Lecture 25: Bezier Subdivision

And he took unto him all these, and divided them in the midst, and laid each piece one against another: Genesis 15:10

1. Divide and Conquer

If we are going to build useful computer models of freeform shapes using Bezier curves and surfaces, we need procedures to do more than simply evaluate points along these curves and surfaces. We need algorithms to analyze their geometry. For example, to ray trace a Bezier patch, we need algorithms both for finding surface normals and for computing ray-surface intersections. In Lecture 24 we developed an algorithm for differentiating Bezier curves and surfaces, which allows us to find the surface normal of a Bezier patch at any parameter value by taking the cross product of the partial derivatives of the patch. Here we shall develop an algorithm that will permit us to compute the intersection of an arbitrary ray with a Bezier patch.

The method we shall employ is a divide and conquer technique called *Bezier subdivision*. The main idea behind Bezier subdivision is that although globally Bezier curves and surfaces represent curved shapes, if we divide these curves and surfaces into a collection of small enough segments, each segment will be almost flat. Flat shapes like lines and planes are simple to analyze. For example, we can easily ray trace a planar polygon. Thus we shall analyze Bezier curves and surfaces by dividing these curves and surfaces into lots of small flat segments, analyzing the individual flat segments, and then recombining the results to build an understanding of the global geometry of the entire curve or surface.

Bezier subdivision is a powerful tool with many applications. In this lecture we shall use Bezier subdivision to develop fast, robust rendering and intersection algorithms for Bezier curves and surfaces, We shall also use Bezier subdivision to prove the variation diminishing property for Bezier curves and to show how to connect two Bezier curves smoothly at their join.

2. The de Casteljau Subdivision Algorithm

The control points of a Bezier curve describe a polynomial curve with respect to a fixed parameter interval $[a,b]$. Sometimes, however, we are interested only in the part of the polynomial curve in a subinterval of $[a,b]$. For example, when rendering a Bezier curve, we may need to clip the curve to a window on the graphics terminal. Since any segment of a Bezier curve is itself a polynomial curve, we should be able to represent each segment of a Bezier curve as a Bezier curve with a new set of control points. Splitting a Bezier curve into smaller pieces is also useful as a divide and conquer strategy for rendering and intersection algorithms. The process of splitting a

Bezier curve into two or more Bezier curves that represent the exact same curve is called *Bezier subdivision* (see Figure 1)

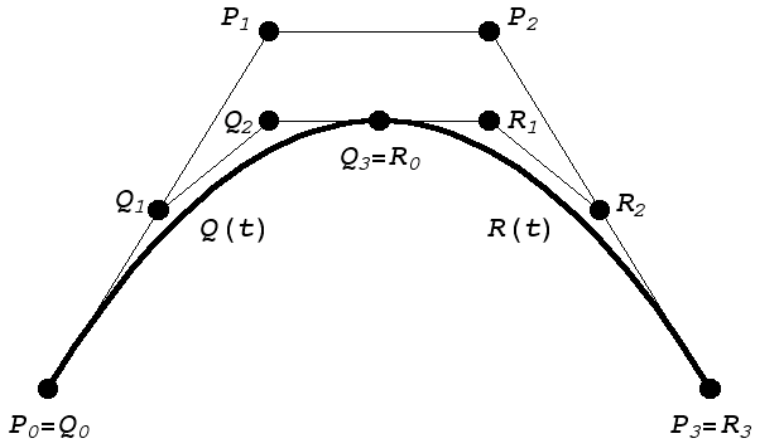


Figure 1: Bezier subdivision for a cubic Bezier curve. The Bezier curve $P(t)$ with control points P_0, P_1, P_2, P_3 is split into two Bezier curves: the left Bezier segment $Q(t)$ has control points Q_0, Q_1, Q_2, Q_3; the right Bezier segment $R(t)$ has control points R_0, R_1, R_2, R_3.

The basic problem in Bezier subdivision is: given a collection of control points $P_0, \ldots, P_n$ that describe a Bezier curve over the interval $[a,b]$, find the control points $Q_0, \ldots, Q_n$ and $R_0, \ldots, R_n$ that describe the segments of the same Bezier curve over the intervals $[a,c]$ and $[c,b]$. Remarkably the solution to this problem is provided by the same de Casteljau algorithm that we use to evaluate points on a Bezier curve.

The de Casteljau Subdivision Algorithm

Let $P(t)$ be a Bezier curve over the interval $[a,b]$ with control points $P_0, \ldots, P_n$. To subdivide $P(t)$ at $t = c$, run the de Casteljau evaluation algorithm for $P(t)$ at $t = c$. The points $Q_0, \ldots, Q_n$ that emerge along the left lateral edge of the de Casteljau diagram are the Bezier control points for the segment of the curve from $t = a$ to $t = c$, and the points $R_0, \ldots, R_n$ that emerge along the right lateral edge of the de Casteljau diagram are the Bezier control points for the segment of the curve from $t = c$ to $t = b$ (see Figure 2).

Notice that when $c = (a+b)/2$ is the midpoint of the parameter interval $[a,b]$, then $(b-c)/(b-a) = 1/2 = (c-a)/(b-a)$, so the labels along all the edges in the de Casteljau subdivision algorithm evaluate to 1/2. Thus for midpoint subdivision the de Casteljau subdivision algorithm is independent of the parameter interval (see Figure 3).

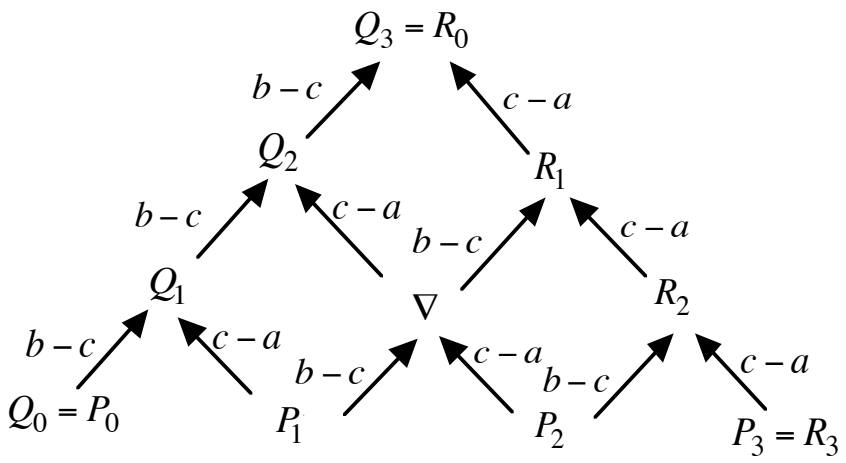


Figure 2: The de Casteljau subdivision algorithm for a cubic Bezier curve with control points P_0, P_1, P_2, P_3. The points Q_0, Q_1, Q_2, Q_3 that emerge along the left edge of the triangle are the Bezier control points for the segment of the original curve from $t = a$ to $t = c$; the points R_0, R_1, R_2, R_3 that emerge along the right edge of the triangle are the Bezier control points for the segment of the original curve from $t = c$ to $t = b$.

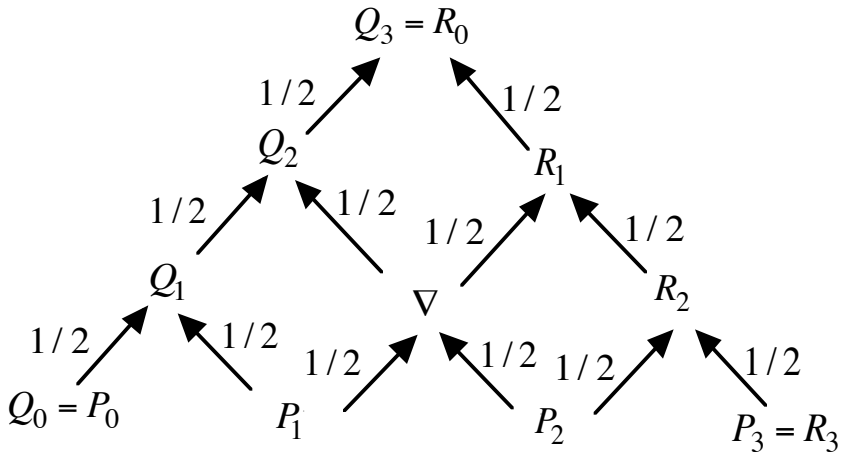


Figure 3: The de Casteljau subdivision algorithm for a cubic Bezier curve at the midpoint of the parameter interval. The labels along all the edges evaluate to 1/2, so midpoint subdivision does not depend on the choice of the parameter interval.

We shall defer the proof of the de Casteljau subdivision algorithm till Lecture 26. In this lecture we will study the consequences of this subdivision algorithm. Figure 4 provides a geometric interpretation for the de Casteljau subdivision algorithm.

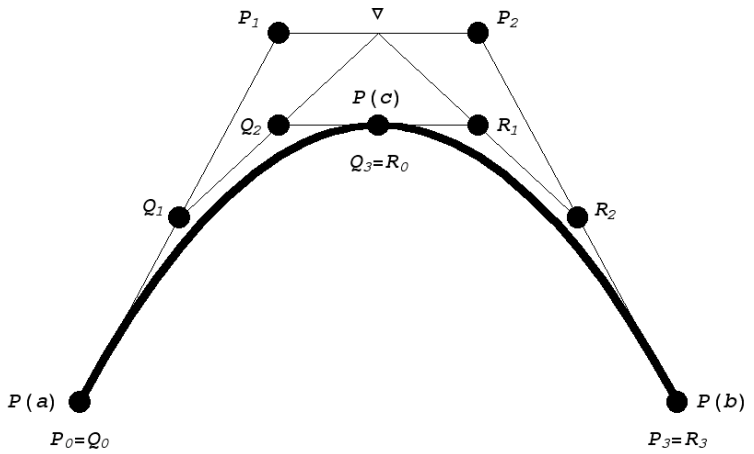


Figure 4: Geometric interpretation of the de Casteljau subdivision algorithm for a cubic Bezier curve $P(t)$ with control points P_0, P_1, P_2, P_3. The points Q_0, Q_1, Q_2, Q_3 are the Bezier control points of the segment of the original curve from $t = a$ to $t = c$, and the points R_0, R_1, R_2, R_3 are the Bezier control points of the segment of the original curve from $t = c$ to $t = b$.

When we subdivide a Bezier curve, the new control polygons appear closer to the Bezier curve than the initial control polygon (see Figures 1 and 4). Suppose we continue recursively subdividing each Bezier segment. Then in the limit, these control polygons converge to a smooth curve (see Figure 5). We shall now show that this limit curve is, in fact, the Bezier curve for the original control polygon. For convenience, we will restrict our attention to recursive subdivision at the midpoint of the parameter interval, though the results are much the same for any value that splits the parameter intervals in a constant ratio,.

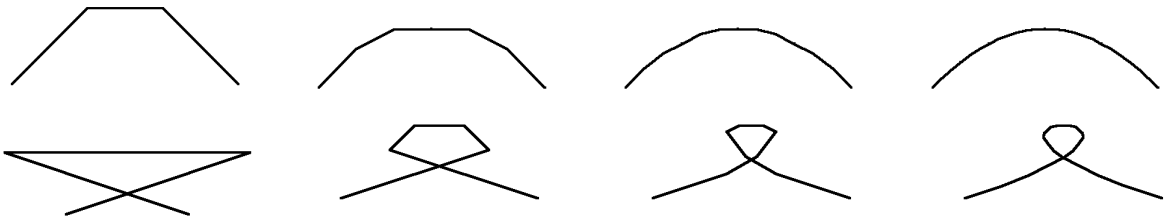


Figure 5: Control polygons converge to Bezier curves under recursive subdivision. Here the original control polygons (left) and the first three levels of subdivision (right) are illustrated.

Theorem 1: *The control polygons generated by recursive subdivision converge to the Bezier curve for the original control polygon.*

Proof: Suppose that the maximum distance between any two adjacent control points is *d*. By construction, the points on any level of the de Casteljau algorithm evaluated at the midpoint of the parameter domain lie at the midpoints of the edges of the polygons generated by the previous level because the labels along all the edges evaluate to 1/2 (see Figure 3). Therefore it follows easily by

induction that adjacent points on any level of the de Casteljau diagram are no further than d units apart (see Figure 6).

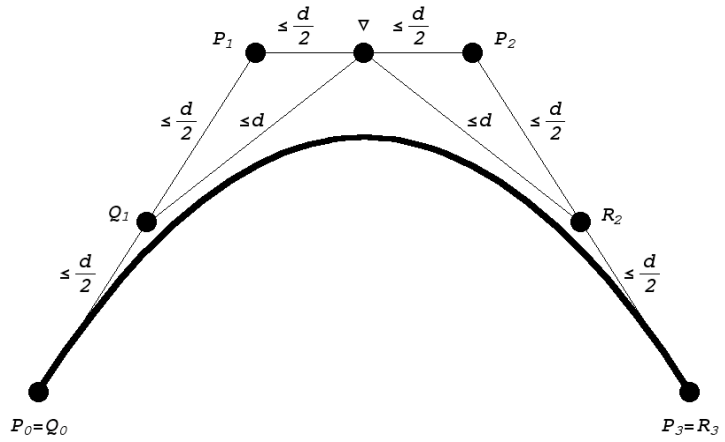


Figure 6: One level of the de Casteljau algorithm for a cubic Bezier curve evaluated at the midpoint of the parameter domain. If adjacent control points P_0, P_1, P_2, P_3 are no further than d units apart, then adjacent points Q_1, ∇, R_2 on the first level of the de Casteljau algorithm can also be no further that d units apart.

By the same midpoint argument, as we proceed up the de Casteljau diagram adjacent points along the left or right lateral edge of the triangle can be no further than $d/2$ units apart. Hence as we apply recursive subdivision, the distance between the control points of any single control polygon must converge to zero. Since the first and last control points of a Bezier control polygon always lie on the Bezier curve, these control polygons must converge to points along the Bezier curve for the original control polygon.

We can also subdivide tensor product Bezier patches. Recall from Lecture 24 that a Bezier surface $B(s,t)$ is defined by applying the de Casteljau algorithm in the parameter s to a collection of control points defined by the Bezier curves $P_0(t),\ldots,P_m(t)$ in the parameter t. To subdivide the Bezier patch along the parameter line $t = t_0$, simply subdivide each of the curves $P_0(t),\ldots,P_m(t)$ at $t = t_0$. Similarly, to subdivide the Bezier patch along a parameter line $s = s_0$ recall that the same Bezier surface $B(s,t)$ is defined by applying the de Casteljau algorithm in the parameter t to a collection of control points defined by the Bezier curves $P_0^*(s),\ldots,P_n^*(s)$ in the parameter s. Thus to subdivide the Bezier patch along the parameter line $s = s_0$, simply subdivide each of the curves $P_0^*(s),\ldots,P_n^*(s)$ at $s = s_0$. Now we have the following result.

Theorem 2: *The control polyhedra generated by recursive subdivision converge to the tensor product Bezier patch for the original control polyhedron provided that the subdivision is done in both the s and t directions.*

The proof of Theorem 2 for Bezier surfaces is much the same as the proof of Theorem 1 for Bezier curves, so we shall not repeat the proof here. Notice, however, that we must be careful to subdivide in both the *s* and the *t* directions. If we subdivide only along one parameter direction, convergence is not assured because in the limit each of the control polyhedra will not necessarily shrink to a single point. Thus to guarantee convergence it is best to alternate subdividing in the *s* and *t* parameter directions.

3. Rendering and Intersection Algorithms

Subdivision is our main mathematical tool for analyzing Bezier curves and surfaces. Here we shall apply recursive subdivision to develop fast, robust algorithms for rendering and intersecting Bezier curves and surfaces.

3.1 Rendering and Intersecting Bezier Curves. To render a Bezier curve, we could begin by applying the de Casteljau evaluation algorithm to compute lots of points along the curve. We could then display a dense collection of points on the curve or we could connect the points on the curve with straight lines and display these line segments.

But Bezier curves are smooth curves. If we applied this approach to rendering Bezier curves, how many points would we need to compute in order for the curve to appear smooth? Some parts of a Bezier curve may be relatively flat, so in these segments we would need to compute only a few points and connect these points with straight lines, whereas other parts of a Bezier curve may be highly bent and in these locations we would need to compute quite a lot of points for the line segments approximating the curve to appear smooth (see Figure 7). The de Casteljau evaluation algorithm allows us to compute lots of points along a Bezier curve. But if we want an accurate portrayal of a Bezier curve, it is not clear how many points to compute or where on the curve to concentrate these computations.

Recursive subdivision solves both of these problems: only a small amount of subdivision is required on segments where the curve is relatively flat while additional subdivision can be performed on segments where the curve is highly bent. The convergence of recursive subdivision is quite fast and the output of recursive subdivision appears smooth (see Figure 5). This rapid convergence and smooth appearance leads to the following recursive subdivision algorithm for rendering Bezier curves.

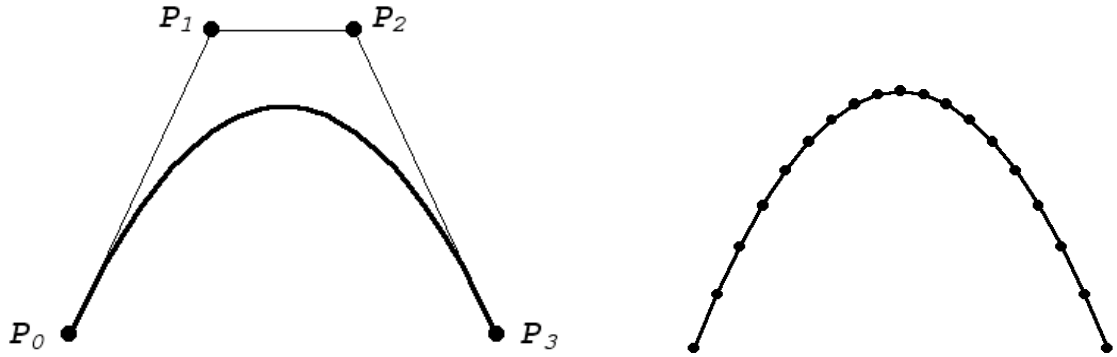


Figure 7: A cubic Bezier curve that is relatively flat near the end points but highly curved towards the center (left). Thus near the end points we need to compute only a few points and connect these points with straight lines, whereas near the center we need to compute quite a lot of points for the line segments approximating the curve to appear smooth (right).

Rendering Algorithm -- Bezier Curves

If the Bezier curve can be approximated to within tolerance by the straight line segment joining its first and last control points, then draw either this line segment or the control polygon.
Otherwise *subdivide* the Bezier curve (at the midpoint of the parameter interval) and render the segments recursively.

To intersect two Bezier curves, we could use the rendering algorithm to generate a piecewise linear approximation for each of the two curves and then intersect all these line segments. But this approach would be highly inefficient because most of the short line segments would not intersect. We can avoid these needless computations by combining recursive subdivision with the convex hull property in order to avoid trying to compute intersections for those parts of the curve that fail to intersect.

Intersection Algorithm -- Bezier Curves

If the convex hulls of the control points of two Bezier curves fail to intersect, then the curves themselves do not intersect.
Otherwise if each Bezier curve can be approximated by the straight line segment joining its first and last control points, then intersect these line segments.
Otherwise *subdivide* the two Bezier curves and intersect the pieces recursively.

To determine whether a Bezier curve can be approximated to within tolerance by the straight line segment joining its first and last control points, it is sufficient, by the convex hull property, to test whether all the interior control points lie within tolerance of this line segment. But recall from Lecture 11, Section 4.1.2 that the distance between a point P and a line L determined by a point Q and a direction vector v is given by

$$dist^2(P,L) = |P-Q|^2 - \frac{\left((P-Q)\bullet v\right)^2}{v\bullet v}. \qquad (3.1)$$

Therefore to test whether the control points P_k, $k = 1,\ldots,n-1$ lie within some tolerance ε of the line determined by the first and last control points P_0, P_n, we set $v = P_n - P_0$ in Equation (3.1) and test whether or not

$$\left|P_k - P_0\right|^2 - \frac{\left((P_k - P_0)\bullet(P_n - P_0)\right)^2}{(P_n - P_0)\bullet(P_n - P_0)} < \varepsilon. \qquad (3.2)$$

Be careful. It may happen that a control point P_k is close to the line L determined by the first and last control points P_0, P_n even though the orthogonal projection of P_k onto L does not lie inside the line segment P_0P_n -- that is, P_k may be close to the line L even though it is not close to the line segment P_0P_n. To be sure that P_k lies close to the line segment P_0P_n, you need only check that in addition to Equation (3.2)

$$0 \le (P_k - P_0)\bullet(P_n - P_0) \le |P_n - P_0|^2. \qquad (3.3)$$

It is relatively easy to test whether or not a Bezier curve can be approximated to within some tolerance by a straight line segment. On the other hand, finding and intersecting the convex hulls of two Bezier curves can be quite difficult and time consuming. In practice, the convex hulls in the intersection algorithm are typically replaced by bounding boxes which are much easier to compute and intersect than the actual convex hulls: simply take the minimum and maximum x and y coordinates of the control points. Since the subdivision algorithm converges rapidly, not much time is lost by replacing convex hulls with bounding boxes.

To complete the intersection algorithm for Bezier curves, we need to be able to intersect two line segments:

$$L_1(s) = (1-s)P_0 + sP_1 = P_0 + s(P_1 - P_0) \qquad 0 \le s \le 1$$
$$L_2(t) = (1-t)Q_0 + tQ_1 = Q_0 + t(Q_1 - Q_0) \qquad 0 \le t \le 1$$

Two infinite lines intersect when $L_1(s) = L_2(t)$ -- that is, when

$$P_0 + s(P_1 - P_0) = Q_0 + t(Q_1 - Q_0)$$

or equivalently when

$$s(P_1 - P_0) - t(Q_1 - Q_0) = (Q_0 - P_0).$$

Dotting both sides first with $P_1 - P_0$ and then with $Q_1 - Q_0$ generates two linear equations in two unknowns.

$$\begin{aligned} s\{(P_1 - P_0)\bullet(P_1 - P_0)\} - t\{(Q_1 - Q_0)\bullet(P_1 - P_0)\} &= (Q_0 - P_0)\bullet(P_1 - P_0) \\ s\{(P_1 - P_0)\bullet(Q_1 - Q_0)\} - t\{(Q_1 - Q_0)\bullet(Q_1 - Q_0)\} &= (Q_0 - P_0)\bullet(Q_1 - Q_0) \end{aligned} \qquad (3.4)$$

which are easy to solve for the parameters s,t (see Lecture 11, Section 5.1). The intersection point lies on the two line segments if and only $0 \le s,t \le 1$. In this case we can compute $L_1(s)$ or $L_2(t)$ to find the actual intersection point.

3.2 Rendering and Intersecting Bezier Surfaces. The rendering and intersection algorithms for Bezier curves can be extended to rendering and intersection algorithms for Bezier surfaces. Line are replaced by planes, line segments are replaced by triangles. and recursive subdivision for Bezier curves is replaced by recursive subdivision for Bezier surfaces. To render a Bezier patch, we must first polygonalize the patch, so we begin with an algorithm to approximate a Bezier patch by a collection of triangles.

Triangulation Algorithm -- Bezier Surfaces

If the Bezier patch can be approximated to within tolerance by two triangles each determined by three of its four corner control points and if the four Bezier boundaries of the Bezier patch can be approximated by straight line segments joining the four corner control points, then triangulate the Bezier patch by the two triangles determined by the four corner control points.
Otherwise *subdivide* the Bezier surface (at the midpoint of the parameter interval in *s* or *t*) and triangulate the segments recursively.

Rendering Algorithm -- Bezier Surfaces

Triangulate the Bezier patch
Apply your favorite shading and hidden surface algorithms to render the triangulated patch.

Ray Tracing Algorithm -- Bezier Surfaces

If the ray does not intersect the convex hull of the control points of the Bezier patch, then the ray and the patch do not intersect.
Otherwise if the Bezier patch can be approximated to within tolerance by two triangles each determined by three of its four corner control points and if the four Bezier boundaries of the Bezier patch can be approximated by straight line segments joining the four corner control points, then intersect the ray with the two triangles determined by the four corner control points.
Otherwise *subdivide* the Bezier patch and ray trace the segments recursively.
Keep the intersection closest to the eye.

We already know from Section 3.1 how to test if a boundary Bezier curve can be approximated by a straight line joining two of the boundary control points. We need to perform this test in the triangulation algorithm to be sure that after subdivision cracks do not appear between adjacent triangles. If a boundary is not a straight line, then subdividing on one side of the boundary

but not the other may introduce a tear between adjacent triangles. But when the boundaries are flat, such cracks will not appear.

To determine whether a Bezier surface can be approximated to within tolerance by a pair of triangles determined by its four corner control points, it is sufficient, by the convex hull property, to test whether each control point lies within some tolerance of at least one of the two triangles. Now recall from Lecture 11, Section 4.1.3 that the distance between a point P and a plane S determined by a point Q and a normal vector N is given by

$$dist(P,S) = \frac{|(P-Q)\bullet N|}{|N|}. \qquad (3.5)$$

Therefore to test whether a control points P_{ij} lies within some tolerance ε of the plane determined by the three corner control points P_{00}, P_{0n}, P_{m0}, we set $N = (P_{0n} - P_{00}) \times (P_{m0} - P_{00})$ in Equation (3.5) and test whether or not

$$\frac{|(P_{ij} - P_{00})\bullet N|}{|N|} < \varepsilon \qquad (3.6)$$

If this test fails, we apply the analogous test with the plane determined by the control points P_{mn}, P_{0n}, P_{m0}. Again we must be careful. It may happen that a control point P_{ij} lies close to the plane S of a triangle Δ even though the orthogonal projection of the point P_{ij} onto the plane S does not lie inside the triangle Δ -- that is, P_{ij} may be close to the plane S even though P_{ij} is not close to the triangle Δ. To be sure that this is not the case, you need only check that the orthogonal projection $P_{ij} - \{(P_{ij} - Q)\bullet N\}N$ lies inside Δ. Now recall from Lecture 19, Section 3 that a point R lies inside a triangle with vertices P_0, P_1, P_2 if and only if

$$\det(N, P_{i+1} - P_i, R - P_i) \geq 0 \qquad i = 0,1,2,$$

where

$$N = (P_1 - P_0) \times (P_2 - P_0)$$

is the normal to the plane of the triangle. These tests should be carried out first with $\Delta P_{00}P_{0n}P_{m0}$ and only if these test fail should the tests then be performed with $\Delta P_{mn}P_{0n}P_{m0}$.

Notice that in the ray tracing algorithm we do not simply triangulate the patch and then ray trace the triangulated approximation. Rather we first perform a convex hull test in order to eliminate as many subpatches as possible before we perform intersection calculations.

Finding the convex hull of a Bezier patch can be quite difficult and time consuming. In practice for surfaces, as with curves, the convex hull is replaced by a bounding box. Again since the subdivision algorithm converges rapidly, not much time is lost in the ray tracing algorithm by replacing convex hulls with bounding boxes.

To complete the ray tracing algorithm for Bezier patches, we need to be able to intersect a line with a planar polygon. Here we can use the algorithm described in Lecture 19, Section 3.

Finally a Bezier patch may self shadow -- that is, a light ray may not be able to reach part of a Bezier patch because the light has already struck another part of the patch. Therefore at the end of the ray tracing algorithm we discard all the intersections of the ray and the patch, except for the intersection closest to the eye.

.

4. The Variation Diminishing Property of Bezier Curves

We introduced Bezier curves because, unlike polynomial interpolation, Bezier approximation does not introduce oscillations not already present in the data. This property of Bezier curves is called the *variation diminishing property*. Recall from Lecture 24 that a curve $B(t)$ for a control polygon P is said to be *variation diminishing* if for *every* line L

$$\#\ \textit{intersections of}\ B(t)\ \textit{and}\ L \le \#\ \textit{intersections of}\ P\ \textit{and}\ L\ .$$

We are now going to use Bezier subdivision to prove the variation diminishing property of Bezier curves.

One way to introduce variation diminishing schemes is by corner cutting. Start with a polygon P and form another polygon Q by cutting a corner off P (see Figure 8). Then it is easy to see that every line L intersects P at least as often as L intersects Q. Thus Q is variation diminishing with respect to P.

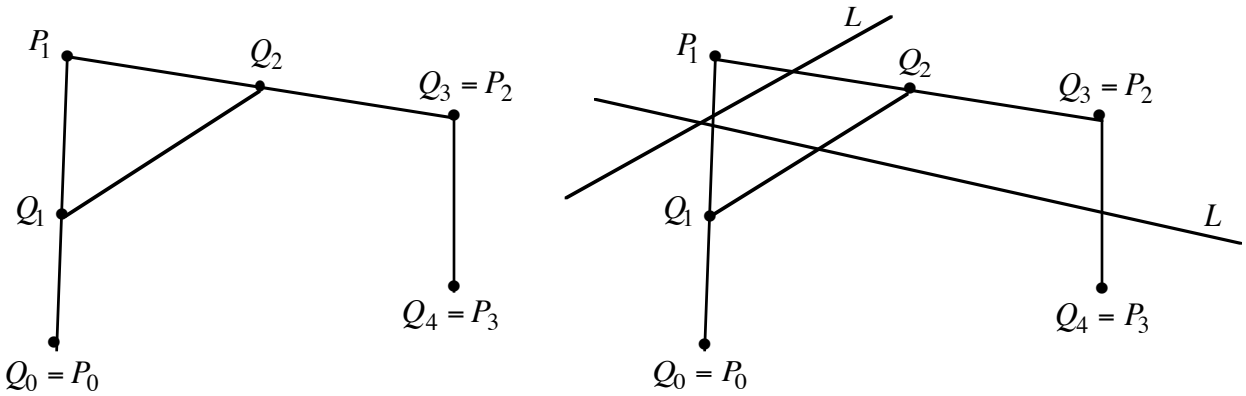
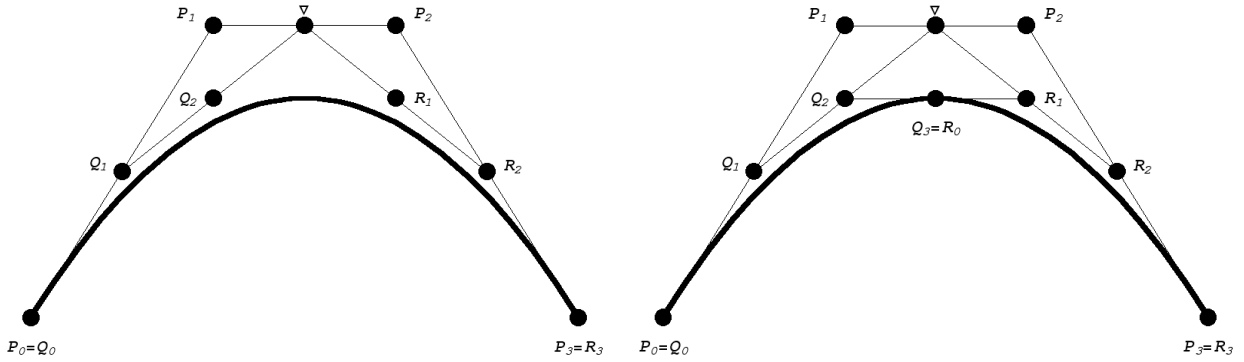


Figure 8: Corner cutting. The polygon with vertices $Q = \{Q_i\}$ is generated from the polygon with vertices $P = \{P_j\}$ by cutting off the corner at P_1 (left). Every line L intersects P at least as often as L intersects Q because if L intersects the line segment Q_1Q_2, then L must intersect either P_0P_1 or P_1P_2 (right).

Now look at the geometric interpretation of the de Casteljau subdivision algorithm in Figure 3. Evidently, the de Casteljau subdivision algorithm is a corner cutting procedure: first the corners at P_1, P_2 are cut off, then the corner at Δ is cut off (see Figure 9).



First step *Second step*

Figure 9: The de Casteljau subdivision algorithm as a sequence of corner cuts. In the first stage corners are cut off at P_1, P_2; in the second stage the corner is cut off at Δ. Compare to Figure 3.

Although we have illustrated only the cubic case in Figure 8, it is easy to verify that the de Casteljau subdivision algorithm is a corner cutting procedure in all degrees. This observation leads to the following result.

Theorem 3: *Bezier curves are variation diminishing*.
Proof: Since recursive subdivision is a corner cutting procedure, the limit curve must be variation diminishing with respect to the original control polygon. But by Theorem 1, the Bezier curve is the limit curve generated by recursive subdivision, so Bezier curves are variation diminishing.

By the way, there is no known analogue of the variation diminishing property for Bezier surfaces because for Bezier surfaces subdivision is not a corner cutting procedure.

5. Joining Bezier Curves Smoothly

In Lecture 24, Section 5 we used our procedure for differentiating the de Casteljau algorithm for Bezier curves to derive constraints on the location of the control points to insure that two Bezier curves meet smoothly where they join. Given a Bezier curve $P(t)$ with control points $P_0,\ldots,P_n$, we derived the following constraints for the location of the control points $Q_0,\ldots,Q_n$ of another Bezier curve $Q(t)$ that meets the first curve and matches the first k derivatives of $P(t)$ where they join.

$$
\begin{aligned}
&k = 0 \;\Rightarrow\; Q_0 = P_n \\
&k = 1 \;\Rightarrow\; Q_1 = P_n + (P_n - P_{n-1}) \qquad : \qquad (5.1) \\
&k = 2 \;\Rightarrow\; Q_2 = P_{n-2} + 4(P_n - P_{n-1})
\end{aligned}
$$

We observed that each derivative generates one new constraint on one additional control point. These constraints are easy to solve, but rather cumbersome to write down. Here we show how to use subdivision to determine the location of the points $Q_0,\ldots,Q_k$ that guarantee k-fold continuity at the join.

We know that the location of the points $Q_0,\ldots,Q_k$ is unique. If we could find a curve $Q(t)$ that meets the original curve $P(t)$ smoothly at the join, then the control points of $Q(t)$ would necessarily give the location of the points $Q_0,\ldots,Q_k$. But we certainly do know such a curve, for $P(t)$ meets itself smoothly at the join! Suppose that $P(t)$ is parameterized over the interval $[a,b]$. Let $Q(t)$ be $P(t)$ over the interval $[b,2b-a]$ -- that is, the interval starting at b and with the same length as $[a,b]$. Then $P(t)$ and $Q(t)$ surely meet smoothly at $t = b$. All we need now are the Bezier control points of $Q(t)$.

To find the Bezier control points of $P(t)$ over the intervals $[a,c]$ and $[c,b]$, we can subdivide the Bezier curve $P(t)$ at $t = c$. Nothing in our subdivision algorithm requires that $a \le c \le b$. If we take $c = 2b - a$, then the de Casteljau subdivision algorithm will generate the Bezier control points for the curve $P(t)$ over the intervals $[a,2b-a]$ and $[2b-a,b]$. By the symmetry property of Bezier curves, (see Lecture 24, Exercise 2) the Bezier control points for the interval $[2b-a,b]$ are the same as the Bezier control points for the interval $[b,2b-a]$ but in reverse order. Thus we can read off the control points $Q_0,\ldots,Q_k$ from the right edge of the de Casteljau subdivision algorithm for $c = 2b - a$. But in this algorithm the labels on all the right pointing arrows are -1 because $\big(b-(2b-a)\big)/(b-a) = -1$ and the labels on all the left pointing arrows are 2 because $\big((2b-a)-a\big)/(b-a) = 2$. Notice that these labels are independent of the interval $[a,b]$; hence this algorithm is independent of the parameter interval. We illustrate this algorithm in Figure 10 for cubic Bezier curves.

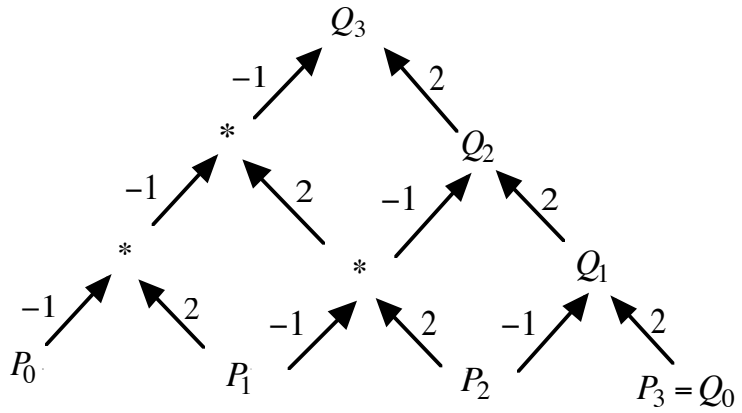


Figure 10: The de Casteljau subdivision algorithm at $c = 2b - a$ for cubic Bezier curves. The first $k+1$ points $Q_0,\ldots,Q_k$ that emerge on the right lateral edge of the diagram are the control points that guarantee *k*-fold continuity at the join. Compare these points to the values in Equations (5.1).

6. Summary

Subdivision is our main technical tool for analyzing Bezier curves and surfaces. In this lecture we used Bezier subdivision to:

i. develop a divide and conquer strategy for rendering and intersecting Bezier curves and surfaces;
ii. prove the variation diminishing property for Bezier curves;
iii. guarantee that two Bezier curves connect with *k*-fold smoothness at their join.

The basic problem in Bezier subdivision is: given a collection of control points $P_0,\ldots,P_n$ that describe a Bezier curve over the parameter interval $[a,b]$, find the control points $Q_0,\ldots,Q_n$ and $R_0,\ldots,R_n$ that describe the segments of the same Bezier curve over the parameter intervals $[a,c]$ and $[c,b]$.

De Casteljau's evaluation algorithm for Bezier curves is also a subdivision procedure. The subdivision control points $Q_0,\ldots,Q_n$ and $R_0,\ldots,R_n$ emerge along the left and right edges of the de Casteljau evaluation algorithm at $t = c$. Bezier subdivision is usually applied at the midpoint of the parameter domain, where the labels along the all the arrows are 1/2 (see Figure 11)..

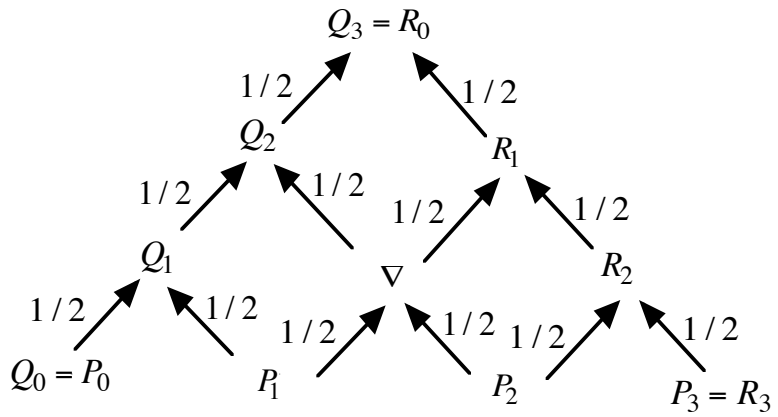


Figure 11: The de Casteljau subdivision algorithm for a cubic Bezier curve at the midpoint of the parameter interval. Midpoint subdivision is independent of the choice of the parameter domain.

Exercises:

1. Let $P(t)$ be a Bezier curve defined over the interval $[a,b]$.
   a. Show that in the de Casteljau subdivision algorithm at $c = (1-r)a + rb$ the labels on all the right pointing arrows are $1-r$ and the labels on all the left pointing arrows are r (see Figure 12).
   b. Conclude from part a that if we subdivide at a fixed ratio of the parameter domain, then the de Casteljau subdivision algorithm is independent of the parameter interval.

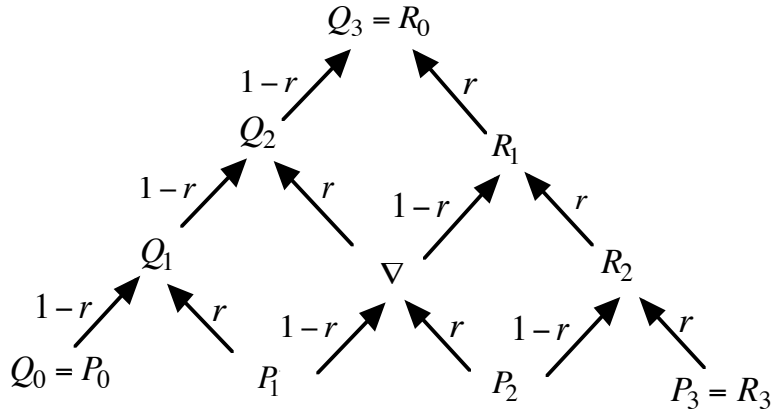


Figure 12: The de Casteljau subdivision algorithm at the parameter value $c = (1-r)a + rb$ for a cubic Bezier curve defined over the interval $[a,b]$. The labels on all the right pointing arrows are $1-r$ and the labels on all the left pointing arrows are r.

2. Consider a Bezier curve defined over the interval [0,1]. Bezier subdivision at $t = 1/2$ generates a binary tree whose nodes are control polygons. Denote the original control polygon by P, and let this polygon be the root of the tree. Let P_0 -- the left child of P -- denote the control polygon for the left segment of the Bezier curve (from $t = 0$ to $t = 1/2$), and let P_1 -- the right child of P -- denote the control polygon for the right portion of the curve (from $t = 1/2$ to $t = 1$). Continue to build this binary tree recursively in this fashion. Thus if P_b is a node in the tree, then P_b represents the control polygon for a portion of the curve, and P_{b0} -- the left child of P_b -- represents the control polygon for the left half of the Bezier segment represented by P_b, while P_{b1} -- the right child of P_b -- represents the control polygon for the right half of the Bezier segment represented by P_b (see Figure 13).

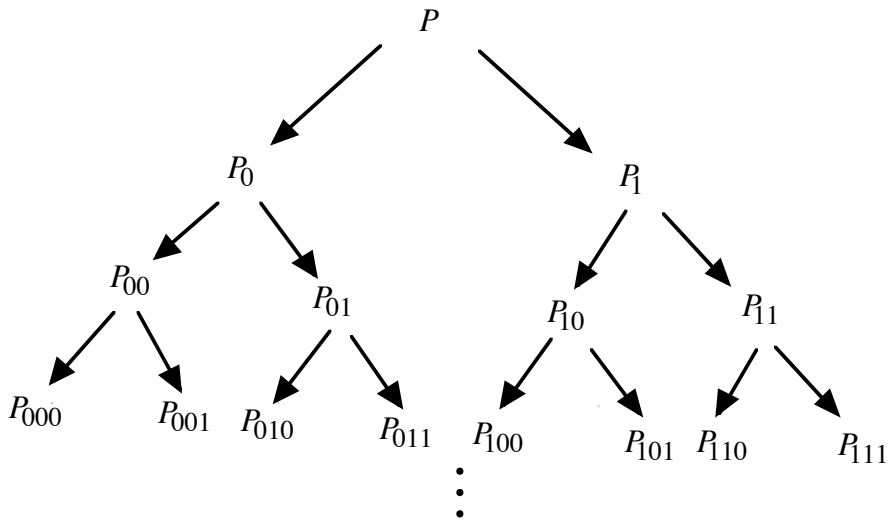


Figure 13: The binary tree of control polygons generated by recursive subdivision of a Bezier curve at $t = 1/2$.

a. Prove that $P_{b_1 \cdots b_n}$ is the control polygon for the original Bezier curve from $t = b$ to $t = b + 2^{-n}$, where b is the binary fraction represented by $0.b_1 \cdots b_n$.
b. Prove that the sequence of control polygons $P_{b_1}, P_{b_1 b_2}, ..., P_{b_1 \cdots b_n}, ...$ converges to the point on the Bezier curve at parameter value $b = Lim_{n \to \infty} 0.b_1 \cdots b_n$.

3. Develop an algorithm to intersect a Bezier curve with a Bezier surface based on recursive subdivision.

4. Develop an algorithm to intersect two Bezier patches based on recursive subdivision.

5. Prove Theorem 2: that the control polyhedra generated by recursive subdivision converge to the tensor product Bezier patch $B(s,t)$ for the original control polyhedron provided that the subdivision is done in both the *s* and *t* directions.

6. Let $P(t)$ be a Bezier curve with control points $P_0,\ldots,P_n$.
   a. Prove that the arc length of a Bezier curve is greater than or equal to the length of the line joining the first and last control points and less than or equal to the perimeter of the control polygon -- that is,

$$|P_n - P_0| \le arclenth(P(t)) \le \sum_{k=0}^{n-1} |P_{k+1} - P_k| .$$

   b. Explain how to use the result of part *a* to compute the arc length of a Bezier curve to within any desired tolerance.

Programming Projects:

1. *Bezier Curves and Surfaces*
   Implement a modeling system based on Bezier curves and surfaces in your favorite programming language using your favorite API.
   a. Include algorithms for rendering Bezier curves and surfaces based on recursive subdivision.
   b. Incorporate the ability to move control points interactively and have the Bezier curve or surface adjust in real time.
   c. Create a new font using Bezier curves.
   d. Build some interesting freeform shapes using Bezier patches.

2. *Root Finding Algorithm for Polynomials in Bezier Form*

A. Implement the following root finding algorithm for polynomials of arbitrary degree *n*.

Input: $c_0,\ldots,c_n$ = Bezier coefficients of $P(t)$ over the interval $[a,b]$

Root Finding Algorithm

1. If $c_i = 0$, $i = 0,\ldots,j-1$, then
   a. there is a multiple root of order *j* at $t = a$; set

$$c_k = \frac{n\cdots(n-j+1)}{(k+j)\cdots(k+1)(b-a)^j} c_{j+k} \qquad k = 0,\ldots,n-j.$$

   $n = n - j$

2. If $c_{n-i} = 0$, $i = 0,\ldots,j-1$, then
   a. there is a multiple root of order *j* at $t = b$; set

$$c_k = \frac{n \cdots (n-j+1)}{(n-k) \cdots (n-j-k+1)(b-a)^j} c_k \qquad k = 0, \ldots, n-j.$$

$$n = n - j$$

3. If $c_k > 0$ for all k or if $c_k < 0$ for all k, then there is no root in the interval $[a,b]$. STOP.
4. If $c_k \geq 0$ for $0 \leq k < i$ and $c_k \leq 0$ for $i \leq k \leq n$ (one sign change), then
   a. there is exactly one root r in the interval $[a,b]$
   b. if $b - a < \varepsilon$, set $r = (a+b)/2$
      otherwise, subdivide the interval at the midpoint and search for the root in each interval recursively.
5. Otherwise, subdivide the interval at the midpoint and find the roots in each interval recursively.

B. Prove that this algorithm finds all the roots of the given polynomial in the interval $[a,b]$.

C. Compare the speed and accuracy of this root finding algorithm to Newton's method.