ starts at the initial control point P_0 and ends at the final control point P_n.

As usual, we can establish these results by examining the de Casteljau algorithm. If we set $t = a$ in de the Casteljau algorithm, then all the right pointing arrows become one and all the left pointing arrows become zero (see Figure 12(a)). Now if we follow the de Casteljau algorithm from the base to the apex, then P_0 appears at each node along the left edge of the diagram. Therefore at the apex, $B(a) = P_0$. Similarly, if we set $t = b$ in de the Casteljau algorithm, then all the right pointing arrows become zero and all the left pointing arrows become one (see Figure 12(b)). Now if we follow the algorithm from the base to the apex, then P_n appears at each node along the right edge of the diagram. Therefore at the apex, $B(b) = P_n$.

Interpolation at the end points is important because we often want to connect two Bezier curves. To insure that two Bezier curves join at their end points, all we need to do is to make sure that the initial control point of the second curve is the same as the final control point of the first curve. This device guarantees 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 compute the derivative of a Bezier curve.



(a) $B(a) = P_0$ (b) $B(b) = P_3$

Figure 12: Interpolation of the first and last control points: $B(a) = P_0$ and $B(b) = P_3$.

5. Differentiating the de Casteljau Algorithm

We can compute the derivative of a Bezier curve directly from its Bernstein representation. Let $B(t)$ be a Bezier curve with control points $P_0,\ldots,P_n$. Recall from Equation (3.1) that

$$B(t) = \sum_{k=0}^{n} B_k^n(t) P_k$$

where

$$B_k^n(t) = \binom{n}{k} \frac{(t-a)^k (b-t)^{n-k}}{(b-a)^n} \qquad k = 0,\ldots,n.$$

Thus

$$B'(t) = \sum_{k=0}^{n} \frac{dB_k^n(t)}{dt} P_k. \tag{5.1}$$

Therefore to compute the derivative of a Bezier curve, we need only differentiate the Bernstein basis functions $B_k^n(t),\ k = 0,\ldots,n$. But it follows easily from the product rule that (see Exercise 8)

$$\frac{dB_k^n(t)}{dt} = n\left(\frac{B_{k-1}^{n-1}(t) - B_k^{n-1}(t)}{b-a} \right). \tag{5.2}$$

Inserting Equation (5.2) into Equation (5.1) and collecting the coefficients of $B_k^{n-1}(t)$ yields

$$B'(t) = n \sum_{k=0}^{n-1} B_k^{n-1}(t) \left(\frac{P_{k+1} - P_k}{b-a} \right). \tag{5.3}$$

Thus the derivative of the degree n Bernstein polynomial with coefficients $P_0,\ldots,P_n$ is n times the Bernstein polynomial of degree $n-1$ with coefficients $\frac{P_1 - P_0}{b-a},\ldots,\frac{P_n - P_{n-1}}{b-a}$.

One important consequence of this observation is that we can compute the derivative of a Bezier curve using the de Casteljau algorithm: Simply place the coefficients $\frac{P_1 - P_0}{b-a},\ldots,\frac{P_n - P_{n-1}}{b-a}$ at the base of the diagram, run $n-1$ levels of the algorithm, and multiply the output by n (see Figure 13).

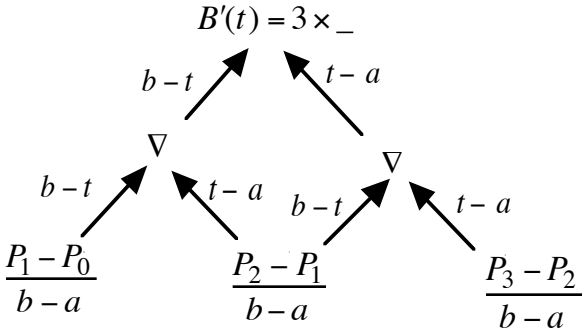


Figure 13: The derivative of a cubic Bezier curve with control points $\{P_k\}$ is, up to a constant multiple, a quadratic Bernstein polynomial with coefficients $\left\{ \frac{P_{k+1} - P_k}{b-a} \right\}$.

We can also compute the derivative of a Bezier curve by differentiating 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 $b-t \to -1$ and $t-a \to 1$ -- on the first level of the de Casteljau algorithm and multiply the output by n (see Figure 14). The validity of this differentiation algorithm follows directly from Equation (5.3) because if we differentiate the labels on the first level and run the algorithm, then the values in the nodes on the first level become $\frac{P_1 - P_0}{b-a}, \ldots, \frac{P_n - P_{n-1}}{b-a}$ (remember the normalizing factor of $b-a$ in the denominator of every label), so the result of the algorithm is exactly the Bernstein polynomial of degree $n-1$ with coefficients $\frac{P_1 - P_0}{b-a}, \ldots, \frac{P_n - P_{n-1}}{b-a}$.

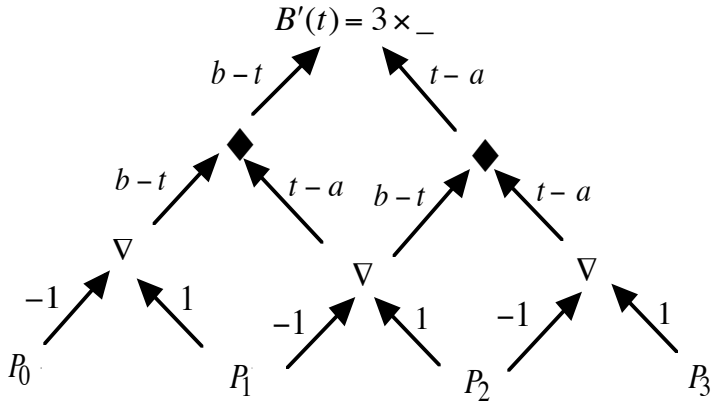


Figure 14: Computing the first derivative of a cubic Bezier curve with control points P_0, P_1, P_2, P_3 by differentiating the labels on the first level of the de Casteljau algorithm. To get the correct derivative $B'(t)$, we must multiply the output of this algorithm by $n = 3$ and we must remember to normalize the labels on each arrow -- even the constants along the arrows on the first level -- by dividing by $b-a$. Compare to Figure 13.

In general, up to a constant multiple, the derivative of a Bezier curve with control points $\{P_k\}$ is a Bernstein polynomial of one lower degree with control vectors $\left\{\frac{P_{k+1} - P_k}{b-a}\right\}$. Therefore, by induction, we can express the *kth* derivative of a degree n Bezier curve as a Bernstein polynomial of degree $n-k$. Moreover, to compute the *kth* derivative of a Bezier curve, we can simply differentiate the labels on the first k levels the de Casteljau algorithm and multiply the output by $\frac{n!}{(n-k)!}$ (see Figure 15).

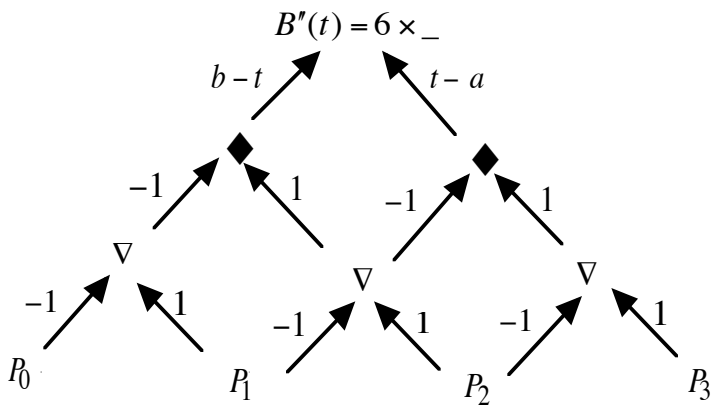


Figure 15: Computing the second derivative of a cubic Bezier curve by differentiating the labels on the bottom two levels of the de Casteljau algorithm. Here to get the correct second derivative $B''(t)$, we must multiply the output of this algorithm by $6 = 3 \times 2$. Also, as in Figure 14, we must remember to normalize the labels along all the arrows, even the constants along the arrows on the bottom two levels, by dividing by $b - a$.

5.1 Smoothly Joining Two Bezier Curves. 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 the derivatives at their end points. Let $B(t)$ be a Bezier curve with control points $P_0,\ldots,P_n$. Substituting $t = a$ or $t = b$ into the differentiation algorithm and recalling the interpolation property at the end points, we see that:

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

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

Suppose then that we are given a Bezier curve $P(t)$ with control points $P_0,\ldots,P_n$ and we want to construct another Bezier curve $Q(t)$ with control points $Q_0,\ldots,Q_n$ so that $Q(t)$ meets $P(t)$ and matches the first r derivatives of $P(t)$ 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 control point at a time. An

alternative approach for finding an explicit formula for the control points $\{Q_k\}$ that avoids all this tedious computation will be presented in Lecture 25. Figure 16 illustrates two cubic Bezier curves that meet with matching first derivatives at their join.

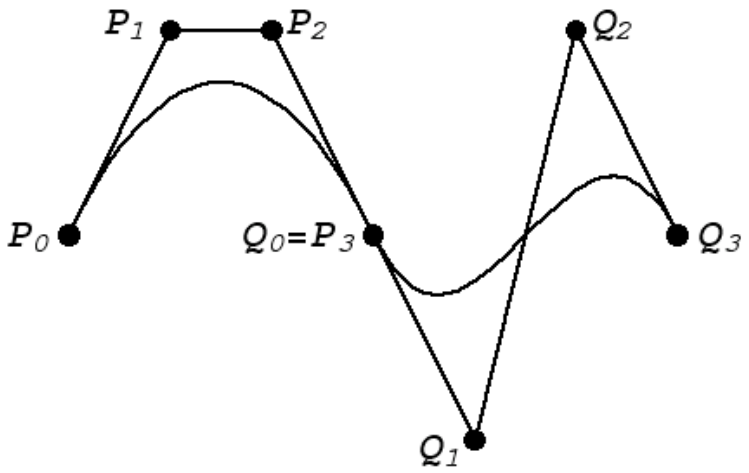


Figure 16: Two cubic Bezier curves -- one with control points P_0,P_1,P_2,P_3 and the other with control points Q_0,Q_1,Q_2,Q_3 -- that meet with matching first derivatives at their join. Here $Q_0 = P_3$ and $Q_1 - Q_0 = P_3 - P_2$.

5.2 Uniqueness of the Bezier Control Points. Another consequence of the differentiation algorithm for Bezier curves is the uniqueness of the Bezier control points of a degree *n* Bezier curve over a fixed parameter interval. For suppose that $P_0,\ldots,P_n$ and $P_0^*,\ldots,P_n^*$ are two distinct sets of control points for the Bezier curve $B(t)$ over the parameter interval $[a,b]$. Substituting $t = a$ into the differentiation algorithm and recalling the interpolation property at the end points, we have the following formulas for the derivatives of $B(t)$ at $t = a$ in terms of $P_0,\ldots,P_n$ and $P_0^*,\ldots,P_n^*$:

$$B(a) = P_0 \qquad B(a) = P_0^*$$

$$B'(a) = n(P_1 - P_0)/(b-a) \qquad B'(a) = n(P_1^* - P_0^*)/(b-a)$$

$$B''(a) = n(n-1)(P_2 - 2P_1 + P_0)/(b-a)^2 \qquad B''(a) = n(n-1)(P_2^* - 2P_1^* + P_0^*)/(b-a)^2$$

and so on up to the *nth* derivative $B^{(n)}(a)$. In general, it follows by induction on *k* that $B^{(k)}(a)$ depends only on the first $k+1$ control points $P_0,\ldots,P_k$ or $P_0^*,\ldots,P_k^*$, and the formulas for these derivatives are identical with respect to $P_0,\ldots,P_k$ and $P_0^*,\ldots,P_k^*$. From the formula for the zeroth derivative, we conclude that $P_0^* = P_0$; from this result and the formula for the first derivative we conclude that $P_1^* = P_1$, and so on. Thus from the formulas for first *n* derivatives, we conclude that $P_k^* = P_k$, $k = 0,\ldots,n$. Hence the control points of a degree *n* Bezier curve over a fixed parameter interval are unique.

6. Tensor Product Bezier Patches

A *tensor product Bezier patch* $B(s,t)$ is a rectangular parametric surface patch -- the image of a planar rectangular domain $[a,b]\times[c,d]$ -- that approximates the shape of a rectangular array of control points $\{P_{ij}\}$, $i=0,\ldots,m$, $j=0,\ldots,n$. 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 a Bezier control polygon controls the shape of a Bezier curve (see Figures 17). In particular, dragging a control point pulls the surface patch in the same general direction as the control point.

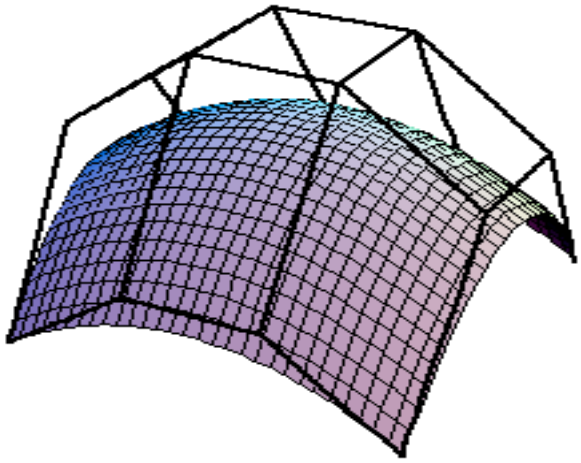


Figure 17: A tensor product Bezier surface with its control polyhedron, formed by connecting control points with adjacent indices.

To construct a tensor product Bezier patch $B(s,t)$ from a rectangular array of control points $\{P_{ij}\}$, $i=0,\ldots,m$, $j=0,\ldots,n$, let $P_i(t)$ be the Bezier curve with control points $P_{i,0},\ldots,P_{i,n}$. For each fixed value of t, let $B(s,t)$ 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 18,19). This surface is called a *tensor product Bezier patch of bidegree* (m,n).

This construction suggests the following evaluation algorithm for tensor product Bezier patches: first use the de Casteljau algorithm $m+1$ times to compute the points at the parameter t along the degree n Bezier curves $P_0(t),\ldots,P_m(t)$; then use the de Casteljau algorithm one more time to compute the point at the parameter s along the degree m Bezier curve with control points $P_0(t),\ldots,P_m(t)$ (see Figure 20).

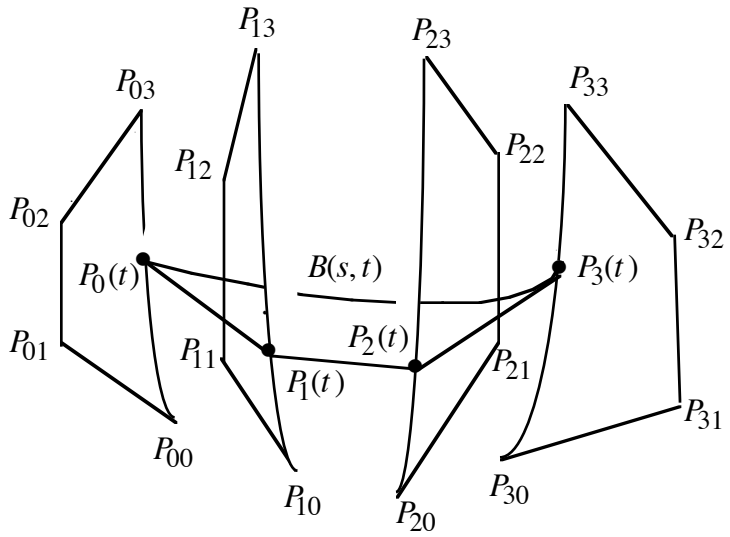


Figure 18: Construction of points on a bicubic tensor product Bezier surface $B(s,t)$. First the Bezier curves $P_i(t)$, $i = 0,...,3$ are constructed from the control points $P_{i0}, P_{i1}, P_{i2}, P_{i3}$. Then for a fixed value of t, the Bezier curve $B(s,t)$ is constructed using the points $P_0(t), P_1(t), P_2(t), P_3(t)$ as control points. As s varies from a to b and t varies from c to d, these curves sweep out a surface patch -- see Figure 19.

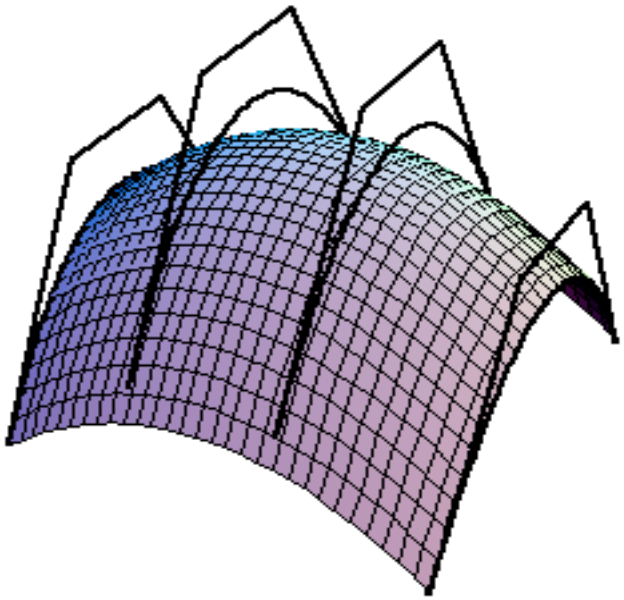


Figure 19: The bicubic Bezier patch in Figure 17 along with its cubic Bezier control curves $P_0(t),...,P_3(t)$. Notice that only the boundary control curves are interpolated by the surface.

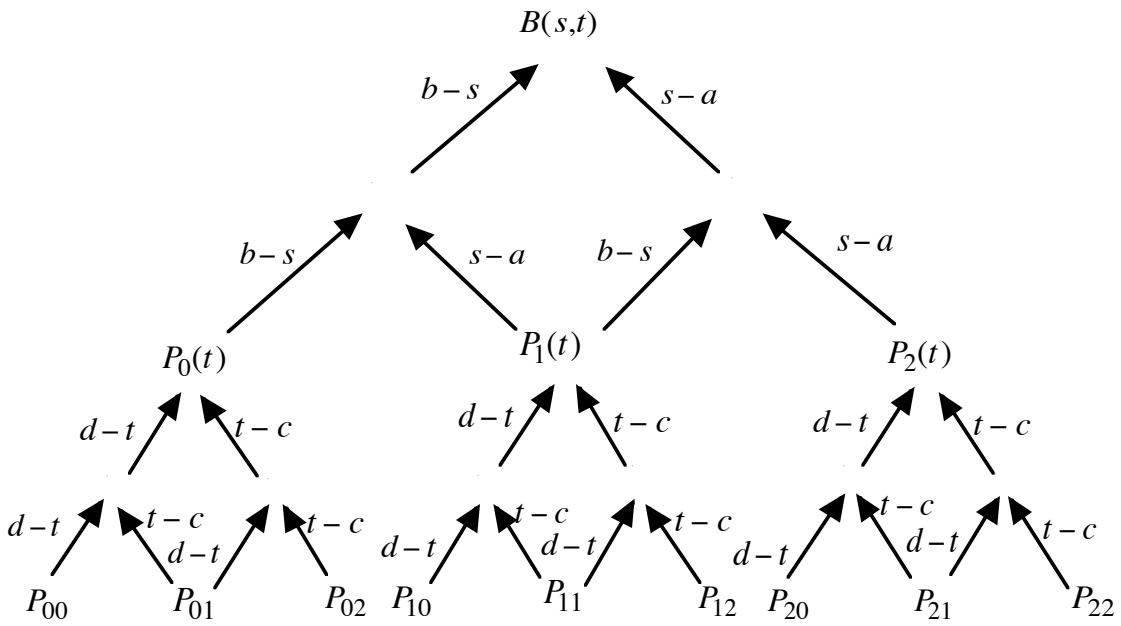


Figure 20: The de Casteljau evaluation algorithm for a biquadratic Bezier patch. The three lower triangles represent Bezier curves in the t direction; the upper triangle blends these curves in the s direction.

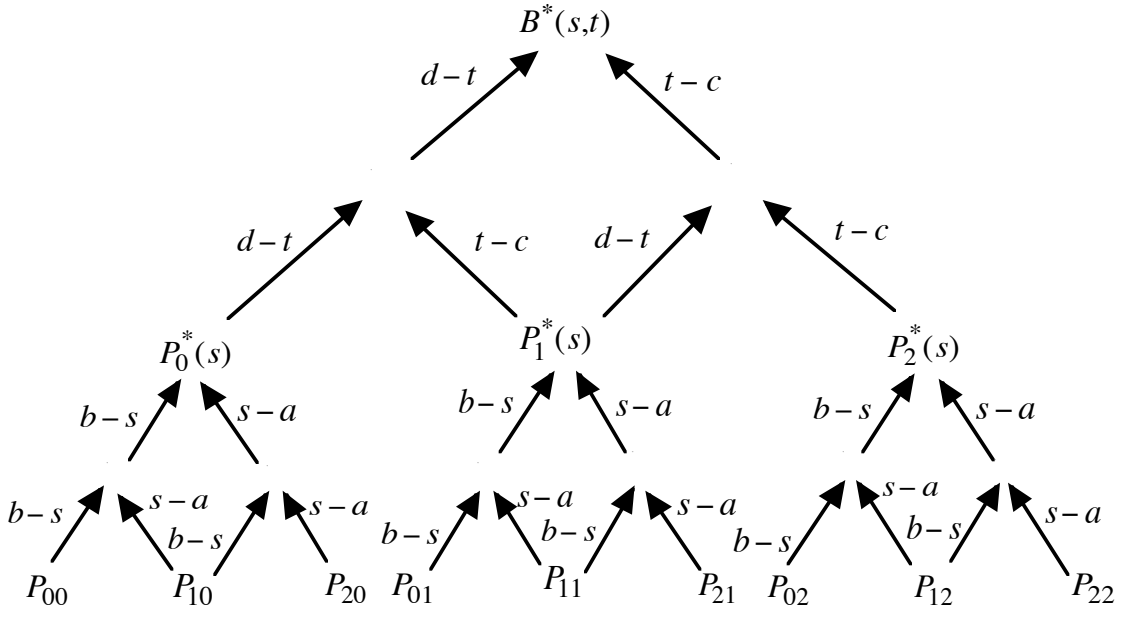


Figure 21: An alternative version of the de Casteljau evaluation algorithm for a biquadratic Bezier patch with the same control points as in Figure 20. The three lower triangles represent Bezier curves in the s direction, and the upper triangle blends these curves in the t direction. Compare to Figure 20.

Alternatively, instead of starting with the Bezier curves $P_0(t),\ldots,P_m(t)$, we could begin with the Bezier curves $P_0^*(s),\ldots,P_n^*(s)$, where $P_j^*(s)$ is the degree m Bezier curve with control points $P_{0,j},\ldots,P_{m,j}$. Now for each fixed value of s, let $B^*(s,t)$ 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, and we can use the de Casteljau algorithm to evaluate points along this surface (see Figure 21).

The surface $B^*(s,t)$ is exactly the same as the surface $B(s,t)$ because both of these surfaces have the same Bernstein representation. To compute the Bernstein representation of the Bezier patch $B(s,t)$, recall that for a fixed value of t, $B(s,t)$ is the Bezier curve with control points $P_0(t),\ldots,P_m(t)$. Therefore

$$B(s,t) = \sum_{i=0}^{m} B_i^m(s) P_i(t). \tag{6.1}$$

Moreover, $P_i(t)$ is the Bezier curve with control points $P_{i,0},\ldots,P_{i,n}$, so

$$P_i(t) = \sum_{j=0}^{n} B_j^n(t)\, P_{i,j}. \tag{6.2}$$

Substituting Equation (6.2) into Equation (6.1) yields the Bernstein representation

$$B(s,t) = \sum_{i=0}^{m} \sum_{j=0}^{n} B_i^m(s) B_j^n(t) P_{i,j}. \tag{6.3}$$

Similarly,

$$B^*(s,t) = \sum_{j=0}^{n} B_j^n(t) P_j^*(s) \tag{6.4}$$

where

$$P_j^*(s) = \sum_{i=0}^{m} B_i^m(s) P_{i,j}, \tag{6.5}$$

so

$$B^*(s,t) = \sum_{j=0}^{n} \sum_{i=0}^{m} B_i^m(s) B_j^n(t) P_{i,j}. \tag{6.6}$$

Hence $B^*(s,t) = B(s,t)$ Thus it does not matter which version of the de Casteljau algorithm we apply; both constructions generate the same surface. However, if $m < n$, then the de Casteljau algorithm for $B(s,t)$ is more efficient, whereas if $n < m$ then the de Casteljau algorithm for $B^*(s,t)$ is more efficient.

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 12,13). These properties follow easily from the de Casteljau algorithm for Bezier patches (Figures 20, 21) 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

$$B(a,t) = P_0(t) \qquad B(s,c) = P_0^*(s)$$
$$\text{and}$$
$$B(b,t) = P_m(t) \qquad B(s,d) = P_n^*(s).$$

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

There is no known analogue of the variation diminishing property for tensor product Bezier patches. Thus although Bezier patches typically follow the shape of their control polyhedra, there is no theorem which guarantees that Bezier surfaces do not oscillate more than their control points.

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 20. 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 the first upper s level of the diagram and multiplying the result by the degree in s (see Figure 22). Symmetric results hold for differentiating the algorithm in Figure 21: simply reverse the roles of s and t. The normal vector N to a Bezier patch $B(s,t)$ is given by setting $N = \frac{\partial B}{\partial s} \times \frac{\partial B}{\partial t}$.

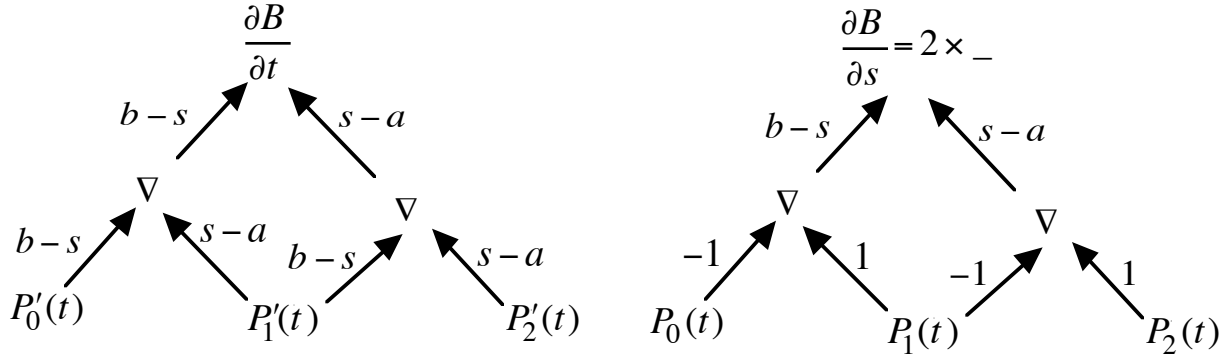
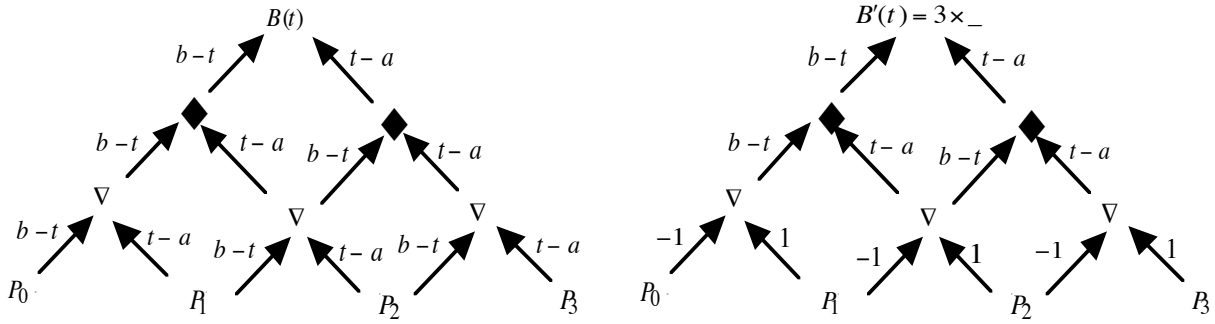


Figure 22: Computing the partial derivatives of a biquadratic Bezier patch by applying the procedure for differentiating the de Casteljau algorithm for Bezier curves. To find $\partial B / \partial t$ (left) simply differentiate the de Casteljau algorithm for each of the Bezier curves $P_0(t), P_1(t), P_2(t)$. To find $\partial B / \partial s$ (right) simply differentiate the first level of the de Casteljau algorithm in s and multiply the result by the degree in s.

7. Summary

The fundamental algorithm for Bezier curves and surfaces is the de Casteljau evaluation algorithm, an algorithm based on repeated linear interpolation (Figure 23(a)). The decomposition of evaluation into successive identical linear interpolation steps makes the evaluation algorithm easy to differentiate (Figure 23(b)), allowing us to compute derivatives as well as points along a Bezier curve using a variant of the de Casteljau algorithm. We applied the differentiation algorithm to derive constraints on the control points to guarantee that two Bezier curves meet smoothly at their join. We also used the differentiation algorithm to establish the uniqueness of the degree n Bezier control points over a fixed parameter interval.



(a) Evaluation (b) Differentiation

Figure 23: The de Casteljau algorithm for (a) evaluation and (b) differentiation.

The Bernstein representation provides an alternative explicit polynomial representation for Bezier curves. If $B(t)$ is the Bezier curve over the interval $[a,b]$ with control points $P_0,\ldots,P_n$, then the Bernstein representation is given by setting

$$B(t) = \sum_{k=0}^{n} B_k^n(t) P_k \qquad a \le t \le b$$

$$B_k^n(t) = \binom{n}{k} \frac{(t-a)^k (b-t)^{n-k}}{(b-a)^n} \qquad k = 0,\ldots,n.$$

Bezier curves have the following important geometric properties:

1. Affine Invariance
2. Convex Hull Property
3. Variation Diminishing Property
4. Interpolate their First and Last Control Points

Tensor product Bezier patches are an extension of the Bezier representation from curves to surfaces. Bezier patches can be evaluated either by extending the de Casteljau algorithm to surfaces or by invoking the Bernstein representation in two variables. Bezier patches are affine invariant, lie

in the convex hull of their control points, and interpolate the boundary Bezier curves defined by their boundary control points. There is, however, no known analogue of the variation diminishing property for Bezier surfaces.

Exercises:

1. Give an example to show that Bezier curves depend on the order of their control points $P_0,...,P_n$ -- that is, if we change the order but not the location of the control points, we may generate a different Bezier curve.

2. Prove that the Bezier curve for the control points $P_n,...,P_0$ is the same as the Bezier curve with the control points $P_0,...,P_n$ but with opposite orientation.

3. Show that a Bezier curve collapses to a single point P if and only if all the Bezier control points are located at P.

4. Consider the Bernstein basis functions

$$B_k^n(t) = \binom{n}{k} \frac{(t-a)^k (b-t)^{n-k}}{(b-a)^n} \qquad k = 0,\ldots,n .$$

Show that:

a. $\sum_{k=0}^{n} B_k^n(t) \equiv 1$

b. $0 \le B_k^n(t) \le 1$ for $a \le t \le b$

c. $B_k^n(a) = \delta_{k,0}$ and $B_k^n(b) = \delta_{k,n}$

5. Using the results in Exercise 4, show that:
   a. Bezier curves are translation invariant.
   c. Bezier curves interpolate their first and last control points.

6. Prove that

a. $$ConvexHull(P_0,\ldots,P_n) = \left\{ \sum_{k=0}^{n} c_k P_k \mid \sum_{k=0}^{n} c_k \equiv 1 \ \ and \ \ c_k \ge 0 \right\}.$$

7. Using the results of Exercises 5 and 6, show that Bezier curves lie in the convex hull of their control points.

8. Consider a Bezier curve

$$B(t) = \sum_{k=0}^{n} B_k^n(t) P_k$$

where

$$B_k^n(t) = \binom{n}{k} \frac{(t-a)^k (b-t)^{n-k}}{(b-a)^n} \qquad k = 0,\ldots,n$$

are the Bernstein basis functions. Show that:

a $\dfrac{dB_k^n(t)}{dt} = n\left(\dfrac{B_{k-1}^{n-1}(t) - B_k^{n-1}(t)}{b-a}\right).$

b. $B'(t) = n \sum_{k=0}^{n-1} B_k^{n-1}(t)\left(\dfrac{P_{k+1} - P_k}{b-a}\right).$

9. Let $P_{i_0\cdots i_n}(t)$ denote the degree n polynomial curve that interpolates the control points $P_{i_0},...,P_{i_n}$ at the parameter values $t_{i_0},...,t_{i_n}$ -- that is, $P_{i_0\cdots i_n}(t_{i_k}) = P_{i_k}$, $k = 0,\ldots,n$. Show that:

a. $P_{01}(t) = \dfrac{t_1 - t}{t_1 - t_0} P_0 + \dfrac{t - t_0}{t_1 - t_0} P_1$

b. $P_{012}(t)$ and $P_{0123}(t)$ can be built using the dynamic programming algorithm (Neville's algorithm) presented in Figure 24.

c. Explain how to extend the dynamic programming algorithm in Figure 24 to $P_{0\cdots n}(t)$.

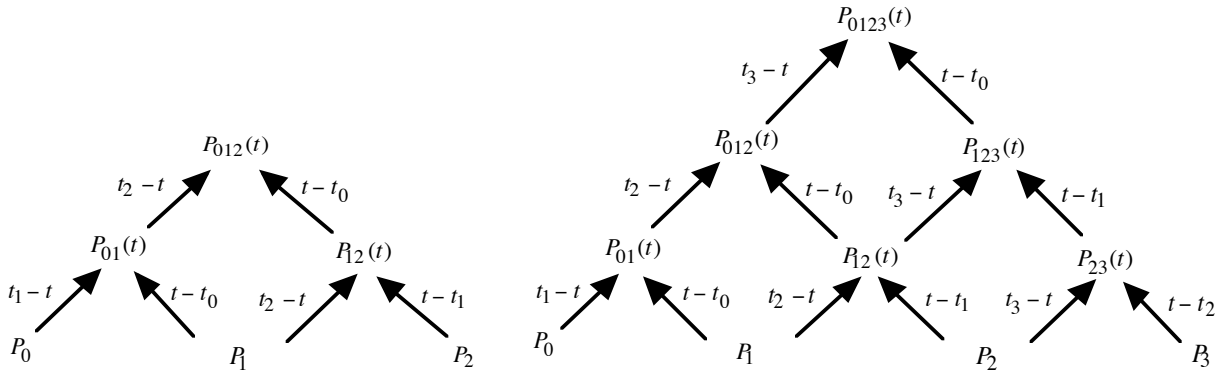


Figure 24: Neville's algorithm for computing points on the polynomial curve that interpolates the points at the base of the diagram. The value at each node must be normalized in the usual manner by dividing by the sum of the labels along the arrows that enter the node.

10. Consider a Bezier curve with control points $P_0,...,P_n$. Show that:
    a. the line segment P_0P_1 is tangent to the curve at P_0;
    b. the line segment $P_{n-1}P_n$ is tangent to the curve at P_n.

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

12. Show that tensor product Bezier patches
    a. are affine invariant;
    b. lie in the convex hull of their control points.

13. Show that every tensor product Bezier patch interpolates
    a. the four Bezier curves defined by the boundary control points;
    b. the four corner control points.