9 Representation of Curves and Surfaces

The classic teapot, shown in Fig. 9.1, is perhaps the best-known icon of computer graphics. Since it was modeled in 1975 by Martin Newell [CROW87], it has been used by dozens of researchers as a structure for demonstrating the latest techniques for producing realistic surfaces and textures. Modeling the elegant teapot required specifying its shape as a collection of smooth surface elements, known as **bicubic patches**. Smooth curves and surfaces must be generated in many computer graphics applications. Many real-world objects are inherently smooth, and much of computer graphics involves modeling the real world. Computer-aided design (CAD), high-quality character typefaces, data plots, and artists' sketches all contain smooth curves and surfaces. The path of a camera or object in an animation sequence is almost always smooth; similarly, a path through intensity or color space (Chapters 12 and 11) often must be smooth.

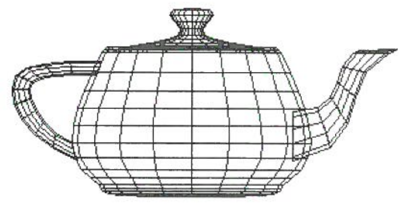


Figure 9.1 The famous *teapot*–a model consisting of an assemblage of smooth, curved surfaces.

The need to represent curves and surfaces arises in two cases: in modeling existing objects (a car, a face, or a mountain) and in modeling *from scratch*, where no preexisting physical object is being represented. In the first case, a mathematical description of the object may be unavailable. Of course, we can use as a model the coordinates of the infinitely many points of the object, but this approach is not feasible for a computer with finite storage. More often, we merely approximate the object with pieces of planes, spheres, or other shapes that are easy to describe mathematically, and require that points on our model be close to corresponding points on the object.

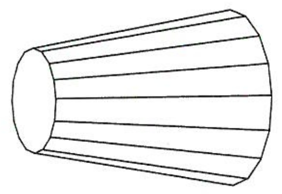


Figure 9.2
A 3D object represented by polygons.

In the second case, when there is no preexisting object to model, the user creates the object in the modeling process; hence, the object matches its representation exactly, because its only embodiment is the representation. To create the object, the user may sculpt the object interactively, describe it mathematically, or give an approximate description to be *filled in* by some program. In CAD, the computer representation is used later to generate physical realizations of the abstractly designed object.

This chapter introduces the general area of **surface modeling.** The area is broad, and only the three most common representations for 3D surfaces are presented here: polygon mesh surfaces, parametric surfaces, and quadric surfaces. We also discuss parametric curves, both because they are interesting in their own right and because parametric surfaces are a generalization of the curves.

Solid modeling, introduced in the next chapter, is the representation of volumes completely surrounded by surfaces, such as a cube, an airplane, or a building. The surface representations discussed in this chapter can be used in solid modeling to define each of the surfaces that bound the volume.

A **polygon mesh** is a set of connected, polygonally bounded planar surfaces. Open boxes, cabinets, and building exteriors can be easily and naturally represented by polygon meshes, as can volumes bounded by planar surfaces. Polygon meshes can be used, although less easily, to represent objects with curved surfaces, as in Fig. 9.2; however, the representation is only approximate. Figure 9.3 shows the cross-section of a curved shape and the polygon mesh representing that shape. We can make the obvious errors in the representation arbitrarily small by using more polygons to create a closer piecewise linear approximation, but this approach increases space requirements and the execution time of algorithms processing the representation. Furthermore, if the image is enlarged, the straight edges again become obvious.

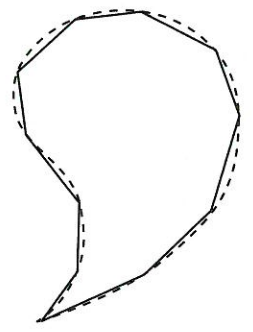


Figure 9.3
A cross-section of a curved shape (dashed line) and its polygonal representation (solid lines).

Parametric polynomial curves define points on a 3D curve by using three polynomials in a parameter t, one for each of x, y, and z. The coefficients of the polynomials are selected such that the curve follows the desired path. Although various degrees of polynomials can be used, we present only the most common case: cubic polynomials (that have powers of the parameter up through the third). The term **cubic curve** will often be used for such curves.

Parametric bivariate (two-variable) **polynomial surface patches** define the coordinates of points on a curved surface by using three bivariate polynomials, one for each of x, y, and z. The boundaries of the patches are parametric polynomial curves. Many fewer bivariate polynomial surface patches than polygonal patches are needed to approximate a curved surface to a given accuracy. The algorithms for working with bivariate polynomials, however, are more complex than are those for polygons. As with curves, polynomials of various degrees can be used, but we discuss here only the common case of polynomials that are cubic in both parameters. The surfaces are accordingly called **bicubic surfaces.**

Quadric surfaces are those defined implicitly by an equation $f(x, y, z) = 0$, where f is a quadric polynomial in x, y, and z. Quadric surfaces are a convenient representation for the familiar sphere, ellipsoid, and cylinder.

Chapter 10, on solid modeling, incorporates these representations into systems to represent not just surfaces, but also bounded (solid) volumes. The surface representations described in this chapter are used, sometimes in combination with one another, to bound a 3D volume.

9.1 POLYGON MESHES

A **polygon mesh** is a collection of edges, vertices, and polygons connected such that each edge is shared by at most two polygons. An edge connects two vertices, and a polygon is a closed sequence of edges. An edge can be shared by two adjacent polygons, a vertex is shared by at least two edges, and every edge is part of *some* polygon. A polygon mesh can be represented in several different ways, each with its advantages and disadvantages. The application programmer's task is to choose the most appropriate representation. Several representations can be used in a single application: one for external storage, another for internal use, and yet another with which the user interactively creates the mesh.

Two basic criteria, space and time, can be used to evaluate different representations. Typical operations on a polygon mesh are finding all the edges incident to a vertex, finding the polygons sharing an edge or a vertex, finding the vertices connected by an edge, finding the edges of a polygon, displaying the mesh, and identifying errors in representation (e.g., a missing edge, vertex, or polygon). In general, the more explicitly the relations among polygons, vertices, and edges are represented, the faster the operations are and the more space the representation requires. Woo [WOO85] has analyzed the time complexity of nine basic access operations and nine basic update operations on a polygon-mesh data structure.

In Sections 9.1.1 and 9.1.2, several issues concerning polygon meshes are discussed: representing polygon meshes, ensuring that a given representation is correct, and calculating the coefficients of the plane of a polygon.

9.1.1 Representing Polygon Meshes

In this section, we discuss three polygon-mesh representations: explicit, pointers to a vertex list, and pointers to an edge list. In the **explicit representation,** each polygon is represented by a list of vertex coordinates:

$$P = ((x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_n, y_n, z_n)).$$

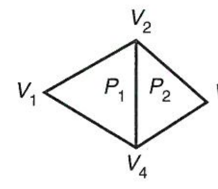
The vertices are stored in the order in which we would encounter them were we traveling around the polygon. There are edges between successive vertices in the list and between the last and first vertices. For a single polygon, this representation is space-efficient; for a polygon mesh, however, much space is lost, because the coordinates of shared vertices are duplicated. Even more of a problem, there is no explicit representation of shared edges and vertices. For instance, to drag a vertex and all its incident edges interactively, we must find all polygons that share the vertex. This search requires comparing the coordinate triples of one polygon with those of all other polygons. The most efficient way to do this would be to sort all N coordinate triples, but this process is at best an $N \log_2 N$ one, and even then there is the danger that the same vertex might, due to computational roundoff, have slightly different coordinate values in each polygon, so a correct match might never be made.

With this representation, displaying the mesh either as filled polygons or as polygon outlines necessitates transforming each vertex and clipping each edge of each polygon. If edges are being drawn, each shared edge is drawn twice, which causes problems on pen plotters, film recorders, and vector displays, due to the overwriting. A problem may also be created on raster displays if the edges are drawn in opposite directions, in which case extra pixels may be intensified.

Polygons defined with **pointers to a vertex list,** the method used by SPHIGS, have each vertex in the polygon mesh stored just once, in the vertex list $V = ((x_1, y_1, z_1), \ldots, (x_n, y_n, z_n))$. A polygon is defined by a list of indices (or pointers) into the vertex list. A polygon made up of vertices 3, 5, 7, and 10 in the vertex list would thus be represented as $P = (3, 5, 7, 10)$.

This representation, an example of which is shown in Fig. 9.4, has several advantages over the explicit polygon representation. Since each vertex is stored just once, considerable space is saved. Furthermore, the coordinates of a vertex can be changed easily. On the other hand, it is still difficult to find polygons that share an edge, and shared polygon edges are still drawn twice when all polygon outlines are displayed. We can eliminate these two problems by representing edges explicitly, as in the next method.

When defining polygons by **pointers to an edge list,** we again have the vertex list V, but represent a polygon as a list of pointers not to the vertex list, but rather to an edge list, in which each edge occurs just once. In turn, each edge in the edge list points to the two vertices in the vertex list defining the edge, and also to the one or

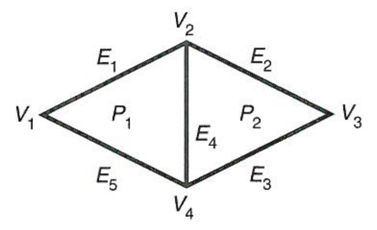


$V = (V_1, V_2, V_3, V_4) = ((x_1, y_1, z_1),\ldots,(x_4, y_4, z_4))$

$P_1 = (1, 2, 4)$

$P_2 = (4, 2, 3)$

Figure 9.4 Polygon mesh defined with indexes into a vertex list.



$V = (V_1, V_2, V_3, V_4) = ((x_1, y_1, z_1), \ldots, (x_4, y_4, z_4))$

$E_1 = (V_1, V_2, P_1, \lambda)$

$E_2 = (V_2, V_3, P_2, \lambda)$

$E_3 = (V_3, V_4, P_2, \lambda)$

$E_4 = (V_4, V_2, P_1, P_2)$

$E_5 = (V_4, V_1, P_1, \lambda)$

$P_1 = (E_1, E_4, E_5)$

$P_2 = (E_2, E_3, E_4)$

Figure 9.5 Polygon mesh defined with edge lists for each polygon (λ represents null).

two polygons to which the edge belongs. Hence, we describe a polygon as $P = (E_1, \ldots, E_n)$, and an edge as $E = (V_1, V_2, P_1, P_2)$. When an edge belongs to only one polygon, either P_1 or P_2 is null. Figure 9.5 shows an example of this representation.

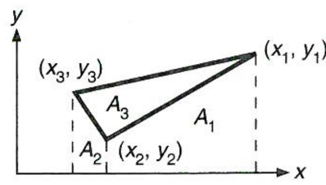
We show polygon outlines by displaying all edges, rather than by displaying all polygons; thus, redundant clipping, transformation, and scan conversion are avoided. Filled polygons are also displayed easily. In some situations, such as the description of a 3D honeycomblike sheet-metal structure, some edges are shared by three polygons. In such cases, the edge descriptions can be extended to include an arbitrary number of polygons: $E = (V_1, V_2, P_1, P_2, \ldots, P_n)$.

In none of these three representations (i.e., explicit polygons, pointers to vertices, pointers to an edge list) is it easy to determine which edges are incident to a vertex: All edges must be inspected. Of course, information can be added explicitly to permit determination of such relationships. For instance, the winged-edge representation used by Baumgart [BAUM75] expands the edge description to include pointers to the two adjoining edges of each polygon, whereas the vertex description includes a pointer to an (arbitrary) edge incident on the vertex, and thus more polygon and vertex information is available.

9.1.2 Plane Equations

When we are working with polygons or polygon meshes, we frequently need to know the equation of the plane in which the polygon lies. In some cases, of course, the equation is known implicitly through the interactive construction methods used to define the polygon. If the equation is not known, we can use the coordinates of three vertices to find the plane. Recall the plane equation

$$Ax + By + Cz + D = 0. \tag{9.1}$$



$$C = \underbrace{\underbrace{\tfrac{1}{2}(y_1 + y_2)(x_2 - x_1)}_{-A_1} + \underbrace{\tfrac{1}{2}(y_2 + y_3)(x_3 - x_2)}_{-A_2} + \underbrace{\tfrac{1}{2}(y_3 + y_1)(x_1 - x_3)}_{A_1 + A_2 + A_3}}_{A_3}$$

Figure 9.6 Calculating the area C of a triangle using Eq. (9.2).

The coefficients A, B, and C define the normal to the plane, $[A\ \ B\ \ C]$. Given points P_1, P_2, and P_3 on the plane, that plane's normal can be computed as the vector cross-product $P_1P_2 \times P_1P_3$ (or $P_2P_3 \times P_2P_1$, etc.). If the cross-product is zero, then the three points are collinear and do not define a plane. Other vertices, if any, can be used instead. Given a nonzero cross-product, we can find D by substituting the normal to $[A\ \ B\ \ C]$ and any one of the three points into Eq. (9.1).

If there are more than three vertices, they may be nonplanar, either for numerical reasons or because of the method by which the polygons were generated. Then another technique for finding the coefficients A, B, and C of a plane that comes close to all the vertices is better than the cross-product method. It can be shown that A, B, and C are proportional to the signed areas of the projections of the polygon onto the (y, z), (z, x), and (x, y) planes, respectively. For example, if the polygon is parallel to the (x, y) plane, then $A = B = 0$, as expected: The projections of the polygon onto the (y, z) and (z, x) planes have zero area. This method is better than the cross-product method, because the areas of the projections are a function of the coordinates of all the vertices and so are not sensitive to the choice of a few vertices that might happen not to be coplanar with most or all of the other vertices, or that might happen to be collinear. For instance, the area (and hence coefficient) C of the polygon projected onto the (x, y) plane in Fig. 9.6 is just the area of the trapezoid A_3, minus the areas of A_1 and A_2. In general,

$$C = \frac{1}{2}\sum_{i=1}^{n}(y_i + y_{i\oplus 1})(x_{i\oplus 1} - x_i), \tag{9.2}$$

where the operator $\oplus$ is normal addition except that $n \oplus 1 = 1$. The areas for A and B are given by similar formulae, except the area for B is negated (see Example 9.1).

Eq. (9.2) gives the sum of the areas of all the trapezoids formed by successive edges of the polygons. If $x_{i\oplus 1} < x_i$, the area makes a negative contribution to the sum. The sign of the sum is also useful: If the vertices have been enumerated in a clockwise direction (as projected onto the plane), then the sign is positive; otherwise, it is negative.

Once we determine the plane equation by using all the vertices, we can estimate how nonplanar the polygon is by calculating the perpendicular distance from the plane to each vertex. This distance d for the vertex at (x, y, z) is

$$d = \frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}}. \tag{9.3}$$

This distance is either positive or negative, depending on which side of the plane the point is located. If the vertex is on the plane, then $d = 0$. Of course, to determine only on which side of a plane a point is, only the sign of d matters, so division by the square root is not needed.

The plane equation is not unique; any nonzero multiplicative constant k changes the equation, but not the plane. It is often convenient to store the plane coefficients with a normalized normal; we can do so by choosing

$$k = \frac{1}{\sqrt{A^2 + B^2 + C^2}}, \tag{9.4}$$

which is the reciprocal of the length of the normal. Then, distances can be computed with Eq. (9.3) more easily, since the denominator is 1.

Example 9.1

Problem: Write a function that calculates the plane equation coefficients, given n vertices of a polygon that is approximately planar. Assume that the polygon vertices are enumerated counterclockwise, as viewed toward the plane from the positive side of the plane. The vertices and the number of vertices are arguments passed to the function.

Answer: Using Eq. (9.2), and similar equations for A and B, the program is simply:

This function calculates plane equation coefficients.

```
FindPlaneCoefficients(float x[], float y[], float z[], int num_verts,
                      float *a, float *b, float *c, float *d)
{
   float  A, B, C, D;
   int    i, j;

   A = B = C = 0.0;
   for (i = 0; i < num_verts; i++) {
      j = (i + 1) % num_verts;
      A += (z[i] + z[j]) * (y[j] – y[i]);
      B += – (x[i] + x[j]) * (z[j] – z[i]);
      C += (y[i] + y[j]) * (x[j] – x[i]);
   }
   A /= 2.0; B /= 2.0; C /= 2.0;
   D = –(A * x[0] + B * y[0] + C * z[0]);

   *a = A;
   *b = B;
   *c = C;
   *d = D;
}
```

9.2 PARAMETRIC CUBIC CURVES

Polylines and polygons are first-degree, piecewise approximations to curves and surfaces, respectively. Unless the curves or surfaces being approximated are also piecewise linear, large numbers of endpoint coordinates must be created and stored if we are to achieve reasonable accuracy. Interactive manipulation of the data to approximate a shape is tedious, because many points have to be positioned precisely.

In this section, a more compact and more manipulable representation of piecewise smooth curves is developed; in Section 9.3 the mathematical development is generalized to surfaces. The general approach is to use functions that are of a degree higher than that of the linear functions. The functions still generally only approximate the desired shapes, but use less storage and offer easier interactive manipulation than do linear functions.

The higher-degree approximations can be based on one of three methods. First, we can express y and z as *explicit* functions of x, so that $y = f(x)$ and $z = g(x)$. The difficulties with this approach are that (1) it is impossible to get multiple values of y for a single x, so curves such as circles and ellipses must be represented by multiple curve segments; (2) such a definition is not rotationally invariant (to describe a rotated version of the curve requires a great deal of work and may in general require breaking a curve segment into many others); and (3) describing curves with vertical tangents is difficult, because a slope of infinity is difficult to represent.

Second, we can choose to model curves as solutions to *implicit* equations of the form $f(x, y, z) = 0$; this method is fraught with its own perils. First, the given equation may have more solutions than we want. For example, in modeling a circle, we might want to use $x^2 + y^2 = 1$, which is fine. But how do we model one-half of a circle? We must add constraints such as $x \geq 0$, which cannot be contained within the implicit equation. Furthermore, if two implicitly defined curve segments are joined together, it may be difficult to determine whether their tangent directions agree at their join point. Tangent continuity is critical in many applications.

These two mathematical forms do permit rapid determination of whether a point lies on the curve or on which side of the curve the point lies, as was done in Chapter 3. Normals to the curve are also computed easily. Hence, we shall discuss briefly the implicit form in Section 9.4.

The **parametric representation** for curves, $x = x(t)$, $y = y(t)$, $z = z(t)$, overcomes the problems caused by functional or implicit forms and offers a variety of other attractions that will become clear in the remainder of this chapter. Parametric curves replace the use of geometric slopes (which may be infinite) with parametric tangent vectors (which, we shall see, are never infinite). Here a curve is approximated by a **piecewise polynomial curve** instead of by the piecewise linear curve used in Section 9.1. Each segment Q of the overall curve is given by three functions, x, y, and z, which are cubic polynomials in the parameter t.

Cubic polynomials are most often used because lower-degree polynomials give too little flexibility in controlling the shape of the curve, and higher-degree polynomials can introduce unwanted wiggles and also require more computation. No lower-degree representation allows a curve segment to interpolate (pass through) two specified endpoints with specified derivatives at each endpoint. Given a cubic polynomial with its four coefficients, four knowns are used to solve for the unknown coefficients. The four knowns might be the two endpoints and the derivatives at the endpoints. Similarly, the two coefficients of a first-order (straight-line) polynomial are determined by the two endpoints. For a straight line, the derivatives at each end are determined by the line itself and cannot be controlled independently. With quadratic (second-degree) polynomials—and hence three coefficients—two endpoints and one other condition, such as a slope or additional point, can be specified.

Also, parametric cubics are the lowest-degree curves that are nonplanar in 3D. You can see this fact by recognizing that a second-order polynomial's three coefficients can be specified completely by three points and that three points define a plane in which the polynomial lies.

Higher-degree curves require more conditions to determine the coefficients and can *wiggle* back and forth in ways that are difficult to control. Despite these complexities, higher-degree curves are used in applications—such as the design of cars and planes—in which higher-degree derivatives must be controlled to create surfaces that are aerodynamically efficient. In fact, the mathematical development for parametric curves and surfaces is often given in terms of an arbitrary degree n. In this chapter, we fix n at 3.

9.2.1 Basic Characteristics

The cubic polynomials that define a curve segment $Q(t) = [x(t)\ \ y(t)\ \ z(t)]^T$ are of the form

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x, \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y, \\ z(t) &= a_z t^3 + b_z t^2 + c_z t + d_z, \qquad 0 \leq t \leq 1. \end{aligned} \tag{9.5}$$

To deal with finite segments of the curve, we restrict the parameter t without loss of generality, to the [0, 1] interval.

With $T = [t^3\ \ t^2\ \ t\ \ 1]^T$, and defining the matrix of coefficients of the three polynomials as

$$C = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix}, \tag{9.6}$$

we can rewrite Eq. (9.5) as

$$Q(t) = [x(t)\ \ y(t)\ \ z(t)]^T = C \cdot T. \tag{9.7}$$

This representation provides a compact way to express Eq. (9.5).

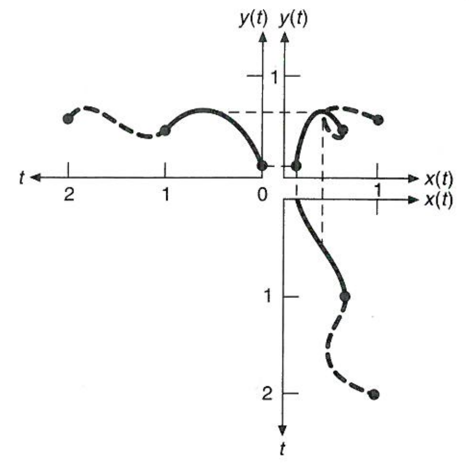


Figure 9.7 Two joined 2D parametric curve segments and their defining polynomials. The dashed lines between the (x, y) plot and the x(t) and y(t) plots show the correspondence between the points on the (x, y) curve and the defining cubic polynomials. The x(t) and y(t) plots for the second segment have been translated to begin at t = 1, rather than at t = 0, to show the continuity of the curves at their join point.

Figure 9.7 shows two joined parametric cubic curve segments and their polynomials; it also illustrates the ability of parametrics to represent easily multiple values of y for a single value of x with polynomials that are themselves single valued. (This figure of a curve, like all others in this section, shows 2D curves represented by $[x(t)\ \ y(t)]^{\mathrm{T}}$.)

Continuity between curve segments. The derivative of $Q(t)$ is the parametric **tangent vector** of the curve. Applying this definition to Eq. (9.7), we have

$$\frac{d}{dt}Q(t) = Q'(t) = \left[\frac{d}{dt}x(t) \quad \frac{d}{dt}y(t) \quad \frac{d}{dt}z(t)\right]^{\mathrm{T}} = \frac{d}{dt}C \cdot T = C \cdot [3t^2 \quad 2t \quad 1 \quad 0]^{\mathrm{T}}$$
$$= [3a_xt^2 + 2b_xt + c_x \quad 3a_yt^2 + 2b_yt + c_y \quad 3a_zt^2 + 2b_zt + c_z]^{\mathrm{T}}. \tag{9.8}$$

If two curve segments join together, the curve has G^0 **geometric continuity**. If the directions (but not necessarily the magnitudes) of the two segments' tangent vectors are equal at a join point, the curve has G^1 geometric continuity. In computer-aided design of objects, G^1 continuity between curve segments often is required. G^1 continuity means that the geometric slopes of the segments are equal at the join point. For two tangent vectors TV_1 and TV_2 to have the same direction, it is necessary that one be a scalar multiple of the other: $TV_1 = k \cdot TV_2$, with $k > 0$ [BARS88].

If the tangent vectors of two cubic curve segments are equal (that is, their directions *and* magnitudes are equal) at the segments' join point, the curve has first-degree continuity in the parameter t, or **parametric continuity**, and is said to be C^1 continuous. If the direction and magnitude of $d^n/dt^n[Q(t)]$ through the nth derivative are equal at the join point, the curve is called C^n **continuous**. Figure 9.8 shows curves with three different degrees of continuity. Note that a parametric curve segment is itself everywhere continuous; the continuity of concern here is at the join points.

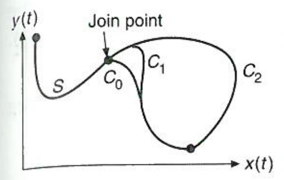


Figure 9.8
Curve segment S joined to segments C_0, C_1, and C_2 with the 0, 1, and 2 degrees of parametric continuity, respectively. The visual distinction between C_1 and C_2 is slight at the join, but obvious away from the join.

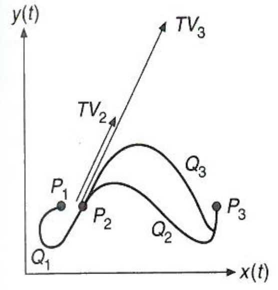


Figure 9.9
Curve segments Q_1, Q_2, and Q_3 join at the point P_2 and are identical except for their tangent vectors at P_2. Q_1 and Q_2 have equal tangent vectors, and hence are both G^1 and C^1 continuous at P_2. Q_1 and Q_3 have tangent vectors in the same direction, but Q_3 has twice the magnitude, so they are only G^1 continuous at P_2. The larger tangent vector of Q_3 means that the curve is pulled more in the tangent-vector direction before heading toward P_3. Vector TV_2 is the tangent vector for Q_2, TV_3 is that for Q_3.

The tangent vector $Q'(t)$ is the *velocity* of a point on the curve with respect to the parameter t. Similarly, the second derivative of $Q(t)$ is the *acceleration*. Suppose a camera were moving along a parametric cubic curve in equal time steps and recording a picture after each step; the tangent vector gives the velocity of the camera along the curve. So that jerky movements in the resulting animation sequence are avoided, the camera velocity and acceleration at join points should be continuous. It is this continuity of acceleration across the join point in Fig. 9.8 that makes the C^2 curve continue farther to the right than the C^1 curve, before bending around to the endpoint.

In general, C^1 continuity implies G^1, but the converse is generally not true. That is, G^1 continuity is generally less restrictive than is C^1, so curves can be G^1 but not necessarily C^1 continuous. However, join points with G^1 continuity will appear just as smooth as those with C^1 continuity, as seen in Fig. 9.9.

The plot of a parametric curve is distinctly different from the plot of an ordinary function, in which the independent variable is plotted on the x axis and the dependent variable is plotted on the y axis. In parametric curve plots, the independent variable t is never plotted at all. Thus we cannot determine, just by looking at a parametric curve plot, the tangent vector to the curve. It is possible to determine the direction of the vector, but not the magnitude. You can see why if you think about it as follows: If $\gamma(t)$, $0 \le t \le 1$ is a parametric curve, its tangent vector at time 0 is $\gamma'(0)$. If we let $\eta(t) = \gamma(2t)$, $0 \le t \le \frac{1}{2}$, then the parametric plots of γ and η are identical. On the other hand, $\eta'(0) = 2\,\gamma'(0)$. Thus, two curves that have identical plots can have different tangent vectors. This fact is the basis for the definition of **geometric continuity:** For two curves to join smoothly, we require only that their tangent-vector directions match; we do not require that their magnitudes match.

Relation to constraints. A curve segment $Q(t)$ is defined by constraints on endpoints, tangent vectors, and continuity between curve segments. Each cubic polynomial of Eq. (9.5) has four coefficients, so four constraints will be needed, allowing us to formulate four equations in the four unknowns, then solving for the unknowns. The three major types of curves discussed in this section are **Hermite,** defined by two endpoints and two endpoint tangent vectors; **Bézier,** defined by two endpoints and two other points that control the endpoint tangent vectors; and several kinds of **splines,** each defined by four control points. The splines have C^1 and C^2 continuity at the join points and come close to their control points, but generally do not interpolate the points. The types of splines are uniform B-splines and nonuniform B-splines.

To see how the coefficients of Eq. (9.5) can depend on four constraints, we recall that a parametric cubic curve is defined by $Q(t) = C \cdot T$. We rewrite the coefficient matrix as $C = G \cdot M$, where M is a 4×4 **basis matrix,** and G is a four-element matrix of geometric constraints, called the **geometry matrix.** The geometric constraints are just the conditions, such as endpoints or tangent vectors, that define the curve. We use G_x to refer to the row vector of just the x components of the geometry matrix. G_y and G_z have similar definitions. G or M, or both G and M, differ for each type of curve.

The elements of G and M are constants, so the product $G \cdot M \cdot T$ is just three cubic polynomials in t. Expanding the product $Q(t) = G \cdot M \cdot T$ gives

$$Q(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = [G_1 \quad G_2 \quad G_3 \quad G_4] \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}. \quad (9.9)$$

We can read this equation in a second way: the point $Q(t)$ is a weighted sum of the *columns* of the geometry matrix G, each of which represents a point or a vector in 3-space.

Multiplying out just $x(t) = G_x \cdot M \cdot T$ gives

$$x(t) = (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41}) g_{1x} + (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42}) g_{2x}$$
$$+ (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43}) g_{3x} + (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44}) g_{4x}. \quad (9.10)$$

Equation (9.10) emphasizes that the curve is a weighted sum of the elements of the geometry matrix. The weights are each cubic polynomials of t, and are called **blending functions.** The blending functions B are given by $B = M \cdot T$. Notice the similarity to a piecewise linear approximation, for which only two geometric constraints (the endpoints of the line) are needed, so each curve segment is a straight line defined by the endpoints G_1 and G_2:

$$x(t) = g_{1x}(1 - t) + g_{2x}(t),$$
$$y(t) = g_{1y}(1 - t) + g_{2y}(t),$$
$$z(t) = g_{1z}(1 - t) + g_{2z}(t). \quad (9.11)$$

Parametric cubics are really just a generalization of straight-line approximations. The cubic curve $Q(t)$ is a combination of the *four* columns of the geometry matrix, just as a straight-line segment is a combination of *two* column vectors.

To see how to calculate the basis matrix M, we turn now to specific forms of parametric cubic curves.

9.2.2 Hermite Curves

The Hermite form (named for the mathematician) of the cubic polynomial curve segment is determined by constraints on the endpoints P_1 and P_4 and tangent vectors at the endpoints R_1 and R_4. (The indices 1 and 4 are used, rather than 1 and 2, for consistency with later sections, where intermediate points P_2 and P_3 will be used, instead of tangent vectors, to define the curve.)

To find the **Hermite basis matrix** M_H, which relates the **Hermite geometry vector** G_H to the polynomial coefficients, we write four equations, one for each of

the constraints, in the four unknown polynomial coefficients, and then solve for the unknowns.

Defining G_{H_x}, the x component of the Hermite geometry matrix, as

$$G_{H_x} = [P_{1_x} \quad P_{4_x} \quad R_{1_x} \quad R_{4_x}], \quad (9.12)$$

and rewriting $x(t)$ from Eqs. (9.5) and (9.9) as

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x = C_x \cdot T = G_{H_x} \cdot M_H \cdot T = G_{H_x} \cdot M_H \, [t^3 \; t^2 \; t \; 1]^T, \quad (9.13)$$

the constraints on $x(0)$ and $x(1)$ are found by direct substitution into Eq. (9.13) as

$$x(0) = P_{1_x} = G_{H_x} \cdot M_H \, [0 \; 0 \; 0 \; 1]^T, \quad (9.14)$$
$$x(1) = P_{4_x} = G_{H_x} \cdot M_H \, [1 \; 1 \; 1 \; 1]^T. \quad (9.15)$$

Just as in the general case we differentiated Eq. (9.7) to find Eq. (9.8), we now differentiate Eq. (9.13) to get $x'(t) = G_{H_x} \cdot M_H \, [3t^2 \; 2t \; 1 \; 0]^T$. Hence, the tangent-vector–constraint equations can be written as

$$x'(0) = R_{1_x} = G_{H_x} \cdot M_H \, [0 \; 0 \; 1 \; 0]^T, \quad (9.16)$$
$$x'(1) = R_{4_x} = G_{H_x} \cdot M_H \, [3 \; 2 \; 1 \; 0]^T. \quad (9.17)$$

The four constraints of Eqs. (9.14)–(9.17) can be rewritten in matrix form as

$$[P_{1_x} \quad P_{4_x} \quad R_{1_x} \quad R_{4_x}] = G_{H_x} = G_{H_x} \cdot M_H \cdot \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}. \quad (9.18)$$

For this equation (and the corresponding expressions for y and z) to be satisfied, M_H must be the inverse of the 4×4 matrix in Eq. (9.18):

$$M_H = \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -3 & 0 & 1 \\ -2 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}. \quad (9.19)$$

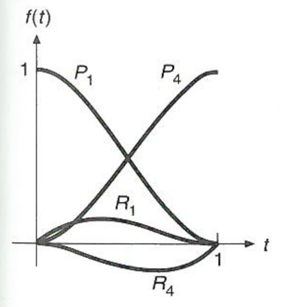


Figure 9.10
The Hermite blending functions, labeled by the elements of the geometry vector that they weight.

M_H can now be used in $x(t) = G_{H_x} \cdot M_H \cdot T$ to find $x(t)$ based on the geometry vector G_{H_x}. Similarly, $y(t) = G_{H_y} \cdot M_H \cdot T$ and $z(t) = G_{H_z} \cdot M_H \cdot T$, so we can write

$$Q(t) = [x(t) \;\; y(t) \;\; z(t)]^T = G_H \cdot M_H \cdot T, \quad (9.20)$$

where G_H is the matrix

$$[P_1 \quad P_4 \quad R_1 \quad R_4].$$

Expanding the product $M_H \cdot T$ in $Q(t) = G_H \cdot M_H \cdot T$ gives the **Hermite blending functions** B_H as the polynomials weighting each element of the geometry matrix:

$$Q(t) = G_H \cdot M_H \cdot T = G_H \cdot B_H$$
$$= (2t^3 - 3t^2 + 1)P_1 + (-2t^3 + 3t^2)P_4 + (t^3 - 2t^2 + t)R_1 + (t^3 - t^2)R_4. \quad (9.21)$$

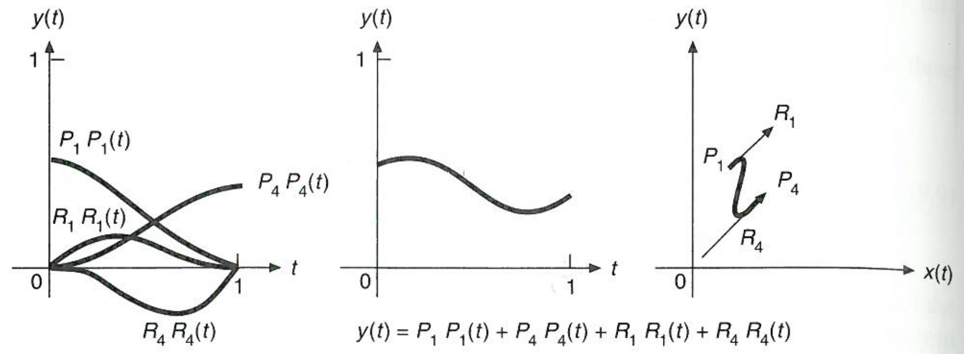


Figure 9.11 A Hermite curve showing the four elements of the geometry vector weighted by the blending functions (leftmost four curves), their sum $y(t)$, and the 2D curve itself (far right). $x(t)$ is defined by a similar weighted sum.

Figure 9.10 shows the four blending functions. Notice that, at $t = 0$, only the function labeled P_1 is nonzero: only P_1 affects the curve at $t = 0$. As soon as t becomes greater than zero, R_1, P_4, and R_4 begin to have an influence. Figure 9.11 shows the four functions weighted by the y components of a specific geometry vector, their sum $y(t)$, and the curve $Q(t)$.

Figure 9.12 shows a series of Hermite curves. The only difference among them is the length of the tangent vector R_1: The directions of the tangent vectors are fixed. The longer the vectors, the greater their effect on the curve. Figure 9.13 is another series of Hermite curves, with constant tangent-vector lengths but with different directions. In an interactive graphics system, the endpoints and tangent vectors of a curve are manipulated interactively by the user to shape the curve. Figure 9.14 shows one way of implementing this type of interaction.

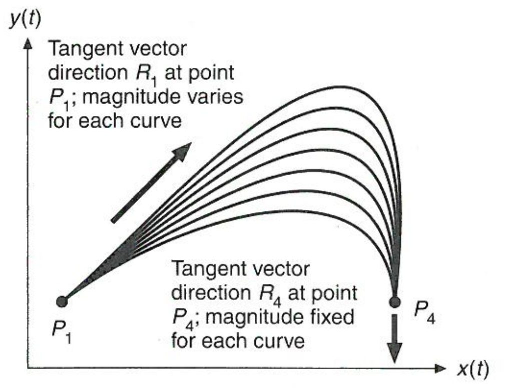

y(t)
Tangent vector direction R_1 at point P_1; magnitude varies for each curve
Tangent vector direction R_4 at point P_4; magnitude fixed for each curve
P_1
P_4
x(t)

Figure 9.12 Family of Hermite parametric cubic curves. Only R_1, the tangent vector at P_1, varies for each curve, increasing in magnitude for the higher curves.

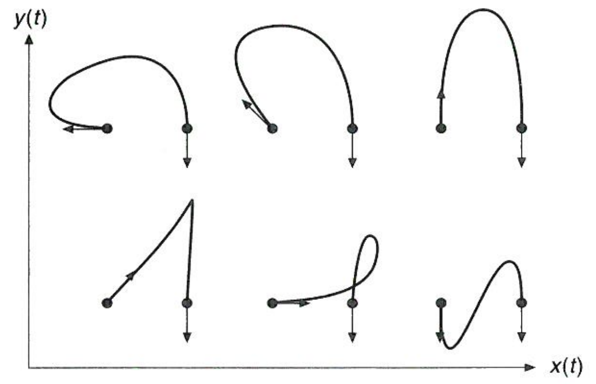


Figure 9.13 Family of Hermite parametric cubic curves. Only the direction of the tangent vector at the left starting point varies; all tangent vectors have the same magnitude. A smaller magnitude would eliminate the loop in the one curve.

Drawing parametric curves. Hermite and other similar parametric cubic curves are simple to display: We evaluate Eq. (9.5) at n successive values of t separated by a step size δ. Program 9.1 gives the code. The evaluation within the { ... } in the **for** loop takes 12 multiplies and 10 additions per 3D point. Use of Horner's rule for factoring polynomials,

$$f(t) = at^3 + bt^2 + ct + d = ((at + b)t + c)t + d, \tag{9.22}$$

would reduce the effort slightly to 10 multiplies and 10 additions per 3D point.

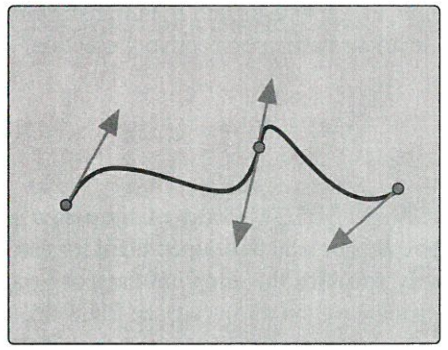


Figure 9.14 Two Hermite cubic curve segments displayed with controls to facilitate interactive manipulation. The user can reposition the endpoints by dragging the dots, and can change the tangent vectors by dragging the arrowheads. The tangent vectors at the join point are constrained to be collinear (to provide C^1 continuity): The user is usually given a command to enforce C^0, C^1, G^1, or no continuity. The tangent vectors at the $t = 1$ end of each curve are drawn in the reverse of the direction used in the mathematical formulation of the Hermite curve, for clarity and for more convenient user interaction.

More efficient ways to display these curves involve forward-difference techniques, as discussed in [FOLE90].

Program 9.1

Program to display a cubic parametric curve.

```
typedef float CoefficientArray[4];
void DrawCurve(CoefficientArray cx, CoefficientArray cy,
                CoefficientArray cz, int n)
  /* cx, cy, and cz are coefficients for x(t), y(t), and z(t) */
  /* e.g., Cx = Gx·M, etc. */
  /* n, number of steps */
{
  float x, y, z, delta, t, t2, t3;
  int   i;

  MoveAbs3( cx[3], cy[3], cz[3] );
  delta = 1.0 / n;
  for (i = 1; i <= n; i++) {
    t = i * delta;
    t2 = t * t;
    t3 = t2 * t;
    x = cx[0] * t3 + cx[1] * t2 + cx[2] * t + cx[3];
    y = cy[0] * t3 + cy[1] * t2 + cy[2] * t + cy[3];
    z = cz[0] * t3 + cz[1] * t2 + cz[2] * t + cz[3];
    DrawAbs3( x, y, z );
  }
}
```

Because the cubic curves are linear combinations (weighted sums) of the four elements of the geometry vector, as seen in Eq. (9.10), we can transform the curves by transforming the geometry vector and then using it to generate the transformed curve, which is equivalent to saying that the curves are invariant under rotation, scaling, and translation. This strategy is more efficient than is generating the curve as a series of short line segments and then transforming each individual line. The curves are *not* invariant under perspective projection, as will be discussed in Section 9.2.6.

9.2.3 Bézier Curves

The Bézier [BEZI70; BEZI74] form of the cubic polynomial curve segment, named after Pierre Bézier who developed them for use in designing automobiles at Rénault, indirectly specifies the endpoint tangent vector by specifying two intermediate points that are not on the curve; see Fig. 9.15. The starting and ending tangent vectors are determined by the vectors P_1P_2 and P_3P_4 and are related to R_1 and R_4 by

$$R_1 = Q'(0) = 3(P_2 - P_1),\quad R_4 = Q'(1) = 3(P_4 - P_3). \tag{9.23}$$

The Bézier curve interpolates the two end control points and approximates the other two. See Exercise 9.9 to understand why the constant 3 is used in Eq. (9.23). The **Bézier geometry matrix** G_B, consisting of four points, is

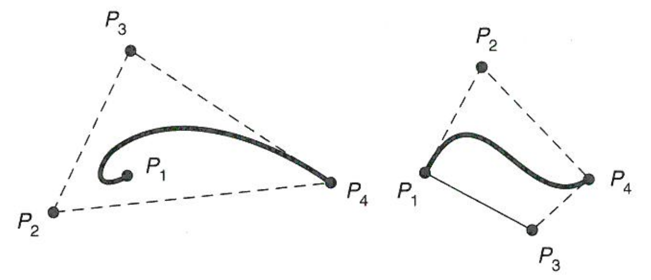


Figure 9.15 Two Bézier curves and their control points. Notice that the convex hulls (the convex polygon formed by the control points), shown as dashed lines, do not need to touch all four control points.

$$G_B = [P_1 \quad P_2 \quad P_3 \quad P_4]\,. \tag{9.24}$$

Then, the matrix M_{HB} that defines the relation $G_H = G_B \cdot M_{HB}$ between the Hermite geometry matrix G_H and the Bézier geometry matrix G_B is just the 4 × 4 matrix in the following equation, which rewrites Eq. (9.24) in matrix form:

$$G_H = [P_1 \quad P_4 \quad R_1 \quad R_4] = [P_1 \quad P_2 \quad P_3 \quad P_4]\begin{bmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{bmatrix} = G_B \cdot M_{HB}\,. \tag{9.25}$$

To find the **Bézier basis matrix** M_B, we use Eq. (9.20) for the Hermite form, substitute $G_H = G_B \cdot M_{HB}$, and define $M_B = M_{HB} \cdot M_H$:

$$Q(t) = G_H \cdot M_H \cdot T = (G_B \cdot M_{HB}) \cdot M_H \cdot T = G_B \cdot (M_{HB} \cdot M_H) \cdot T = G_B \cdot M_B \cdot T. \tag{9.26}$$

Carrying out the multiplication $M_B = M_{HB} \cdot M_H$ gives

$$M_B = M_{HB} \cdot M_H = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \tag{9.27}$$

and the product $Q(t) = G_B \cdot M_B \cdot T$ is

$$Q(t) = (1-t)^3 P_1 + 3t(1-t)^2 P_2 + 3t^2(1-t) P_3 + t^3 P_4. \tag{9.28}$$

The four polynomials $B_B = M_B \cdot T$, which are the weights in Eq. (9.28), are called the **Bernstein polynomials,** and are shown in Fig. 9.16.

Joining of curve segments. Figure 9.17 shows two Bézier curve segments with a common endpoint. G^1 continuity is provided at the endpoint when $P_3 - P_4 = k(P_4 - P_5)$, $k > 0$. That is, the three points P_3, P_4, and P_5 must be distinct and collinear. In the more restrictive case when $k = 1$, there is C^1 continuity in addition to G^1 continuity.

If we refer to the polynomials of two curve segments as x^{l} (for the left segment) and x^{r} (for the right segment), we can find the conditions for C^0 and C^1 continuity at their join point:

$$x^{\mathrm{l}}(1) = x^{\mathrm{r}}(0), \quad \frac{d}{dt}x^{\mathrm{l}}(1) = \frac{d}{dt}x^{\mathrm{r}}(0) \cdot \tag{9.29}$$

Working with the x component of Eq. (9.29), we have

$$x^{\mathrm{l}}(1) = x^{\mathrm{r}}(0) = P_{4_x}, \frac{d}{dt}x^{\mathrm{l}}(1) = 3(P_{4_x} - P_{3_x}), \frac{d}{dt}x^{\mathrm{r}}(0) = 3(P_{5_x} - P_{4_x}). \tag{9.30}$$

As always, the same conditions are true of y and z. Thus, we have C^0 and C^1 continuity when $P_4 - P_3 = P_5 - P_4$, as expected.

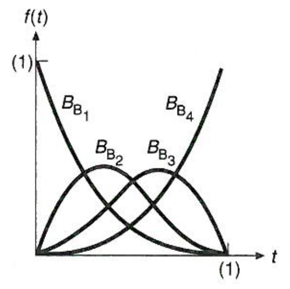


Figure 9.16 The Bernstein polynomials, which are the weighting functions for Bézier curves. At $t = 0$, only B_{B_1} is nonzero, so the curve interpolates P_1; similarly, at $t = 1$, only B_{B_4} is nonzero, and the curve interpolates P_4.

Importance of the convex hull. Examining the four B_B polynomials in Eq. (9.28) and Fig. 9.16, we note that their sum is everywhere unity and that each polynomial is everywhere nonnegative for $0 \le t < 1$. Thus, $Q(t)$ is just a weighted average of the four control points. This condition means that each curve segment, which is just the sum of four control points weighted by the polynomials, is completely contained in the **convex hull** of the four control points. The convex hull for 2D curves is the convex polygon formed by the four control points: Think of it as the polygon that you would form by putting a rubberband around the points (Fig. 9.15). For 3D curves, the convex hull is the convex polyhedron formed by the control points: Think of it as the polyhedron you would form by stretching a rubber sheet around the four points.

This convex-hull property holds for all cubics defined by weighted sums of control points if the blending functions are nonnegative and sum to one. In general, the weighted average of n points falls within the convex hull of the n points; this can be seen intuitively for $n = 2$ and $n = 3$, and the generalization follows. Another consequence of the fact that the four polynomials sum to unity is that we can find the value of the fourth polynomial for any value of t by subtracting the first three from unity—a fact that can be used to reduce computation time.

The convex-hull property is also useful for clipping curve segments: Rather than clip each short line piece of a curve segment to determine its visibility, we first apply a polygonal clip algorithm, for example, the Sutherland–Hodgman algorithm discussed in Chapter 3—to clip the convex hull or its extent against the clip region. If the convex hull (extent) is completely within the clip region, so is the entire curve segment. If the convex hull (extent) is completely outside the clip region, so is the curve segment. Only if the convex hull (extent) intersects the clip region does the curve segment itself need to be examined.

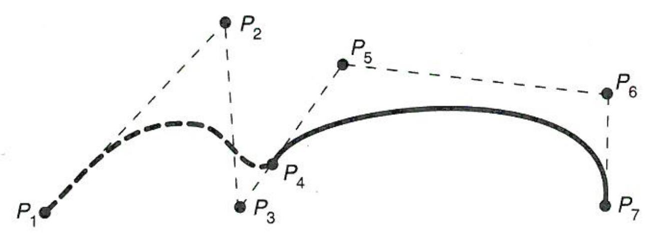


Figure 9.17 Two Bézier curves joined at P_4. Points P_3, P_4, and P_5 are collinear.

Example 9.2

Problem: Write a program, using SRGP, that allows a user to specify the four control points for a 2D Bézier curve and then draws the curve using the approach of Prog. 9.1. You should provide a way of specifying an arbitrary number of Bézier curves, clearing the SRGP window, and terminating the program.

Answer: We implement the DrawCurve function by using Eq. (9.28), which relates the curve $Q(t)$ to the four control points. In general, this implementation sacrifices efficiency for clarity. We do, however, use the SRGP_polyLine function, which is the most efficient way to draw the curve. The rest of the implementation follows the model of Prog. 9.1.

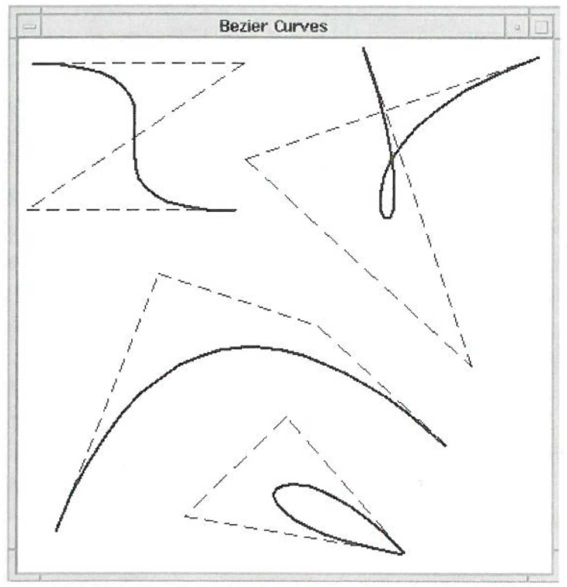
We have arbitrarily specified the window size and number of steps used to approximate the curve as 400 and 20, respectively. There are many possible ways to implement the interactive part of the program; we have elected to use a combination of locator and keyboard devices. The right locator button is used to specify the beginning of a new sequence of control points, whereas the left button is used to define the remaining three points. A rubber line echo helps to guide the layout of the points. The Bézier curve is drawn as soon as the last point is entered.

Finally, the window is cleared when the user presses the "c" key; pressing the "q" key terminates the program. A typical set of curves produced by the program is shown in the accompanying figure.

Interactive Bézier curve program.

```
#include "srgp.h"
#include <stdio.h>

#define KEYMEASURE_SIZE   80
#define WINDOW_SIZE       400
#define NUM_STEPS         20

void  DrawCurve(point *ControlPoints, int n)
{
   int    i;
   float  t, delta;
   point  CurvePoints[n];

   CurvePoints[0].x = ControlPoints[0].x;     /* Bézier curves interpolate the first */
   CurvePoints[0].y = ControlPoints[0].y;     /* and last control points */
   delta = 1.0 / n;                     /* The curve is to be approximated by n points */
                                        /* t ranges from 0.0 to 1.0 */
   for (i = 1; i <= n; i++) {
      t = i * delta;
      CurvePoints[i].x = ControlPoints[0].x * (1.0 – t) * (1.0 – t) * (1.0 – t)
```

```
            + ControlPoints[1].x * 3.0 * t * (1.0 – t) * (1.0 – t)
            + ControlPoints[2].x * 3.0 * t * t * (1.0 – t)
            + ControlPoints[3].x * t * t * t;

        CurvePoints[i].y = ControlPoints[0].y * (1.0 – t) * (1.0 – t) * (1.0 – t)
            + ControlPoints[1].y * 3.0 * t * (1.0 – t) * (1.0 – t)
            + ControlPoints[2].y * 3.0 * t * t * (1.0 – t)
            + ControlPoints[3].y * t * t * t;
    }
    SRGP_polyLine(n + 1, CurvePoints);             /* Draw the complete curve */
}
```



Typical output from the Bézier curve program.

```
main()
{
    locator_measure locMeasure, pastlocMeasure;
    char   keyMeasure[KEYMEASURE_SIZE];
    int    device;
    int    numCtl;
    boolean terminate;
    rectangle screen;
    point  ControlPoints[4];

    SRGP_begin("Bezier Curves", WINDOW_SIZE, WINDOW_SIZE, 1, FALSE);
    SRGP_setLocatorEchoType(CURSOR);
    SRGP_setLocatorButtonMask(LEFT_BUTTON_MASK|RIGHT_BUTTON_MASK);
    pastlocMeasure.position = SRGP_defPoint(–1, –1);     /* Initialize position to */
    SRGP_setLocatorMeasure(pastlocMeasure.position);   /* arbitrary location */
    SRGP_setKeyboardProcessingMode(RAW);
    SRGP_setInputMode(LOCATOR, EVENT);                 /* Both locator (mouse) */
```

```
    SRGP_setInputMode(KEYBOARD, EVENT);            /* and keyboard are active */
    screen = SRGP_defRectangle(0, 0, WINDOW_SIZE – 1, WINDOW_SIZE – 1);

/* Main event loop */
    terminate = FALSE;
    do {
        device = SRGP_waitEvent(INDEFINITE);
        switch (device) {
        case KEYBOARD:{
            SRGP_getKeyboard(keyMeasure, KEYMEASURE_SIZE);
            switch (keyMeasure[0]) {
            case 'q':                               /* Quitting the program */
                terminate = TRUE;
                break;
            case 'c':                               /* Clearing the window */
                SRGP_setColor(0);
                SRGP_fillRectangle(screen);
                SRGP_setColor(1);
                break;
            }
            break;
        }                                           /* keyboard case */
        case LOCATOR:{
            SRGP_getLocator(&locMeasure);
            switch (locMeasure.buttonOfMostRecentTransition) {
            case LEFT_BUTTON:             /* Defining remaining control points */
                if ((locMeasure.buttonChord[LEFT_BUTTON] == DOWN) &&
                    pastlocMeasure.position.x > 0) {
                    SRGP_setLocatorEchoRubberAnchor(locMeasure.position);
                    SRGP_line(pastlocMeasure.position, locMeasure.position);
                    pastlocMeasure = locMeasure;
                    ControlPoints[numCtl] = locMeasure.position;
                    numCtl++;
                    if (numCtl == 4) {
                        SRGP_setLineStyle(CONTINUOUS);      /* To draw curve */
                        SRGP_setLineWidth(2);
                        DrawCurve(ControlPoints, NUM_STEPS);
                        pastlocMeasure.position.x = –1;
                        SRGP_setLocatorEchoType(CURSOR);
                    }
                    break;
            case RIGHT_BUTTON:            /* Start new set of control points */
                    SRGP_setLocatorEchoRubberAnchor(locMeasure.position);
                    pastlocMeasure = locMeasure;
                    SRGP_setLocatorEchoType(RUBBER_LINE);
                    SRGP_setLineStyle(DASHED);     /* To draw control polygon */
                    SRGP_setLineWidth(1);
                    ControlPoints[0] = locMeasure.position;
                    numCtl = 1;
                    break;
                    }
                }               /* button handling */
            }                   /* locator case */
        }                       /* device switch */
    } while (!terminate);
    SRGP_end();
}
```

9.2.4 Uniform Nonrational B-Splines

The term **spline** goes back to the long flexible strips of metal used by draftsmen to lay out the surfaces of airplanes, cars, and ships. "Ducks," which are weights attached to the splines, were used to pull the spline in various directions. The metal splines, unless severely stressed, had second-order continuity. The mathematical equivalent of these strips, the **natural cubic spline,** is a C^0, C^1, and C^2 continuous cubic polynomial that interpolates (passes through) the control points. This polynomial has one more degree of continuity than is inherent in the Hermite and Bézier forms. Thus, splines are inherently smoother than are the previous forms.

The polynomial coefficients for natural cubic splines, however, are dependent on all n control points; their calculation involves inverting an $n + 1$ by $n + 1$ matrix [BART87]. This characteristic has two disadvantages: moving any one control point affects the entire curve, and the computation time needed to invert the matrix can interfere with rapid interactive reshaping of a curve.

B-splines, discussed in this section, consist of curve segments whose polynomial coefficients depend on just a few control points. This behavior is called **local control.** Thus, moving a control point affects only a small part of a curve. In addition, the time needed to compute the coefficients is greatly reduced. B-splines have the same continuity as do natural splines, but do not interpolate their control points.

In the following discussion, we change our notation slightly, since we must discuss an entire curve consisting of several curve segments, rather than its individual segments. A curve segment does not need to pass through its control points, and the two continuity conditions on a segment come from the adjacent segments. This behavior results from sharing control points between segments, so it is best to describe the process in terms of all the segments at once.

Cubic B-splines approximate a series of $m + 1$ control points $P_0, P_1, \dots P_m$, $m \geq 3$, with a curve consisting of $m - 2$ cubic polynomial curve segments $Q_3, Q_4, \dots, Q_m$. Although such cubic curves might be defined each on its own domain $0 \leq t < 1$, we can adjust the parameter (making a substitution of the form $t = t + k$) such that the parameter domains for the various curve segments are sequential. Thus, we say that the parameter range on which Q_i is defined is $t_i \leq t < t_{i+1}$, for $3 \leq i \leq m$. In the particular case of $m = 3$, there is a single curve segment Q_3 that is defined on the interval $t_3 \leq t < t_4$ by four control points, P_0 to P_3.

For each $i \geq 4$, there is a join point or **knot** between Q_{i-1} and Q_i at the parameter value t_i; the parameter value at such a point is called a **knot value**. The initial

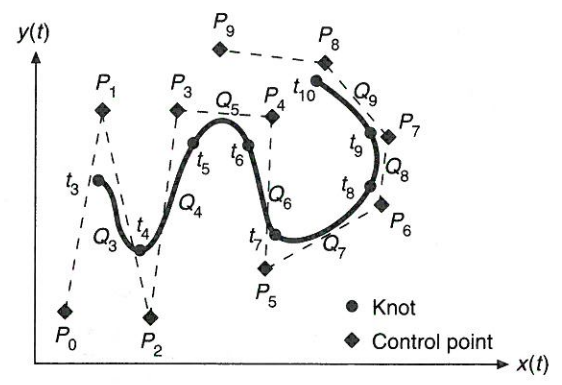


Figure 9.18 A B-spline with curve segments Q_3 through Q_9. This figure and many others in this chapter were created with a program written by Carles Castellsaquè.

and final points at t_3 and t_{m+1} are also called knots, so there are a total of $m - 1$ knots. Figure 9.18 shows a 2D B-spline curve with its knots marked. A closed B-spline curve is easy to create: The control points P_0, P_1, P_2 are repeated at the end of the sequence—$P_0, P_1, \dots, P_m, P_0, P_1, P_2$.

The term **uniform** means that the knots are spaced at equal intervals of the parameter t. Without loss of generality, we can assume that $t_3 = 0$, and the interval $t_{i+1} - t_i = 1$. Nonuniform nonrational B-splines, which permit unequal spacing between the knots, are discussed in Section 9.2.5. (In fact, the concept of knots is introduced in this section to set the stage for nonuniform splines.) The term **nonrational** is used to distinguish these splines from rational cubic polynomial curves, discussed in Section 9.2.6, where $x(t)$, $y(t)$, and $z(t)$ are each defined as the ratio of two cubic polynomials. The "B" stands for basis, since the splines can be represented as weighted sums of polynomial basis functions, in contrast to the natural splines, for which the weighted-sum property is not true.

Each of the $m - 2$ curve segments of a B-spline curve is defined by four of the $m + 1$ control points. In particular, curve segment Q_i is defined by points P_{i-3}, P_{i-2}, P_{i-1}, and P_i. Thus, the **B-spline geometry matrix** G_{Bs_i} for segment Q_i is

$$G_{\mathrm{Bs}_i} = [P_{i-3} \;\; P_{i-2} \;\; P_{i-1} \;\; P_i], \quad 3 \leq i \leq m. \tag{9.31}$$

The first curve segment, Q_3, is defined by the points P_0 through P_3 over the parameter range $t_3 = 0$ to $t_4 = 1$, Q_4 is defined by the points P_1 through P_4 over the parameter range $t_4 = 1$ to $t_5 = 2$, and the last curve segment, Q_m, is defined by the points P_{m-3}, P_{m-2}, P_{m-1}, and P_m over the parameter range $t_m = m - 3$ to $t_{m+1} = m - 2$. In general, curve segment Q_i begins somewhere near point P_{i-2} and ends somewhere near point P_{i-1}. We shall see that the B-spline blending functions are everywhere nonnegative and sum to unity, so the curve segment Q_i is constrained to the convex hull of its four control points.

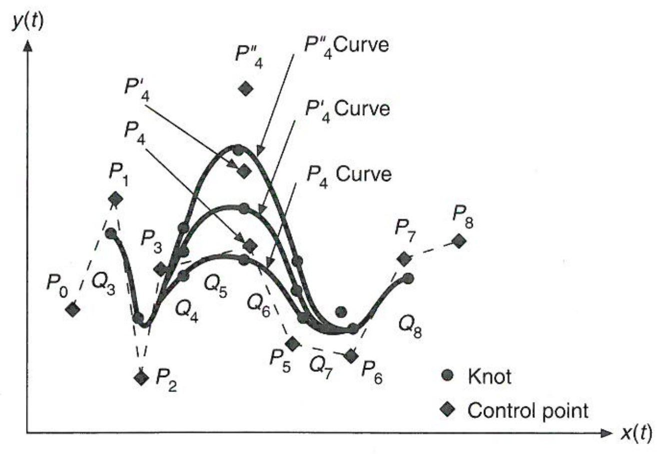


Figure 9.19 A B-spline with control point P_4 in several different locations.

Just as each curve segment is defined by four control points, each control point (except for those at the beginning and end of the sequence $P_0, P_1, \ldots, P_m$) influences four curve segments. Moving a control point in a given direction moves the four curve segments it affects in the same direction; the other curve segments are totally unaffected (see Fig. 9.19). This behavior is the local control property of B-splines and of all the other splines discussed in this chapter.

If we define T_i as the column vector $[(t-t_i)^3 \quad (t-t_i)^2 \quad (t-t_i) \quad 1]^{\mathrm{T}}$, then the B-spline formulation for curve segment i is

$$Q_i(t) = G_{\mathrm{Bs}_i} \cdot M_{\mathrm{Bs}} \cdot T_i, \quad t_i \le t < t_{i+1}. \tag{9.32}$$

We generate the entire curve by applying Eq. (9.32) for $3 \le i \le m$.

The **B-spline basis matrix**, M_{Bs}, relates the geometrical constraints G_{Bs} to the blending functions and the polynomial coefficients:

$$M_{\mathrm{Bs}} = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}. \tag{9.33}$$

This matrix is derived in [BART87].

The B-spline blending functions B_{Bs} are given by the product $M_{\mathrm{Bs}} \cdot T_i$, analogously to the previous Bézier and Hermite formulations. Note that the blending functions for each curve segment are exactly the same, because, for each segment i, the values of $t - t_i$ range from 0 at $t = t_i$ to 1 at $t = t_{i+1}$. If we replace $t - t_i$ by t, and replace the interval $[t_i, t_{i+1}]$ by $[0, 1]$, we have

$$\begin{aligned} B_{\mathrm{Bs}} &= M_{\mathrm{Bs}} \cdot T = [B_{\mathrm{Bs\text{-}3}} \quad B_{\mathrm{Bs\text{-}2}} \quad B_{\mathrm{Bs\text{-}1}} \quad B_{\mathrm{Bs0}}]^{\mathrm{T}} \\ &= \frac{1}{6}[-t^3 + 3t^2 - 3t + 1 \quad 3t^3 - 6t^2 + 4 \quad -3t^3 + 3t^2 + 3t + 1 \quad t^3]^{\mathrm{T}} \\ &= \frac{1}{6}[(1-t)^3 \quad 3t^3 - 6t^2 + 4 \quad -3t^3 + 3t^2 + 3t + 1 \quad t^3]^{\mathrm{T}}, \quad 0 \le t < 1. \end{aligned} \tag{9.34}$$

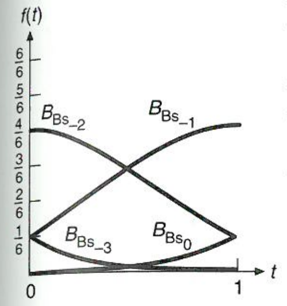


Figure 9.20
The four B-spline blending functions from Eq. (9.34). At $t = 0$ and $t = 1$, just three of the functions are nonzero.

Figure 9.20 shows the B-spline blending functions B_{Bs}. Because the four functions sum to 1 and are nonnegative, the convex-hull property holds for each curve segment of a B-spline. See [BART87] to understand the relation between these blending functions and the Bernstein polynomial basis functions.

Expanding Eq. (9.32), again replacing $t - t_i$ with t at the second equal-to sign, we have

$$\begin{aligned} Q_i(t - t_i) &= G_{\mathrm{Bs}_i} \cdot M_{\mathrm{Bs}} \cdot T_i = G_{\mathrm{Bs}_i} \cdot M_{\mathrm{Bs}} \cdot T \\ &= G_{\mathrm{Bs}_i} \cdot B_{\mathrm{Bs}} = P_{i-3} \cdot B_{\mathrm{Bs}-3} + P_{i-2} \cdot B_{\mathrm{Bs}-2} + P_{i-1} \cdot B_{\mathrm{Bs}-1} + P_i \cdot \mathrm{B}_{\mathrm{Bs}0} \\ &= \frac{(1-t)^3}{6} P_{i-3} + \frac{3t^3 - 6t^2 + 4}{6} P_{i-2} + \frac{-3t^3 + 3t^2 + 3t + 1}{6} P_{i-1} \\ &\quad + \frac{t^3}{6} P_i, \quad 0 \le t < 1. \end{aligned} \tag{9.35}$$

It is easy to show that Q_i and Q_{i+1} are C^0, C^1, and C^2 continuous where they join. The additional continuity afforded by B-splines is attractive, but it comes at the cost of less control of where the curve goes. We can force the curve to interpolate specific points by replicating control points; this is useful both at endpoints and at intermediate points on the curve. For instance, if $P_{i-2} = P_{i-1}$, the curve is pulled closer to this point because curve segment Q_i is defined by just three different points, and the point $P_{i-2} = P_{i-1}$ is weighted twice in Eq. (9.35)—once by $B_{\mathrm{Bs}-2}$ and once by $B_{\mathrm{Bs}-1}$.

If a control point is used three times—for instance, if $P_{i-2} = P_{i-1} = P_i$—then Eq. (9.35) becomes

$$Q_i(t) = P_{i-3} \cdot B_{\mathrm{Bs}-3} + P_i \cdot (B_{\mathrm{Bs}-2} + B_{\mathrm{Bs}-1} + B_{\mathrm{Bs}0}). \tag{9.36}$$

Q_i is clearly a straight line. Furthermore, the point P_{i-2} is interpolated by the line at $t = 1$, where the three weights applied to P_i sum to 1, but P_{i-3} is not in general interpolated at $t = 0$. Another way to think of this behavior is that the convex hull for Q_i is now defined by just two distinct points, so Q_i has to be a line. Figure 9.21 shows the effect of multiple control points at the interior of a B-spline.

Another technique for interpolating endpoints, **phantom vertices,** is discussed in [BARS83; BART87]. We shall see in the next section that, with nonuniform B-splines, endpoints and internal points can be interpolated in a more natural way than they can with the uniform B-splines.

9.2.5 Nonuniform, Nonrational B-Splines

Nonuniform, nonrational B-splines differ from the uniform, nonrational B-splines discussed in Section 9.2.4 in that the parameter interval between successive

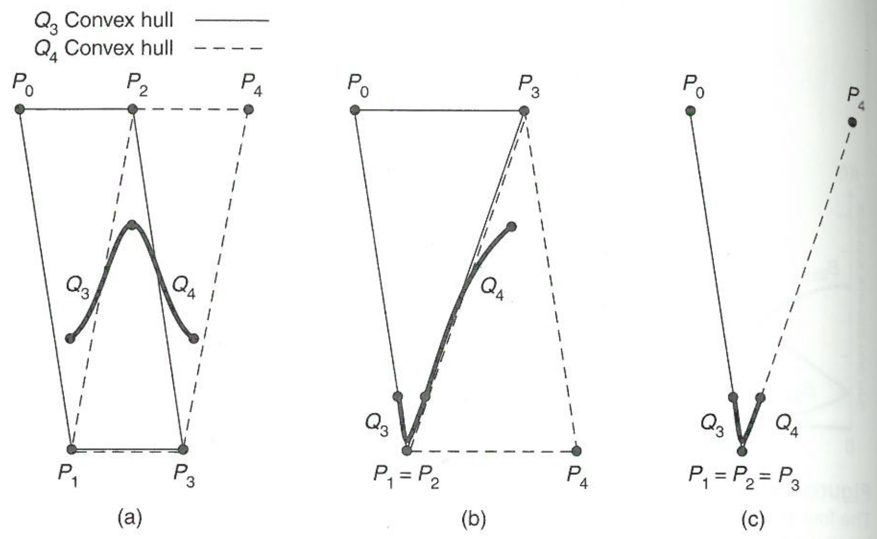


Figure 9.21 The effect of multiple control points on a uniform B-spline curve. In (a), there are no multiple control points. The convex hulls of the two curves overlap; the join point between Q_3 and Q_4 is in the region shared by both convex hulls. In (b), there is a double control point, so the two convex hulls share edge P_2P_3; the join point is therefore constrained to lie on this edge. In (c), there is a triple control point, and the two convex hulls are straight lines that share the triple point; hence, the join point is also at the triple point. Because the convex hulls are straight lines, the two curve segments must also be straight lines. There is C^2 but only G^0 continuity at the join.

knot values is not necessarily uniform. The nonuniform knot-value sequence means that the blending functions are no longer the same for each interval, but rather vary from curve segment to curve segment.

These curves have several advantages over uniform B-splines. First, continuity at selected join points can be reduced from C^2 to C^1 to C^0 to none. If the continuity is reduced to C^0, then the curve interpolates a control point, but without the undesirable effect of uniform B-splines, where the curve segments on either side of the interpolated control point are straight lines. Also, starting and ending points can be easily interpolated exactly, without at the same time introducing linear segments. As is discussed in [FOLE90], it is possible to add an additional knot and control point to nonuniform B-splines, so the resulting curve can be easily reshaped, whereas this modification cannot be done with uniform B-splines.

The increased generality of nonuniform B-splines requires a notation slightly different from that used for uniform B-splines. As before, the spline is a piecewise continuous curve made up of cubic polynomials, approximating the control points P_0 through P_m. The **knot-value sequence** is a nondecreasing sequence of knot values t_0 through t_{m+4} (that is, there are four more knots than there are control points). Because the smallest number of control points is four, the smallest knot sequence has eight knot values and the curve is defined over the parameter interval from t_3 to t_4.

The only restriction on the knot sequence is that it be nondecreasing, which allows successive knot values to be equal. When this occurs, the parameter value is called a **multiple knot** and the number of identical parameter values is called the **multiplicity** of the knot (a single unique knot has multiplicity of 1). For instance, in the knot sequence (0, 0, 0, 0, 1, 1, 2, 3, 4, 4, 5, 5, 5, 5), the knot value 0 has multiplicity 4; value 1 has multiplicity 2; values 2 and 3 have multiplicity 1; value 4 has multiplicity 2; and value 5 has multiplicity 4.

Curve segment Q_i is defined by control points P_{i-3}, P_{i-2}, P_{i-1}, P_i and by blending functions $B_{i-3,4}(t)$, $B_{i-2,4}(t)$, $B_{i-1,4}(t)$, $B_{i,4}(t)$, as the weighted sum

$$Q_i(t) = P_{i-3} \cdot B_{i-3,4}(t) + P_{i-2} \cdot B_{i-2,4}(t) + P_{i-1} \cdot B_{i-1,4}(t) + P_i \cdot B_{i,4}(t)$$
$$3 \le i \le m, \quad t_i \le t < t_{i+1}. \tag{9.37}$$

The curve is not defined outside the interval t_3 through t_{m+1}. When $t_i = t_{i+1}$ (a multiple knot), curve segment Q_i is a single point. It is this notion of a curve segment reducing to a point that provides the extra flexibility of nonuniform B-splines.

There is no single set of blending functions, as there was for other types of splines. The functions depend on the intervals between knot values and are defined recursively in terms of lower-order blending functions. $B_{i,j}(t)$ is the jth-order blending function for weighting control point P_i. Because we are working with fourth-order (that is, third-degree, or cubic) B-splines, the recursive definition ends with $B_{i,4}(t)$ and can be presented easily in its "unwound" form. The recurrence for cubic B-splines is

$$B_{i,1}(t) = \begin{cases} 1, & t_i \le t < t_{i+1} \\ 0, & \text{otherwise,} \end{cases}$$

$$B_{i,2}(t) = \frac{t - t_i}{t_{i+1} - t_i} B_{i,1}(t) + \frac{t_{i+2} - t}{t_{i+2} - t_{i+1}} B_{i+1,1}(t),$$

$$B_{i,3}(t) = \frac{t - t_i}{t_{i+2} - t_i} B_{i,2}(t) + \frac{t_{i+3} - t}{t_{i+3} - t_{i+1}} B_{i+1,2}(t),$$

$$B_{i,4}(t) = \frac{t - t_i}{t_{i+3} - t_i} B_{i,3}(t) + \frac{t_{i+4} - t}{t_{i+4} - t_{i+1}} B_{i+1,3}(t). \tag{9.38}$$

It can be shown that the blending functions are nonnegative and sum to one, so nonuniform B-spline curve segments lie within the convex hulls of their four control points. For knots of multiplicity greater than one, the denominators can be zero because successive knot values can be equal: Division by zero is defined to yield zero.

Increasing knot multiplicity has two effects. First, the spline, evaluated at any known knot value t_i, will automatically yield a point within the convex hull of the points P_{i-3}, P_{i-2}, and P_{i-1}. If t_i and t_{i+1} are equal, they must lie in the convex hull of P_{i-3}, P_{i-2}, and P_{i-1}, *and* in the convex hull of P_{i-2}, P_{i-1}, and P_i. Thus, they must actually lie on the line segment between P_{i-2} and P_{i-1}. In the same way, if $t_i = t_{i+1} = t_{i+2}$, then this knot must lie *at* P_{i-1}. If $t_i = t_{i+1} = t_{i+2} = t_{i+3}$, then the knot must lie both at P_{i-1} and at P_i—the curve becomes broken. Second, the multiple knots will reduce parametric continuity: from C^2 to C^1 continuity for one extra knot (multiplicity 2); from C^1 to C^0 continuity for two extra knots (multiplicity 3); from C^0 to no continuity for three extra knots (multiplicity 4).

Interactive creation of nonuniform splines typically involves pointing at control points, with multiple control points indicated simply by successive selection of the same point. Another way is to point directly at the curve with a multibutton mouse: A double click on one button can indicate a double control point; a double click on another button can indicate a double knot.

9.2.6 Nonuniform, Rational Cubic Polynomial Curve Segments

General rational cubic curve segments are ratios of polynomials:

$$x(t) = \frac{X(t)}{W(t)}, \quad y(t) = \frac{Y(t)}{W(t)}, \quad z(t) = \frac{Z(t)}{W(t)}, \tag{9.39}$$

where $X(t)$, $Y(t)$, $Z(t)$, and $W(t)$ are all cubic polynomial curves whose control points are defined in homogeneous coordinates. We can also think of the curve as existing in homogeneous space as $Q(t) = [X(t) \quad Y(t) \quad Z(t) \quad W(t)]^T$. As always, moving from homogeneous space to 3-space involves dividing by $W(t)$. We can transform any nonrational curve to a rational curve by adding $W(t) = 1$ as a fourth element. In general, the polynomials in a rational curve can be Bézier, Hermite, or any other type. When they are B-splines, we have nonuniform rational B-splines, sometimes called **NURBS** [FORR80].

Rational curves are useful for two reasons. The first and most important reason is that they are invariant under rotation, scaling, translation, and perspective transformations of the control points (nonrational curves are invariant under only rotation, scaling, and translation). Thus, the perspective transformation needs to be applied to only the control points, which can then be used to generate the perspective transformation of the original curve. The alternative to converting a nonrational curve to a rational curve prior to a perspective transformation is first to generate points on the curve itself, and then to apply the perspective transformation to *each* point—a far less efficient process. An analogous observation is that the perspective transformation of a sphere is not the same as a sphere whose center and radius are the transformed center and radius of the original sphere.

A second advantage of rational splines is that, unlike nonrationals, they can define precisely any of the conic sections. We can only approximate a conic with nonrationals, by using many control points close to the conic. This second property is useful in those applications, particularly CAD, where general curves and surfaces as well as conics are needed. Both types of entities can be defined with NURBS.

For further discussion of conics and NURBS, see [FAUX79; BÖHM84; TILL83].

9.2.7 Fitting Curves to Digitized Points

An engineer or artist often has a nonelectronic representation of a complex shape that can be digitized as a series of discrete points. For example, a paper hardcopy

of a shape may be all that is available. For additional manipulation of the shape we might like to fit a smooth curve or series of curves to the (usually) imprecise digitized representation. Various curve-fitting techniques have been published; all have various advantages and disadvantages. Schneider [SCHN90] has developed a method for approximating digitized curves with piecewise Bézier segments. Advantages over previous approaches are geometric continuity, stability, and ease of implementation. A complete C implementation of the algorithm is available in [SCHN90]. Figure 9.22 shows an example of the method applied to a digitized shape.

9.2.8 Comparison of the Cubic Curves

The different types of parametric cubic curves can be compared by several different criteria, such as ease of interactive manipulation, degree of continuity at join points, generality, and speed of various computations using the representations. Of course, it is not necessary to choose a single representation, since it is possible to convert among all representations, as discussed in [FOLE90]. For instance, nonuniform rational B-splines can be used as an internal representation, while the user might interactively manipulate Bézier control points or Hermite control points and tangent vectors. Some interactive graphics editors provide the user with Hermite curves while representing them internally in the Bézier form supported by PostScript [ADOB85]. In general, the user of an interactive CAD system may be given several choices, such as Hermite, Bézier, uniform B-splines, and nonuniform B-splines. The nonuniform rational B-spline representation is likely to be used internally, because it is the most general.

Table 9.1 compares most of the curve forms mentioned in this section. Ease of interactive manipulation is not included explicitly in the table, because that attribute is quite application specific. *Number of parameters controlling a curve*

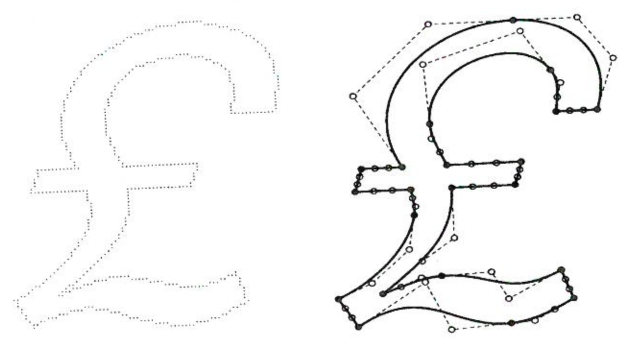


Figure 9.22 A digitized character, showing the original sample, the fitted curves, and the Bézier control points. (Courtesy of Academic Press, Inc.)

Table 9.1 Comparison of Four Different Forms of Parametric Cubic Curves

	Hermite	Bézier	Uniform B-Spline	Nonuniform B-spline
Convex hull defined by control points	N/A	Yes	Yes	Yes
Interpolates some control points	Yes	Yes	No	No
Interpolates all control points	Yes	No	No	No
Ease of subdivision	Good	Best	Average	High
Continuities inherent in representation	C^0 G^0	C^0 G^0	C^2 G^2	C^2 G^2
Continuities achieved easily	C^1 G^1	C^1 G^1	C^2 G^{2*}	C^2 G^{2*}
Number of parameters controlling a curve segment	4	4	4	5

*Except for special case discussed in Section 9.2.

segment is the four geometrical constraints plus other parameters, such as knot spacing for nonuniform splines. *Continuity achieved easily* refers to constraints such as forcing control points to be collinear to allow G^1 continuity. Because C^n continuity is more restrictive than is G^n, any form that can attain C^n can by definition also attain at least G^n.

When only geometric continuity is required, as is often the case for CAD, the choice is narrowed to the various types of splines, all of which can achieve both G^1 and G^2 continuity. Of the three types of splines in the table, uniform B-splines are the most limiting. The possibility of multiple knots afforded by nonuniform B-splines gives more shape control to the user. Of course, a good user interface that allows the user to exploit this power easily is important.

It is customary to provide the user with the ability to drag control points or tangent vectors interactively, continually displaying the updated spline. Figure 9.19 shows such a sequence for B-splines. One of the disadvantages of B-splines in some applications is that the control points are not on the spline itself. It is possible, however, not to display the control points, allowing the user instead to interact with the knots (which must be marked so they can be selected).

9.3 PARAMETRIC BICUBIC SURFACES

Parametric bicubic surfaces are a generalization of parametric cubic curves. Recall the general form of the parametric cubic curve $Q(t) = G \cdot M \cdot T$, where G, the geometry matrix, is a constant. First, for notational convenience, we replace t with s, giving $Q(s) = G \cdot M \cdot S$. If we now allow the points in G to vary in 3D along some path that is parameterized on t, we have

$$Q(s,t) = [G_1(t) \;\; G_2(t) \;\; G_3(t) \;\; G_4(t)] \cdot M \cdot S\,. \tag{9.40}$$

Now, for a fixed t_1, $Q(s, t_1)$ is a curve because $G(t_1)$ is constant. Allowing t to take on some new value—say, t_2—where $t_2 - t_1$ is very small, $Q(s, t)$ is a slightly different curve. By repeating this process for arbitrarily many other values of t_2 between 0 and 1, we define an entire family of curves, each member arbitrarily close to another curve. The set of all such curves defines a surface. If the $G_i(t)$ are themselves cubics, the surface is said to be a **parametric bicubic surface.**

Continuing with the case that the $G_i(t)$ are cubics, each can be represented as $G_i(t) = \mathbf{G}_i \cdot M \cdot T$, where $\mathbf{G}_i = [g_{i1} \; g_{i2} \; g_{i3} \; g_{i4}]$ (the $\mathbf{G}$ and g are used to distinguish from the G used for the curve). Hence, g_{i1} is the first element of the geometry matrix for curve $G_i(t)$, and so on.

Now let us transpose the equation $G_i(t) = \mathbf{G}_i \cdot M \cdot T$, using the identity $(A \cdot B \cdot C)^{\mathrm{T}} = C^{\mathrm{T}} \cdot B^{\mathrm{T}} \cdot A^{\mathrm{T}}$. The result is $G_i(t) = T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \mathbf{G}_i^{\mathrm{T}} = T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot [g_{i1} \; g_{i2} \; g_{i3} \; g_{i4}]^{\mathrm{T}}$. If we now substitute this result in Eq. (9.40) for each of the four points, we have

$$Q(s,t) = T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \begin{bmatrix} g_{11} & g_{21} & g_{31} & g_{41} \\ g_{12} & g_{22} & g_{32} & g_{42} \\ g_{13} & g_{23} & g_{33} & g_{43} \\ g_{14} & g_{24} & g_{34} & g_{44} \end{bmatrix} \cdot M \cdot S\,, \tag{9.41}$$

or

$$Q(s,t) = T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \mathbf{G} \cdot M \cdot S, \quad 0 \le s, t \le 1. \tag{9.42}$$

Written separately for each of x, y, and z, the form is

$$x(s,t) = T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \mathbf{G}_x \cdot M \cdot S,$$

$$y(s,t) = T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \mathbf{G}_y \cdot M \cdot S,$$

$$z(s,t) = T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \mathbf{G}_z \cdot M \cdot S. \tag{9.43}$$

Given this general form, we now move on to examine specific ways to specify surfaces using different geometry matrices.

9.3.1 Hermite Surfaces

Hermite surfaces are completely defined by a 4×4 geometry matrix $\mathbf{G}_{\mathrm{H}}$. Derivation of $\mathbf{G}_{\mathrm{H}}$ follows the same approach that we used to find Eq. (9.42). We further elaborate the derivation here, applying it just to $x(s, t)$. First, we replace t by s in Eq. (9.13), to get $x(s) = G_{H_x} \cdot M_H \cdot S$. Rewriting this expression further such that the Hermite geometry matrix G_{H_x} is not constant, but is rather a function of t, we obtain

$$x(s, t) = G_{H_x}(t) \cdot M_H \cdot S = [P_{1_x}(t) \;\; P_{4_x}(t) \;\; R_{1_x}(t) \;\; R_{4_x}(t)] \cdot M_H \cdot S \,. \qquad (9.44)$$

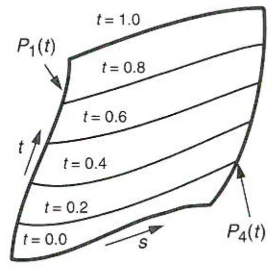


Figure 9.23
Lines of constant parameter values on a bicubic surface: $P_1(t)$ is at $s = 0$, and $P_4(t)$ is at $s = 1$.

The functions $P_{1_x}(t)$ and $P_{4_x}(t)$ define the x components of the starting and ending points for the curve in parameter s. Similarly, $R_{1_x}(t)$ and $R_{4_x}(t)$ are the tangent vectors at these points. For any specific value of t, there are two specific endpoints and tangent vectors. Figure 9.23 shows $P_1(t)$, $P_4(t)$, and the cubic in s that is defined when $t = 0.0$, 0.2, 0.4, 0.6, 0.8, and 1.0. The surface patch is essentially a cubic interpolation between $P_1(t) = Q(0, t)$ and $P_4(t) = Q(1, t)$ or, alternatively, between $Q(s, 0)$ and $Q(s, 1)$.

In the special case that the four interpolants $Q(0, t)$, $Q(1, t)$, $Q(s, 0)$, and $Q(s, 1)$ are straight lines, the result is a **ruled surface**. If the interpolants are also coplanar, then the surface is a four-sided planar polygon.

Continuing with the derivation, let each of $P_{1_x}(t)$, $P_{4_x}(t)$, $R_{1_x}(t)$, and $R_{4_x}(t)$ be represented in Hermite form as

$$P_{1_x}(t) = [g_{11} \; g_{12} \; g_{13} \; g_{14}]_x \cdot M_H \cdot T \,, \qquad P_{4_x}(t) = [g_{21} \; g_{22} \; g_{23} \; g_{24}]_x \cdot M_H \cdot T \,,$$
$$R_{1_x}(t) = [g_{31} \; g_{32} \; g_{33} \; g_{34}]_x \cdot M_H \cdot T \,, \qquad R_{4_x}(t) = [g_{41} \; g_{42} \; g_{43} \; g_{44}]_x \cdot M_H \cdot T \,. \qquad (9.45)$$

These four cubics can be rewritten together as a single equation:

$$[P_{1_x}(t) \;\; P_{4_x}(t) \;\; R_{1_x}(t) \;\; R_{4_x}(t)]^{\mathrm{T}} = \boldsymbol{G}_{H_x} \cdot M_H \cdot T, \qquad (9.46)$$

where

$$\boldsymbol{G}_{H_x} = \begin{bmatrix} g_{11} & g_{12} & g_{13} & g_{14} \\ g_{21} & g_{22} & g_{23} & g_{24} \\ g_{31} & g_{32} & g_{33} & g_{34} \\ g_{41} & g_{42} & g_{43} & g_{44} \end{bmatrix}_x . \qquad (9.47)$$

Transposing both sides of Eq. (9.46) results in

$$[P_{1_x}(t) \;\; P_{4_x}(t) \;\; R_{1_x}(t) \;\; R_{4_x}(t)] = T^{\mathrm{T}} \cdot M_H^{\mathrm{T}} \cdot \begin{bmatrix} g_{11} & g_{21} & g_{31} & g_{41} \\ g_{12} & g_{22} & g_{32} & g_{42} \\ g_{13} & g_{23} & g_{33} & g_{43} \\ g_{14} & g_{24} & g_{34} & g_{44} \end{bmatrix}_x = T^{\mathrm{T}} \cdot M_H^{\mathrm{T}} \cdot \boldsymbol{G}_{H_x} . \qquad (9.48)$$

Substituting Eq. (9.48) into Eq. (9.44) yields

$$x(s, t) = T^{\mathrm{T}} \cdot M_H{}^{\mathrm{T}} \cdot \boldsymbol{G}_{H_x} \cdot M_H \cdot S; \qquad (9.49)$$

similarly,

$$y(s, t) = T^{\mathrm{T}} \cdot M_H{}^{\mathrm{T}} \cdot \boldsymbol{G}_{H_y} \cdot M_H \cdot S, \quad z(s, t) = T^{\mathrm{T}} \cdot M_H{}^{\mathrm{T}} \cdot \boldsymbol{G}_{H_z} \cdot M_H \cdot S. \qquad (9.50)$$

The three 4×4 matrixes $\boldsymbol{G}_{H_x}$, $\boldsymbol{G}_{H_y}$, and $\boldsymbol{G}_{H_z}$ play the same role for Hermite surfaces as did the single matrix G_H for curves.

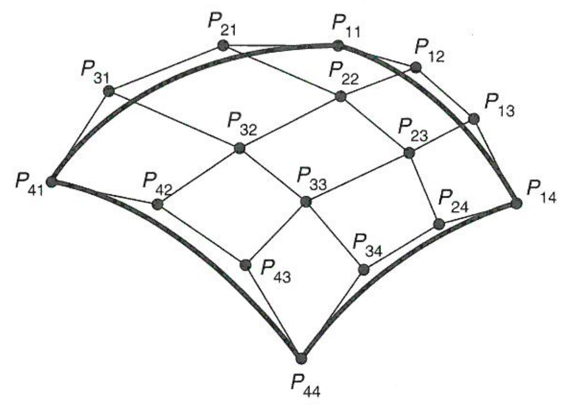


Figure 9.24 Sixteen control points for a Bézier bicubic patch.

The Hermite bicubic permits C^1 and G^1 continuity from one patch to the next in much the same way the Hermite cubic permits C^1 and G^1 continuity from one curve segment to the next. Details can be found in Chapter 11 of [FOLE90].

9.3.2 Bézier Surfaces

The Bézier bicubic formulation can be derived in exactly the same way as the Hermite cubic. The results are

$$\begin{aligned} x(s, t) &= T^{\mathrm{T}} \cdot M_B{}^{\mathrm{T}} \cdot \boldsymbol{G}_{B_x} \cdot M_B \cdot S, \\ y(s, t) &= T^{\mathrm{T}} \cdot M_B{}^{\mathrm{T}} \cdot \boldsymbol{G}_{B_y} \cdot M_B \cdot S, \\ z(s, t) &= T^{\mathrm{T}} \cdot M_B{}^{\mathrm{T}} \cdot \boldsymbol{G}_{B_z} \cdot M_B \cdot S. \end{aligned} \qquad (9.51)$$

The Bézier geometry matrix $\boldsymbol{G}$ consists of 16 control points, as shown in Fig. 9.24. Bézier surfaces are attractive in interactive design for the same reason as Bézier curves are: Some of the control points interpolate the surface, giving convenient precise control, whereas tangent vectors also can be controlled explicitly. When Bézier surfaces are used as an internal representation, their convex-hull property is attractive.

We create C^0 and G^0 continuity across patch edges by making the four common control points equal. G^1 continuity occurs when the two sets of four control points on either side of the edge are collinear with the points on the edge. In Fig. 9.25, the following sets of control points are collinear and define four line segments whose lengths all have the same ratio k: (P_{13}, P_{14}, P_{15}), (P_{23}, P_{24}, P_{25}), (P_{33}, P_{34}, P_{35}), and (P_{43}, P_{44}, P_{45}). The teapot shown in Fig. 9.1 was modeled by 32 Bézier patches, all joined to ensure G^1 continuity.

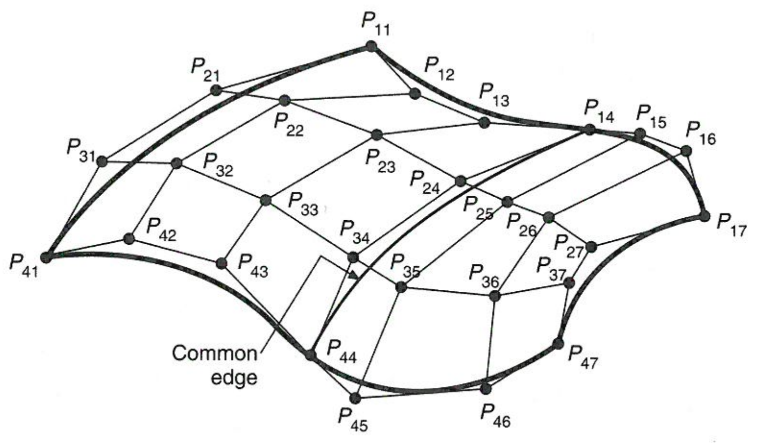


Figure 9.25 Two Bézier patches joined along the edge P_{14}, P_{24}, P_{34}, and P_{44}.

9.3.3 B-Spline Surfaces

B-spline patches are represented as

$$\begin{aligned} x(s,t) &= T^{\mathrm{T}} \cdot M_{\mathrm{Bs}}{}^{\mathrm{T}} \cdot \boldsymbol{G}_{\mathrm{Bs}_x} \cdot M_{\mathrm{Bs}} \cdot S, \\ y(s,t) &= T^{\mathrm{T}} \cdot M_{\mathrm{Bs}}{}^{\mathrm{T}} \cdot \boldsymbol{G}_{\mathrm{Bs}_y} \cdot M_{\mathrm{Bs}} \cdot S, \qquad (9.52) \\ z(s,t) &= T^{\mathrm{T}} \cdot M_{\mathrm{Bs}}{}^{\mathrm{T}} \cdot \boldsymbol{G}_{\mathrm{Bs}_z} \cdot M_{\mathrm{Bs}} \cdot S. \end{aligned}$$

C^2 continuity across boundaries is automatic with B-splines; no special arrangements of control points are needed, except to avoid duplicate control points, which create discontinuities.

Bicubic nonuniform and rational B-spline surfaces and other rational surfaces are similarly analogous to their cubic counterparts. All the techniques for display carry over directly to the bicubic case.

9.3.4 Normals to Surfaces

The normal to a bicubic surface, needed for shading (Chapter 14), for performing interference detection in robotics, for calculating offsets for numerically controlled machining, and for doing other calculations, is easy to find. From Eq. (9.42), the s tangent vector of the surface $Q(s, t)$ is

$$\begin{aligned} \frac{\partial}{\partial s} Q(s,t) &= \frac{\partial}{\partial s}(T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \boldsymbol{G} \cdot M \cdot S) = T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \boldsymbol{G} \cdot M \cdot \frac{\partial}{\partial s}(S) \\ &= T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \boldsymbol{G} \cdot M \cdot [3s^2 \;\; 2s \;\; 1 \;\; 0]^{\mathrm{T}}, \qquad (9.53) \end{aligned}$$

and the t tangent vector is

$$\begin{aligned} \frac{\partial}{\partial t} Q(s,t) &= \frac{\partial}{\partial t}(T^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \boldsymbol{G} \cdot M \cdot S) = \frac{\partial}{\partial t}(T^{\mathrm{T}}) \cdot M^{\mathrm{T}} \cdot \boldsymbol{G} \cdot M \cdot S \\ &= [3t^2 \;\; 2t \;\; 1 \;\; 0]^{\mathrm{T}} \cdot M^{\mathrm{T}} \cdot \boldsymbol{G} \cdot M \cdot S. \qquad (9.54) \end{aligned}$$

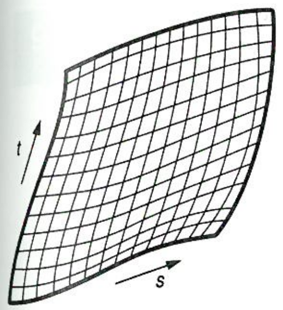


Figure 9.26 A single surface patch displayed as curves of constant s and constant t.

Both tangent vectors are parallel to the surface at the point (s, t), and their cross-product is therefore perpendicular to the surface. Notice that, if both tangent vectors are zero, the cross-product is zero, and there is no meaningful surface normal. Recall that a tangent vector can go to zero at join points that have C^1 but not G^1 continuity.

Each of the tangent vectors is, of course, a 3-tuple, because Eq. (9.42) represents the x, y, and z components of the bicubic. With the notation x_s for the x component of the s tangent vector, y_s for the y component, and z_s for the z component, the normal is

$$\frac{\partial}{\partial s} Q(s,t) \times \frac{\partial}{\partial t} Q(s,t) = [y_s z_t - y_t z_s \quad z_s x_t - z_t x_s \quad x_s y_t - x_t y_s]. \qquad (9.55)$$

The surface normal is a biquintic (two-variable, fifth-degree) polynomial and hence is fairly expensive to compute. [SCHW82] gives a bicubic approximation that is satisfactory as long as the patch itself is relatively smooth.

9.3.5 Displaying Bicubic Surfaces

Like curves, surfaces can be displayed by iterative evaluation of the bicubic polynomials. Iterative evaluation is best suited for displaying bicubic patches in the style of Fig. 9.26. Each of the curves of constant s and constant t on the surface is itself a cubic, so display of each of the curves is straightforward, as in Prog. 9.2.

Brute-force iterative evaluation for surfaces is even more expensive than for curves, because the surface equations must be evaluated about $2/\delta^2$ times. For $\delta = 0.1$, this value is 200; for $\delta = 0.01$, it is 20,000. These numbers make the alternative, forward-differencing method even more attractive than it is for curves. This method and other useful ways to display bicubic surfaces are presented in [FOLE90; FORR79].

Program 9.2

Function to display bicubic patch as a grid. Functions X(s,t), Y(s,t), and Z(s,t) evaluate the surface using the coefficient matrix coefficients.

```
typedef float Coeffs[4][4][3];

void DrawSurface( Coeffs coefficients, int ns, int nt, int n )
   /* the variable coefficients are the coefficients for Q(s,t) */
   /* ns and nt are the number of curves of constant s and t to be drawn */
{
   float del, dels, delt, s, t;
   int i, j;
   /* Initialize */
   del = 1.0 / n;           /* Step size to use in drawing each curve */
   dels = 1.0 / (ns – 1);   /* Step size in s when moving to next curve of constant t */
   delt = 1.0 / (nt – 1);   /* Step size in t when moving to next curve of constant s */
```

```
    /* Draw ns curves of constant s, for s=0.0, dels, 2dels, ... 1.0 */
    for ( i = 0; i < ns; i++ ) {
        s = i * dels;
        /* Draw a curve of constant s, varying t from 0.0 to 1.0 */
        /* X, Y, and Z are functions to evaluate the bicubics for a given s and t */
        MoveAbs3(X(s, 0.0), Y(s, 0.0), Z(s, 0.0));
        for ( j = 0; j < n; j++ ) {
            t = j * del;
            /* n steps are used as t varies from 0.0 to 1.0 for each curve */
            LineAbs3(X(s, t), Y(s, t), Z(s, t));
        }
    }

    /* Draw nt curves of constant t, for t=0.0, delt, 2delt, ... 1.0 */
    for ( i = 0; i < nt; i++ ) {
        t = i * delt;
        /* Draw a curve of constant t, varying s from 0.0 to 1.0 */
        MoveAbs3(X(0.0, t), Y(0.0, t), Z(0.0, t));
        for ( j = 0; j < n; j++ ) {
            s = j * del;
            /* n steps are used as s varies from 0 to 1 for each curve */
            LineAbs3(X(s, t), Y(s, t), Z(s, t));
        }
    }
}
```

The functions $X(s, t)$, $Y(s, t)$, and $Z(s, t)$ can be easily developed for a specific type of surface. As an example, we will consider a Bézier surface. From Eq. (9.51), the $x(s, t)$ equation can be rewritten as

$$x(s,t) = [(1-t)^3 \;\; 3t(1-t)^2 \;\; 3t^2(1-t) \;\; t^3] \cdot \boldsymbol{G}_{\mathrm{B}_x} \cdot \begin{bmatrix} (1-s)^3 \\ 3s(1-s)^2 \\ 3s^2(1-s) \\ s^3 \end{bmatrix}, \qquad (9.56)$$

just by multiplying out the $T^{\mathrm{T}} \cdot M_{\mathrm{B}}{}^{\mathrm{T}}$ and $M_{\mathrm{B}} \cdot S$ matrices. Recall that $\boldsymbol{G}_{\mathrm{B}_x}$ is the matrix of the x component of the control points, which are shown in Fig. 9.24, so it can be written as

$$\boldsymbol{G}_{\mathrm{B}_x} = \begin{bmatrix} \boldsymbol{P}_{11} & \boldsymbol{P}_{21} & \boldsymbol{P}_{31} & \boldsymbol{P}_{41} \\ \boldsymbol{P}_{12} & \boldsymbol{P}_{22} & \boldsymbol{P}_{32} & \boldsymbol{P}_{42} \\ \boldsymbol{P}_{13} & \boldsymbol{P}_{23} & \boldsymbol{P}_{33} & \boldsymbol{P}_{43} \\ \boldsymbol{P}_{14} & \boldsymbol{P}_{24} & \boldsymbol{P}_{34} & \boldsymbol{P}_{44} \end{bmatrix}_x .$$

Finally, the completely expanded form of Eq. (9.56) can be written as

$$\begin{aligned} x(s,t) \;=\;& (1-s)^3(\boldsymbol{P}_{11_x}(1-t)^3 + 3\boldsymbol{P}_{12_x}(1-t)^2t + 3\boldsymbol{P}_{13_x}(1-t)t^2 + \boldsymbol{P}_{14_x}t^3) \\ +\;& 3(1-s)^2s(\boldsymbol{P}_{21_x}(1-t)^3 + 3\boldsymbol{P}_{22_x}(1-t)^2t + 3\boldsymbol{P}_{23_x}(1-t)t^2 + \boldsymbol{P}_{24_x}t^3) \\ +\;& 3(1-s)s^2(\boldsymbol{P}_{31_x}(1-t)^3 + 3\boldsymbol{P}_{32_x}(1-t)^2t + 3\boldsymbol{P}_{33_x}(1-t)t^2 + \boldsymbol{P}_{34_x}t^3) \\ +\;& s^3(\boldsymbol{P}_{41_x}(1-t)^3 + 3\boldsymbol{P}_{42_x}(1-t)^2t + 3\boldsymbol{P}_{43_x}(1-t)t^2 + \boldsymbol{P}_{44_x}t^3) . \end{aligned}$$

The equations for $y(s, t)$ and $z(s, t)$ are derived in an identical fashion. The functions $X(s, t)$, $Y(s, t)$, and $Z(s, t)$, which are required for Prog. 2.2, can be coded directly from the $x(s, t)$, $y(s, t)$, and $z(s, t)$ expressions. Functions for drawing other types of surfaces can be developed just as easily as for the Bézier surface.

9.4 QUADRIC SURFACES

The implicit surface equation of the form

$$f(x, y, z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0 \quad (9.57)$$

defines the family of quadric surfaces. For example, if $a = b = c = -k = 1$ and the remaining coefficients are zero, a unit sphere is defined at the origin. If a through f are zero, a plane is defined. Quadric surfaces are particularly useful in specialized applications such as molecular modeling [PORT79; MAX79] and have also been integrated into solid-modeling systems. Recall, too, that rational cubic curves can represent conic sections; similarly, rational bicubic surfaces can represent quadrics. Hence, the implicit quadratic equation is an alternative to rational surfaces, *if* only quadric surfaces are being represented. Other reasons for using quadrics include ease of

- Computing the surface normal
- Testing whether a point is on the surface [just substitute the point into Eq. (9.57), evaluate, and test for a result within some ϵ of zero]
- Computing z given x and y (important in hidden-surface algorithms—see Chapter 13)
- Calculating intersections of one surface with another.

An alternative representation of Eq. (9.57) is

$$P^{\mathrm{T}} \cdot Q \cdot P = 0, \qquad (9.58)$$

$$\text{with} \quad Q = \begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \qquad (9.59)$$

The surface represented by Q can be easily translated and scaled. Given a 4×4 transformation matrix M of the form developed in Chapter 5, the transformed quadric surface Q' is given by

$$Q' = (M^{-1})^{\mathrm{T}} \cdot Q \cdot M^{-1}. \qquad (9.60)$$

The normal to the implicit surface defined by $f(x, y, z) = 0$ is the vector $[df/dx \;\; df/dy \;\; df/dz]$. This surface normal is much easier to evaluate than is the surface normal to a bicubic surface discussed in Section 9.3.4.

9.5 SPECIALIZED MODELING TECHNIQUES

In this chapter we have concentrated on geometric models; we will as well in Chapter 10. In a world made entirely of simple geometric objects, these models would suffice. But many natural phenomena are not efficiently represented by geometric models, at least not on a large scale. Fog, for example, is made up of tiny drops of water, but using a model in which each drop must be individually placed is out of the question. Furthermore, this water-drop model does not accurately represent our perception of fog: We see fog as a blur in the air in front of us, not as millions of drops. Our visual perception of fog is based on how fog alters the light reaching our eyes, not on the shape or placement of the individual drops. Thus, to model the perceptual effect of fog efficiently, we need a different model. In the same way, the shape of a leaf of a tree may be modeled with polygons and its stem may be modeled with a spline tube, but to place explicitly every limb, branch, twig, and leaf of a tree would be impossibly time consuming and cumbersome.

You will find a comprehensive discussion of advanced modeling techniques in Chapter 20 of [FOLE90]; here we will discuss two specialized methods, which are surprisingly easy to implement and which produce startlingly realistic images.

9.5.1 Fractal Models

Fractals have recently attracted much attention [VOSS87; MAND82; PEIT86]. The images resulting from them are spectacular, and many different approaches to generating fractals have been developed. The term **fractal** has been generalized by the computer graphics community to include objects outside Mandelbrot's original definition. It has come to mean anything which has a substantial measure of exact or statistical self-similarity, and that is how we use it here, although its precise mathematical definition requires statistical self-similarity at all resolutions. Thus, only fractals generated by infinitely recursive processes are true fractal objects. On the other hand, those generated by finite processes may exhibit no visible change in detail after some stage, so they are adequate approximations of the ideal. What we mean by **self-similarity** is best illustrated by an example, the von Koch snowflake. Starting with a line segment with a bump on it, as shown in Fig. 9.27, we

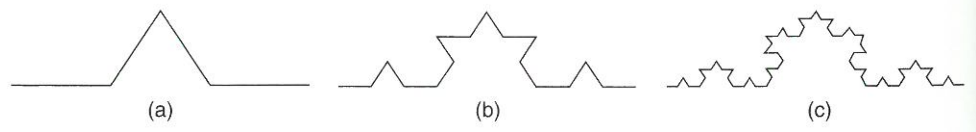


Figure 9.27 Construction of the von Koch snowflake: each segment in (a) is replaced by an exact copy of the entire figure, shrunk by a factor of 3. The same process is applied to the segments in (b) to generate those in (c).

replace each segment of the line by a figure exactly like the original line. This process is repeated: Each segment in part (b) of the figure is replaced by a shape exactly like the entire figure. (It makes no difference whether the replacement is by the shape shown in part (a) or by the shape shown in part (b); if the one in part (a) is used, the result after 2^n steps is the same as the result after n steps if each segment of the current figure is replaced by the entire current figure at each stage.) If this process is repeated infinitely many times, the result is said to be **self-similar**: The entire object is similar (i.e., can be translated, rotated, and scaled) to a subportion of itself.

Associated with this notion of self-similarity is the notion of **fractal dimension**. To define fractal dimension, we shall examine some properties of objects whose dimension we know. A line segment is 1D; if we divide a line into N equal parts, the parts each look like the original line scaled down by a factor of $N = N^{1/1}$. A square is 2D: if we divide it into N parts, each part looks like the original scaled down by a factor of $\sqrt{N} = N^{1/2}$. (For example, a square divides nicely into nine subsquares; each one looks like the original scaled by a factor of $\frac{1}{3}$.) What about the von Koch snowflake? When it is divided into four pieces (the pieces associated with the original four segments in Fig. 9.27(a), each resulting piece looks like the original scaled down by a factor of 3. We would like to say it has a dimension d, where $4^{1/d} = 3$. The value of d must be log(4)/log(3) = 1.26.... This is the definition of fractal dimension.

The most famous two fractal objects deserve mention here: the Julia–Fatou set and the Mandelbrot set. These objects are generated from the study of the rule $x \rightarrow x^2 + c$ (and many other rules as well—this is the simplest and best known). Here x is a **complex number**,[1] $x = a + bi$. If a complex number has modulus < 1, then squaring it repeatedly makes it go toward zero. If it has a modulus > 1, repeated squaring makes it grow larger and larger. Numbers with modulus 1 still have modulus 1 after repeated squarings. Thus, some complex numbers "fall toward zero" when they are repeatedly squared, some "fall toward infinity," and some do neither—the last group forms the boundary between the numbers attracted to zero and those attracted to infinity.

Suppose we repeatedly apply the mapping $x \rightarrow x^2 + c$ to each complex number x for some nonzero value of c, such as $c = -0.12375 + 0.056805i$; some complex numbers will be attracted to infinity, some will be attracted to finite numbers, and some will go toward neither. Drawing the set of points that go toward neither, we get the Julia–Fatou set shown in Fig. 9.28(a).

Notice that the region in Fig. 9.28(b) is not as well connected as is that in part (a) of the figure. In part (b), some points fall toward each of the three black dots shown, some go to infinity, and some do neither. These last points are the ones drawn as the outline of the shape in part (b). The shape of the Julia–Fatou set evidently depends on the value of the number c. If we compute the Julia sets for all

[1] If you are unfamiliar with complex numbers, it suffices to treat i as a special symbol and merely to know the definitions of addition and multiplication of complex numbers. If $z = c + di$ is a second complex number, then $x + z$ is defined to be $(a + c) + (b + d)i$, and xz is defined to be $(ac - bd) + (ad + bc)i$. We can represent complex numbers as points in the plane by identifying the point (a, b) with the complex number $(a + bi)$. The *modulus* of the number $a + bi$ is the real number $(a^2 + b^2)^{1/2}$, which gives a measure of the "size" of the complex number.

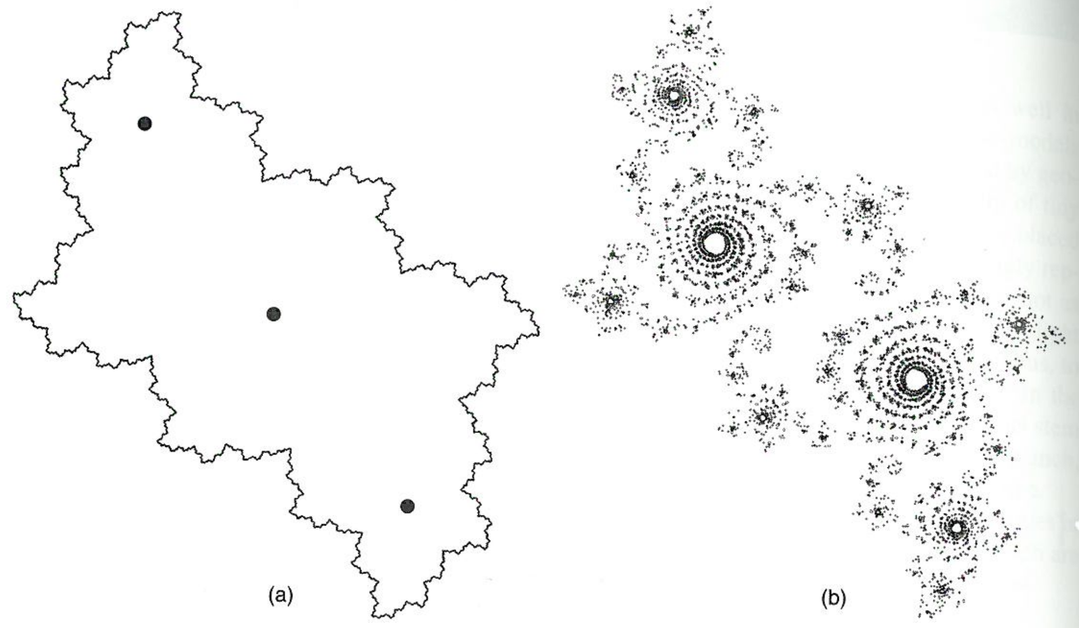


Figure 9.28 The Julia–Fatou set. (a) $c = -0.12375 + 0.056805i$. (b) $c = -0.012 + 0.74i$.

possible values of c and color the point c black when the Julia–Fatou set is connected (i.e, is made of one piece, not broken into disjoint "islands") and white when the set is not connected, we get the object shown in Fig. 9.29, which is known as the **Mandelbrot set**. Note that the Mandelbrot set is self-similar in that, around the edge of the large disk in the set, there are several smaller sets, each looking a great deal like the large one scaled down.

Fortunately, there is an easier way to generate approximations of the Mandelbrot set: For each value of c, take the complex number $0 = 0 + 0i$ and apply the process $x \rightarrow x^2 + c$ to it some finite number of times (perhaps 1000). If after this many iterations it is outside the disk defined by modulus < 100, then we color c white; otherwise, we color it black. As the number of iterations and the radius of the disk are increased, the resulting picture becomes a better approximation of the set. Peitgen and Richter [PEIT86] give explicit directions for generating many spectacular images of Mandelbrot and Julia–Fatou sets.

These results are extremely suggestive for modeling natural forms, since many natural objects seem to exhibit striking self-similarity. Mountains have peaks and smaller peaks and rocks and gravel, which all look similar; trees have limbs and branches and twigs, which all look similar; coastlines have bays and inlets and estuaries and rivulets and drainage ditches, which all look similar. Hence, modeling self-similarity at some scale seems to be a way to generate appealing-looking models of natural phenomena. The scale at which the self-similarity breaks down

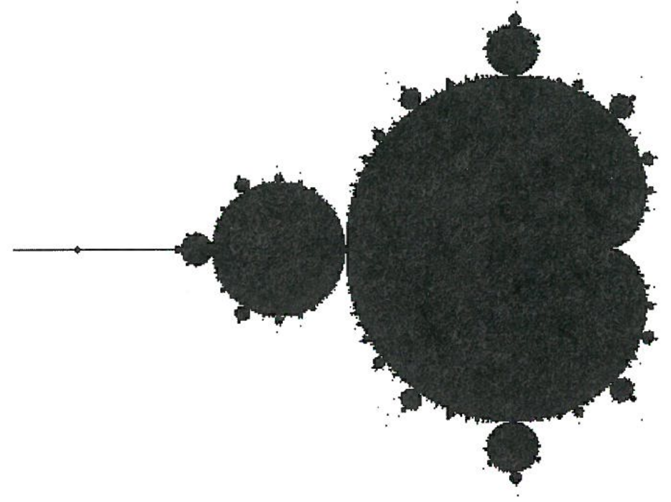


Figure 9.29 The Mandelbrot set. Each point c in the complex plane is colored black if the Julia set for the process $x \rightarrow x^2 + c$ is connected.

is not particularly important here, since the intent is modeling rather than mathematics. Thus, when an object has been generated recursively through enough steps that all further changes happen at well below pixel resolution, there is no need to continue.

Fournier, Fussell, and Carpenter [FOUR82] developed a mechanism for generating a class of fractal mountains based on recursive subdivision. It is easiest to explain in 1D. Suppose we start with a line segment lying on the x axis, as shown in Fig. 9.30(a). If we now subdivide the line into two halves and then move the midpoint some distance in the y direction, we get the shape shown in Fig. 9.30(b). To continue subdividing each segment, we compute a new value for the midpoint of the segment from (x_i, y_i) to (x_{i+1}, y_{i+1}) as follows: $x_{\text{new}} = \frac{1}{2}(x_i + x_{i+1})$, $y_{\text{new}} = \frac{1}{2}(y_i + y_{i+1}) + P(x_{i+1} - x_i)\, R(x_{\text{new}})$, where $P()$ is a function determining the extent of the perturbation in terms of the size of the line being perturbed, and $R()$ is a random number[2] between 0 and 1 selected on the basis of x_{new} (see Fig. 9.30c). If $P(s) = s$, then the first point cannot be displaced by more than 1, each of the next two points (which are at most at height $\frac{1}{2}$ already) cannot be displaced by more than $\frac{1}{2}$, and so on. Hence, all the resulting points fit in the unit square. For $P(s) = s^a$, the shape of the result depends on the value of a; smaller values of a yield larger perturbations, and vice versa. Of course, other functions, such as $P(s) = 2^{-s}$, can be used as well.

[2] $R()$ is actually a *random variable*, a function taking real numbers and producing randomly distributed numbers between 0 and 1. If this is implemented by a pseudorandom-number generator, it has the advantage that the fractals are repeatable: We can generate them again by supplying the same seed to the pseudorandom-number generator.

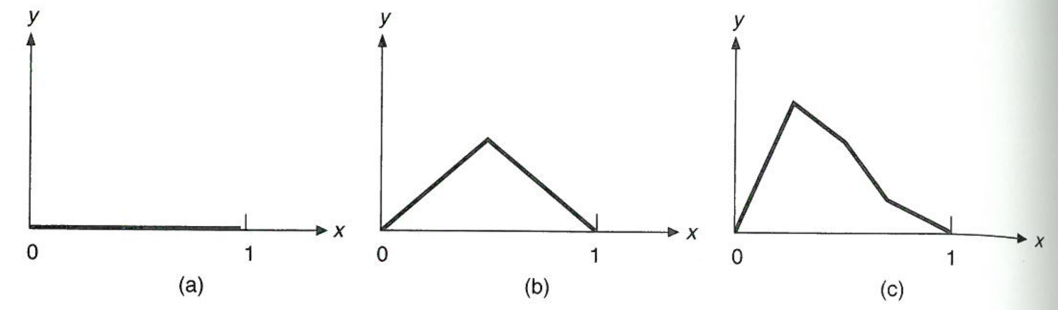


Figure 9.30 A line segment on the *x* axis. (b) The midpoint of the line has been translated in the *y* direction by a random amount. (c) The result of one further iteration.

Fournier, Fussell, and Carpenter use this process to modify 2D shapes in the following fashion. They start with a triangle, mark the midpoint of each edge, and connect the three midpoints, as shown in Fig. 9.31(a). The *y* coordinate of each midpoint is then modified in the manner we have described, so that the resulting set of four triangles looks like Fig. 9.31(b). This process, when iterated, produces quite realistic-looking mountains, as shown in Color Plate 11 (although, in an overhead view, one perceives a very regular polygonal structure).

Notice that we can start with an arrangement of triangles that have a certain shape, then apply this process to generate the finer detail. This ability is particularly important in some modeling applications, in which the layout of objects in a scene may be stochastic at a low level but ordered at a high level: The foliage in an ornamental garden may be generated by a stochastic mechanism, but its arrangement in the garden must follow strict rules. On the other hand, the fact that the

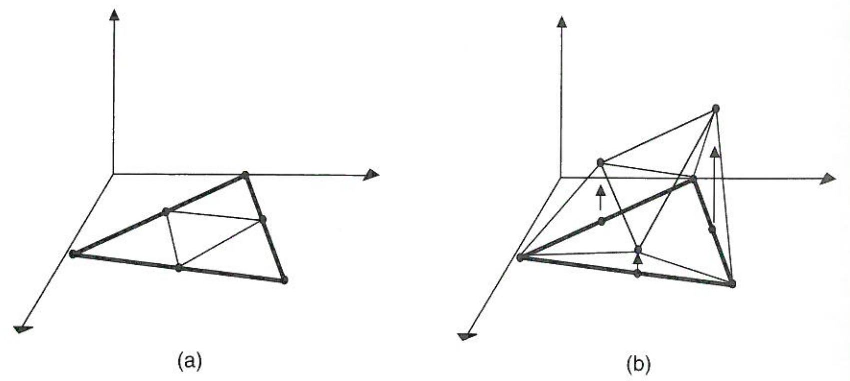


Figure 9.31 (a) The subdivision of a triangle into four smaller triangles. The midpoints of the original triangle are perturbed in the *y* direction to yield the shape in (b).

high-level structure of the initial triangle arrangement persists in the iterated subdivisions may be inappropriate in some applications (in particular, the fractal so generated does not have all the statistical self-similarities present in fractals based on Brownian motion [MAND82]). Also, since the position of any vertex is adjusted only once and is stationary thereafter, creases tend to develop in the surface along the edges between the original triangles, and these may appear unnatural.

Rendering fractals can be difficult. If the fractals are rendered into a *z*-buffer, displaying the entire object takes a long time because of the huge number of polygons involved. In scan-line rendering, it is expensive to sort all the polygons so that only those intersecting the scan line are considered. But ray tracing fractals is extremely difficult, since each ray must be checked for intersection with each of the possibly millions of polygons involved. Kajiya [KAJI83] gave a method for ray tracing fractal objects of the class described in [FOUR82], and Bouville [BOUV85] improves this algorithm by finding a better bounding volume for the objects.

9.5.2 Grammar-Based Models

Smith [SMIT84] presents a method for describing the structure of certain plants, originally developed by Lindenmayer [LIND68], by using parallel graph grammar languages (**L-grammars**), which Smith called **graftals**. These languages are described by a grammar consisting of a collection of productions, all of which are applied at once. Lindenmayer extended the languages to include brackets, so the alphabet contained the two special symbols, "[" and "]." A typical example is the grammar with alphabet {A, B, [,]} and two production rules:

1. A → AA
2. B → A[B]AA[B]

Starting from the axiom A, the first few generations are A, AA, AAAA, and so on; starting from the axiom B, the first few generations are

0. B
1. A[B]AA[B]
2. AA[A[B]AA[B]]AAAA[A[B]AA[B]]

and so on. If we say that a word in the language represents a sequence of segments in a graph structure and that bracketed portions represent portions that branch from the symbol preceding them, then the figures associated with these three levels are as shown in Fig. 9.32.

This set of pictures has a pleasing branching structure, but a somewhat more balanced tree would be appealing. If we add the parentheses symbols, "(" and ")," to the language and alter the second production to be A[B]AA(B), then the second generation becomes

2. AA[A[B]AA(B)]AAAA(A[B]AA(B))

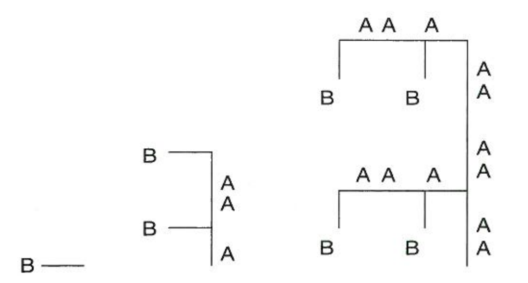


Figure 9.32 Tree representations of the first three words of the language. All branches are drawn to the left of the current main axis.

If we say that square brackets denote a left branch and parentheses denote a right branch, then the associated pictures are as shown in Fig. 9.33. By progressing to later generations in such a language, we get graph structures representing extremely complex patterns. These graph structures have a sort of self-similarity, in that the pattern described by the nth-generation word is contained (repeatedly, in this case) in the $(n + 1)$th-generation word.

Generating an object from such a word is a process separate from that of generating the word itself. Here, the segments of the tree have been drawn at successively smaller lengths, the branching angles have all been 45°, and the branches go to the left or to the right. Choosing varying branching angles for different depth branches, and varying thicknesses for the lines (or even cylinders) representing the segments, gives different results; drawing a "flower" or "leaf" at each terminal node of the tree further enhances the picture. The grammar itself has no inherent geometric content, so using a grammar-based model requires both a grammar and a geometric interpretation of the language.

This sort of enhancement of the languages and the interpretation of words in the language (i.e., pictures generated from words) has been carried out by several

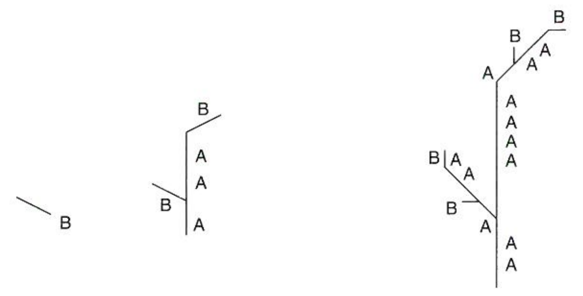


Figure 9.33 Tree representations of the first three words, but in the language with two-sided branching. We have made each segment of the tree shorter as we progress into further generations.

researchers [REFF88; PRUS88]. The grammars have been enriched to allow us to keep track of the "age" of a letter in a word, so that the old and young letters are transformed differently (this recording of ages can be done with rules of the form A → B, B → C, C → D,..., Q → QG[Q], so that no interesting transitions occur until the plant has "aged"). Much of the work has been concentrated on making grammars that accurately represent the biology of plants during development.

At some point, however, a grammar becomes unwieldy as a descriptor for plants: Too many additional features are added to it or to the interpretation of a word in it. In Reffye's model [REFF88], the simulation of the growth of a plant is controlled by a small collection of parameters that are described in biological terms and that can be cast in an algorithm. The productions of the grammar are applied probabilistically, rather than deterministically.

In this model, we start as before with a single stem. At the tip of this stem is a **bud**, which can undergo one of several transitions: it may die, it may flower and die, it may sleep for some period of time, or it may become an **internode**, a segment of the plant between buds. The process of becoming an internode has three stages: the original bud may generate one or more **axillary buds** (buds on one side of the joint between internodes), a process that is called **ramification**; the internode is added; and the end of the new internode becomes an **apical bud** (a bud at the very end of a sequence of internodes). Figure 9.34(a) shows examples of the transition from bud to internode.

Each of the buds in the resulting object can then undergo similar transitions. If we say the initial segment of the tree is of **order 1**, we can define the order of all other internodes inductively: Internodes generated from the apical bud of an order-i internode are also of order-i; those generated from axillary buds of an order-i internode are of order $(i + 1)$. Thus, the entire trunk of a tree is order 1, the limbs are order 2, the branches on those limbs are order 3, and so on. Figure 9.34(b) shows a more complicated plant and the orders of various internodes in the plant.

Converting this description into an actual image of a tree requires a model for the shapes of its various components: an order-1 internode may be a large tapered cylinder, and an order-7 internode may be a small green line, for example. The sole requirement is that there must be a leaf at each axillary node (although the leaf may fall at some time).

Finally, to simulate the growth of a plant in this model, then, we need the following biological information: the current age of the model, the growth rate of each order of internode, the number of axillary buds at the start of each internode (as a function of the order of the internode), and the probabilities of death, pause, ramification, and reiteration as functions of age, dimension, and order. We also need certain geometric information: the shape of each internode (as a function of order and age), the branching angles for each order and age, and the tropism of each axis (whether each sequence of order-i internodes is a straight line, or curves toward the horizontal or vertical). To draw an image of the plant, we need still more information: the color and texture of each of the entities to be drawn—internodes of various orders, leaves of various ages, and flowers of different ages. Very convincing tree models can be produced by grammar-based models; see Color Plate 12.

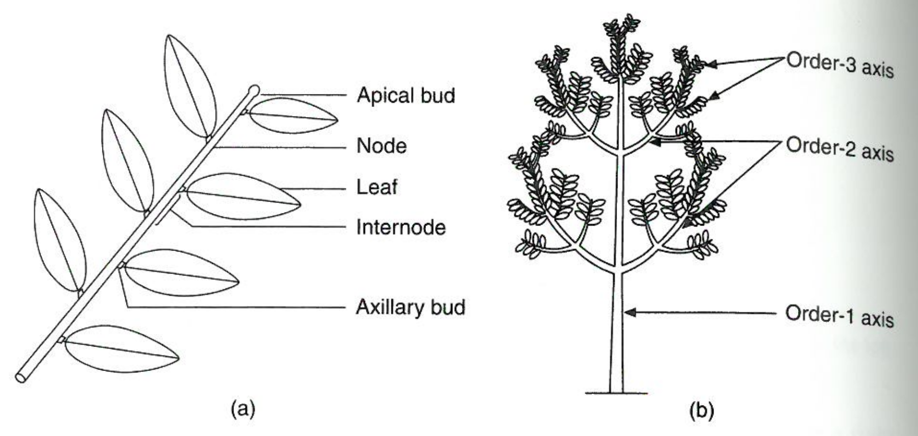


Figure 9.34 Examples of plant growth. (a) The bud at the tip of a segment of the plant can become an internode; in so doing, it creates a new bud (the **axillary bud**), a new segment (**the internode**), and a new bud at the tip (the **apical bud**). (b) A more complex plant, with orders attached to the various internodes.

SUMMARY

This chapter has only touched on important ideas concerning curve and surface representation, but it has given sufficient information so that you can implement interactive systems using these representations. Theoretical treatments of the material can be found in texts such as [BART87; DEBO78; FAUX79; MORT85].

Polygon meshes, which are piecewise linear, are well suited for representing flat-faced objects, but are seldom satisfactory for curve-faced objects. Piecewise continuous parametric cubic curves and bicubic surfaces are widely used in computer graphics and CAD to represent curve-faced objects because they

- Permit multiple values for a single value of x or y
- Represent infinite slopes
- Provide local control, such that changing a control point affects only a local piece of the curve
- Can be made either to interpolate or to approximate control points, depending on application requirements
- Are computationally efficient
- Are easily transformed by transformation of control points.

Although we have discussed only cubics, higher- and lower-order surfaces also can be used. The texts mentioned previously generally develop parametric curves and surfaces for the general case of order n.

We have also discussed briefly some techniques for modeling natural phenomena; in particular, fractal and grammar-based approaches.

Exercises

9.1 Find the geometry matrix and basis matrix for the parametric representation of a straight line given in Eq. (9.11).

9.2 Show that, for a 2D curve $[x(t) \quad y(t)]^T$, G^1 continuity means that the geometric slope dy/dx is equal at the join points between segments.

9.3 Let $\gamma(t) = (t, t^2)$ for $0 \le t \le 1$, and let $\eta(t) = (2t + 1, t^3 + 4t + 1)$ for $0 \le t \le 1$. Notice that $\gamma(1) = (1, 1) = \eta(0)$, so γ and η join with C^0 continuity.

a. Plot $\eta(t)$ and $\gamma(t)$ for $0 \le t \le 1$.

b. Determine whether $\eta(t)$ and $\gamma(t)$ meet with C^1 continuity at the join point. (You will need to compute the vectors $\frac{d\gamma}{dt}(1)$ and $\frac{d\eta}{dt}(0)$ to check your answer.)

c. Determine whether $\eta(t)$ and $\gamma(t)$ meet with G^1 continuity at the join point.(You will need to check ratios from part(b) to check your answer.)

9.4 Consider the paths

$$\gamma(t) = (t^2 - 2t + 1, t^3 - 2t^2 + t) \quad \text{and} \quad \eta(t) = (t^2 + 1, t^3),$$

both defined on the interval $0 \le t \le 1$. The curves join, since $\gamma(1) = (1, 0) = \eta(0)$. Show that they meet with C^1 continuity, but not with G^1 continuity. Plot both curves as functions of t to demonstrate exactly why this behavior occurs.

9.5 Show that the two curves $\gamma(t) = (t^2 - 2t, t)$ and $\eta(t) = (t^2 + 1, t + 1)$ are both C^1 and G^1 continuous where they join at $\gamma(1) = \eta(0)$.

9.6 Analyze the effect on a B-spline of having in sequence four collinear control points.

9.7 Write a program to accept an arbitrary geometry matrix, basis matrix, and list of control points, and to draw the corresponding curve.

9.8 Find the conditions under which two joined Hermite curves have C^1 continuity.

9.9 Suppose that the equations relating the Hermite geometry to the Bézier geometry are of the form $R_1 = \beta(P_2 - P_1)$, $R_4 = \beta(P_4 - P_3)$. Consider the four equally spaced Bézier control points $P_1 = (0, 0)$, $P_2 = (1, 0)$, $P_3 = (2, 0)$, $P_4 = (3, 0)$. Show that, for the parametric curve $Q(t)$ to have constant velocity from P_1 to P_4, the coefficient β must be equal to 3.

9.10 Explain why Eq. (9.35) for uniform B-splines is written as $Q_i(t - t_i)$, whereas Eq. (9.37) for nonuniform B-splines is written as $Q_i(t)$.

9.11 Given a 2D nonuniform B-spline and an (x, y) value on the curve, write a program to find the corresponding value of t. Be sure to consider the possibility that, for a given value of x (or y), there may be multiple values of y (or x).

9.12 Apply the methodology used to derive Eqs. (9.49) and (9.50) for Hermite surfaces to derive Eq. (9.51) for Bézier surfaces.

9.13 Let $t_0 = 0$, $t_1 = 1$, $t_2 = 3$, $t_3 = 4$, $t_4 = 5$. Using these values, compute $B_{0,4}$ and each of the functions used in its definition. Then plot these functions on the interval $-3 \le t \le 8$.

9.14 Develop a program, similar to Example 9.2, for displaying Bézier surface patches using the framework of Prog. 9.2. Your program should offer the option of displaying the control points for a specified patch, so that the user can select any one of them (using a locator) and move it to a new location. The patch should then be redrawn to reflect the new geometric constraint. Since the locator input is 2D, how will you associate it with a 3D control point? You can specify your own patch geometry or use existing data, such as that for the teapot in Fig. 9.1. Complete data for the teapot are included in [CROW87].

10 Solid Modeling

The representations discussed in Chapter 9 allow us to describe curves and surfaces in 2D and 3D. Just as a set of 2D lines and curves does not need to describe the boundary of a closed area, a collection of 3D planes and surfaces does not necessarily bound a closed volume. In many applications, however, it is important to distinguish among the inside, outside, and surface of a 3D object and to be able to compute properties of the object that depend on this distinction. In CAD/CAM, for example, if a solid object can be modeled in a way that adequately captures its geometry, then a variety of useful operations can be performed before the object is manufactured. We may wish to determine whether two objects interfere with each other, for example, whether a robot arm will bump into objects in its environment, or whether a cutting tool will cut only the material it is intended to remove. In simulating physical mechanisms, such as a gear train, it may be important to compute properties such as volume and center of mass. Finite-element analysis is applied to solid models to determine response to factors such as stress and temperature through finite-element modeling. A satisfactory representation for a solid object may even make it possible to generate instructions automatically for computer-controlled machine tools to create that object or to rapidly prototype it using a technique such as stereolithography, a process which uses a laser beam to form a hardened object out of a bath of molten plastic. In addition, some graphical techniques, such as modeling refractive transparency, depend on being able to determine where a beam of light enters and exits a solid object. These applications are all examples of **solid modeling**. The need to model objects as solids has resulted in the development of a variety of specialized ways to represent them. This chapter provides a brief introduction to these representations.