CHAPTER

Interpolating Values

3

This chapter presents the foundation upon which much of computer animation is built—that of interpolating values. Changing values over time that somehow affect the visuals produced is, essentially, animation. Interpolation implies that the boundary values have been specified and all that remains is to fill in the intermediate details. But even filling in the values requires skill and knowledge. This chapter starts by presenting the basics of various types of interpolating and approximating curves followed by techniques to control the motion of a point along a curve. The interpolation of orientation is then considered. The chapter concludes with a section on working with paths.

3.1 Interpolation

The foundation of most computer animation is the interpolation of values. One of the simplest examples of animation is the interpolation of the position of a point in space. But even to do this correctly is nontrivial and requires some discussion of several issues: the appropriate interpolating function, the parameterization of the function based on distance traveled, and maintaining the desired control of the interpolated position over time.

Most of this discussion will be in terms of interpolating spatial values. The reader should keep in mind that any changeable value involved in the animation (and display) process such as an object's transparency, the camera's focal length, or the color of a light source could be subjected to interpolation over time.

Often, an animator has a list of values associated with a given parameter at specific frames (called *key frames* or *extremes*) of the animation. The question to be answered is how best to generate the values of the parameter for frames between the key frames. The parameter to be interpolated may be a coordinate of the position of an object, a joint angle of an appendage of a robot, the transparency attribute of an object, or any other parameter used in the manipulation and display of computer graphics elements. Whatever the case, values for the parameter of interest must be generated for all of the frames between the key frames.

For example, if the animator wants the position of an object to be $(-5, 0, 0)$ at frame 22 and the position to be $(5, 0, 0)$ at frame 67, then values for the position need to be generated for frames 23 to 66. Linear interpolation could be used. But what if the object should appear to be stopped at frame 22 and needs to accelerate from that position, reach a maximum speed by frame 34, start to decelerate at frame 50, and come to a stop by frame 67? Or perhaps instead of stopping at frame 67, the object should continue to position (5, 10, 0) and arrive there at frame 80 by following a nice curved path. The next

several sections address these issues of generating points along a path defined by control points and distributing the points along the path according to timing considerations.

3.1.1 The appropriate function

Appendix B.5 contains a discussion of various specific interpolation techniques. In this section, the discussion covers general issues that determine how to choose the most appropriate interpolation technique and, once it is chosen, how to apply it in the production of an animated sequence.

The following issues need to be considered in order to choose the most appropriate interpolation technique: interpolation versus approximation, complexity, continuity, and global versus local control.

Interpolation versus approximation

Given a set of points to describe a curve, one of the first decisions an animator must make is whether the given values represent actual positions that the curve should pass through (*interpolation*) or whether they are meant merely to control the shape of the curve and do not represent actual positions that the curve will intersect (*approximation*) (see Figure 3.1). This distinction is usually dependent on whether the data points are sample points of a desired curve or whether they are being used to design a new curve. In the former case, the desired curve is assumed to be constrained to travel through the sample points, which is, of course, the definition of an *interpolating spline*.[1] In the latter case, an approximating spline can be used as the animator quickly gets a feel for how repositioning the control points influences the shape of the curve.

Commonly used interpolating functions are the Hermite formulation and the Catmull-Rom spline. The Hermite formulation requires tangent information at the endpoints, whereas Catmull-Rom uses only positions the curve should pass through. Parabolic blending, similar to Catmull-Rom, is another useful interpolating function that requires only positional information. Functions that approximate some or all of the control information include Bezier and B-spline curves. See Appendix B.5 for a more detailed discussion of these functions.

An interpolating spline in which the spline passes through the interior control points

An approximating spline in which only the endpoints are interpolated; the interior control points are used only to design the curve

FIGURE 3.1

Comparing interpolation and approximating splines.

[1]The term *spline* comes from flexible strips used by shipbuilders and draftsmen to draw smooth curves. In computer graphics, it generally refers to a wide class of interpolating or smoothing functions.

Complexity

The complexity of the underlying interpolation equation is of concern because this translates into computational efficiency. The simpler the underlying equations of the interpolating function, the faster its evaluation. In practice, polynomials are easy to compute, and piecewise cubic polynomials are the lowest degree polynomials that provide sufficient smoothness while still allowing enough flexibility to satisfy other constraints such as beginning and ending positions and tangents. A polynomial whose degree is lower than cubic does not provide for a point of inflection between two endpoints; therefore, it might not fit smoothly to certain data points. Using a polynomial whose degree is higher than cubic typically does not provide any significant advantages and is more costly to evaluate.

Continuity

The smoothness in the resulting curve is a primary consideration. Mathematically, smoothness is determined by how many of the derivatives of the curve equation are continuous. *Zero-order continuity* refers to the continuity of values of the curve itself. Does the curve make any discontinuous jumps in its values? If a small change in the value of the parameter always results in a small change in the value of the function, then the curve has zero-order, or positional, continuity. If the same can be said of the first derivative of the function (the instantaneous change in values of the curve), then the function has *first-order*, or *tangential*, *continuity*. *Second-order continuity* refers to continuous curvature or instantaneous change of the tangent vector (see Figure 3.2). In some geometric design environments, second-order continuity of curves and surfaces may be needed, but often in animation applications, first-order continuity suffices for spatial curves. As explained later in this chapter, when dealing with time-distance curves, second-order continuity can be important.

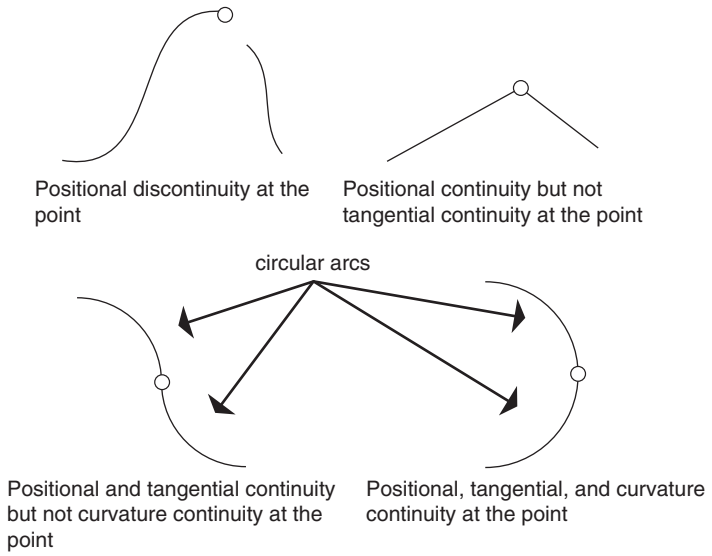


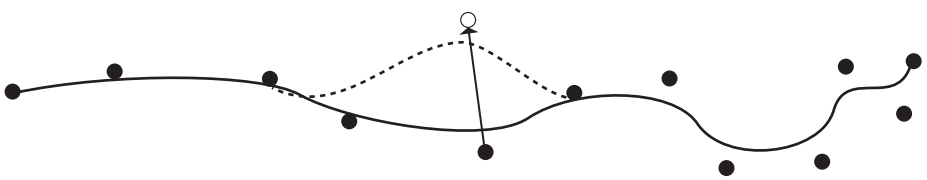
FIGURE 3.2

Continuity (at the point indicated by the small circle).

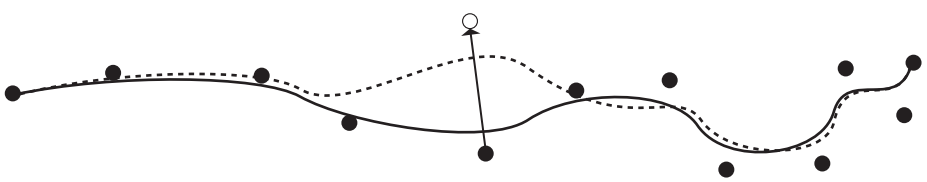
In most applications, a curve interpolating or approximating more than a few points is defined piecewise; the curve is defined by a sequence of segments where each segment is defined by a single, vector-valued parametric function and shares its endpoints with the functions of adjacent segments. For the many types of curve segments considered here, the segment functions are cubic polynomials. The issue then becomes, not the continuity within a segment (which in the case of polynomial functions is of infinite order), but the continuity enforced at the junction between adjacent segments. Hermite, Catmull-Rom, parabolic blending, and cubic Bezier curves (see Appendix B.5) can all produce first-order continuity between curve segments. There is a form of compound Hermite curves that produces second-order continuity between segments at the expense of local control (see the following section). A cubic B-spline is second-order continuous everywhere. All of these curves provide sufficient continuity for many animation applications. Appendix B.5 discusses continuity in more mathematical terms, with topics including the difference between *parametric continuity* and *geometric continuity*.

Global versus local control

When designing a curve, a user often repositions one or just a few of the points that control the shape of the curve in order to tweak just part of the curve. It is usually considered an advantage if a change in a single control point has an effect on a limited region of the curve as opposed to affecting the entire curve. A formulation in which control points have a limited effect on the curve is referred to as providing *local control*. If repositioning one control point redefines the entire curve, however slightly, then the formulation provides only *global control*. The well-known Lagrange interpolation [5] is an example of an interpolating polynomial with global control. Figure 3.3 illustrates the difference between local control and global control. Local control is almost always viewed as being the more desirable of the two. Almost all of the composite curves provide local control: parabolic blending, Catmull-Rom splines, composite cubic Bezier, and cubic B-spline. The form of Hermite curves that enforces second-order continuity at the segment junctions does so at the expense of local control. Higher order continuity Bezier and B-spline curves have less localized control than their cubic forms.



Local control: moving one control point only changes the curve over a finite bounded region



Global control: moving one control point changes the entire curve; distant sections may change only slightly

FIGURE 3.3

Comparing local and global effect of moving a control point.

3.1.2 Summary

There are many formulations that can be used to interpolate values. The specific formulation chosen depends on the desired continuity, whether local control is needed, the degree of computational complexity involved, and the information required from the user. The Catmull-Rom spline is often used in creating a path through space because it is an interpolating spline and requires no additional information from the user other than the points that the path is to pass through. Bezier curves that are constrained to pass through given points are also often used. Parabolic blending is an often overlooked technique that also affords local control and is interpolating. Formulations for these curves appear in Appendix B.5. See Mortenson [5] and Rogers and Adams [6] for more in-depth discussions.

3.2 Controlling the motion of a point along a curve

Designing the shape of a curve is only the first step in creating an animation path. The speed at which the curve is traced out as the parametric value is increased has to be under the direct control of the animator to produce predictable results. If the relationship between a change in parametric value and the corresponding distance along the curve is not known, then it becomes much more difficult for the animator to produce desired effects. The first step in giving the animator control is to establish a method for stepping along the curve in equal increments. Once this is done, methods for speeding up and slowing down along the curve can be made available to the animator.

For this discussion, it is assumed that an interpolating technique has been chosen and that a function $\boldsymbol{P}(u)$ has been selected that, for a given value of u, will produce a value that is a point in space, that is, $\boldsymbol{p} = \boldsymbol{P}(u)$. Such vector valued functions will be shown in bold. Because a position in three-dimensional space is being interpolated, $\boldsymbol{P}(u)$ can be considered to represent three functions. The x-, y-, and z-coordinates for the positions at the key frames are specified by the user. Key frames are associated with specific values of the time parameter, u. The x-, y-, and z-coordinates are considered independently so that, for example, the x-coordinates of the points are used as control values for the interpolating curve so that $X = P_x(u)$, where P_x denotes an interpolating function; the subscript x is used to denote that this specific curve was formed using the x-coordinates of the key positions. Similarly, $Y = P_y(u)$ and $Z = P_z(u)$ are formed so that for any specified time, u, a position (X, Y, Z) can be produced as $(P_x(u), P_y(u), P_z(u)) = \boldsymbol{P}(u)$.

It is very important to note that varying the parameter of interpolation (in this case u) by a constant amount does not mean that the resulting values (in this case Euclidean position) will vary by a constant amount. Thus, if positions are being interpolated by varying u at a constant rate, the positions that are generated will not necessarily, in fact will seldom, represent a constant speed (e.g., see Figure 3.4).

To ensure a constant speed for the interpolated value, the interpolating function has to be parameterized by *arc length*, that is, distance along the curve of interpolation. Some type of reparameterization by arc length should be performed for most applications. Usually this reparameterization can be approximated without adding undue overhead or complexity to the interpolating function.

Three approaches to establishing the reparameterization by arc length are discussed here. One approach is to analytically compute arc length. Unfortunately, many curve formulations useful in animation do not lend themselves to the analytic method, so numeric methods must be applied. Two numeric methods are presented, both of which create a table of values to establish a relationship between parametric value and approximate arc length. This table can then be used to approximate

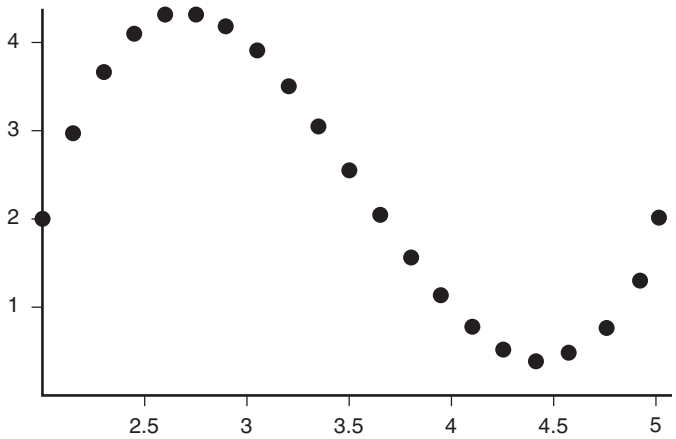


FIGURE 3.4

Example of points produced by equal increments of an interpolating parameter for a typical cubic curve. Notice the variable distance between adjacent points.

parametric values at equal-length steps along the curve. The first of these numeric methods constructs the table by supersampling the curve and uses summed linear distances to approximate arc length. The second numeric method uses Gaussian quadrature to numerically estimate the arc length. Both methods can benefit from an adaptive subdivision approach to controlling error.

3.2.1 Computing arc length

To specify how fast the object is to move along the path defined by the curve, an animator may want to specify the time at which positions along the curve should be attained. Referring to Figure 3.5 as an example in two-dimensional space, the animator specifies the following frame number and position pairs: (0, *A*), (10, *B*), (35, *C*), and (60, *D*).

Alternatively, instead of specifying time constraints, the animator might want to specify the relative velocities that an object should have along the curve. For example, the animator might specify that an object, initially at rest at position *A*, should smoothly accelerate until frame 20, maintain a constant

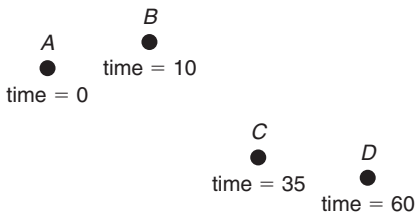


FIGURE 3.5

Position-time pairs constraining the motion.

speed until frame 35, and then smoothly decelerate until frame 60 at the end of the curve at position D. These kinds of constraints can be accommodated in a system that can compute the distance traveled along any span of the curve.

Assume that the position of an object in three-dimensional space (also referred to as three-space) is being interpolated. The objective is to define a parameterized function that evaluates to a point in three-dimensional space as a function of the parametric value; this defines a *space curve*. Assume for now that the function is a cubic polynomial as a function of a single parametric variable (as we will see, this is typically the case), that is, Equation 3.1.

$$\boldsymbol{P}(u) = \boldsymbol{a}u^3 + \boldsymbol{b}u^2 + \boldsymbol{c}u + \boldsymbol{d} \tag{3.1}$$

Remember that in three-space this really represents three equations: one for the x-coordinate, one for the y-coordinate, and one for the z-coordinate. Each of the three equations has its own constants a, b, c, and d. The equation can also be written explicitly representing these three equations, as in Equation 3.2.

$$\begin{aligned} \boldsymbol{P}(u) &= (x(u),\ y(u),\ z(u)) \\ x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\ y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y \\ z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z \end{aligned} \tag{3.2}$$

Each of the three equations is a cubic polynomial of the form given in Equation 3.1. The curve itself can be specified using any of the standard ways of generating a spline (see Appendix B.5 or texts on the subject, e.g., [6]). Once the curve has been specified, an object is moved along it by choosing a value of the parametric variable, and then the x-, y-, and z-coordinates of the corresponding point on the curve are calculated.

It is important to remember that in animation the path swept out by the curve in space is not the only important thing. Equally important is how the path is swept out over time. A very different effect will be evoked by the animation if an object travels over the curve at a strictly constant speed instead of smoothly accelerating at the beginning and smoothly decelerating at the end. As a consequence, it is important to discuss both the curve that defines the path to be followed by the object and the function that relates time to distance traveled. The former is the previously mentioned *space curve* and the term *distance-time function* will be used to refer to the latter. In discussing the distance-time function, the curve that represents the function will be referred to often. As a result, the terms *curve* and *function* will be used interchangeably in some contexts.

Notice that a function is desired that relates time to a position on the space curve. The user supplies, in one way or another (to be discussed in the sections that follow), the distance-time function that relates time to the distance traveled along the curve. The distance along a curve is defined as *arc length* and, in this text, is denoted by s. When the arc length computation is given as a function of a variable u and this dependence is noteworthy, then $s = S(u)$ is used. The arc length at a specific parametric value, such as $S(u_i)$, is often denoted as s_i. If the arc length computation is specified as a function of time, then $S(t)$ is used. Similarly, the function of arc length that computes a parametric value is specified as $u = U(s)$.

The interpolating function relates parametric value to position on the space curve. The relationship between distance along the curve and parametric value needs to be established. This relationship is the *arc length parameterization* of the space curve. It allows movement along the curve at a constant speed by evaluating the curve at equal arc length intervals. Further, it allows acceleration and deceleration along the curve by controlling the distance traveled in a given time interval.

For an arbitrary curve, it is usually not the case that a constant change in the parameter will result in a constant distance traveled. Because the value of the parameterizing variable is not the same as arc length, it is difficult to choose the values of the parameterizing variable so that the object moves along the curve at a desired speed. The relationship between the parameterizing variable and arc length is usually nonlinear. In the special case when a unit change in the parameterizing variable results in a unit change in curve length, the curve is said to be parameterized by arc length. Many seemingly difficult problems in controlling the motion along a path become very simple if the curve can be parameterized by arc length or, if arc length can be numerically computed, given a value for the parameterizing variable.

Let the function $LENGTH(u_a, u_b)$ be the length along the space curve from the point $P(u_a)$ to the point $P(u_b)$ (see Figure 3.6). Then the two problems to solve are

1. Given parameters u_a and u_b, find $LENGTH(u_a, u_b)$.
2. Given an arc length s and a parameter value u_a, find u_b such that $LENGTH(u_a, u_b) = \mathbf{s}$.
   This is equivalent to finding the solution to the equation $s - LENGTH(u_a, u_b) = 0$.

The first step in controlling the timing along a space curve is to establish the relationship between parametric values and arc length. This can be done by specifying the function $s = S(u)$, which computes, for any given parametric value, the length of the curve from its starting point to the point that corresponds to that parametric value. Then if the inverse, $u = S^{-1}(s) = U(s)$, can be computed (or estimated), the curve can be effectively parameterized by arc length, that is, $P(U(s))$. Once this is done, the second step is for the animator to specify, in one way or another, the distance the object should move along the curve for each time step.

In general, neither of the problems above has an analytic solution, so numerical solution techniques must be used. As stated previously, the first step in controlling the motion of an object along a space curve is to establish the relationship between the parametric value and arc length. If possible, the curve should be explicitly parameterized by arc length by analyzing the space curve equation. Unfortunately, for most types of parametric space curves, it is difficult or impossible to find a closed-form algebraic formula to describe an arc length parameter. For example, it has been shown that B-spline curves cannot, in general, be parameterized this way [3]. As a result, several approximate parameterization techniques have been developed. But first, it is useful to look at the analytic approach.

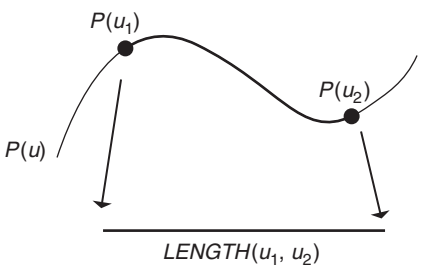


FIGURE 3.6

$LENGTH(u_1, u_2)$.

The analytic approach to computing arc length

The length of the curve from a point $\boldsymbol{P}(u_a)$ to any other point $\boldsymbol{P}(u_b)$ can be found by evaluating the arc length integral [4] by

$$s = \int_{u_a}^{u_b} |dP/du| du \tag{3.3}$$

where the derivative of the space curve with respect to the parameterizing variable is defined to be that shown in Equations 3.4 and 3.5.

$$d\boldsymbol{P}/du = ((dx(u)/(du)), (dy(u)/(du)), (dz(u)/(du))) \tag{3.4}$$

$$|d\boldsymbol{P}/du| = \sqrt{(dx(u)/du)^2 + (dy(u)/du)^2 + (dz(u)/du)^2} \tag{3.5}$$

For a cubic curve in which $\boldsymbol{P}(u) = au3 + bu2 + cu + d$, the derivative of the x-coordinate equation with respect to u is $dx(u)/du = 3a_x u2 + 2b_x u + c_x$. After squaring this and collecting the terms similarly generated by the y- and z-coordinate equations, the expression inside the radical takes the form of Equation 3.6.

$$Au^4 + Bu^3 + Cu^2 + Du + E \tag{3.6}$$

For the two-dimensional case, the coefficients are given in Equation 3.7; the extension to three-dimensional is straightforward.

$$\begin{aligned} A &= 9(a_x^2 + a_y^2) \\ B &= 12(a_x b_x + a_y b_y) \\ C &= 6(a_x c_x + a_y c_y) + 4(b_x^2 + b_y^2) \\ D &= 4(b_x c_x + b_y c_y) \\ E &= c_x^2 + c_y^2 \end{aligned} \tag{3.7}$$

Estimating arc length by forward differencing

As mentioned previously, analytic techniques are not tractable for many curves of interest to animation. The easiest and conceptually simplest strategy for establishing the correspondence between parameter value and arc length is to sample the curve at a multitude of parametric values. Each parametric value produces a point along the curve. These sample points can then be used to approximate the arc length by computing the linear distance between adjacent points. As this is done, a table is built up of arc lengths indexed by parametric values. For example, given a curve $\boldsymbol{P}(u)$, approximate the positions along the curve for $u = 0.00, 0.05, 0.10, 0.15, \ldots, 1.0$. The table, recording summed distances and indexed by entry number, is represented here as $G[i]$. It would be computed as follows:

$G[0] = 0.0$
$G[1] =$ the distance between $\boldsymbol{P}(0.00)$ and $\boldsymbol{P}(0.05)$
$G[2] = G[1]$ plus the distance between $\boldsymbol{P}(0.05)$ and $\boldsymbol{P}(0.10)$
$G[3] = G[2]$ plus the distance between $\boldsymbol{P}(0.10)$ and $\boldsymbol{P}(0.15)$
...
$G[20] = G[19]$ plus the distance between $\boldsymbol{P}(0.95)$ and $\boldsymbol{P}(1.00)$

Table 3.1 Parameter, Arc Length Pairs

Index	Parametric Value (V)	Arc Length (G)
0	0.00	0.000
1	0.05	0.080
2	0.10	0.150
3	0.15	0.230
4	0.20	0.320
5	0.25	0.400
6	0.30	0.500
7	0.35	0.600
8	0.40	0.720
9	0.45	0.800
10	0.50	0.860
11	0.55	0.900
12	0.60	0.920
13	0.65	0.932
14	0.70	0.944
15	0.75	0.959
16	0.80	0.972
17	0.85	0.984
18	0.90	0.994
19	0.95	0.998
20	1.00	1.000

For example, consider the table of values, V, for u and corresponding values of the function G in Table 3.1.

As a simple example of how such a table could be used, consider the case in which the user wants to know the distance (arc length) from the beginning of the curve to the point on the curve corresponding to a parametric value of 0.73. The parametric entry closest to 0.73 must be located in the table. Because the parametric entries are evenly spaced, the location in the table of the closest entry to the given value can be determined by direct calculation. Using Table 3.1, the index, i, is determined by Equation 3.8 using the distance between parametric entries, $d = 0.05$ in this case, and the given parametric value, $v = 0.73$ in this case.

$$i = \left\lfloor \frac{v}{d} + 0.5 \right\rfloor = \left\lfloor \frac{0.73}{0.05} + 0.5 \right\rfloor = 15 \tag{3.8}$$

A crude approximation to the arc length is obtained by using the arc length entry located in the table at index 15, that is, 0.959. A better approximation can be obtained by interpolating between the arc lengths corresponding to entries on either side of the given parametric value. In this case, the index of the largest parametric entry that is less than the given value is desired (Eq. 3.9).

$$i = \left\lfloor \frac{v}{d} \right\rfloor = \left\lfloor \frac{0.73}{0.05} \right\rfloor = 14 \tag{3.9}$$

An arc length, s, can be linearly interpolated from the approximated arc lengths, G, in the table by using the differences between the given parametric value and the parametric values on either side of it in the table, as in Equation 3.10.

$$\begin{aligned} s &= G[i] + \frac{v - V[i]}{V[i+1] - V[i]}(G[i+1] - G[i]) \\ &= 0.944 + \frac{0.73 - 0.70}{0.75 - 0.70}(0.959 - 0.944) \\ &= 0.953 \end{aligned} \tag{3.10}$$

The solution to the first problem cited above (given two parameters u_a and u_b, find the distance between the corresponding points) can be found by applying this calculation twice and subtracting the respective distances.

The converse situation, that of finding the value of u given the arc length, is dealt with in a similar manner. The table is searched for the closest arc length entry to the given arc length value, and the corresponding parametric entry is used to estimate the parametric value. This situation is a bit more complicated because the arc length entries are not evenly spaced; the table must actually be searched for the closest entry. Because the arc length entries are monotonically increasing, a binary search is an effective method of locating the closest entry. As before, once the closest arc length entry is found the corresponding parametric entry can be used as the approximate parametric value, or the parametric entries corresponding to arc length entries on either side of the given arc length value can be used to linearly interpolate an estimated parametric value.

For example, if the task is to estimate the location of the point on the curve that is 0.75 unit of arc length from the beginning of the curve, the table is searched for the entry whose arc length is closest to that value. In this case, the closest arc length is 0.72 and the corresponding parametric value is 0.40. For a better estimate, the values in the table that are on either side of the given distance are used to linearly interpolate a parametric value. In this case, the value of 0.75 is three-eighths of the way between the table values 0.72 and 0.80. Therefore, an estimate of the parametric value would be calculated as in Equation 3.11.

$$u = 0.40 + \frac{3}{8}(0.45 - 0.40) = 0.41875 \tag{3.11}$$

The solution to the second problem (given an arc length s and a parameter value u_a, find u_b such that $LENGTH(u_a, u_b) = s$) can be found by using the table to estimate the arc length associated with u_a, adding that to the given value of s, and then using the table to estimate the parametric value of the resulting length.

The advantages of this approach are that it is easy to implement, intuitive, and fast to compute. The downside is that both the estimate of the arc length and the interpolation of the parametric value introduce errors into the calculation. These errors can be reduced in a couple of ways.

The curve can be supersampled to help reduce errors in forming the table. For example, ten thousand equally spaced values of the parameter could be used to construct a table consisting of a thousand entries by breaking down each interval into ten subintervals. This is useful if the curve is given beforehand and the table construction can be performed as a preprocessing step.

Better methods of interpolation can be used to reduce errors in estimating the parametric value. Instead of linear interpolation, higher-degree interpolation procedures can be used in computing the parametric value. Of course, higher-degree interpolation slows down the computation somewhat, so a decision about the speed/accuracy trade-off has to be made.

These techniques reduce the error somewhat blindly. There is no measure for the error in the calculation of the parametric value; these techniques only reduce the error globally instead of investing more computation in the areas of the curve in which the error is highest.

Adaptive approach

To better control error, an *adaptive forward differencing* approach can be used that invests more computation in areas of the curve that are estimated to have large errors. The approach does this by considering a section of the curve and testing to see if the estimate of the segment's length is within some error tolerance of the sum of the estimates for each of the segment's halves. If the difference is greater than can be tolerated, then the segment is split in half and each half is put on the list of segments to be tested. In addition, at each level of subdivision the tolerance is reduced by half to reflect the change in scale. If the difference is within the tolerance, then its estimated length is taken to be a good enough estimate and is used for that segment. Initially, the entire curve is the segment to be tested.

As before, a table is to be constructed that relates arc length to parametric value. Each element of the table records a parametric value and the associated arc length from the start of the curve. It is also useful to record the coordinates of the associated point on the curve. A linked list is an appropriate structure to hold this list because the number of entries that will be generated is not known beforehand; after all points are generated, the linked list can then be copied to a linear list to facilitate a binary search. In addition to the list of entries, a sorted list of segments to be tested is maintained. A segment on the list is defined and sorted according to its range of parametric values.

The adaptive approach begins by initializing the table with an entry for the first point of the curve, $<0.0, \boldsymbol{P}(0)>$, and initializing the list of segments to be tested with the entire curve, $<0.0,1.0>$. The procedure operates on segments from the list to be tested until the list is empty. The first segment on the list is always the one tested next. The segment's midpoint is computed by evaluating the curve at the middle of the range of its parametric value. The curve is also evaluated at the endpoint of the segment; the position of the start of the segment is already in the table and can be retrieved from there. The length of the segment and the lengths of each of its halves are estimated by the linear distance between the points on the curve. The sum of the estimated lengths of the two halves of the segment is compared to the estimated length of the segment. Equation 3.12 shows the test for the initial entire-curve segment.

$$\left|\,||P(0.0) - P(1.0)|| - \left(||P(0.0) - P(0.5)|| + ||P(0.5) - P(1.0)||\right)\right| < \varepsilon \tag{3.12}$$

If the difference between these two values is above some user-specified threshold, then both halves, in order, are added to the list of segments to be tested along with their error tolerance ($\varepsilon/2^n$ where n is the level of subdivision). If the values are within tolerance, then the parametric value of the midpoint is recorded in the table along with the arc length of the first point of the segment plus the distance from the first point to the midpoint. Also added to the table is the last parametric value of the segment along with the arc length to the midpoint plus the distance from the midpoint to the last point. When the list of segments to be tested becomes empty, a list of $<$parametric value, arc length$>$ has been generated for the entire curve.

One problem with this approach is that at a premature stage in the procedure two half segments might indicate that the subdivision can be terminated (Figure 3.7). It is usually wise to force the subdivision down to a certain level and then embark on the adaptive subdivision.

Because the table has been adaptively built, it is not possible to directly compute the index of a given parametric entry as it was with the nonadaptive approach. Depending on the data structure used

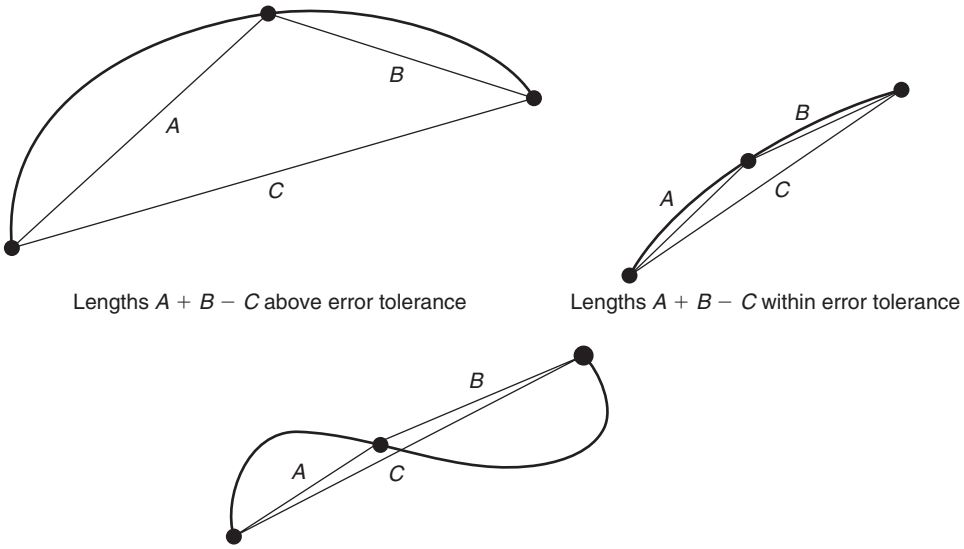


FIGURE 3.7

Tests for adaptive subdivision.

to hold the final table, a sequential search or, possibly, a binary search must be used. Once the appropriate entries are found, then, as before, a corresponding table entry can be used as an estimate for the value or entries can be interpolated to produce better estimates.

Estimating the arc length integral numerically

For cases in which efficiency of both storage and time are of concern, calculating the arc length function numerically may be desirable. Calculating the length function involves evaluating the arc length integral (refer to Eq. 3.3). Many numerical integration techniques approximate the integral of a function with the weighted sum of values of the function at various points in the interval of integration. Techniques such as Simpson's and trapezoidal integration use evenly spaced sample intervals. Gaussian quadrature [1] uses unevenly spaced intervals in an attempt to get the greatest accuracy using the smallest number of function evaluations. Because evaluation of the derivatives of the space curve accounts for most of the processing time in this algorithm, minimizing the number of evaluations is important for efficiency. Also, because the higher derivatives are not continuous for some piecewise curves, this should be done on a per segment basis.

Gaussian quadrature, as commonly used, is defined over an integration interval from -1 to 1. The function to be integrated is evaluated at fixed points in the interval -1 to $+1$, and each function evaluation is multiplied by a precalculated weight (see Eq. 3.13).

$$\int_{-1}^{1} f(u) = \sum_{i} w_i f(u_i) \tag{3.13}$$

A function, $g(t)$, defined over an arbitrary integration interval $t \in [a, b]$ can be converted to a function, $h(u) = g(f(u))$, defined over the interval $u \in [-1, 1]$ by the linear transformation shown in Equation 3.14. By making this substitution and the corresponding adjustments to the derivative, any range can be handled by the Gaussian integration as shown in Equation 3.15 for arbitrary limits, a and b.

$$t = f(u) = \frac{(b-a)}{2}u + \frac{b+a}{2} \tag{3.14}$$

$$\begin{aligned} \int_a^b g(t)dt &= \int_{-1}^{+1} g(f(u))f'(u)du \\ &= \left(\frac{b-a}{2}\right)\int_{-1}^{+1} g\left(\frac{(b-a)}{2}u + \frac{b+a}{2}\right)du \end{aligned} \tag{3.15}$$

The weights and evaluation points for different orders of Gaussian quadrature have been tabulated and can be found in many mathematical handbooks (see Appendix B.8.1 for additional details).

In using Gaussian quadrature to compute the arc length of a cubic curve, Equations 3.5–3.7 are used to define the arc length function in the form shown in Equation 3.16 after common terms have been collected into the new coefficients. Sample code to implement this is given below in the discussion on adaptive Gaussian integration.

$$\int_{-1}^{1} \sqrt{Au^4 + B\mathrm{u}^3 + Cu^2 + Du + E} \tag{3.16}$$

Adaptive Gaussian integration

Some space curves have derivatives that vary rapidly in some areas and slowly in others. For such curves, Gaussian quadrature will undersample some areas of the curve or oversample some other areas. In the former case, unacceptable error will accumulate in the result. In the latter case, time is wasted by unnecessary evaluations of the function. This is similar to what happens when using nonadaptive forward differencing.

To address this problem, an adaptive approach can be used [2]. Adaptive Gaussian integration is similar to the previously discussed adaptive forward differencing. Each interval is evaluated using Gaussian quadrature. The interval is then divided in half, and each half is evaluated using Gaussian quadrature. The sum of the two halves is compared with the value calculated for the entire interval. If the difference between these two values is less than the desired accuracy, then the procedure returns the sum of the halves; otherwise, as with the forward differencing approach, the two halves are added to the list of intervals to be subdivided along with the reduced error tolerance, $\varepsilon/2^n$, where, as before, ε is the user-supplied error tolerance and n is the level of subdivision.

Initially the length of the entire space curve is calculated using adaptive Gaussian quadrature. During this process, a table of the subdivision points is created. Each entry in the table is a pair (u, s) where s is the arc length at parameter value u. When calculating $LENGTH(0, u)$, the table is searched to find the values u_i, u_{i+1} such that $u_i < u < u_{i+1}$. The arc length from u_i to u is then calculated using nonadaptive Gaussian quadrature. This can be done because the space curve has been subdivided in such a way that nonadaptive Gaussian quadrature will give the required accuracy over the interval from u_i to $u_i + 1$. $LENGTH(0, u)$ is then equal to $s_i + LENGTH(u_i, u)$. $LENGTH(u_a, u_b)$ can be found by calculating $LENGTH(0, u_a)$ and $LENGTH(0, u_b)$ and then subtracting. The code for the adaptive integration and table-building procedure is shown in Figure 3.8.

```
/* ------------------------------------------------------------
 * STRUCTURES
*/


// the structure to hold entries of the table consisting of
// parameter value (u) and estimated length (length)
typedef struct table_entry_structure {
  double u,length;
} table_entry_td;


// the structure to hold an interval of the curve, defined by
// starting and ending parameter values and the estimated
// length (to be filled in by the adaptive integration
// procedure)
typedef struct interval_structure {
  double ua,ub;
  double length;
} interval_td;


// coefficients for a 2D cubic curve
typedef struct cubic_curve_structure {
  double ax,bx,cx,dx;
  double ay,by,cy,dy;
} cubic_curve_td;


// polynomial function structure; a quadric function is generated
// from a cubic curve during the arclength computation
typedef struct polynomial_structure {
  double *coeff;
  int  degree;
} polynomial_td;



/* ------------------------------------------------------------------
 * ADAPTIVE INTEGRATION
 * this is the high-level call used whenever a curve's length is to be computed
*/
void adaptive_integration(cubic_curve_td *curve, double ua, double ub,double
tolerance)
{
   double subdivide();
   polynomial_td func;
   interval_td full_interval;
   double total_length;
   double integrate_func();
   double temp;

   func.degree = 4;
   func.coeff = (double *)malloc(sizeof(double)*5);
```

FIGURE 3.8

Adaptive Gaussian integration of arc length.

(Continued)

```
func.coeff[4] = 9*(curve->ax*curve->ax + curve->ay*curve->ay);
   func.coeff[3] = 12*(curve->ax*curve->bx + curve->ay*curve->by);
   func.coeff[2] = (6*(curve->ax*curve->cx + curve->ay*curve->cy) +
               4*(curve->bx*curve->bx + curve->by*curve->by)  );
   func.coeff[1] = 4*(curve->bx*curve->cx + curve->by*curve->cy);
   func.coeff[0] = curve->cx*curve->cx + curve->cy*curve->cy;
   full_interval.ua = ua; full_interval.ub = ub;
   temp = integrate_func(&func,&full_interval);
   printf("\nInitial guess = %lf; %lf:%lf",temp,ua,ub);
   full_interval.length = temp;
   total_length = subdivide(&full_interval,&func,0.0,tolerance);
   printf("\n total length = %lf\n",total_length);
}


/* ---------------------------------------------------------------
 * SUBDIVIDE
 * 'total_length' is the length of the curve up to, but not including,
 *         the 'full interval'
 * if the difference between the interval and the sum of its halves is
 *         less than 'tolerance,' stop the recursive  subdivision
 * 'func' is a polynomial function
 */
double subdivide(interval_td *full_interval, polynomial_td *func, double
total_length, double tolerance)
{
   interval_td left_interval, right_interval;
   double left_length,right_length;
   double midu;
   double subdivide();
   double integrate_func();
   double temp;
   void add_table_entry();

   midu = (full_interval->u1+full_interval->u2)/2;
   left_interval.ua = full_interval->ua; left_interval.ub = midu;
   right_interval.ua = midu; right_interval.ub = full_interval->ub;
   left_length = integrate_func(func, & left_interval);
   right_length = integrate_func(func, & right_interval);
   temp = fabs(full_interval->length - (left_length+right_length));
   if (temp > tolerance) {
     left_interval.length = left_length;
     right_interval.length = right_length;
     total_length = subdivide(&left_interval, func, total_length, tolerance/2.0);
     total_length = subdivide(&right_interval, func, total_length, tolerance/2.0);
     return(total_length);
   }
   else {
     total_length = total_length + left_length;
     add_table_entry(midu,total_length);
     total_length = total_length + right_length;
     add_table_entry(full_interval->ub,total_length);
     return(total_length);
}
}
```

FIGURE 3.8—Cont'd

```
/* ------------------------------------------------------------------
 * ADD TABLE ENTRY
 * In an actual implementation, this code would create and add an
 *      entry to a linked list.
 * The code below simply prints out the values.
*/
void add_table_entry(double u, double length)
{
   /* add entry of (u, length) */
   printf("\ntable entry: u: %lf, length: %lf",u,length);
}


/* ------------------------------------------------------------------
 * INTEGRATE FUNCTION
* use Gaussian quadrature to integrate square root of given function
 *          in the given interval
*/
double integrate_func(polynomial_td *func,interval_td *interval)
{
   double x[5] = {.1488743389,.4333953941,.6794095682,.8650633666,
   .9739065285};
   double w[5] =
   {.2955242247,.2692667193,.2190863625,.1494513491,.0666713443};
   double length, midu, dx, diff;
   int  i;
   double evaluate_polynomial();
   double ua,ub;
   ua = interval->ua;
   ub = interval->ub;
   midu = (ua+ub)/2.0;
   diff = (ub-ua)/2.0;
   length = 0.0;
   for (i=0; i<5; i++) {
      dx = diff*x[i];
      length += w[i]*(sqrt(evaluate_polynomial(func,midu+dx)) +
            sqrt(evaluate_polynomial(func,midu-dx)));
   }
   length *= diff;
   return (length);
}


/* ------------------------------------------------------------
 * EVALUATE POLYNOMIAL
 * evaluate a polynomial using the Horner scheme
*/
double evaluate_polynomial(polynomial_td *poly, double u)
{
   int   i;
   double value;
   value = 0.0;
   for (i=poly->degree; i<=0; i--) {
      value = value*u+poly->coeff[i];
   }
   return value;
}
```

FIGURE 3.8—Cont'd

This solves the first problem posed above; that is, given u_a and u_b find $LENGTH(u_a, u_b)$. To solve the second problem of finding u, which is a given distance, s, away from a given u_a, numerical root-finding techniques must be used as described next.

Find a point that is a given distance along a curve

The solution of the equation $s - LENGTH(u_a, u) = 0$ gives the value of u such that the point $P(u)$ is at arc length s from the point $P(u_a)$ along the curve. Since the arc length is a strictly monotonically increasing function of u, the solution is unique provided that the length of $d\boldsymbol{P}(u)/du$ is not identically 0 over some interval. Newton-Raphson iteration can be used to find the root of the equation because it converges rapidly and requires very little calculation at each iteration. Newton-Raphson iteration consists of generating the sequence of points $\{p_n\}$ as in Equation 3.17.

$$p_n = p_{n-1} - f(p_{n-1})/f'(p_{n-1}) \tag{3.17}$$

In this case, f is equal to $s - LENGTH(u_1, U(p_{n-1}))$ where $U(p)$ is a function that computes the parametric value of a point on a parameterized curve. The function $U()$ can be evaluated at p_{n-1} using precomputed tables of parametric value and points on the curve similar to the techniques discussed above for computing arc length; f' is dP/du evaluated at p_{n-1}.

Two problems may be encountered with Newton-Raphson iteration: some of the p_n may not lie on the space curve at all (or at least within some tolerance) and $d\boldsymbol{P}/du$ may be 0, or very nearly 0, at some points on the space curve. The first problem will cause all succeeding elements $p_{n+1}, p_{n+2}, \ldots$ to be undefined, while the latter problem will cause division by 0 or by a very small number. A zero parametric derivative is most likely to arise when two or more control points are placed in the same position. This can easily be detected by examining the derivative of f in Equation 3.17. If it is small or 0, then binary subdivision is used instead of Newton-Raphson iteration. Binary subdivision can be used in a similar way to handle the case of undefined p_n. When finding u such that $LENGTH(0, u) = s$, the subdivision table is searched to find the values s_i, s_{i+1} such that $s_i < s < s_{i+1}$. The solution u lies between u_i and u_{i+1}. Newton-Raphson iteration is then applied to this subinterval.

An initial approximation to the solution is made by linearly interpolating s between the endpoints of the interval and using this as the first iterate. Although Newton-Raphson iteration can diverge under certain conditions, it does not happen often in practice. Since each step of Newton-Raphson iteration requires evaluation of the arc length integral, eliminating the need to do adaptive Gaussian quadrature, the result is a significant increase in speed. At this point it is worth noting that the integration and root-finding procedures are completely independent of the type of space curve used. The only procedure that needs to be modified to accommodate new curve types is the derivative calculation subroutine, which is usually a short program.

3.2.2 Speed control

Once a space curve has been parameterized by arc length, it is possible to control the speed at which the curve is traversed. Stepping along the curve at equally spaced intervals of arc length will result in a constant speed traversal. More interesting traversals can be generated by speed control functions that relate an equally spaced parametric value (e.g., *time*) to arc length in such a way that a controlled traversal of the curve is generated. The most common example of such speed control is *ease-in/ease-out*

traversal. This type of speed control produces smooth motion as an object accelerates from a stopped position, reaches a maximum velocity, and then decelerates to a stopped position.

In this discussion, the speed control function's input parameter is referred to as t, for *time*, and its output is *arc length*, referred to as *distance* or simply as s. Constant-velocity motion along the space curve can be generated by evaluating it at equally spaced values of its arc length where arc length is a linear function of t. In practice, it is usually easier if, once the space curve has been reparameterized by arc length, the parameterization is then normalized so that the parametric variable varies between 0 and 1 as it traces out the space curve; the *normalized arc length parameter* is just the arc length parameter divided by the total arc length of the curve. For this discussion, the normalized arc length parameter will still be referred to simply as the arc length.

Speed along the curve can be controlled by varying the arc length values at something other than a linear function of t; the mapping of time to arc length is independent of the form of the space curve itself. For example, the space curve might be linear, while the arc length parameter is controlled by a cubic function with respect to time. If t is a parameter that varies between 0 and 1 and if the curve has been parameterized by arc length and normalized, then ease-in/ease-out can be implemented by a function $s = ease(t)$ so that as t varies uniformly between 0 and 1, s will start at 0, slowly increase in value and gain speed until the middle values, and then decelerate as it approaches 1 (see Figure 3.9). Variable s is then used as the interpolation parameter in whatever function produces spatial values.

The control of motion along a parametric space curve will be referred to as *speed control*. Speed control can be specified in a variety of ways and at various levels of complexity. But the final result is to produce, explicitly or implicitly, a distance-time function $S(t)$, which, for any given time t, produces the distance traveled along the space curve (arc length). The space curve defines *where* to go, while the distance-time function defines *when*.

Such a function $S(t)$ would be used as follows. At a given time t, $S(t)$ is the desired distance to have traveled along the curve at time t. The arc length of the curve segment from the beginning of the curve to the point is $s = S(t)$ (within some tolerance). If, for example, this position is used to translate an object through space at each time step, it translates along the path of the space curve at a speed indicated by $S'(t)$. An arc length table (see Section 3.2.1) can then be used to find the corresponding parametric value $u = U(s)$ that corresponds to that arc length. The space curve is then evaluated at u to produce a point on the space curve, $p = \boldsymbol{P}(u)$. Thus, $\boldsymbol{p} = \boldsymbol{P}(U(S(t)))$.

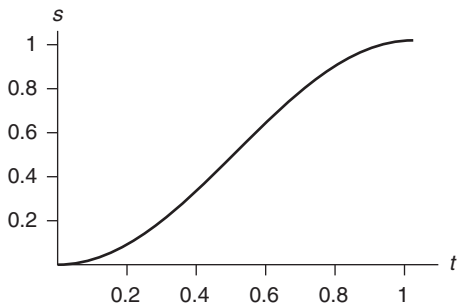


FIGURE 3.9

An example of an *ease*(*t*) function.

There are various ways in which the distance-time function can be specified. It can be explicitly defined by giving the user graphical curve-editing tools. It can be specified analytically. It can also be specified by letting the user define the velocity-time curve, or by defining the acceleration-time curve. The user can even work in a combination of these spaces. In the following discussion, the common assumption is that the entire arc length of the curve is to be traversed during the given total time. Two additional optional assumptions (constraints) that are typically used are listed below. In certain situations, it might be desirable to violate one or both of these constraints.

1. The distance-time function should be monotonic in t, that is, the curve should be traversed without backing up along the curve.
2. The distance-time function should be continuous, that is, there should be no instantaneous jumps from one point on the curve to a nonadjacent point on the curve.

Following the assumptions stated above, the entire space curve is to be traversed in the given total time. This means that $0.0 = S(0.0)$ and $total\_distance = S(total\_time)$. As mentioned previously, the distance-time function may also be normalized so that all such functions end at $1.0 = S(1.0)$. Normalizing the distance-time function facilitates its reuse with other space curves. An example of an analytic definition is $S(t) = (2 - t)*t$ (although this does not have the shape characteristic of ease-in/ease-out motion control; see Figure 3.10).

3.2.3 Ease-in/ease-out

Ease-in/ease-out is one of the most useful and most common ways to control motion along a curve. There are several ways to incorporate ease-in/ease-out control. The standard assumption is that the motion starts and ends in a complete stop and that there are no instantaneous jumps in velocity (first-order continuity). There may or may not be an intermediate interval of constant speed, depending on the technique used to generate the speed control. The speed control function will be referred to as $s = ease(t)$, where t is a uniformly varying input parameter meant to represent time and s is the output parameter that is the distance (arc length) traveled as a function of time.

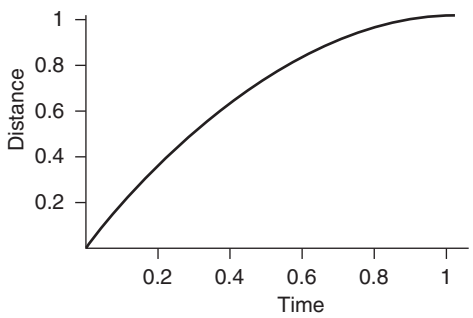


FIGURE 3.10

Graph of sample analytic distance-time function $(2 - t)*t$.

Sine interpolation

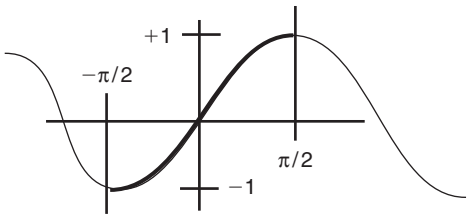
One easy way to implement ease-in/ease-out is to use the section of the sine curve from $-\pi/2$ to $+\pi/2$ as the *ease*(t) function. This is done by mapping the parameter values of 0 to $+1$ into the domain of $-\pi/2$ to $+\pi/2$ and then mapping the corresponding range of the sine functions of -1 to $+1$ in the range 0 to 1. See Equation 3.18 and the corresponding Figure 3.11.

$$s = ease(t) = \frac{\sin\left(t_\pi - \frac{\pi}{2}\right) + 1}{2} \tag{3.18}$$

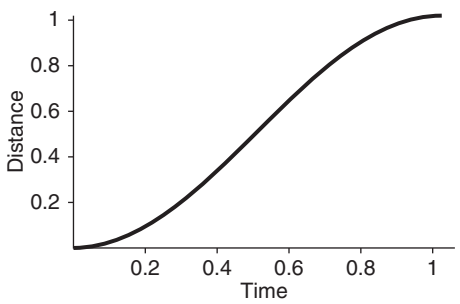
In this function, t is the input that is to be varied uniformly from 0 to 1 (e.g., 0.0, 0.1, 0.2, 0.3, . . .). The output, s, also goes from 0 to 1 but does so by starting off slowly, speeding up, and then slowing down. For example, at $t = 0.25$, $ease(t) = 0.1465$, and at $t = 0.75$, $ease(t) = 0.8535$. With the sine/cosine ease-in/ease-out function presented here, the "speed" of s with respect to t is never constant over an interval, rather it is always accelerating or decelerating, as can be seen by the ever-changing slope of the curve. Notice that the derivative of this ease function is 0 at $t = 0$ and at $t = 1$. Zero derivatives at 0 and 1 indicate a smooth acceleration from a stopped position at the beginning of the motion and a smooth deceleration to a stop at the end.

Using sinusoidal pieces for acceleration and deceleration

To provide an intermediate section of the distance-time function that has constant speed, pieces of the sine curve can be constructed at each end of the function with a linear intermediate segment. Care must be taken so that the tangents of the pieces line up to provide first-order continuity. There are various ways to approach this, but as an example, assume that the user specifies fragments of the unit interval that should be devoted to initial acceleration and final deceleration. For example, the user may specify that acceleration ceases at 0.3 and the deceleration starts at 0.75. That is, acceleration occurs from time 0 to 0.3 and deceleration occurs from time 0.75 to the end of the interval at 1.0. Referring to the user-specified values as k_1 and k_2 respectively, a sinusoidal curve segment is used to implement an acceleration from time 0 to k_1. A sinusoidal curve is also used for velocity to implement deceleration from time k_2 to 1. Between times k_1 and k_2, a constant velocity is used.



Sine curve segment to use as ease-in/ease-out control



Sine curve segment mapped to useful values

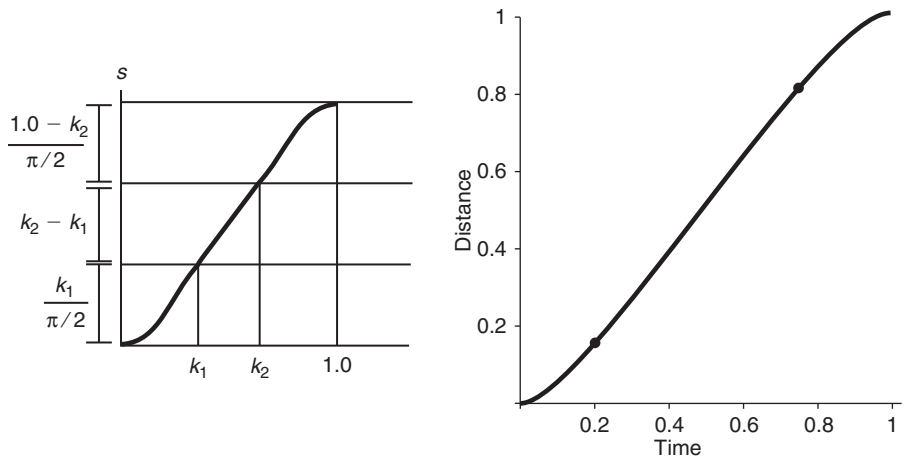
FIGURE 3.11

Using a sine curve segment as the ease-in/ease-out distance-time function.

The solution is formed by piecing together a sine curve segment from $-\pi/2$ to 0 with a straight line segment (indicating constant speed) inclined at 45 degrees (because the slope of the sine curve at 0 is equal to 1) followed by a sine curve segment from 0 to $\pi/2$. The initial sine curve segment is uniformly scaled by $k_1/(\pi/2)$ so that the length of its domain is k_1. The length of the domain of the line segment is $k_2 - k_1$. And the final sine curve segment is uniformly scaled by $1.0 - k_2/(\pi/2)$ so that the length of its domain is $1.0 - k_2$. The sine curve segments must be uniformly scaled to preserve C^1 (first-order) continuity at the junction of the sine curves with the straight line segment. This means that for the first sine curve segment to have a domain from 0 to k_1, it must have a range of length $k_1/(\pi/2)$; similarly, the ending sine curve segment must have a range of length $1 - k_2/(\pi/2)$. The middle straight line segment, with a slope of 1, will travel a distance of $k_2 - k_1$.

To normalize the distance traveled, the resulting three-segment curve must be scaled down by a factor equal to the total distance traveled as computed by $k_1/(\pi/2) + k_2 - k_1 + 1 - k_2/(\pi/2)$ (see Figure 3.12).

The ease function described above and shown in Equation 3.19 is implemented in the code of Figure 3.13.



Ease-in/ease-out curve as it is initially pieced together

Curve segments scaled into useful values with points marking segment junctions

FIGURE 3.12

Using sinusoidal segments with constant speed intermediate interval.

```
double ease(float t, float k1, float k2)
{
   double f,s;
   f = k1*2/PI + k2 - k1 + (1.0 - k2)*2/PI;
   if(t < k1)        s = k1*(2/PI)*(sin((t/k1)*PI/2 - PI/2)+1);
   else if (t < k2)  s = (2*k1/PI + t - k1);
         else        s = 2*k1/PI + k2 - k1 + ((1-k2)*(2/PI)) *
                           sin(((t - k2)/(1.0 - k2))*PI/2);
   return (s/f);
}
```

FIGURE 3.13

Code to implement ease-in/ease-out using sinusoidal ease-in, followed by constant velocity and sinusoidal ease-out.

$$ease(t)\begin{cases} = \left(k_1\frac{2}{\pi}\left(\sin\left(\frac{t}{k_1}\frac{\pi}{2}-\frac{\pi}{2}\right)+1\right)\right)/f & t \le k_1 \\ = \left(\frac{k_1}{\pi/2}+t-k_1\right)/f & k_1 \le t \le k_2 \\ = \left(\frac{k_1}{\pi/2}+k_2-k_1+\left((1-k_2)\frac{2}{\pi}\right)\sin\left(\left(\frac{t-k_2}{1-k_2}\right)\frac{\pi}{2}\right)\right)/f & k_2 \le t \end{cases} \tag{3.19}$$

where $f = k_1(2/\pi + k_2 - k_1 + (1-k_2)(2/\pi)$

Single cubic polynomial ease-in/ease-out

A single polynomial can be used to approximate the sinusoidal ease-in/ease-out control (Eq. 3.20). It provides accuracy within a couple of decimal points while avoiding the transcendental function[2] invocations. It passes through the points (0,0) and (1,1) with horizontal beginning and ending tangents of 0. Its drawback is that there is no intermediate segment of constant speed (see Figure 3.14).

$$s = -2t^3 + 3t^2 \tag{3.20}$$

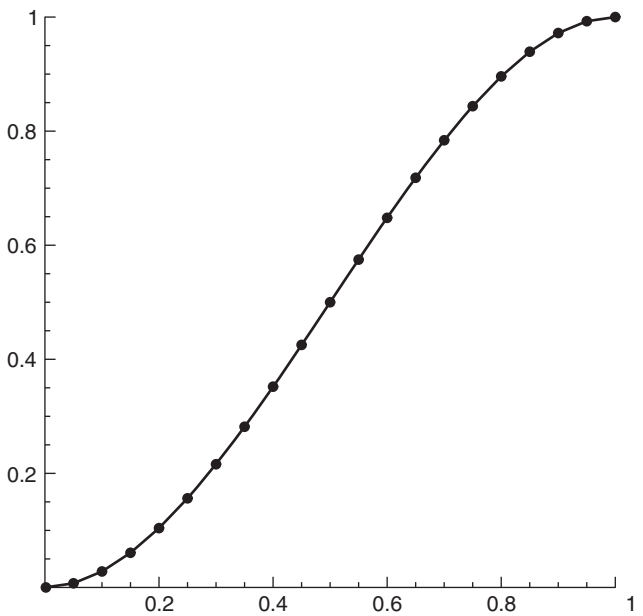


FIGURE 3.14

Ease-in/ease-out polynomial $s = -2t^3 + 3t^2$.

[2]A transcendental function is one that cannot be expressed algebraically. That is, it transcends expression in algebra. Transcendental functions include the exponential function as well as the trigonometric functions such as sine.

Constant acceleration: parabolic ease-in/ease-out

To avoid the transcendental function evaluation while still providing for a constant speed interval between the ease-in and ease-out intervals, an alternative approach for the ease function is to establish basic assumptions about the acceleration that in turn establish the basic form that the velocity-time curve can assume. The user can then set parameters to specify a particular velocity-time curve that can be integrated to get the resulting distance-time function.

The simple case of no ease-in/ease-out would produce a constant zero acceleration curve and a velocity curve that is a horizontal straight line at some value v_0 over the time interval from 0 to total time, t_{total}. The actual value of v_0 depends on the total distance covered, d_{total}, and is computed using the relationship

$$\text{distance} = \textit{speed}\cdot\textit{time}$$

so that

$$v_0 = \frac{\mathrm{d_{total}}}{\mathrm{t_{total}}}$$

In the case where normalized values of 1 are used for total distance covered and total time, $v_0 = 1$ (see Figure 3.15).

The distance-time curve is defined by the integral of the velocity-time curve and relates time and distance along the space curve through a function $S(t)$. Similarly, the velocity-time curve is defined by the integral of the acceleration-time curve and relates time and velocity along the space curve.

To implement an ease-in/ease-out function, constant acceleration and deceleration at the beginning and end of the motion and zero acceleration during the middle of the motion are assumed. The assumptions of beginning and ending with stopped positions mean that the velocity starts out at 0 and ends at 0. In order for this to be reflected in the acceleration/deceleration curve, the area under the curve marked

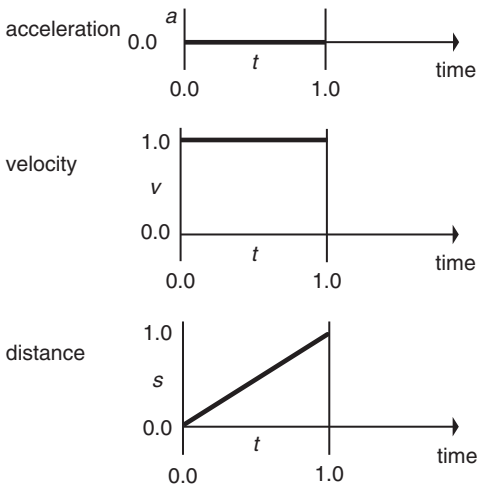
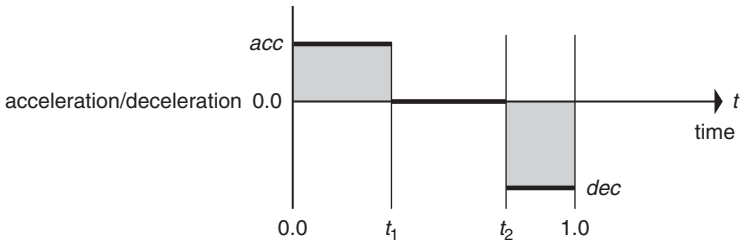


FIGURE 3.15

Acceleration, velocity, and distance curves for constant speed.

"acc" must be equal to the area above the curve marked "dec," but the actual values of the acceleration and deceleration do not have to be equal to each other. Thus, three of the four variables (acc, dec, t_1, t_2) can be specified by the user and the system can solve for the fourth to enforce the constraint of equal areas (see Figure 3.16).

This piecewise constant acceleration function can be integrated to obtain the velocity function. The resulting velocity function has a linear ramp for accelerating, followed by a constant velocity interval, and ends with a linear ramp for deceleration (see Figure 3.17). The integration introduces a constant into the velocity function, but this constant is 0 under the assumption that the velocity starts out at 0 and ends at 0. The constant velocity attained during the middle interval depends on the total distance that



$a = acc \quad 0 < t < t_1$
$a = 0.0 \quad t_1 < t < t_2$
$a = dec \quad t_2 < t < 1.0$

FIGURE 3.16

Constant acceleration/deceleration graph.

$$v = v_0 \cdot \frac{t}{t_1} \qquad 0.0 < t < t_1$$

$$v = v_0 \qquad t_1 < t < t_2$$

$$v = v_0 \cdot \left(1.0 - \frac{t - t_2}{1 - t_2}\right) \qquad t_2 < t < 1.0$$

FIGURE 3.17

Velocity-time curve for constant acceleration. Area under the curve equals the distance traveled.

must be covered during the time interval; the velocity is equal to the area below (above) the *acc* (*dec*) segment in Figure 3.16. In the case of normalized time and normalized distance covered, the total time and total distance are equal to 1. The total distance covered will be equal to the area under the velocity curve (Figure 3.17). The area under the velocity curve can be computed as in Equation 3.21.

$$\begin{aligned} 1 &= A_{acc} + A_{constant} + A_{dec} \\ 1 &= \frac{1}{2} v_0 t_1 + v_0 (t_2 - t_1) + \frac{1}{2} v_0 (1 - t_2) \end{aligned} \tag{3.21}$$

Because integration introduces arbitrary constants, the acceleration-time curve does not bear any direct relation to total distance covered. Therefore, it is often more intuitive for the user to specify ease-in/ease-out parameters using the velocity-time curve. In this case, the user can specify two of the three variables, t_1, t_2, and v_0, and the system can solve for the third in order to enforce the "total distance covered" constraint. For example, if the user specifies the time over which acceleration and deceleration take place, then the maximum velocity can be found by using Equation 3.22.

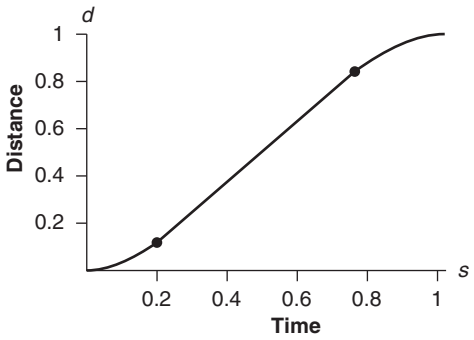
$$v_0 = \frac{2}{(t_2 - t_2 + 1)} \tag{3.22}$$

The velocity-time function can be integrated to obtain the final distance-time function. Once again, the integration introduces an arbitrary constant, but, with the assumption that the motion begins at the start of the curve, the constant is constrained to be 0. The integration produces a parabolic ease-in segment, followed by a linear segment, followed by a parabolic ease-out segment (Figure 3.18).

The methods for ease control based on one or more sine curve segments are easy to implement and use but are less flexible than the acceleration-time and velocity-time functions. These last two functions allow the user to have even more control over the final motion because of the ability to set various parameters.

3.2.4 General distance-time functions

When working with velocity-time curves or acceleration-time curves, one finds that the underlying assumption that the total distance is to be traversed during the total time presents some interesting issues. Once the total distance and total time are given, the average velocity is fixed. This average



$$d = v_0 \frac{t^2}{2t_1} \qquad 0.0 < t < t_1$$

$$d = v_0 \frac{t_1}{2} + v_0 (t - t_1) \qquad t_1 < t < t_2$$

$$d = v_0 \frac{t_1}{2} + v_0 (t_2 - t_1) + \left[v_0 - \frac{\left(v_0 \frac{t - t_2}{1 - t_2} \right)}{2} \right] (t - t_2) \qquad t_2 < t < 1.0$$

FIGURE 3.18

Distance-time function for constant acceleration.

velocity must be maintained as the user modifies, for example, the shape of the velocity-time curve. This can create a problem if the user is working only with the velocity-time curve. One solution is to let the absolute position of the velocity-time curve float along the velocity axis as the user modifies the curve. The curve will be adjusted up or down in absolute velocity in order to maintain the correct average velocity. However, this means that if the user wants to specify certain velocities, such as starting and ending velocities, or a maximum velocity, then other velocities must change in response in order to maintain total distance covered.

An alternative way to specify speed control is to fix the absolute velocities at key points and then change the interior shape of the curve to compensate for average velocity. However, this may result in unanticipated (and undesirable) changes in the shape of the velocity-time curve. Some combinations of values may result in unnatural spikes in the velocity-time curve in order to keep the area under the curve equal to the total distance (equal to 1, in the case of normalized distance). Consequently, undesirable accelerations may be produced, as demonstrated in Figure 3.19.

Notice that negative velocities mean that a point traveling along the space curve backs up along the curve until the time is reached when velocity becomes positive again. Usually, this is not desirable behavior.

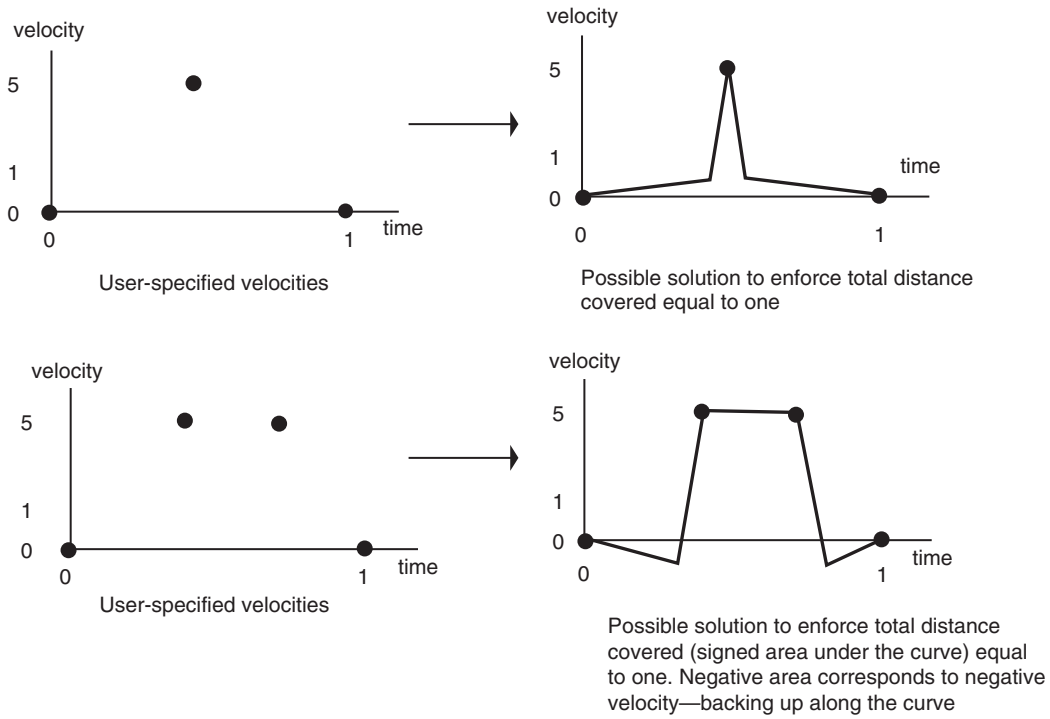


FIGURE 3.19

Some non-intuitive results of user-specified values on the velocity-time curve.

The user may also work directly with the distance-time curve. For some users, this is the most intuitive approach. Assuming that a point starts at the beginning of the curve at $t = 0$ and traverses the curve to arrive at the end of the curve at $t = 1$, then the distance-time curve is constrained to start at (0, 0) and end at (1, 1). If the objective is to start and stop with zero velocity, then the slope at the start and end should be 0. The restriction that a point traveling along the curve may not back up any time during the traversal means that the distance-time curve must be monotonically increasing (i.e., always have a non-negative slope). If the point may not stop along the curve during the time interval, then there can be no horizontal sections in the distance-time curve (no internal zero slopes) (see Figure 3.20).

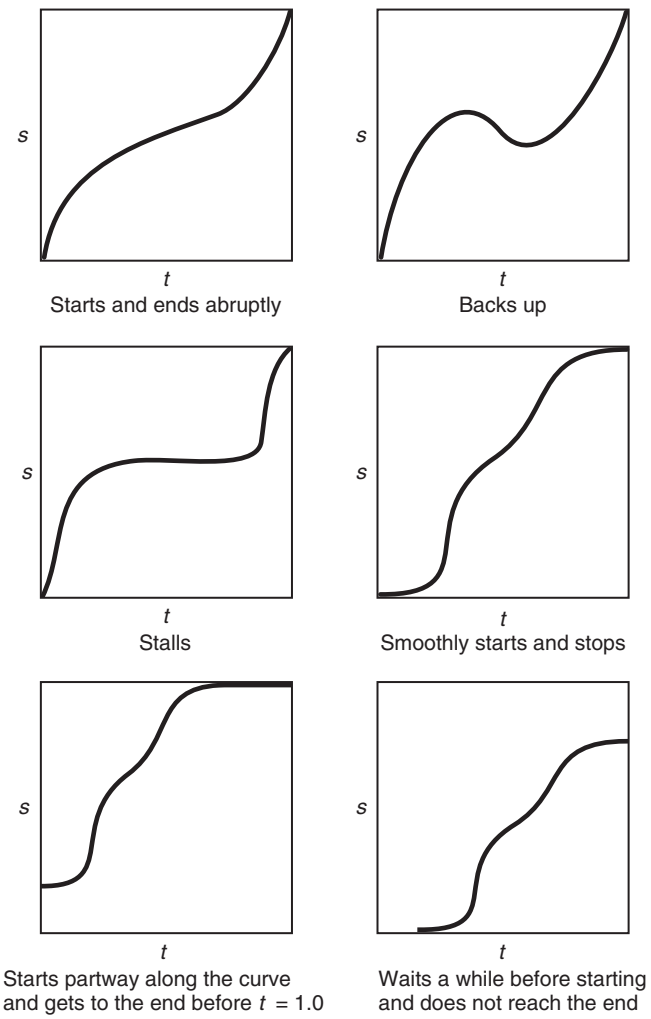


FIGURE 3.20

Sample distance-time functions.

As mentioned before, the space curve that defines the path along which the object moves is independent of the speed control curves that define the relative velocity along the path as a function of time. A single space curve could have several velocity-time curves defined along it. A single distance-time curve, such as a standard ease-in/ease-out function, could be applied to several different space curves. Reusing distance-time curves is facilitated if normalized distance and time are used.

Motion control frequently requires specifying positions and speeds along the space curve at specific times. An example might be specifying the motion of a hand as it reaches out to grasp an object; initially the hand accelerates toward the object and then, as it comes close, it slows down to almost zero speed before picking up the object. The motion is specified as a sequence of constraints on time, position (in this case, arc length traveled along a space curve), velocity, and acceleration. Stating the problem more formally, each point to be constrained is an n-tuple,$<t_i, s_i, v_i, a_i, \ldots,>$, where s_i is position, v_i is velocity, a_i is acceleration, and t_i is the time at which all the constraints must be satisfied (the ellipses, . . ., indicate that higher order derivatives may be constrained). Define the zero-order constraint problem to be that of satisfying sets of two-tuples, $<t_i, s_i>$, while velocity, acceleration, and so on are allowed to take on any values necessary to meet the position constraints at the specified times. Zero-order constrained motion is illustrated at the top of Figure 3.21. Notice that there is continuity of position but not of speed. By extension, the first-order constraint problem requires satisfying sets of three-tuples, $<s_i, v_i, t_i>$, as shown in the bottom illustration in Figure 3.21. Standard interpolation techniques (see Appendix B.5) can be used to aid the user in generating distance-time curves.

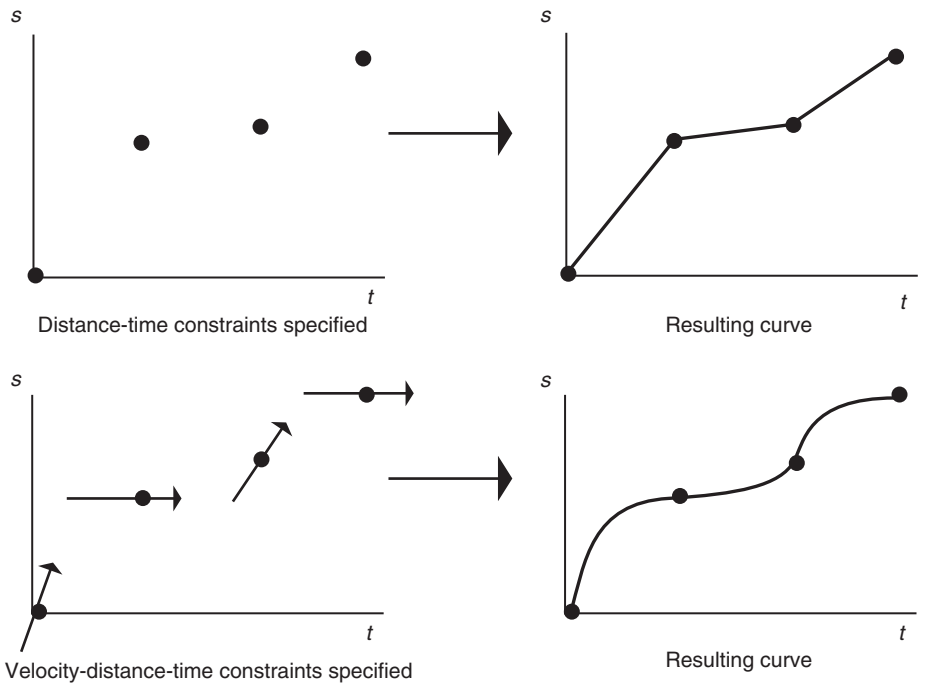


FIGURE 3.21

Specifying motion constraints.

3.2.5 Curve fitting to position-time pairs

If the animator has specific positional constraints that must be met at specific times, then the time-parameterized space curve can be determined directly. Position-time pairs can be specified by the animator, as in Figure 3.22, and the control points of the curve that produce an interpolating curve can be computed from this information [6].

For example, consider the case of fitting a B-spline[3] curve to values of the form (P_i, t_i), $i = 1, \ldots, j$. Given the form of B-spline curves shown in Equation 3.23 of degree k with $n + 1$ defining control vertices, and expanding in terms of the j given constraints $(2 \leq k \leq n + 1 \leq j)$, Equation 3.24 results. Put in matrix form, it becomes Equation 3.25, in which the given points are in the column matrix P, the unknown defining control vertices are in the column matrix B, and N is the matrix of basis functions evaluated at the given times $(t_1, t_2, \ldots, t_j)$.

$$P(t) = \sum_{i=1}^{n+1} B_i N_{i,k}(t) \tag{3.23}$$

$$\begin{aligned} P_1 &= N_{1,k}(t_1)B_1 + N_{2,k}(t_1)B_2 + \ldots + N_{n+1,k}(t_1)B_{n+1} \\ P_2 &= N_{1,k}(t_2)B_1 + N_{2,k}(t_2)B_2 + \ldots + N_{n+1,k}(t_2)B_{n+1} \\ &\ldots \\ P_j &= N_{1,k}(t_j)B_1 + N_{2,k}(t_j)B_2 + \ldots + N_{n+1,k}(t_j)B_{n+1} \end{aligned} \tag{3.24}$$

$$P = NB \tag{3.25}$$

If there are the same number of given data points as there are unknown control points $(2 \leq k \leq n + 1 = j)$, then N is square and the defining control vertices can be solved by inverting the matrix, as in Equation 3.26.

$$B = N^{-1}P \tag{3.26}$$

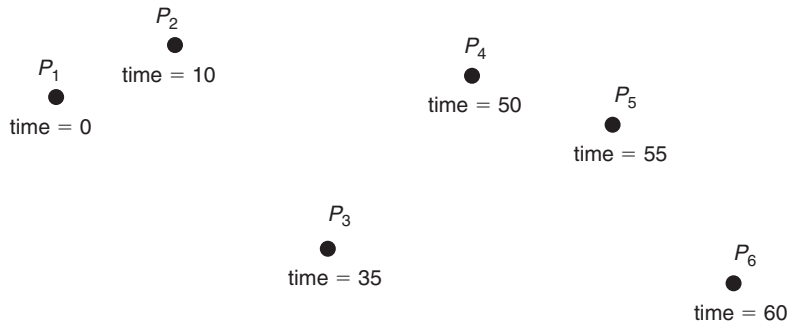


FIGURE 3.22

Position-time constraints.

[3]Refer to Appendix B.5.12 for more information on B-spline curves.

The resulting curve is smooth but can sometimes produce unwanted wiggles. Specifying fewer control points ($2 \leq k \leq n+1 < j$) can remove these wiggles but N is no longer square. To solve the matrix equation, the use of the pseudoinverse of N is illustrated in Equation 3.27, but in practice a more numerically stable and robust algorithm for solving linear least-squares problems should be used (e.g., see [1]).

$$\begin{aligned} P &= NB \\ N^T P &= N^T NB \\ [N^T N]^{-1} N^T P &= B \end{aligned} \tag{3.27}$$

3.3 Interpolation of orientations

The previous section discussed the interpolation of position through space and time. However, when dealing with objects, animators are concerned not only with the object's position, but also with the object's orientation. An object's orientation can be interpolated just like its position can be interpolated and the same issues apply. Instead of interpolating x, y, z positional values, an object's orientation parameters are interpolated. Just as an object's position is changed over time by applying translation transformations, an object's orientation is changed over time by applying rotational transformations. As discussed in the previous chapter, quaternions are a very useful representation for orientation and they will be the focus here.

3.3.1 Interpolating quaternions

One of the most important reasons for choosing quaternions to represent orientation is that they can be easily interpolated. Most important, the quaternion representation avoids the condition known as *gimbal lock*, which can trouble the other commonly used representations such as fixed angles and Euler angles.

Because the magnitude of a quaternion does not affect the orientation it represents, unit quaternions are typically used as the canonical representation for an orientation. Unit quaternions can be considered as points on a four-dimensional unit sphere.

Given two orientations represented by unit quaternions, intermediate orientations can be produced by linearly interpolating the individual quantities of the quaternion four-tuples. These orientations can be viewed as four-dimensional points on a straight-line path from the first quaternion to the second quaternion. While equal-interval, linear interpolation between the two quaternions will produce orientations that are reasonably seen as between the two orientations to be interpolated, it will not produce a constant speed rotation. This is because the unit quaternions to which the intermediate orientations map will not create equally spaced intervals on the unit sphere. Figure 3.23 shows the analogous effect when interpolating a straight-line path between points on a two-dimensional circle.

Intermediate orientations representing constant-speed rotation can be calculated by interpolating directly on the surface of the unit sphere, specifically along the great arc between the two quaternion points. In the previous chapter, it was pointed out that a quaternion, $[s, v]$, and its negation, $[-s, -v]$, represent the same orientation. That means that interpolation from one orientation, represented by the quaternion q_1, to another orientation, represented by the quaternion q_2, can also be carried out from q_1 to $-q_2$. The difference is that one interpolation path will be longer than the other. Usually, the shorter path is the more desirable because it represents the more direct way to get from one orientation to the other. The shorter path is the one indicated by the smaller angle between the four-dimensional

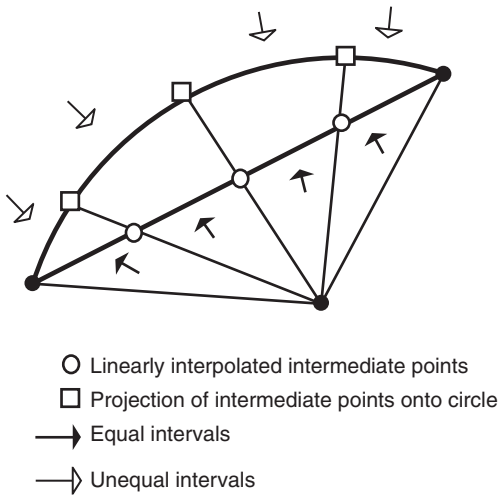


FIGURE 3.23

Equally spaced linear interpolations of straight-line path between two points on a circle generate unequal spacing of points after projecting onto a circle. Interpolating quaternion four-tuples exhibit the same problem.

quaternion vectors. This can be determined by using the four-dimensional dot product of the quaternions to compute the cosine of the angle between q_1 and q_2 (Eq. 3.28). If the cosine is positive, then the path from q_1 to q_2 is shorter; otherwise the path from q_1 to $-q_2$ is shorter (Figure 3.24).

$$\cos(\theta) = q_1 \cdot q_2 = s_1 s_2 + v_1 \cdot v_2 \tag{3.28}$$

The formula for spherical linear interpolation (*slerp*) between unit quaternions q_1 and q_2 with parameter u varying from 0 to 1 is given in Equation 3.29, where $q_1 \cdot q_2 = \cos(\theta)$. Notice that this does

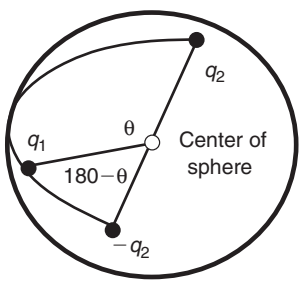


FIGURE 3.24

The closer of the two representations of orientation, q_2, is the better choice to use in interpolation. In this case $-q_2$ is closer to q_1 than q_2. The angle between q_1 and q_2, as measured from the center of the sphere, is computed by taking the dot product of q_1 and q_2. The same is done for $-q_2$.

not necessarily produce a unit quaternion, so the result must be normalized if a unit quaternion is desired.

$$slerp(q_1, q_2, u) = \frac{\sin((1-u)\theta)}{sin(\theta)} q_1 + \frac{\sin(u\theta)}{sin(\theta)} q_2 \tag{3.29}$$

Notice that in the case $u = 1/2$, $slerp(q_1, q_2, u)$ can be easily computed within a scale factor, as $q_1 + q_2$, which can then be normalized to produce a unit quaternion.

When interpolating between a series of orientations, slerping between points on a spherical surface has the same problem as linear interpolation between points in Euclidean space: that of first-order discontinuity (see Appendix B.5). Shoemake [7] suggests using cubic Bezier interpolation to smooth the interpolation between orientations. Reasonable interior control points are automatically calculated to define cubic segments between each pair of orientations.

To introduce this technique, assume for now that there is a need to interpolate between a sequence of two-dimensional points, $[\ldots, p_{n-1}, p_n, p_{n+1}, \ldots]$; these will be referred to as the *interpolated points*. (How to consider the calculations using quaternion representations is discussed below.) Between each pair of points, two control points will be constructed. For each of the interpolation points, p_n, two control points will be associated with it: the one immediately before it, b_n, and the one immediately after it, a_n.

To calculate the control point following any particular point p_n, take the vector defined by p_{n-1} to p_n and add it to the point p_n. Now, take this point (marked "1" in Figure 3.25) and find the average of it and p_{n+1}. This becomes one of the control points (marked "a_n" in Figure 3.25).

Next, take the vector defined by a_n to p_n and add it to p_n to get b_n (Figure 3.26). Points b_n and a_n are the control points immediately before and after the point p_n. This construction ensures first-order

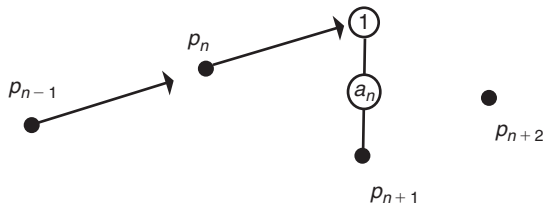


FIGURE 3.25

Finding the control point after p_n: to p_n, add the vector from p_{n-1} to p_n and average that with p_{n+1}.

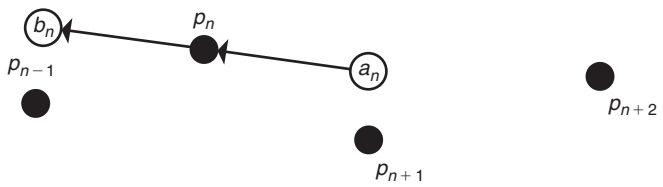


FIGURE 3.26

Finding the control point before p_n: to p_n, add the vector from a_n to p_n.

continuity at the junction of adjacent curve segments (in this case, p_n) because the control points on either side of the point are colinear with, and equidistant to, the control point itself.

The end conditions can be handled by a similar construction. For example, the first control point, a_0, is constructed as the vector from the third interpolated point to the second point ($p_1 - p_2$) is added to the second point (Figure 3.27).

In forming a control point, the quality of the interpolated curve can be affected by adjusting the distance the control point is from its associated interpolated point while maintaining its direction from that interpolated point. For example, a new b_n' can be computed with a user-specified constant, k, as in Equation 3.30 (and optionally computing a corresponding new a_n').

$$b_n' = p_n + k(b_n - p_n) \tag{3.30}$$

Between any two interpolated points, a cubic Bezier curve segment is then defined by the points p_n, a_n, b_{n+1}, and p_{n+1}. The control point b_{n+1} is defined in exactly the same way that b_n is defined except for using p_n, p_{n+1}, and p_{n+2}. The cubic curve segment is then generated between p_n and p_{n+1} (see Figure 3.28).

It should be easy to see how this procedure can be converted into the four-dimensional spherical world of quaternions. Instead of adding vectors, rotations are concatenated using quaternion multiplication. Averaging of orientations can easily be done by slerping to the halfway orientation, which is implemented by adding quaternions (and optionally normalizing). Adding the difference between two orientations, represented as unit quaternions p and q, to the second orientation, q, can be affected by constructing the representation of the result, r, on the four-dimensional unit sphere using Equation 3.31. See Appendix B.3.4 for further discussion.

$$\mathrm{r} = 2\,(\mathrm{p}.\mathrm{q})\,\mathrm{q} - \mathrm{p} \tag{3.31}$$

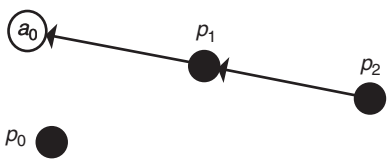


FIGURE 3.27

Constructing the first interior control point.

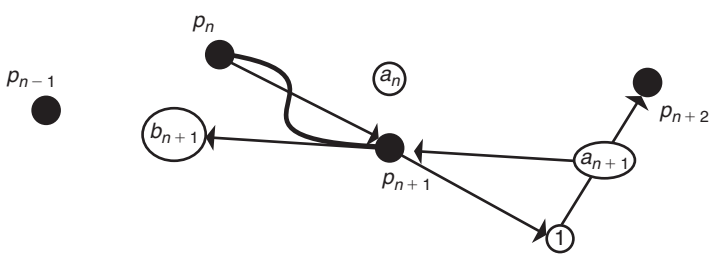


FIGURE 3.28

Construction of b_{n+1} and the resulting curve.

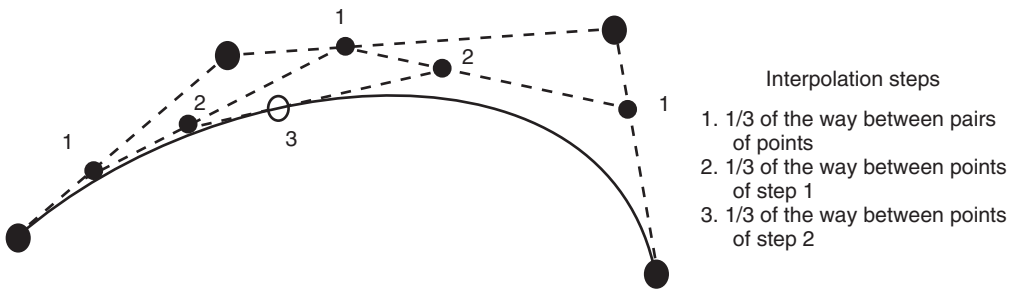


FIGURE 3.29

de Casteljau construction of point on cubic Bezier segment at 1/3 (the point labeled "3").

Once the internal control points are computed, the de Casteljau algorithm can be applied to interpolate points along the curve. An example of the de Casteljau construction procedure in the two-dimensional Euclidean case of Bezier curve interpolation is shown in Figure 3.29. See Appendix B.5.10 for a more complete discussion of the procedure.

The same procedure can be used to construct the Bezier curve in four-dimensional spherical space. For example, to obtain an orientation corresponding to the $u = 1/3$ position along the curve, the following orientations are computed:

$$
\begin{aligned}
p1 &= \mathrm{slerp}(q_n, a_n, 1/3)\\
p2 &= \mathrm{slerp}(a_n, b_{n+1}, 1/3)\\
p3 &= \mathrm{slerp}(b_{n11}, q_{n11}, 1/3)\\
p12 &= \mathrm{slerp}(p_1, p_2, 1/3)\\
p23 &= \mathrm{slerp}(p_2, p_3, 1/3)\\
p &= \mathrm{slerp}(p12, p23, 1/3)
\end{aligned}
$$

where p is the quaternion representing an orientation 1/3 along the spherical cubic spline.

The procedure can be made especially efficient in the case of quaternion representations when calculating points at positions along the curve corresponding to u values that are powers of 1/2. For example, consider calculating a point at $u = 1/4$.

$$
\begin{aligned}
&\mathrm{temp} = \mathrm{slerp}(q_n, a_n, 1/2) = (q_n + a_n)/||q_n + a_n||\\
&p_1 = \mathrm{slerp}(q_n, \mathrm{temp}, 1/2) = (q_n + \mathrm{temp})/||q_n + \mathrm{temp}||\\
&\mathrm{temp} = \mathrm{slerp}(a_n, b_{n11}, 1/2) = (a_n + b_{n11})/||\, a_n + b_{n11}||\\
&p_2 = \mathrm{slerp}(a_n, \mathrm{temp}, 1/2) = (a_n + \mathrm{temp})/||\, a_n + \mathrm{temp}||\\
&\mathrm{temp} = \mathrm{slerp}(b_{n11}, q_{n11}, 1/2) = (b_{n+1} + q_{n11})/||\, b_{n+1} + q_{n11}||\\
&p_3 = \mathrm{slerp}(b_{n11}, \mathrm{temp}, 1/2) = (b_{n+1} + \mathrm{temp})/||\, b_{n+1} + \mathrm{temp}\,||\\
&\mathrm{temp} = \mathrm{slerp}(p_1, p_2, 1/2) = (p_1 + p_2)/||\, p_1 + p_2||\\
&p_{12} = \mathrm{slerp}(p_1, \mathrm{temp}, 1/2) = (p_1 + \mathrm{temp})/||\, p_1 + \mathrm{temp}\,||\\
&\mathrm{temp} = \mathrm{slerp}(p_2, p_3, 1/2) = (p_2 + p_3)/||\, p_2 + p_3||\\
&p_{23} = \mathrm{slerp}(p_2, \mathrm{temp}, 1/2) = (p_2 + \mathrm{temp})/||\, p_2 + \mathrm{temp}\,||\\
&\mathrm{temp} = \mathrm{slerp}(p_{12}, p_{23}, 1/2) = (p_{12} + p_{23})/||\, p_{12} + p_{23}||\\
&p = \mathrm{slerp}(p_{12}, \mathrm{temp}, 1/2) = (p_{12} + \mathrm{temp})/||\, p_{12} + \mathrm{temp}\,||
\end{aligned}
$$

The procedure can be made more efficient if the points are generated in order according to binary subdivision (1/2, 1/4, 3/4, 1/8, 3/8, 5/8, 7/8, 1/16, ...) and temporary values are saved for use in subsequent calculations.

3.4 Working with paths

3.4.1 Path following

Animating an object by moving it along a path is a very common technique and usually one of the simplest to implement. As with most other types of animation, however, issues may arise that make the task more complicated than first envisioned. An object (or camera) following a path requires more than just translating along a space curve, even if the curve is parameterized by arc length and the speed is controlled using ease-in/ease-out. Changing the orientation of the object also has to be taken into consideration. If the path is the result of a digitization process, then often it must be smoothed before it can be used. If the path is constrained to lie on the surface of another object or needs to avoid other objects, then more computation is involved. These issues are discussed below.

3.4.2 Orientation along a path

Typically, a local coordinate system (u, v, w) is defined for an object to be animated. In this discussion, a right-handed coordinate system is assumed, with the origin of the coordinate system determined by a point along the path $\boldsymbol{P}(s)$. As previously discussed, this point is generated based on the frame number, arc length parameterization, and possibly ease-in/ease-out control. This position will be referred to as POS. The direction the object is facing is identified with the w-axis, the up vector is identified with the v-axis, and the local u-axis is perpendicular to these two, completing the coordinate system. To form a right-handed coordinate system, for example, the u-axis points to the left of the object as someone at the object's position (POS) looks down the w-axis with the head aligned with the v-axis (Figure 3.30).

There are various ways to handle the orientation of the camera as it travels along a path. Of course, which method to use depends on the desired effect of the animation. The orientation is specified by determining the direction it is pointing (the w-axis) and the direction of its up vector (the v-axis); the u-axis is then fully specified by completing the left-handed camera coordinate system (eye space).

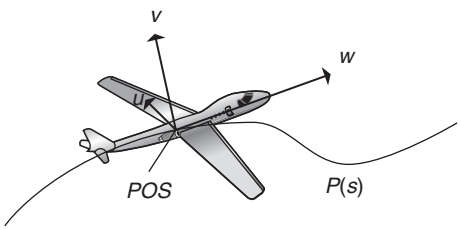


FIGURE 3.30

Object-based, right-handed, local coordinate system.

Use of the Frenet frame

If an object or camera is following a path, then its orientation can be made directly dependent on the properties of the curve. The Frenet frame[4] can be defined along the curve as a moving coordinate system, (u, v, w), determined by the curve's tangent and curvature. The Frenet frame changes orientation over the length of the curve. It is defined as normalized orthogonal vectors with w in the direction of the first derivative ($\boldsymbol{P}'(s)$), v orthogonal to w and in the general direction of the second derivative ($\boldsymbol{P}''(s)$), and u formed to complete, for example, a left-handed coordinate system as computed in a right-handed space (Figure 3.31). Specifically, at a given parameter value s, the left-handed Frenet frame is calculated according to Equation 3.32, as illustrated in Figure 3.32. The vectors are then normalized.

$$\begin{aligned} \mathrm{w} &= \mathrm{P}'(\mathrm{s}) \\ \mathrm{u} &= \mathrm{P}'(\mathrm{s}) \times \mathrm{P}''(\mathrm{s}) \\ \mathrm{v} &= \mathrm{u} \times \mathrm{w} \end{aligned} \tag{3.32}$$

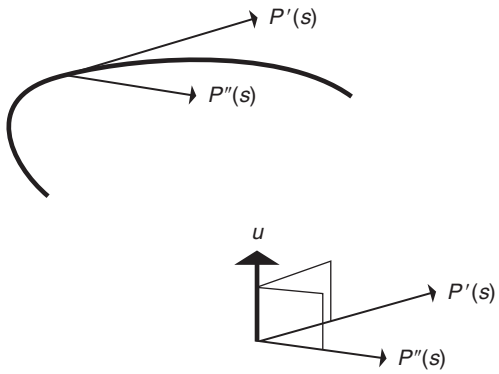


FIGURE 3.31

The derivatives at a point along the curve are used to form the u vector.

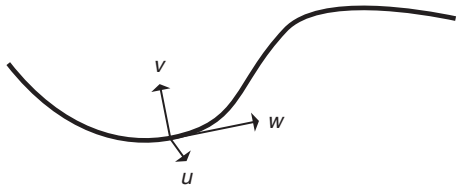


FIGURE 3.32

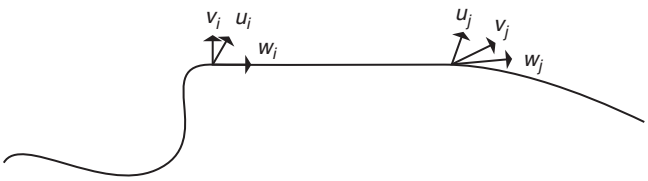
Left-handed Frenet frame at a point along a curve.

[4]Note the potential source of confusion between the use of the term frame to mean (1) a frame of animation and (2) the moving coordinate system of the Frenet frame. The context in which the term frame is used should determine its meaning.

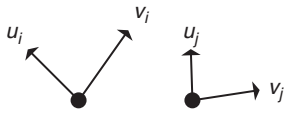
While the Frenet frame provides useful information about the curve, there are a few problems with using it directly to control the orientation of a camera or object as it moves along a curve. First, there is no concept of "up" inherent in the formation of the Frenet frame. The *v* vector merely lines up with the direction of the second derivative. Another problem occurs in segments of the curve that have no curvature ($\boldsymbol{P}''(u) = 0$) because the Frenet frame is undefined. These undefined segments can be dealt with by interpolating a Frenet frame along the segment from the Frenet frames at the boundary of the segment. By definition, there is no curvature along this segment, so the boundary Frenet frames must differ by only a rotation around *w*. Assuming that the vectors have already been normalized, the angular difference between the two can be determined by taking the arccosine of the dot product between the two *v* vectors so that $\theta = \text{acos}(v_1 \cdot v_2)$. This rotation can be linearly interpolated along the no-curvature segment (Figure 3.33).

The problem is more difficult to deal with when there is a discontinuity in the curvature vector as is common with piecewise cubic curves. Consider, for example, a curve composed of two semicircular segments positioned so that they form an **S** (*sigmoidal*) shape. The curvature vector, which for any point along this curve will point to the center of the semicircle that the point is on, will instantaneously switch from pointing to one center point to pointing to the other center point at the junction of the two semicircular segments. In this case, the Frenet frame is defined everywhere but has a discontinuous jump in orientation at the junction (see Figure 3.34).

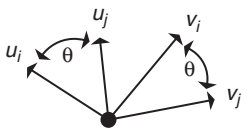
However, the main problem with using the Frenet frame as the local coordinate frame to define the orientation of the camera or object following the path is that the resulting motions are usually too extreme and not natural looking. Using the *w*-axis (tangent vector) as the view direction of a camera can be undesirable. Often, the tangent vector does not appear to correspond to the direction of "where



Frenet frames on the boundary of an undefined Frenet frame segment because of zero curvature



The two frames sighted down the (common) *w* vector



The two frames superimposed to identify angular difference

FIGURE 3.33

Interpolating Frenet frames to determine the undefined segment.

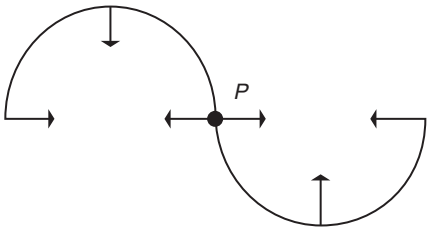


FIGURE 3.34

The curvature vector, defined everywhere but discontinuous, instantaneously switches direction at point *P*.

it's going" even though it is in the instantaneous (analytic) sense. The more natural orientation, for someone riding in a car or riding a bike, for example, would be to look farther ahead along the curve rather than to look tangential to the current location along the curve.

In addition, if the v-axis is equated with the up vector of an object, the object will rotate wildly about the path even when the path appears to mildly meander through an environment. With three-dimensional space curves, the problem becomes more obvious as the path bends up and down in space; the camera will flip between being upright and traveling along the path upside down. While the Frenet frame provides useful information to the animator, its direct use to control object orientation is clearly of limited value. When modeling the motion of banking into a turn, the curvature vector (u-axis) does indicate which way to bank and can be used to control the magnitude of the bank. For example, the horizontal component (the component in the x-z plane) of the u-axis can be used as the direction/magnitude indicator for the bank. For a different effect, the animator may want the object to tilt away from the curvature vector to give the impression of the object feeling a force that throws it off the path, such as a person riding a roller coaster.

Camera path following

The simplest method of specifying the orientation of a camera is to set its center of interest (COI) to a fixed point in the environment or, more elegantly, to use the center point of one of the objects in the environment. In either case, the COI is directly determined and is available for calculating the view vector, $w = COI - POS$. This is usually a good method when the path that the camera is following is one that circles some arena of action on which the camera's attention must be focused.

This still leaves one degree of freedom to be determined in order to fully specify the local coordinate system. For now, assume that the up vector, v, is to be kept "up." *Up* in this case means "generally in the positive y-axis direction," or, for a more mathematical definition, the v-axis is at right angles to the view vector (w-axis) and is to lie in the plane defined by the local w-axis and the global y-axis. The local coordinate system can be computed as shown in Equation 3.33 for a left-handed camera coordinate-system computed in right-handed space. The vectors can then be normalized to produce unit length vectors.

$$\begin{aligned} w &= COI - POS \\ u &= w \times (0,1,0) \\ v &= u \times w \end{aligned} \tag{3.33}$$

For a camera traveling down a path, the view direction can be automatically set in several ways. As previously mentioned, the COI can be set to a specific point in the environment or to the position of a

specific object in the environment. This works well as long as there is a single point or single object on which the camera should focus and as long as the camera does not pass too close to the point or object. Passing close to the COI will result in radical changes in view direction (in some cases, the resulting effect may be desirable). Other methods use points along the path itself, a separate path through the environment, or interpolation between positions in the environment. The up vector can also be set in several ways. The default orientation is for the up vector to lie in the plane of the view vector and the global y-axis. Alternatively, a tilt of the up vector can be specified away from the default orientation as a user-specified value (or interpolated set of values). And, finally, the up vector can be explicitly specified by the user.

The simplest method of setting the view vector is to use a delta parametric value to define the COI. If the position of the camera on a curve is defined by $\boldsymbol{P}(s)$, then the COI will be $\boldsymbol{P}(s + \Delta s)$. This, of course, should be after reparameterization by arc length. Otherwise, the actual distance to the COI along the path will vary over time (although if a relatively large Δu is used, the variation may not be noticeable). At the end of the curve, once $s + \Delta s$ is beyond the path parameterization, the view direction can be interpolated to the end tangent vector as s approaches 1 (in the case that distance is normalized).

Often, updating the COI to be a specific point along the curve can result in views that appear jerky. In such cases, averaging some number of positions along the curve to be the COI point can smooth the view. However, if the number of points used is too small or the points are too close together, the view may remain jerky. If n is too large and the points are spaced out too much, the view direction may not change significantly during the traversal of the path and will appear too static. The number of points to use and their spread along the curve are dependent on the path itself and on the effect desired by the animator.

An alternative to using some function of the position path to produce the COI is to use a separate path altogether to define the COI. In this case, the camera's position is specified by $\boldsymbol{P}(s)$, while the COI is specified by some $\boldsymbol{C}(s)$. This requires more work by the animator but provides greater control and more flexibility.

Similarly, an up vector path, $\boldsymbol{U}(s)$, might be specified so that the general up direction is defined by $\boldsymbol{U}(s) - \boldsymbol{P}(s)$. This is just the general direction because a valid up vector must be perpendicular to the view vector. Thus, the coordinate frame for the camera could be defined as in Equation 3.34.

$$\begin{aligned} w &= C(s) - P(s) \\ u &= w \times (U(s) - P(s)) \\ v &= u \times w \end{aligned} \tag{3.34}$$

Instead of using a separate path for the COI, a simple but effective strategy is to fix it at one location for an interval of time and then move it to another location (using linear spatial interpolation and ease-in/ease-out temporal interpolation) and fix it there for a number of frames, and so on. The up vector can be set as before in the default "up" direction.

3.4.3 Smoothing a path

For cases in which the points making up a path are generated by a digitizing process, the resulting curve can be too jerky because of noise or imprecision. To remove the jerkiness, the coordinate values of the data can be smoothed by one of several approaches. For this discussion, the following set of data will be used: {(1, 1.6), (2, 1.65), (3, 1.6), (4, 1.8), (5, 2.1), (6, 2.2), (7, 2.0), (8, 1.5), (9, 1.3), (10, 1.4)}. This is plotted in Figure 3.35.

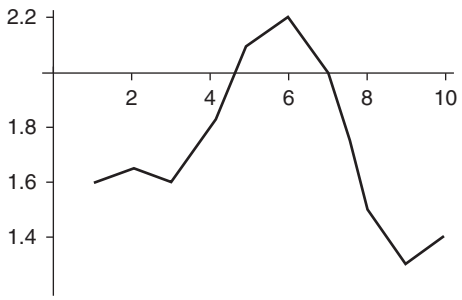


FIGURE 3.35

Sample data for path smoothing.

Smoothing with linear interpolation of adjacent values

An ordered set of points in two-space can be smoothed by averaging adjacent points. In the simplest case, the two points, one on either side of an original point, P_i, are averaged. This point is averaged with the original data point (Eq. 3.35). Figure 3.36 shows the sample data plotted with the original data. Notice how linear interpolation tends to draw the data points in the direction of local concavities. Repeated applications of the linear interpolation to further smooth the data would continue to draw the reduced concave sections and flatten out the curve.

$$P_i' = \frac{P_i + \frac{P_{i-1}+P_{i+1}}{2}}{2} = \frac{1}{4}P_{i-1} + \frac{1}{2}P_i + \frac{1}{4}P_{i+1} \tag{3.35}$$

Smoothing with cubic interpolation of adjacent values

To preserve the curvature but still smooth the data, the adjacent points on either side of a data point can be used to fit a cubic curve that is then evaluated at its midpoint. This midpoint is then averaged with the original point, as in the linear case. A cubic curve has the form shown in Equation 3.36. The two data points on either side of an original point, P_i, are used as constraints, as shown in Equation 3.37. These equations can be used to solve for the constants of the cubic curve (a, b, c, d). Equation 3.36 is then

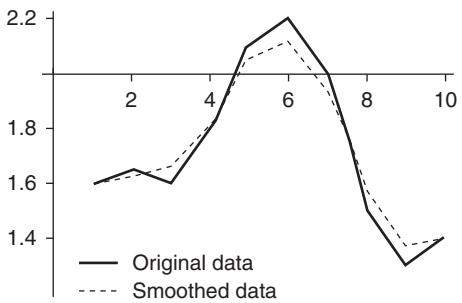


FIGURE 3.36

Sample data smoothed by linear interpolation.

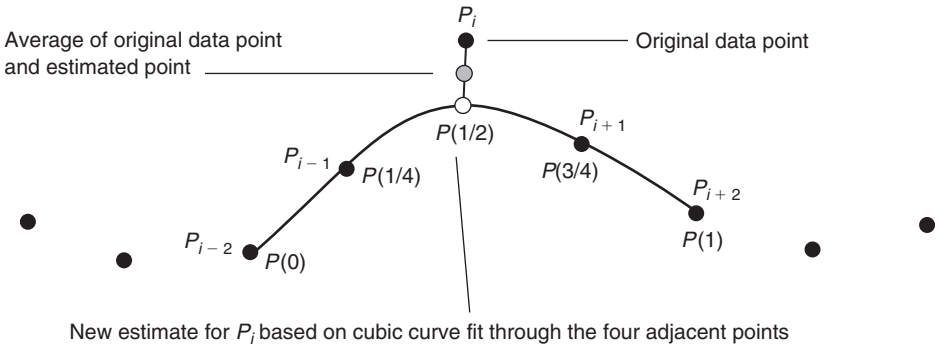


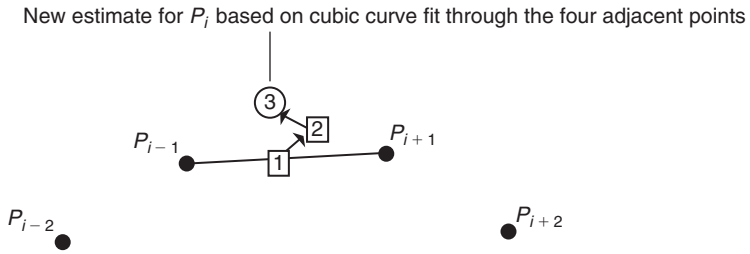
FIGURE 3.37

Smoothing data by cubic interpolation.

evaluated at $u = 1/2$; and the result is averaged with the original data point (see Figure 3.37). Solving for the coefficients and evaluating the resulting cubic curve is a bit tedious, but the solution needs to be performed only once and can be put in terms of the original data points, $P_{i\text{-}2}, P_{i\text{-}1}, P_{i+1}$, and P_{i+2}. This is illustrated in Figure 3.38.

$$\boldsymbol{P}(u) = au^3 + bu^2 + cu + d \tag{3.36}$$

$$\begin{aligned} P_{i-2} &= \boldsymbol{P}(0) = d \\ P_{i-1} &= \boldsymbol{P}(1/4) = a\frac{1}{64} + b\frac{1}{16} + c\frac{1}{4} + d \\ P_{i+1} &= \boldsymbol{P}(3/4) = a\frac{27}{64} + b\frac{9}{16} + c\frac{3}{4} + d \\ P_{i+2} &= a + b + c + d \end{aligned} \tag{3.37}$$



1. Average P_{i-1} and P_{i+1}
2. Add 1/6 of the vector from P_{i-2} *to* P_{i-1}
3. Add 1/6 of the vector from P_{i+2} to P_{i+1} to get new estimated point
4. (Not shown) Average estimated point with original data point

FIGURE 3.38

Geometric construction of a cubic estimate for smoothing a data point.

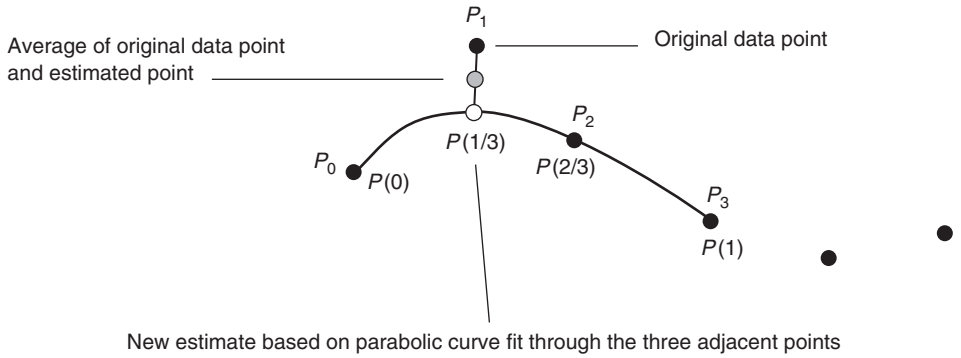


FIGURE 3.39

Smoothing data by parabolic interpolation.

For the end conditions, a parabolic arc can be fit through the first, third, and fourth points, and an estimate for the second point from the start of the data set can be computed (Figure 3.39). The coefficients of the parabolic equation, $\boldsymbol{P}(u) = au^2 + bu + c$, can be computed from the constraints in Equation 3.38, and the equation can be used to solve for the position $P_1 = \boldsymbol{P}(1/3)$.

$$P_0 = \boldsymbol{P}(0),\ P_2 = \boldsymbol{P}\left(\frac{2}{3}\right),\ P_3 = \boldsymbol{P}(1) \tag{3.38}$$

This can be rewritten in geometric form and the point can be constructed geometrically from the three points $P_1^{'} = P_2 + (1/3)\ (P_0 - P_3)$ (Figure 3.40). A similar procedure can be used to estimate the data point second from the end. The very first and very last data points can be left alone if they represent hard constraints, or parabolic interpolation can be used to generate estimates for them as well, for

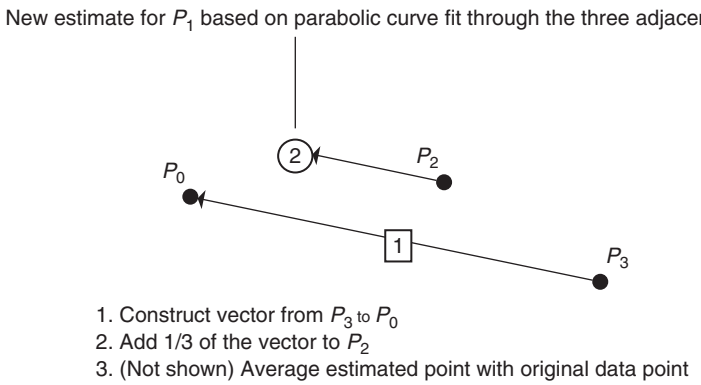


FIGURE 3.40

Geometric construction of a parabolic estimate for smoothing a data point.

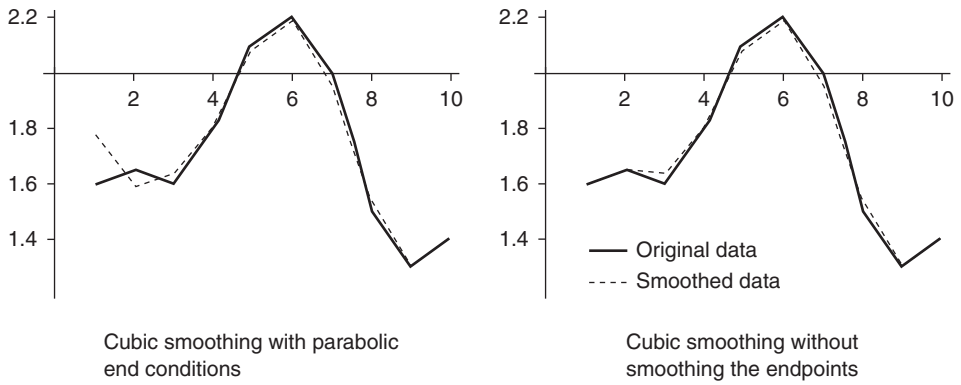


FIGURE 3.41

Sample data smoothed with cubic interpolation.

example, $P_0' = P_3 + 3(P_1 - P_2)$. Figure 3.41 shows cubic interpolation to smooth the data with and without parabolic interpolation for the endpoints.

Smoothing with convolution kernels

When the data to be smoothed can be viewed as a value of a function, $y_i = f(x_i)$, the data can be smoothed by convolution. Figure 3.42 shows such a function where the x_i are equally spaced. A smoothing kernel can be applied to the data points by viewing them as a step function (Figure 3.43). Desirable attributes of a smoothing kernel include the following: it is centered around 0, it is symmetric, it has finite support, and the area under the kernel curve equals 1. Figure 3.44 shows examples of some possibilities. A new point is calculated by centering the kernel function at the position where the new point is to be computed. The new point is calculated by summing the area under the curve that results from multiplying the kernel function, $g(u)$, by the corresponding segment of the step function, $f(x)$, beneath

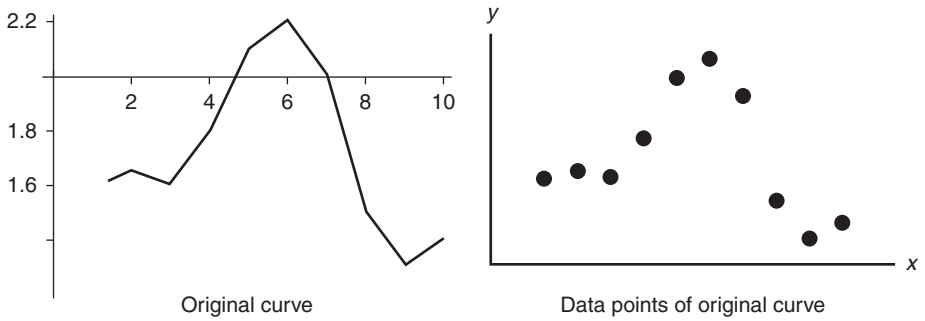


FIGURE 3.42

Sample function to be smoothed.

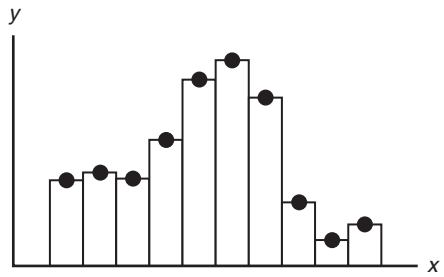


FIGURE 3.43

Step function defined by data points of original curve.

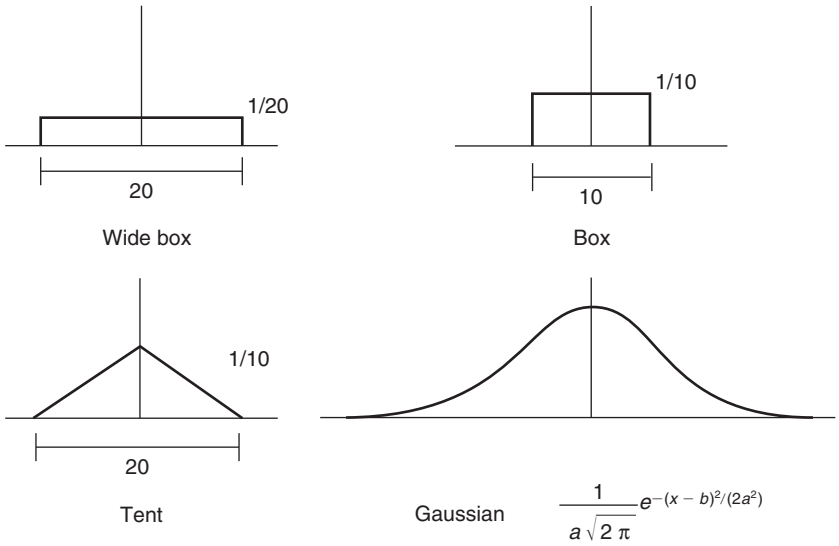


FIGURE 3.44

Sample smoothing kernels.

it (i.e., *convolution*). Figure 3.45 shows a simple tent-shaped kernel applied to a step function. In the continuous case, this becomes the integral, as shown in Equation 3.39, where $[-s, \ldots, s]$ is the extent of the support of the kernel function.

$$\boldsymbol{P}(x) = \int_{-s}^{s} f(x+u)g(u)du \tag{3.39}$$

The integral can be analytically computed or approximated by discrete means. This can be done either with or without averaging down the number of points making up the path. Additional points can also be

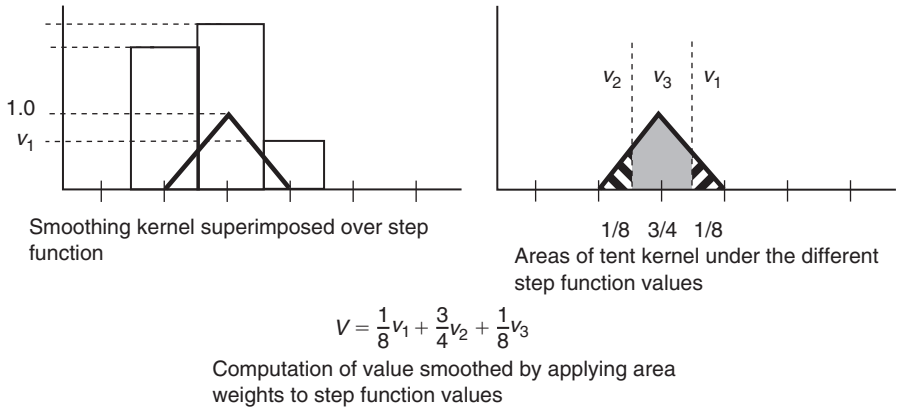


FIGURE 3.45

Example of a tent-shaped smoothing filter.

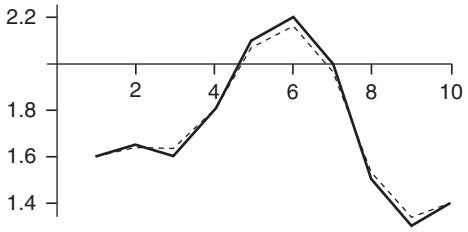


FIGURE 3.46

Sample data smoothed with convolution using a tent kernel.

interpolated. At the endpoints, the step function can be arbitrarily extended so as to cover the kernel function when centered over the endpoints. Often the first and last points must be fixed because of animation constraints, so care must be taken in processing these. Figure 3.45 shows how a tent kernel is used to average the step function data; Figure 3.46 shows the sample data smoothed with the tent kernel.

Smoothing by B-spline approximation

Finally, if an approximation to the curve is sufficient, then points can be selected from the curve, and, for example, B-spline control points can be generated based on the selected points. The curve can then be regenerated using the B-spline control points, which ensures that the regenerated curve is smooth even though it no longer passes through the original points.

3.4.4 Determining a path along a surface

If one object is to move across the surface of another object, then a path across the surface must be determined. If start and destination points are known, it can be computationally expensive to find the shortest path between the points. However, it is not often necessary to find the absolute shortest path. Various alternatives exist for determining suboptimal yet reasonably direct paths.

An easy way to determine a path along a polygonal surface mesh is to determine a plane that contains the two points and is generally perpendicular to the surface. *Generally perpendicular* can be defined, for example, as the plane containing the two vertices and the average of the two vertex normals that the path is being formed between. The intersection of the plane with the faces making up the surface mesh will define a path between the two points (Figure 3.47).

If the surface is a higher order surface and the known points are defined in the u, v coordinates of the surface definition, then a straight line (or curve) can be defined in the parametric space and transferred to the surface. A linear interpolation of u, v coordinates does not guarantee the most direct route, but it should produce a reasonably short path.

Over a complex surface mesh, a greedy-type algorithm can be used to construct a path of edges along a surface from a given start vertex to a given destination vertex. For each edge emanating from the current vertex (initially the start vertex), calculate the projection of the edge onto the unit vector from the current vertex to the destination vertex. Divide this distance by the length of the edge to get the cosine of the angle between the edge and the straight line. The edge with the largest cosine is the edge most in the direction of the straight line; choose this edge to add to the path. Keep applying this until the destination edge is reached. Improvements to this approach can be made by allowing the path to cut across polygons to arrive at points along edges rather than by going vertex to vertex. For example, using the current path point and the line from the start and end vertices to form a plane, intersect the plane with the plane of the current triangle. If the line of intersection falls inside of the current triangle, use the segment inside of the triangle as part of the path. If the line of intersection leaving the current vertex is outside of the triangle, find the adjacent triangle and use it as the current triangle, and repeat. For the initial plane, the start and end vertex can be used along with some point on the surface that is judged to lie in the general direction of the path that is yet to be determined.

If a path downhill from an initial point on the surface is desired, then the surface normal and global "up" vector can be used to determine the downhill vector. The cross-product of the normal and global up vector defines a vector that lies on the surface perpendicular to the downhill direction. So the cross-product of this vector and the normal vector define the downhill (and uphill) vector on a plane (Figure 3.48). This same approach works with curved surfaces to produce the instantaneous downhill vector.

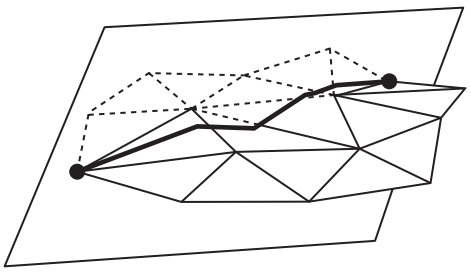


FIGURE 3.47

Determining a path along a polygonal surface mesh by using plane intersection.

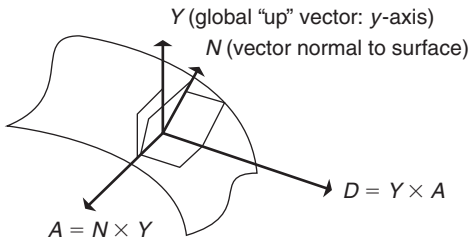


FIGURE 3.48

Calculating the downhill vector, *D*.

3.4.5 Path finding

Finding a collision-free path in an arbitrarily complex environment can be a complicated task. In the simple case, a point is to be moved through an otherwise stationary environment. If the obstacles are moving, but their paths are known beforehand, then the problem is manageable. If the movements of the obstructions are not known, then path finding can be very difficult. A further complication is when the object to be moved is not just a point, but has spatial extent. If the object has a nontrivial shape (i.e., not a sphere) and can be arbitrarily rotated as it moves, then the problem becomes even more interesting. A few ideas for path finding are mentioned here.

In finding a collision-free path among static obstacles, it is often useful to break the problem into simpler subproblems by introducing *via points*. These are points that the path should pass through—or at least educated guesses of points that the path might pass through. In this manner, finding the path is reduced to a number of simpler problems of finding paths between the via points.

If the size of the moving object might determine whether it can pass between obstacles, then finding a collision-free path can be a problem. However, if the moving object can be approximated by a sphere, then increasing the size of the obstacles by the size of the approximating sphere changes the problem into one of finding the path of a point through the expanded environment. More complex cases, in which the moving object has to adjust its orientation in order to pass by obstacles, are addressed in the robotics literature, for example.

Situations in which the obstacles are moving present significant challenges. A path through the static objects can be established and then movement along this path can be used to avoid moving obstacles, although there is no guarantee that a successful movement can be found. If the environment is not cluttered with moving obstacles, then a greedy-type algorithm can be used to avoid one obstacle at a time, by angling in front of the moving obstacle or angling behind it, in order to produce a path. This type of approach assumes that the motion of the obstacles, although unknown, is predictable.

3.5 Chapter summary

Interpolation is fundamental to much animation, and the ability to understand and control the interpolation process is very important in computer animation programming. Interpolation of values takes many forms, including arc length, image pixel color, and shape parameters. The control of the interpolation process can be key framed, scripted, or analytically determined. But in any case, interpolation forms the foundation upon which most computer animation takes place, including those animations that use advanced algorithms.

References

[1] Burden R, Faires J. Numerical Analysis. 8th ed. Brooks-Cole; 2005.
[2] Guenter B, Parent R. Computing the Arc Length of Parametric Curves. IEEE Comput Graph Appl May 1990; 10(3):72–8.
[3] Judd C, Hartley P. Parametrization and Shape of B-Spline Curves for CAD. Computer Aided Design September 1980; 12(5):235–8.
[4] Litwinowicz P, Williams L. Animating Images with Drawings. In: Glassner A, editor. Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series (July 1994, Orlando, Fla.). Orlando, Florida: ACM Press. p. 409–12. ISBN 0-89791-667-0.
[5] Mortenson M. Geometric Modeling. New York: John Wiley & Sons; 1985.
[6] Rogers D, Adams J. Mathematical Elements for Computer Graphics. New York: McGraw-Hill; 1976.
[7] Shoemake K. Animating Rotation with Quaternion Curves. In: Barsky BA (Ed.), Computer Graphics (Proceedings of SIGGRAPH 85) (August 1985, San Francisco, Calif.), vol. 19(3). p. 143–52.