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

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

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).

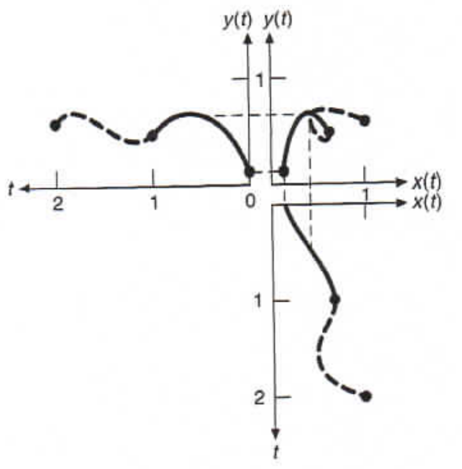


Figure 9.7 Two joined 2D parametric curve segments and their defining polynomials. The dashed lines between the (x, y) plot and the x(t) and y(t) plots show the correspondence between the points on the (x, y) curve and the defining cubic polynomials. The x(t) and y(t) plots for the second segment have 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

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

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 $k > 0$ [BARS

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

ashed lines
veen the
ılots for the
how the

d their poly-
sily multiple
s single val-
urves repre-

e parametric
have

1 0]^T

(9.8)

tinuity. If the
tangent vec-
In computer-
n is required.
ual at the join
n, it is neces-
0 [BARS88].
(that is, their
the curve has
and is said to
rough the nth

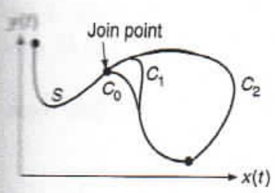


Figure 9.8
Curve segment S joined to segments C_0, C_1, and C_2 with the 0, 1, and 2 degrees of parametric continuity, respectively. The visual distinction between C_1 and C_2 is slight at the join, but obvious away from the join.

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.

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.

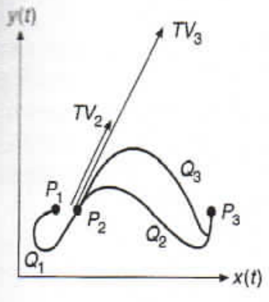


Figure 9.9
Curve segments Q_1, Q_2, and Q_3 join at the point P_2 and are identical except for their tangent vectors at P_2. Q_1 and Q_2 have equal tangent vectors, and hence are both G^1 and C^1 continuous at P_2. Q_1 and Q_3 have tangent vectors in the same direction, but Q_3 has twice the magnitude, 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 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

$$Q(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \underset{1\times 4}{[G_1 \quad G_2 \quad G_3 \quad G_4]} \underset{4\times 4}{\begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix}} \underset{4\times 1}{\begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}}. \quad (9.9)$$

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

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

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:

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

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

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

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

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

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

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

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

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

$$x'(0) = R_{1_x} = G_{H_x} \cdot M_H \; [0 \;\; 0 \;\; 1 \;\; 0]^T, \tag{9.16}$$

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

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

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

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):

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

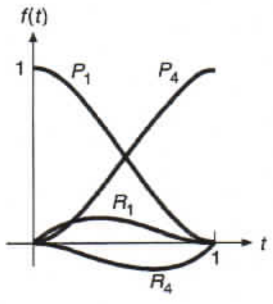


Figure 9.10
The Hermite blending functions, labeled by the elements of the geometry vector that they weight.

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

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

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:

$$\begin{aligned} Q(t) &= G_H \cdot M_H \cdot T = G_H \cdot B_H \\ &= (2t^3 - 3t^2 + 1)P_1 + (-2t^3 + 3t^2)P_4 + (t^3 - 2t^2 + t)R_1 + (t^3 - t^2)R_4. \end{aligned} \tag{9.21}$$

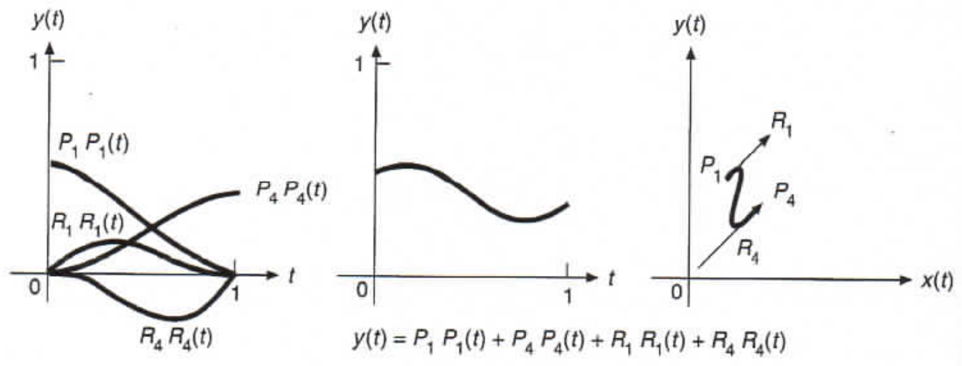
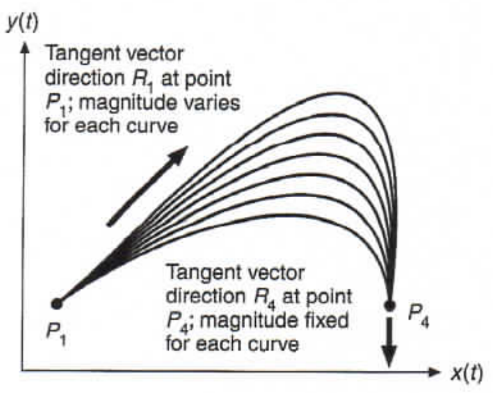


Figure 9.11 A Hermite curve showing the four elements of the geometry vector weighted by the blending functions (leftmost four curves), their sum $y(t)$, and the 2D curve itself (far right). $x(t)$ is defined by a similar weighted sum.

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.

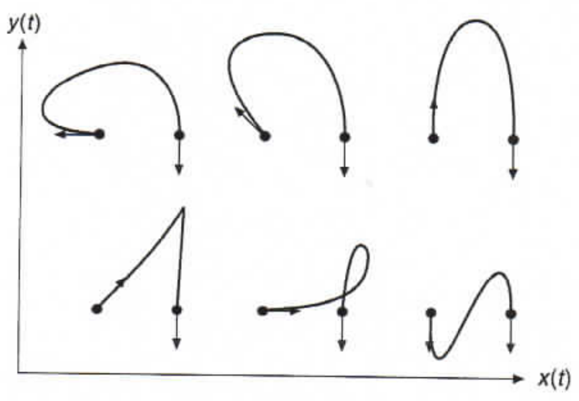


Figure 9.13 Family of Hermite parametric cubic curves. Only the direction of the tangent vector at the left starting point varies; all tangent vectors have the same magnitude. A smaller magnitude would eliminate the loop in the one curve.

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,

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

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

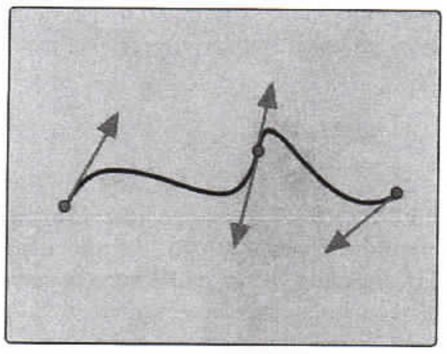


Figure 9.14 Two Hermite cubic curve segments displayed with controls to facilitate interactive manipulation. The user can reposition the endpoints by dragging the dots, and can change the tangent vectors by dragging the arrowheads. The tangent vectors at the join point are constrained to be collinear (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

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

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

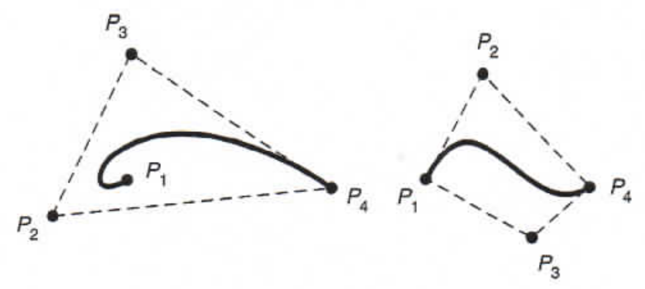


Figure 9.15 Two Bézier curves and their control points. Notice that the convex hulls (the convex polygon formed by the control points), shown as dashed lines, do not need to touch all four control points.

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

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 $G_H = M_{HB} \cdot G_B$, and define $M_B = M_H \cdot M_{HB}$:

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

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:

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

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

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

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.

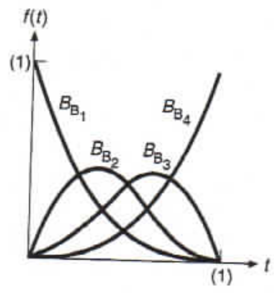


Figure 9.16 The Bernstein polynomials, which are the weighting functions for Bézier curves. At $t = 0$, only B_{B_1} is nonzero, so the curve interpolates P_1; similarly, at $t = 1$, only B_{B_4} is nonzero, and the curve interpolates P_4.

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 acurve segment to determine its visibility, we first apply a polygonal clip algorithm, for example, the Sutherland–Hodgman

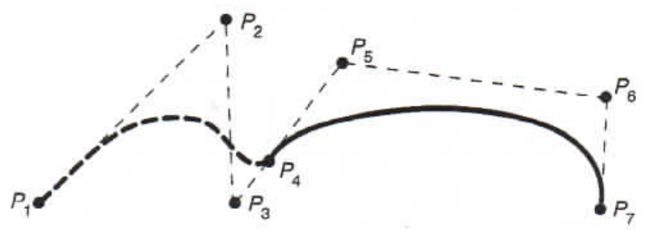


Figure 9.17 Two Bézier curves joined at P_4. Points P_3, P_4, and P_5 are collinear.

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.

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.

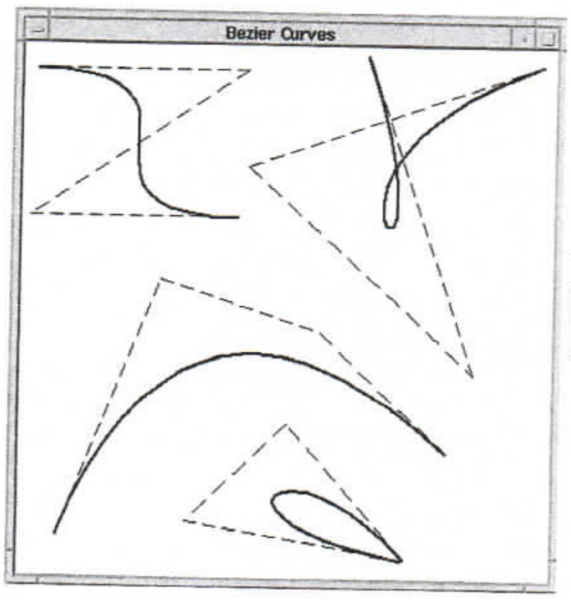
We have arbitrarily specified the window size and number of steps used to approximate the curve as 400 and 20, respectively. There are many possible ways to implement the interactive part of the program; we have elected to use a combination of locator and keyboard devices. The right locator button is 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 $P_0, P_1, \ldots P_m$, $m \geq 3$, with a curve consisting of $m - 2$ cubic polynomial curve segments $Q_3, Q_4, \ldots, Q_m$. 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