19
Computer Animation

Webster defines "animate" as "to give life to; to make alive." This is precisely what we feel when we watch a well-made piece of animation, and this is the reason why traditional animation has always been popular and why computer animation is such a successful field. In fact, computer graphics has developed over the years in three stages. The first stage was to display a single image consisting of smooth, curved, realistic-looking surfaces. The second stage was to create and display an entire animation made of many *frames*, where each frame is an image. The third stage is *virtual reality*, where the user can interact with the animation.

> There is increasing interaction between images and language. One might say that living in society today is almost like living in a vast comic strip.
>
> —Jean-Luc Godard (as Narrator) in *Deux ou trois choses que je sais d'elle* (1966).

19.1 Background

Animation is based on the way our eye and brain work. If the eye is presented with a slow sequence of images, the brain interprets them as separate. If the images are speeded up more and more, the brain starts interpreting them first as motion with flickers, then as continuous motion, and finally as a blur. The physiological property that allows our eye and brain to turn a sequence of individual images into a continuous stream is called *persistence of vision*.

The rate of animation should be fast enough to create the perception of continuous motion but slow enough so as not to waste resources. In practice, playback rates of 24 or 30 frames per second are normally used. With cheap animation, however, each frame may be displayed several consecutive times, producing a sampling rate (or update rate) that's much lower than the playback rate.

D. Salomon, *The Computer Graphics Manual*, Texts in Computer Science,
DOI 10.1007/978-0-85729-886-7_19, © Springer-Verlag London Limited 2011

The computer is used to automate parts of the overall task of animation, letting the animator work on an abstract level, concentrating on scene design and specifying the important information. In practice, this means that the animator enters information about the state of the animation at certain *key frames* and the software uses this to create the images for all the frames by interpolating between consecutive key frames. We say that the computer does the in-betweening.

Different pieces of animation can have different characteristics. Artistic animation, cheap cartoons, and flight simulation are all animations, but they are very different in their approach, attention to detail, the use of color, and the amount of information displayed. As a result, different tools, techniques, and algorithms have to be used, depending on the type of animation at hand.

Computer animation is divided into *computer-aided* (or two-dimensional) animation and *computer-generated* (or three-dimensional) animation. The former uses the computer to interpolate between two-dimensional shapes, whereas the latter uses it to build three-dimensional objects, to move both camera and objects along their paths, and to stop and take a snapshot at each frame. The term two-and-a-half-dimensional animation is also sometimes used. It refers to two-dimensional animation where each frame consists of several shapes drawn on separate slides. They represent objects at different distances from the viewer (for example, a nearby dog and trees in the background) and are moved different distances between frames (the background trees are shifted to the right while the dog is running to the left) to simulate parallax.

A complete piece of animation is sometimes called a *presentation*. It consists of a number of *acts*, where each act is broken down into several *scenes*. A scene is made of several *shots* or *sequences* of animation, each a succession of *animation frames*, where there is a small change in scene and camera position between consecutive frames. Thus, the hierarchy is

$$\text{piece} \rightarrow \text{act} \rightarrow \text{scene} \rightarrow \text{sequence} \rightarrow \text{frame}.$$

Each sequence is tested before it is actually produced, by displaying it with low-quality rendering and a small number of frames. Objects may be displayed as wire frames, or without removal of hidden parts, or in low resolution. The camera may be moved large distances between frames. The test is played back and watched by animators, which may lead them to change features such as timing, the camera path, the arrangement of objects, or the background color.

Time is an important term in animation and should be discussed further. Time is used in animation as a discrete quantity and can be varied by changing the number of frames. Speeding up an action is traditionally done by deleting certain frames, while slowing down the animation requires adding new frames. In traditional animation, the new frames that are added are identical to existing ones. For example, if every other frame is duplicated, the same sequence takes 50% longer to run. In computer animation, time can be controlled in a sophisticated way and there is no need to add or delete frames. If the animator decides, based on a test, to slow down an n-frame sequence by, say, 50%, the software is simply told to recreate the entire sequence from scratch using 50% more frames ($1.5n$ frames instead of n). The entire action is interpolated between the frames and the result is that no two frames are identical.

References [carlson 11] and [morrison 10] discuss the history of computer graphics, including the history of computer animation.

Early computer animation employed film as the output medium. Either 16-, 35-, or 70-mm film was used at 24 frames per second (fps). Very high quality can be achieved with 70-mm film, but high resolution (at least $3K \times 3K$) is required. The advantage of film is high resolution (Appendix E), a large number of colors, and insensitivity of the medium to magnetic fields. The disadvantages are the need for developing (the film cannot be watched immediately) and non-reusability of the medium (the same film cannot be used twice).

It is possible to place a camera in front of the computer screen and shoot frames, but old CRT displays normally had curved screens, resulting in image distortions. Another drawback is the fact that the screen is refreshed all the time. Taking a quick snapshot may produce a picture that's partly bright (from those parts of the screen that have just been refreshed) and partly dark (from other parts). The shot should therefore be slow, covering several screen refreshes, or it should be synchronized with the refresh so that the camera shutter remains open during an entire screen refresh. It is because of these reasons that better results are obtained with a film recorder. Such a device has a special flat screen and can be used with different cameras to take high-quality pictures.

If the animation is produced for television or for home entertainment, where it is going to be played back from a VCR or a DVD player, it makes sense to record it on video tape or a DVD. A video tape can be viewed immediately, is easy to copy, lasts a long time, and can be reused. Its main drawback is low resolution. The NTSC standard calls for 525 scan lines per image, of which only 480 actually contain the image. The NTSC aspect ratio is $4:3$, leading to 640 pixels per scan line. Currently, a resolution of 480×640 is considered low. The new HDTV standard (Section 26.2.2) doubles both the horizontal and vertical resolutions and employs a $16:9$ aspect ratio. This results in a high-resolution video image.

Placing a video camera in front of the computer screen involves the same problems as with film. It is therefore better to output the bitmap from memory directly to the camera, which is done by means of a special interface card plugged into the computer.

The main problems in computer animation are as follows:

- How to display on the screen only those parts of the scene that would be seen by an actual camera located at a certain point. This involves general perspective projection and clipping. In computer animation, the term *camera* replaces the word *observer*. This term refers to what is displayed on the screen (what we want the camera to see). In the computer, the camera is represented by several numbers describing its position, direction of view, an "up" direction, the distance k between it and the projection screen, and the two viewing half-angles h and v (Section 6.10).

- How to move the camera along any desired path and rotate it during movement so it always points to the center of interest (generally a different direction in each frame). Its "up" direction may also have to be rotated to achieve the desired animation effects.

- How to move the scene along another path (mathematically this is the same as moving the camera) and move parts of the scene in different ways (imagine a person walking, moving hands and feet in a complex pattern).

> Nothing is more revealing than movement.
>
> —Martha Graham.

A typical *sequence* in a computer-generated piece of animation involves a camera moving smoothly along a curved path around a scene composed of objects. The objects may also move at the same time. Creating such a sequence requires the following tasks:

- Defining the camera path. This may be a long, complex curve but the software should be able to follow it and to stop at many points (frames) for a snapshot. The frames should be equally spaced if uniform camera speed and smooth animation are important. Special effects may require the camera to accelerate or to slow down. The problem is that a typical parametric curve $\mathbf{P}(t)$ has variable velocity; varying t in equal increments advances unequal segments on the curve.
- At each point, the camera may have to be rotated so that it points in the right direction (normally directly at the scene, but sometimes off it). This is where *spherical interpolation* is used (Section 19.5).
- When the camera is properly positioned, a snapshot is taken. This is done by projecting the scene (or part of it) on the screen, which is assumed to be perpendicular to the line of sight of the camera, at a distance of k units from it. The y axis of the screen should be in the "up" direction of the camera. Perspective projection is normally used, since an image generated by other types of projection may look unnatural.
- The objects constituting the scene may also have to be moved and rotated (imagine a camera flying over a moving train). This task can use the techniques and tools developed for tasks 1 and 2. In fact, the case where only the camera moves and the objects of the scene are stationary is special and is referred to as a "walk-through" or a "flyby."

Perspective projection has been discussed in detail in Chapter 6. Here we show simple ways to approach the first two tasks.

19.2 Interpolating Positions

Task 1, defining a curve and moving along it at a constant speed, can be done by an interpolating Bézier curve. Section 13.12 shows how such a curve can be constructed. Given a set of $n+1$ points $\mathbf{P}_0$ through $\mathbf{P}_n$, the curve goes from $\mathbf{P}_1$ to $\mathbf{P}_{n-1}$ (not from $\mathbf{P}_0$ to $\mathbf{P}_n$) and is constructed of $n-2$ segments $\mathbf{P}_i(t)$, each connecting one pair of points. The pairs are $(\mathbf{P}_1,\mathbf{P}_2)$, $(\mathbf{P}_2,\mathbf{P}_3)$, up to $(\mathbf{P}_{n-2},\mathbf{P}_{n-1})$. Each segment $\mathbf{P}_i(t)$ is based on four points, the two exterior points are $\mathbf{P}_i$ and $\mathbf{P}_{i+1}$ and the two interior ones, $\mathbf{X}_i$ and $\mathbf{Y}_i$, are automatically calculated by Equation (13.29), duplicated here:

$$\mathbf{X}_i = \mathbf{P}_i + \frac{1}{6}(\mathbf{P}_{i+1} - \mathbf{P}_{i-1}); \quad \mathbf{Y}_i = \mathbf{P}_{i+1} - \frac{1}{6}(\mathbf{P}_{i+2} - \mathbf{P}_i). \tag{13.29}$$

No segments connect $\mathbf{P}_0$ to $\mathbf{P}_1$ or $\mathbf{P}_{n-1}$ to $\mathbf{P}_n$. The two extreme points, $\mathbf{P}_0$ and $\mathbf{P}_n$, are used as guide points, to control the initial and final directions of the curve. Point $\mathbf{X}_1$ is obtained by adding vector $(\mathbf{P}_2 - \mathbf{P}_0)/6$ to point $\mathbf{P}_1$. Point $\mathbf{Y}_{n-2}$ is similarly obtained by subtracting vector $(\mathbf{P}_n - \mathbf{P}_{n-2})/6$ from point $\mathbf{P}_{n-1}$. Figure 19.1 shows such a curve.

In practical animation work, the animator should have a rough idea of the shape of the path along which the camera should move. The animator inputs the coordinates of

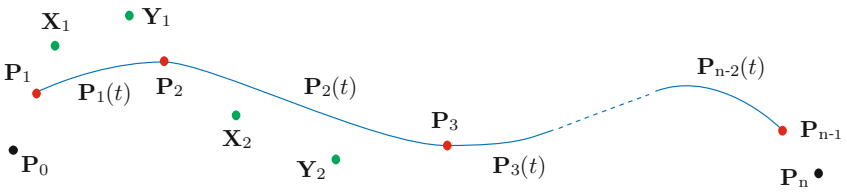


Figure 19.1: An Interpolating Bézier Curve for $n+1$ Points.

$n-1$ key points $\mathbf{P}_i$ on the path (they should be fairly close to each other and roughly equally spaced) followed by the two extreme guide points $\mathbf{P}_0$ and $\mathbf{P}_n$ to control the start and end directions of the path. The $n-1$ points are called the animation *key frames*. The software calculates and displays all the interior points $\mathbf{X}_i$ and $\mathbf{Y}_i$, and the $n-2$ individual Bézier segments $\mathbf{P}_i(t)$. The path (which is normally three-dimensional) is then examined by rotating it and watching it from different directions. If the path is not satisfactory, it can be edited either by deleting some key frames, moving them, or adding new ones. The interior points can also be manually repositioned at this time.

When the right path is finally obtained, the software moves the camera along the path, segment by segment, varying t from 0 to 1 in F steps, called *frames*, (where F is a parameter) for **each** of the $n-2$ segments. The value of t for frame f is, thus, given by $t=(f-1)/(F-1)$. The original $n+1$ points are converted in this way to $n-2$ Bézier segments that produce the final $(n-2)F$ equally spaced animation frames. At each frame, the camera is rotated to point in the right direction and a snapshot taken. In professional jargon, this process is called *in-betweening*. The computer stops the camera and generates F frames between each pair of key frames supplied by the animator.

> And when a damp fell round the path of Milton, in his hand The thing became a trumpet; whence he blew Soul-animating strains,–alas! too few.
>
> —William Wordsworth, *Scorn Not The Sonnet*.

19.3 Constant Speed: I

To obtain smooth animation, the camera should move along its path in equal steps, covering equal arc lengths in each step. In principle, this can be achieved by a parameter substitution. Suppose that we substitute some parameter $s(t)$ for t, such that our curve becomes $\mathbf{P}(s)$ instead of $\mathbf{P}(t)$. Clearly, the best choice for s is the arc length. If $s(0.2)$ is the length of the curve from its start $\mathbf{P}(0)$ to point $\mathbf{P}(0.2)$, then incrementing s in equal steps will advance equal arc lengths on the curve. Section 8.2 shows that the arc length of the entire curve $\mathbf{P}(t)$ is given by the integral

$$\int |d\mathbf{P}(t)| = \int_0^1 |\mathbf{P}^t(t)|\,dt.$$

The arc length $s(u)$ from $t = 0$ to $t = u$ is therefore given by

$$s(u) = \int_0^u |\mathbf{P}^t(t)|\, dt.$$

The trouble is that such integrals are normally impossible to calculate analytically; they must be computed numerically (they belong to the family of *elliptic integrals*). A simple alternative that's sometimes satisfactory is to calculate a large number of points on the curve, to replace the curve with the polyline made by these points, and to calculate approximate arc lengths by computing the lengths of the polyline segments. The steps are as follows:

1. Vary t from 0 to 1 in $n+1$ small, equal steps and calculate $n+1$ points $\mathbf{P}(t)$ on the curve.

2. Compute the n straight line distances between the points.

3. Accumulate the distances of step 2, such that accumulated distance i will give the total (approximate) distance from the start of the curve to point $t = i$.

4. Divide all the accumulated distances by the last one, resulting in a table T of normalized accumulated distances (whose values are between 0 and 1).

5. Find the entries in T that are closest to the required arc lengths. Suppose, for example, that we want to select the six points on the curve where the normalized accumulated arc lengths are 0, 0.2, 0.4, 0.6, 0.8, and 1. We find the entries of table T that are the closest to these values. Assume that these are entries 1, 43, 61, 78, 95, and 100. The parameter t should be set to the normalized values of these six entries.

Figure 19.2 is a listing of *Mathematica* code that illustrates this method for a four-point Bézier curve.

> Speed is scarcely the noblest virtue of graphic composition, but it has its curious rewards. There is a sense of getting somewhere fast, which satisfies a native American urge.
>
> —James Thurber.

19.4 Constant Speed: II

Given a space curve $\mathbf{P}(t) = (x(t), y(t), z(t))$, we denote by $\text{Len}(t_1, t_2)$ the arc length from $\mathbf{P}(t_1)$ to $\mathbf{P}(t_2)$. This section presents a numerical approach—proposed by [Guenter and Parent 90] and called *adaptive subdivision*—to two problems:

Problem 1. Given values t_1 and t_2 of the time parameter, calculate $\text{Len}(t_1, t_2)$ numerically.

Problem 2. Given a value t_1 and an arc length s, find a value t_2 such that $\text{Len}(t_1, t_2) = s$.

We first show why these problems are important. If we want to move the animation camera along $\mathbf{P}(t)$ at a constant speed, we can use problem 1 to find the length $S =$

```
p0={0,1}; p1={5,1}; p2={5,0}; p3={4,.5};
Bez[t_]:=(1-t)^3p0+3t(1-t)^2p1+3t^2(1-t)p2+t^3p3;
tbl=Table[Bez[t], {t,0,1,.01}];
(* tab1 is a list of lengths of straight segments *)
tab1=Table[Sqrt[(tbl[[i+1,1]]-tbl[[i,1]])^2
 +(tbl[[i+1,2]]-tbl[[i,2]])^2], {i,1,100}];
(* tab2 is a list of accumulated lengths *)
tab2={tab1[[1]]};
Do[tab2=Append[tab2,tab1[[i]]+tab2[[i-1]]],{i,2,100}];
tab2=tab2/tab2[[100]]; (* normalize tab2 *)
tab3={0}; d=.1;
(* tab3 is a list of non-equally-spaced parameter values *)
Do[If[tab2[[i]]>d, {tab3=Append[tab3,i/100], d=d+.1}], {i,1,100}];
tab3=Append[tab3,1];
len=Length[tab3];
tab4=Table[Bez[tab3[[i]]], {i,1,len}];
 (* use tab3 as the parameter values *)
ListPlot[tab4] (* display equally-spaced points *)
ListPlot[tbl]  (* display 101 non-equally-spaced points *)
```

Figure 19.2: Normalized Accumulated Arc Lengths.

$\mathrm{Len}(0,1)$ of the entire curve, then divide it into $n-1$ equal parts $s = S/(n-1)$ and use problem 2 to find values $t_1 = 0 < t_2 < t_3 < \cdots < t_n = 1$ such that

$$\mathrm{Len}(t_1, t_2) = \mathrm{Len}(t_2, t_3) = \cdots = \mathrm{Len}(t_{n-1}, t_n) = s.$$

The t_i values should then be used to specify n equally spaced frames along the curve. A similar method can be used for more complex cases where we want to move the camera at a nonuniform speed along the curve. We divide S into parts s_i of different sizes and use problem 2 to find values t_i such that $\mathrm{Len}(t_i, t_{i+1}) = s_i$. Acceleration will result if $s_i < s_{i+1}$, but any nonuniform motion can be generated by carefully selecting the values of s_i. Here is how to approach the two problems.

Problem 1: Section 8.2 shows that the arc length of a curve $\mathbf{P}(t)$ is given by

$$\int_0^1 |\mathbf{P}^t(t)|\, dt.$$

Since $\mathbf{P}(t) = (x(t), y(t), z(t))$, we get

$$\mathbf{P}^t(t) = \frac{d\mathbf{P}(t)}{dt} = \left(\frac{dx(t)}{dt}, \frac{dy(t)}{dt}, \frac{dz(t)}{dt}\right),$$

and from this,

$$|\mathbf{P}^t(t)| = \sqrt{\left(\frac{dx(t)}{dt}\right)^2 + \left(\frac{dy(t)}{dt}\right)^2 + \left(\frac{dz(t)}{dt}\right)^2}. \tag{19.1}$$

Gaussian quadrature is used to numerically integrate Equation (19.1) from 0 to t_i for certain values of i. Each such integral results in an arc length s_i from the start of the curve to point $\mathbf{P}(t_i)$. The pairs of values (t_i, s_i) are stored in a table and are later used to solve problem 1 in the following way. Given two values t_1 and t_2, the arc length $\text{Len}(t_1, t_2)$ is determined as follows:

1. Find entry i in the table such that $t_i \le t_1 < t_{i+1}$.
2. Using Gaussian quadrature, integrate Equation (19.1) from t_i to t_1 to obtain arc length s_1 (if $t_i = t_1$, skip the integration and set $s_1 = 0$).
3. Set $\text{Len}(0, t_1) = s_i + s_1$.
4. Do the same thing for t_2. Find entry j in the table such that $t_j \le t_2 < t_{j+1}$, integrate from t_j to t_2 to obtain s_2 (or set $s_2 = 0$, if $t_j = t_2$), and set $\text{Len}(0, t_2) = s_j + s_2$.
5. Subtract $\text{Len}(0, t_2) - \text{Len}(0, t_1) = s_j + s_2 - (s_i + s_1)$ to obtain $\text{Len}(t_1, t_2)$.

The question is what values of i to select for the table, and the answer should now be obvious. Since we integrate from t_i to t_1, we can relate the distance between two consecutive values t_i and t_{i+1} to the curvature of $\mathbf{P}(t)$ in that region. If the curvature is low (the curve between points $\mathbf{P}(t_i)$ and $\mathbf{P}(t_{i+1})$ is close to a straight line), we can place t_{i+1} well away from t_i. The integral from t_i to t_1 would be done over a region of the curve that may be long but is close to a straight line. The result would therefore be quick and accurate. If the curvature is high, the two values t_i and t_{i+1} have to be close by. The integral from t_i to t_1 would be done in this case over a curvy but *short* region of the curve, so, again, it would be accurate.

Instead of calculating the curvature, the method uses a recursive procedure `Subdivide`, and a threshold parameter `eps`. The procedure is given a range $[t_l, t_r]$, it uses Gaussian integration to find the arc length s_{lr} of this range, then divides the range in the middle $t_m = (t_l + t_r)/2$, integrates each part to get arc lengths s_{lm} and s_{mr}, and calculates the difference $|s_{lr} - (s_{lm} + s_{mr})|$. If this difference is less than `eps`, the procedure assumes that the curvature of $\mathbf{P}(t)$ in the region $[t_l, t_r]$ is small enough and it stores the pair $(t_m, s_{0l} + s_{lm})$ in the table. Otherwise, it calls itself recursively for the two ranges $[t_l, t_m]$ and $[t_m, t_r]$. Figure 19.3 lists C++ code for this procedure.

Problem 2: Given a value t_1 and an arc length s, find a value t_2 such that $\text{Len}(t_1, t_2) = s$. We define a function $f(t) = \text{Len}(t_1, t) - s$ that reduces problem 2 to that of finding a zero of $f(t)$ in the range $[t_1, 1]$. Perhaps the simplest method for finding a zero of a function is binary subdivision. The range $[t_1, 1]$ is divided in the middle, $t_m = (t_1 + 1)/2$. If $f(t_m) = 0$ (or if it is very close to 0), we are done. Otherwise, if $f(t_1)$ and $f(t_m)$ have the same sign, we divide the range $[t_1, t_m]$ in the middle and perform the same tests. Otherwise, we divide the range $[t_m, 1]$.

Finding the zero of a function can also be done by using the well-known Newton–Raphson method (discussed in texts on numerical analysis). This is a fast method, but it has two disadvantages.

1. It requires the derivative of the function. In our case, the derivative depends on the particular curve used, so it has to be implemented by the user for each curve separately.
2. The derivative may be zero, or very close to zero. Since this method divides by the derivative, the division may result in overflow.

```
#include <stdio.h>
#include <math.h> // for function fabs
float totl_arc; // global variable
void Add_tabl(float, float);
float Gauss(float, float);
float Subdivide(float left, float right, float full_intr, float eps){
float mid, left_arc, right_arc, left_sub;
 mid=(left+right)/2;
 left_arc=Gauss(left,mid);
 right_arc=Gauss(mid,right);
 if(fabs(full_intr-left_arc-right_arc)<eps)
  {left_sub=Subdivide(left,mid,left_arc,eps/2.0);
  totl_arc=totl_arc+left_sub;
  Add_tabl(mid,totl_arc);
  return(Subdivide(mid,right,right_arc,eps/2.0)+left_sub);}
 else
  return(left_arc+right_arc);
}
int main(){
float left, right, full_intr, eps;
left=0; right=1.0; totl_arc=0; eps=0.001;
 full_intr=Gauss(left,right);
 Subdivide(left,right,full_intr,eps);
 }
```

Figure 19.3: Procedure Subdivide.

19.5 Interpolating Orientations: I

Now comes the second task. The animator should supply the animation software with the data needed for orienting the camera. This process is based on the following fact, proved by Leonhard Euler in 1752. Imagine a rigid object positioned at point $\mathbf{P}$ and having a certain orientation. We now send the object flying through space. It may roll and tumble in a complicated way, but its position and orientation at any moment can be completely described by two transformations, a translation from $\mathbf{P}$ to its present position and a rotation through an angle θ about some axis $\mathbf{v}$.

Our imaginary camera may be considered such an object. It may have to move around the scene along a complicated path and change its orientation all the time, so that it always points in the right direction. The animator should have a rough idea of how to move the camera and in what direction it should look. The animator should therefore input $n-2$ direction vectors (the directions in which the camera should look when positioned at the *key frames*) and the animation software should use these vectors to interpolate between key frames and determine the orientation of the camera at any point.

Before we discuss how to interpolate the direction vectors, we have to introduce one more complication, namely the "up" vector. Imagine a camera placed in the xy

plane, looking in the positive x direction $(1,0,0)$ with its top pointing in the positive z direction. We now rotate the camera in small steps until it points in the positive y direction $(0,1,0)$ with its top still pointing in the positive z direction (Figure 19.4a). At any time during this rotation the camera points in a direction $(a,b,0)$, i.e., somewhere in the xy plane. Now, imagine that while rotating the camera from x to y, the animator also wants to rotate it about its direction of view $(a,b,0)$ such that when it reaches its final direction, its top will be pointing in the *negative* z direction (Figure 19.4b). If such an effect is called for, then the animator also has to specify an "up" direction in each key frame. These "up" vectors should be interpolated between key frames and should be used to indicate the top of the screen each time a snapshot is taken. (When the software calls a procedure to project the scene on the screen, it should transfer to the procedure, as parameters, the position of the camera, its direction of view, the direction of the "top" of the screen, and any other necessary data.) The interpolation method discussed below should therefore be applied to the direction of view of the camera, as well as to its "up" direction, if this direction is explicitly defined.

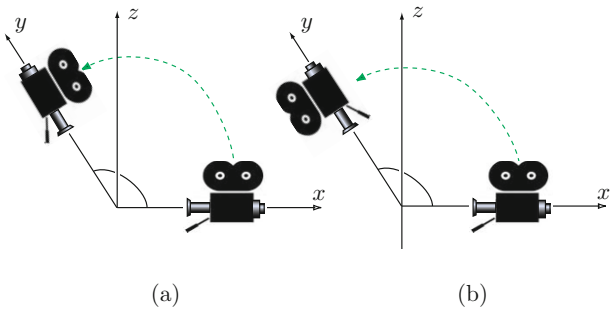


Figure 19.4: Illustrating the "Up" Direction.

(One special, important case of camera orientation, namely the case where the camera follows a moving object along its path, should be mentioned. Imagine a camera following an airplane, repeating all its maneuvers while staying the same distance behind it all the time. This case is easy to implement. When the camera is located at point $\mathbf{P}_i(t)$, it should look at point $\mathbf{P}_i(t+f)$, where f is a constant. If f is negative, then the camera is located in front of the object, flying backward and constantly looking at the object.)

For each of the $n-1$ key frames, the animator has to input the direction $\mathbf{D}_i$ in which the camera should be looking. The software interpolates these vectors to orient the camera between successive key frames. Figure 19.5 shows several vectors $\mathbf{D}_i$. To interpolate $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$ we need to compute the angle θ_i between them. This is done by first normalizing the two direction vectors (dividing each by its length to obtain unit vectors), then computing their dot product $\mathbf{D}_i \bullet \mathbf{D}_{i+1}$ which equals $\cos\theta_i$. We will see that $\sin\theta_i$ is also necessary, but it can always be calculated as $\sin\theta_i = \pm\sqrt{1-\cos^2\theta_i}$.

◇ **Exercise 19.1:** How does the software decide what sign to use for $\sqrt{1-\cos^2\theta}$?

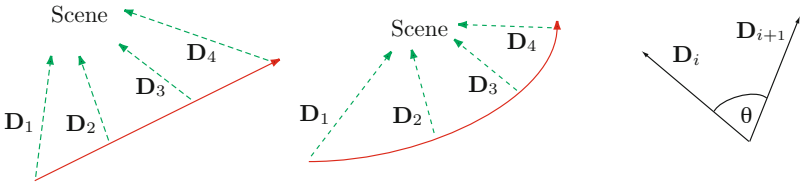


Figure 19.5: Rotating the Camera at Key Frames.

We now need a function to correctly interpolate direction vectors. For each of the $n-2$ segments $\mathbf{P}_i(t)$ that constitute the camera path, we start with a direction $\mathbf{D}_i$ in animation frame 1 (where $t=0$) and end with a direction $\mathbf{D}_{i+1}$ in animation frame F ($t=1$). A linear interpolation $(1-t)\mathbf{D}_i + t\mathbf{D}_{i+1}$ is simple but produces nonuniform moves. When t is varied from 0 to 1 in equal steps, the interpolation steps are not the same; they start large and get smaller. The reason is that rotation has to do with spherical symmetry, whereas linear interpolation has to do with straight lines.

To derive a proper interpolation function, we have to think in terms of moving along a circular arc. We start with a two-dimensional example. Imagine two unit vectors $\mathbf{D}_1$ and $\mathbf{D}_2$ in two-dimensional space. The vector $\mathbf{D}_l$ that's defined as the combination $\mathbf{D}_l(t) = (1-t)\mathbf{D}_1 + t\mathbf{D}_2$ rotates from $\mathbf{D}_1$ to $\mathbf{D}_2$ such that its tip moves along a straight line (i.e., the magnitude of $\mathbf{D}_l$ keeps changing, Figure 19.6a). To move from $\mathbf{D}_1$ to $\mathbf{D}_2$ along a circular arc, we should use *spherical linear interpolation* (slerp, Figure 19.6b). We use the expression

$$\mathbf{D}_s(t) = \frac{\sin((1-t)\theta)}{\sin\theta}\mathbf{D}_1 + \frac{\sin(t\theta)}{\sin\theta}\mathbf{D}_2, \tag{19.2}$$

where θ is the angle between vectors $\mathbf{D}_1$ and $\mathbf{D}_2$ (note that $\mathbf{D}_1 \bullet \mathbf{D}_2 = \cos\theta$, since these are unit vectors). $\mathbf{D}_s(t)$ is a unit vector that changes direction from $\mathbf{D}_1$ (when $t=0$) to $\mathbf{D}_2$ (when $t=1$) in equal increments. Its tip describes a circular arc (Figure 19.6b).

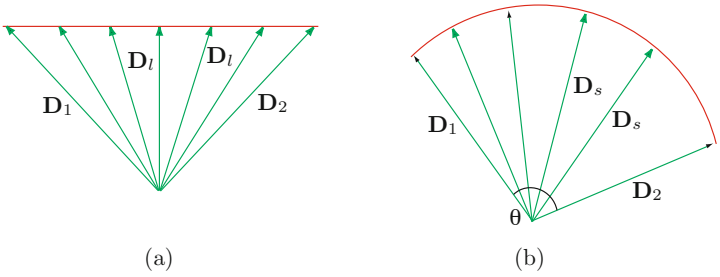


Figure 19.6: Linear and Spherical Interpolations.

◇ **Exercise 19.2:** Prove the above claim.

Another way to look at spherical interpolation is to consider all the two-dimensional unit vectors. They have the same size, but they point in all possible directions. Placing them with their tails at the origin creates a unit circle about the origin. Spherically interpolating two such vectors, $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$, is equivalent to moving along an arc on this circle.

The same is true for two unit vectors in three-dimensional space. We can imagine all the three-dimensional unit vectors to form a unit sphere. Spherically interpolating two such vectors is equivalent to moving along a great arc on this sphere. A good spherical interpolation function for direction vectors $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$ would therefore be

$$\mathbf{D}_{i+t} = \frac{\sin((1-t)\theta)}{\sin\theta}\mathbf{D}_i + \frac{\sin(t\theta)}{\sin\theta}\mathbf{D}_{i+1}. \tag{19.3}$$

When t is incremented in equal steps, this function produces smooth movements of the animation camera from the orientation specified by $\mathbf{D}_i$ to that specified by $\mathbf{D}_{i+1}$. A two-dimensional vector $\mathbf{D} = (x, y)$ can be considered a complex number. When we use this interpretation, the spherical interpolation of two unit vectors can also be written $\mathbf{D}_i(\mathbf{D}_i^{-1}\mathbf{D}_{i+1})^t$ (complex numbers can be multiplied, they have an inverse, and they can be raised to a power). When t varies from 0 to 1, this expression varies from $\mathbf{D}_i$ to $\mathbf{D}_{i+1}$. Since both $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$ are unit vectors, the absolute value of this triple product is 1.

Spherical interpolation involves a division by $\sin\theta$, so the case $\sin\theta = 0$ should be discussed. This case occurs when $\theta = 0°$ or $\theta = 180°$, but the latter case can be excluded since it does not make sense to use two direction vectors going in opposite directions in two consecutive key frames (see Exercise 19.3). The case $\theta = 0°$ means two parallel direction vectors in two consecutive key frames. This case is common (it means that the camera's orientation should not change between two consecutive key frames), so the interpolation software should check for it and perform the trivial interpolation $\mathbf{D}_{i+t} = \mathbf{D}_i$.

◇ **Exercise 19.3:** Explain why the case $\theta = 180°$ can be excluded.

Table 19.7 illustrates the difference between linear and spherical interpolations. Given the two unit vectors $\mathbf{D}_1 = (1, 0)$ and $\mathbf{D}_2 = (0, 1)$ with a 90° angle between them, the table shows the results of the linear interpolation $(1-t)\mathbf{D}_1 + t\mathbf{D}_2$ and the spherical interpolation

$$\frac{\sin(90(1-t))}{\sin 90°}\mathbf{D}_1 + \frac{\sin(90t)}{\sin 90°}\mathbf{D}_2$$

for 11 values of t. The spherical interpolation results in equal increments of 9°, while the linear interpolation results in angle increments (row "Diff" in the table) that initially get bigger, then get smaller.

t:	.1	.2	.3	.4	.5	.6	.7	.8	.9	1
Linear:	6.34	14.04	23.20	33.69	45.00	56.31	66.80	75.96	83.66	90.00
Diff:	6.34	7.70	9.16	10.49	11.31	11.31	10.49	9.16	7.70	6.34
Spherical:	9	18	27	36	45	54	62	72	81	90

Table 19.7: Linear and Spherical Interpolations.

```
(* Two interpolations of vectors with 90 deg *)
d1={1,0}; d2={0,1};
(* Generate 11 linearly interpolated vectors in 'vec' *)
vec=Table[(1-t)d1+t d2,{t,0,1,.1}];
(* Normalize these vectors *)
Do[vec[[i]]=vec[[i]]/Sqrt[vec[[i,1]]^2+vec[[i,2]]^2], {i,1,11}];
(* Show them *)
Table[ArcCos[vec[[1]].vec[[i+1]]]/Degree, {i,1,10}]
Table[ArcCos[vec[[i]].vec[[i+1]]]/Degree, {i,1,10}]
(* Generate 11 spherically interpolated vectors in 'vec' *)
vec=Table[(Sin[90(1-t)Degree]d1+Sin[90t Degree]d2),{t,0,1,.1}];
(* Normalize these vectors *)
Do[vec[[i]]=vec[[i]]/Sqrt[vec[[i,1]]^2+vec[[i,2]]^2], {i,1,11}];
(* Show them *)
Table[ArcCos[vec[[1]].vec[[i+1]]]/Degree, {i,1,10}]
Table[ArcCos[vec[[i]].vec[[i+1]]]/Degree, {i,1,10}]
```

Mathematica Code for Table 19.7.

19.6 SLERP

This section explains why slerp (shorthand for spherical linear interpolation) is the right interpolation for vector rotation. Imagine a sphere of radius R intersected by a plane that passes through the center of the sphere. The intersection of the sphere and the plane is a great circle of the sphere. Imagine two points $\mathbf{x}$ and $\mathbf{y}$ on the surface of a unit radius sphere. The shortest path between them is called a geodesic and is part of a great circle. (The case where $\mathbf{x}$ and $\mathbf{y}$ are antipodal must be excluded because in this case there is no unique shortest path between them.)

Given a number $t \in [0,1]$, we want to locate the point $\mathbf{z}$ on the sphere located a fraction t of the distance from $\mathbf{x}$ to $\mathbf{y}$ on the geodesic connecting them. Figure 19.8a,b makes it clear that $\mathbf{z} = \cos(t\theta)\mathbf{x} + \sin(t\theta)\mathbf{w}$, where $\mathbf{w}$ is the unit vector perpendicular to $\mathbf{x}$. It only remains to express $\mathbf{w}$ in terms of the given quantities $\mathbf{y}$ and θ.

Section 8.1.2 discusses the projections of one vector onto another, and demonstrates the following. Given two vectors $\mathbf{v}$ and $\mathbf{u}$, the components (or projections) of $\mathbf{v}$ in the direction of $\mathbf{u}$ and perpendicular to it are $(\mathbf{u}\cdot\mathbf{v})\mathbf{u}$, and $\mathbf{v} - (\mathbf{u}\cdot\mathbf{v})\mathbf{u}$, respectively.

Figure 19.8b shows vector $\mathbf{v}$, the projection of $\mathbf{y}$ perpendicular to $\mathbf{x}$. This vector satisfies $\mathbf{v} = \mathbf{y} - (\cos\theta)\mathbf{x} = \mathbf{y} - (\mathbf{y}\cdot\mathbf{x})\mathbf{x}$. Vector $\mathbf{w}$ is a unit vector in the direction of $\mathbf{v}$,

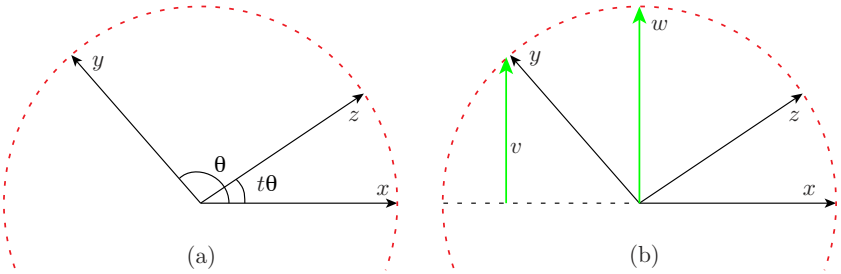


Figure 19.8: Derivation of the slerp Function.

so it satisfies $\mathbf{w} = \mathbf{v}/\sin\theta = \mathbf{v}/\sqrt{\mathbf{v}\cdot\mathbf{v}}$. From these relations we obtain the final result

$$\begin{aligned}
\text{slerp}(\mathbf{x},\mathbf{y},\theta) &= \cos(t\theta)\mathbf{x} + \sin(t\theta)\mathbf{w} \\
&= \cos(t\theta)\mathbf{x} + \sin(t\theta)\frac{\mathbf{y}-(\cos\theta)\mathbf{x}}{\sin\theta} \\
&= \left[\cos(t\theta) - \sin(t\theta)\frac{\cos\theta}{\sin\theta}\right]\mathbf{x} + \frac{\sin(t\theta)}{\sin\theta}\mathbf{y} \\
&= \frac{\sin\theta\cos(t\theta) - \sin(t\theta)\cos\theta}{\sin\theta}\mathbf{x} + \frac{\sin(t\theta)}{\sin\theta}\mathbf{y} \\
&= \frac{\sin(\theta - t\theta)}{\sin\theta}\mathbf{x} + \frac{\sin(t\theta)}{\sin\theta}\mathbf{y} \\
&= \frac{\sin((1-t)\theta)}{\sin\theta}\mathbf{x} + \frac{\sin(t\theta)}{\sin\theta}\mathbf{y}
\end{aligned},$$

(where the trigonometric identity $\sin(a-b) = \sin a\cos b - \sin b\cos a$ was used in the next-to-last line.)

19.7 Summary

Animating the camera starts with the animator specifying the key frames. For key frame i, the animator should specify the camera position $\mathbf{P}_i$, its direction of view $\mathbf{D}_i$, and if necessary, also its "up" direction (the values of k, h, and v may also vary from one key frame to the next). The software prepares the entire path as described in Section 19.2. It then varies the time parameter t in F steps for each path segment, to obtain all the $(n-2)F$ frames. For each frame, the two direction vectors $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$ are spherically interpolated, the camera is pointed in the new direction, and the entire scene is projected on the projection plane (the screen), a process that may require clipping. Each path segment should be short and should not deviate much from a straight line, so varying t in equal steps would cover roughly equal distances on the segment even though the velocity of the segment is normally variable.

I'm appropriately animate for a human being in the context in which I exist.
—Woody Allen in *Wild Man Blues* (1998).

19.7.1 Example 1

This example is in three dimensions, but to make it easier to visualize the way the camera moves, we restrict the scene and the camera path to the xy plane. The scene is assumed to be located about the origin and the camera path, Figure 19.9a, is assumed to be in the xy plane. The camera should therefore start at point $\mathbf{P}_1$, pointing toward the origin (i.e., in the positive y direction) and should rotate about the z axis as it moves, in order to always point toward the origin. We define the camera path by means of the seven points

$$\mathbf{P}_0 = (1.5, -2, 0), \quad \mathbf{P}_1 = (0, -2, 0), \quad \mathbf{P}_2 = (-2, 0, 0), \quad \mathbf{P}_3 = (1.5, 2, 0),$$
$$\mathbf{P}_4 = (5, 0, 0), \quad \mathbf{P}_5 = (3, -2, 0), \quad \mathbf{P}_6 = \mathbf{P}_0.$$

The two extreme points $\mathbf{P}_0$ and $\mathbf{P}_6$ control the start and end directions of the path, respectively. The path itself is made of four segments defined by means of the four overlapping groups $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2\mathbf{P}_3$, $\mathbf{P}_1\mathbf{P}_2\mathbf{P}_3\mathbf{P}_4$, $\mathbf{P}_2\mathbf{P}_3\mathbf{P}_4\mathbf{P}_5$, and $\mathbf{P}_3\mathbf{P}_4\mathbf{P}_5\mathbf{P}_6$.

Equation (13.29) is used to calculate the four sets of $\mathbf{X}$ and $\mathbf{Y}$ points:

$$\begin{aligned}
\mathbf{X}_1 &= \mathbf{P}_1 + \tfrac{1}{6}(\mathbf{P}_2 - \mathbf{P}_0) = \left(-\tfrac{7}{12}, -\tfrac{5}{3}, 0\right), & \mathbf{Y}_1 &= \mathbf{P}_2 - \tfrac{1}{6}(\mathbf{P}_3 - \mathbf{P}_1) = \left(-\tfrac{9}{4}, -\tfrac{2}{3}, 0\right),\\
\mathbf{X}_2 &= \mathbf{P}_2 + \tfrac{1}{6}(\mathbf{P}_3 - \mathbf{P}_1) = \left(-\tfrac{21}{12}, \tfrac{2}{3}, 0\right), & \mathbf{Y}_2 &= \mathbf{P}_3 - \tfrac{1}{6}(\mathbf{P}_4 - \mathbf{P}_2) = \left(\tfrac{2}{6}, 2, 0\right),\\
\mathbf{X}_3 &= \mathbf{P}_3 + \tfrac{1}{6}(\mathbf{P}_4 - \mathbf{P}_2) = \left(\tfrac{8}{3}, 2, 0\right), & \mathbf{Y}_3 &= \mathbf{P}_4 - \tfrac{1}{6}(\mathbf{P}_5 - \mathbf{P}_3) = \left(\tfrac{19}{4}, \tfrac{2}{3}, 0\right),\\
\mathbf{X}_4 &= \mathbf{P}_4 + \tfrac{1}{6}(\mathbf{P}_5 - \mathbf{P}_3) = \left(\tfrac{21}{4}, -\tfrac{2}{3}, 0\right), & \mathbf{Y}_4 &= \mathbf{P}_5 - \tfrac{1}{6}(\mathbf{P}_6 - \mathbf{P}_4) = \left(\tfrac{43}{12}, -\tfrac{5}{3}, 0\right).
\end{aligned}$$

Thus, the path is made of the four Bézier segments:

$$\begin{aligned}
\mathbf{P}_1(t) &= (1-t)^3\mathbf{P}_1 + 3t(1-t)^2\mathbf{X}_1 + 3t^2(1-t)\mathbf{Y}_1 + t^3\mathbf{P}_2,\\
\mathbf{P}_2(t) &= (1-t)^3\mathbf{P}_2 + 3t(1-t)^2\mathbf{X}_2 + 3t^2(1-t)\mathbf{Y}_2 + t^3\mathbf{P}_3,\\
\mathbf{P}_3(t) &= (1-t)^3\mathbf{P}_3 + 3t(1-t)^2\mathbf{X}_3 + 3t^2(1-t)\mathbf{Y}_3 + t^3\mathbf{P}_4,\\
\mathbf{P}_4(t) &= (1-t)^3\mathbf{P}_4 + 3t(1-t)^2\mathbf{X}_4 + 3t^2(1-t)\mathbf{Y}_4 + t^3\mathbf{P}_5.
\end{aligned}$$

We assume that the scene is located at the origin and the camera should always be positioned to look at it. The five direction vectors $\mathbf{D}_1$ through $\mathbf{D}_5$ are thus the vectors from each of the points $\mathbf{P}_i$ to the origin (Figure 19.9b). They are shown as unit vectors:

$$\mathbf{D}_1 = -\mathbf{P}_1 = (0, 1, 0), \quad \mathbf{D}_2 = -\mathbf{P}_2 = (1, 0, 0), \quad \mathbf{D}_3 = -\mathbf{P}_3 = (-3/5, -4/5, 0),$$
$$\mathbf{D}_4 = -\mathbf{P}_4 = (-1, 0, 0), \quad \mathbf{D}_5 = -\mathbf{P}_5 = (-0.83, 0.55, 0).$$

The angles between them are determined by means of dot products:

$$(\mathbf{D}_1 \bullet \mathbf{D}_2) = 0 \rightarrow \theta_{12} = 90^\circ, \quad (\mathbf{D}_2 \bullet \mathbf{D}_3) = -3/5 \rightarrow \theta_{23} = 126.87^\circ,$$
$$(\mathbf{D}_3 \bullet \mathbf{D}_4) = 3/5 \rightarrow \theta_{34} = 53.13^\circ, \quad (\mathbf{D}_4 \bullet \mathbf{D}_5) = .83 \rightarrow \theta_{45} = 33.9^\circ.$$

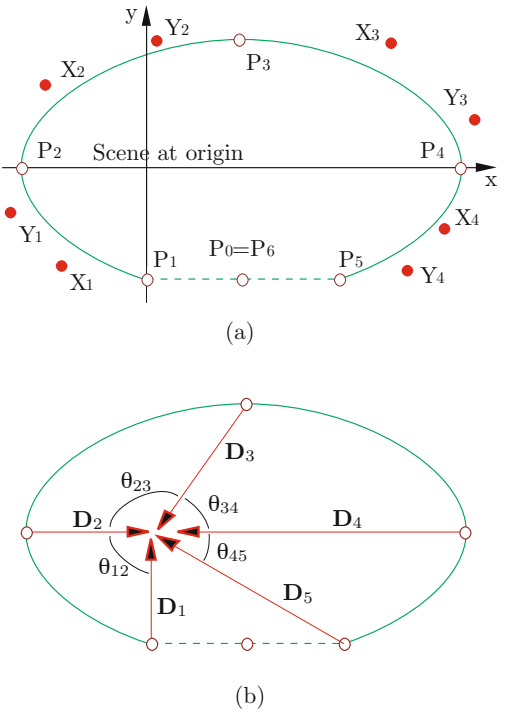


Figure 19.9: (a) A Seven-Point Animation Path. (b) Five Direction Vectors.

To produce the first animation frame, we assume that the camera is located at $\mathbf{P}_1$, looking in direction $\mathbf{D}_1$ and we take a snapshot, i.e., we assume a projection plane perpendicular to $\mathbf{D}_1$, located at a distance k from $\mathbf{P}_1$, on which the scene can now be projected.

To produce more animation frames, we start with the first Bézier segment, $\mathbf{P}_1(t)$, vary t in steps, and for each value t_i calculate a position $\mathbf{P}_1(t_i)$ on the curve and a direction $\mathbf{D}_{1+t_i}$ that's a spherical interpolation of $\mathbf{D}_1$ and $\mathbf{D}_2$:

$$\mathbf{D}_{1+t_i} = \frac{\sin(\theta_{12} - t_i\theta_{12})}{\sin\theta_{12}}\mathbf{D}_1 + \frac{\sin(t_i\theta_{12})}{\sin\theta_{12}}\mathbf{D}_2.$$

Once $\mathbf{D}_{1+t_i}$ is obtained for frame i, we use it to take a snapshot.

As an example, for $t = 0.5$ the camera should be moved to point

$$\begin{aligned}\mathbf{P}_1(0.5) &= 0.5^3(0,-2,0) + 3\cdot 0.5\cdot 0.5^2(-7/12,-5/3,0)\\ &\quad + 3\cdot 0.5^2\cdot 0.5(-9/4,-2/3,0) + 0.5^3(-2,0,0)\\ &= 0.5^3[(0,-2,0) + 3(-7/12,-5/3,0) + 3(-9/4,-2/3,0) + (-2,0,0)]\\ &= (-1.3125,-1.125,0)\end{aligned}$$

and should look in direction $\mathbf{D}_{1+0.5}$,

$$\begin{aligned}\mathbf{D}_{1+0.5} &= \frac{\sin((1-0.5)\theta_{12})}{\sin\theta_{12}}\mathbf{D}_1 + \frac{\sin(0.5\theta_{12})}{\sin\theta_{12}}\mathbf{D}_2\\ &= \frac{\sin 45^\circ}{\sin 90^\circ}(\mathbf{D}_1 + \mathbf{D}_2),\\ &= 0.7071[(0,1,0) + (1,0,0)]\\ &= (0.7071, 0.7071, 0).\end{aligned}$$

Notice that $\mathbf{D}_{1+0.5}$ is a unit vector pointing in a 45° direction in the xy plane.

We now have the new position $(-1.3125,-1.125,0)$ and new direction of view $(0.7071, 0.7071, 0)$ of the camera and we can use a perspective projection technique, such as the one described in Section 6.10, to calculate the projections of all the points in the scene. We assume that the projection plane is perpendicular to $\mathbf{D}_{1+0.5} = (0.7071, 0.7071, 0)$ and is located at a distance k from the camera (where k is a user-controlled parameter that may vary from frame to frame). Notice that the technique of Section 6.10 assumes that two viewing half-angles h and v are given. They correspond to the size of the projection plane. Any image point that would be projected outside that size should be ignored.

◇ **Exercise 19.4:** The new camera direction is $(0.7071, 0.7071, 0)$. In order to point at the origin, the camera should be located at a point with coordinates $(-c,-c,0)$, i.e., the x and y coordinates should be identical. We, however, got $\mathbf{P}_1(0.5) = (-1.3125,-1.125,0)$. What's the explanation?

◇ **Exercise 19.5:** If at $\mathbf{P}_1$ the camera should look at the positive y direction, and at $\mathbf{P}_2$, at the positive x direction, then midway it should look between these directions, i.e., at 45° or in the $(1,1,0)$ direction. In fact, if we want the camera to stop, for example, at $\mathbf{P}_1$, $\mathbf{P}_2$, and at three other points equally spaced in between, we know we should point the camera at angles of 0°, 22.5°, 45°, 67.5°, and 90° to the positive y direction at the five points and there seems to be no need for the direction vectors $\mathbf{D}_i$. What's the explanation?

◇ **Exercise 19.6:** Perform the same calculation for the second Bézier segment. Find the coordinates of point $\mathbf{P}_2(0.5)$ and compute the new camera direction $\mathbf{D}_{2+0.5}$ as a spherical interpolation of $\mathbf{D}_2$ and $\mathbf{D}_3$.

◇ **Exercise 19.7:** Calculate $\mathbf{D}_{3+0.5}$ for the third path segment.

19.7.2 Example 2

This is the same as Example 1, except that point $\mathbf{P}_2$ is moved to location $(-2,0,1)$. The camera path in this example is therefore not completely contained in the $z=0$ plane. The only direction vector that is different is $\mathbf{D}_2$, which becomes $(2,0,-1)$ or, after normalization, $(2,0,-1)/\sqrt{5}$. Only two angles are affected:

$$(\mathbf{D}_1 \bullet \mathbf{D}_2) = \frac{1}{\sqrt{5}}(0,1,0) \bullet (2,0,-1) = 0 \rightarrow \theta_{12} = 90^\circ,$$
$$(\mathbf{D}_2 \bullet \mathbf{D}_3) = \frac{1}{\sqrt{5}}(2,0,-1) \bullet (-3/5,-4/5,0) = -\frac{6}{5\sqrt{5}} \approx -0.5367 \rightarrow \theta_{23} = 122.46^\circ.$$

The two interpolations $\mathbf{D}_{1+0.5}$ and $\mathbf{D}_{2+0.5}$ are shown

$$\begin{aligned}\mathbf{D}_{1+0.5} &= \frac{\sin 45^\circ}{\sin 90^\circ}(\mathbf{D}_1+\mathbf{D}_2) = 0.7071\left[(0,1,0)+\frac{1}{\sqrt{5}}(2,0,-1)\right]\\ &= (0.6324, 0.7071, -0.3162),\\ \mathbf{D}_{2+0.5} &= \frac{\sin 61.23^\circ}{\sin 122.46^\circ}(\mathbf{D}_2+\mathbf{D}_3) = \frac{0.8766}{0.8438}\left[\frac{1}{\sqrt{5}}(2,0,-1)+(-3/5,-4/5,0)\right]\\ &= (0.3059, -0.8311, -0.4646).\end{aligned}$$

The new camera directions have a negative z component, since the camera itself is now located at points with positive z and should be looking at the origin. However, it is impossible to tell just by examining the interpolated directions whether they are the right ones. The best test is to actually implement the example in software.

19.7.3 Example 3

This is still a simple example, but this time the camera is aimed at different points in the scene while moving along its path. We assume a camera path that's a straight line from $\mathbf{P}_1 = (2,2,0)$ to $\mathbf{P}_2 = (1,1,0)$ (Figure 19.10a). The equation of this line is, of course, $\mathbf{P}_1(t) = (1-t)\mathbf{P}_1 + t\mathbf{P}_2 = (2-t, 2-t, 0)$, but notice that this equation is also easy to obtain with an interpolating Bézier curve, which is our standard method. All that's necessary is two more points, $\mathbf{P}_0 = (3,3,0)$ and $\mathbf{P}_3 = (0,0,0)$ which will define the start and end directions, respectively, of the curve, and will make it a straight line. We first calculate the two new interior points

$$\mathbf{X}_1 = \mathbf{P}_1 + \frac{1}{6}(\mathbf{P}_2 - \mathbf{P}_0) = \left(\frac{5}{3}, \frac{5}{3}, 0\right), \qquad \mathbf{Y}_1 = \mathbf{P}_2 - \frac{1}{6}(\mathbf{P}_3 - \mathbf{P}_1) = \left(\frac{4}{3}, \frac{4}{3}, 0\right).$$

The curve is, as usual,

$$\begin{aligned}\mathbf{P}_1(t) &= (1-t)^3\mathbf{P}_1 + 3t(1-t)^2\mathbf{X}_1 + 3t^2(1-t)\mathbf{Y}_1 + t^3\mathbf{P}_2\\ &= (1-t)^3(2,2,0) + 3t(1-t)^2\left(\frac{5}{3}, \frac{5}{3}, 0\right) + 3t^2(1-t)\left(\frac{4}{3}, \frac{4}{3}, 0\right) + t^3(1,1,0)\\ &= (2-t, 2-t, 0).\end{aligned}$$

We arbitrarily decide that at $\mathbf{P}_1$, the camera should look at point $\mathbf{S}_1 = (1.75, 1.75, -1)$, while at $\mathbf{P}_2$, it should look at $\mathbf{S}_2 = (1.25, 1.25, -1)$. The idea is to slide the camera along its simple path while panning it, so it covers the area between $\mathbf{S}_1$ and $\mathbf{S}_2$.

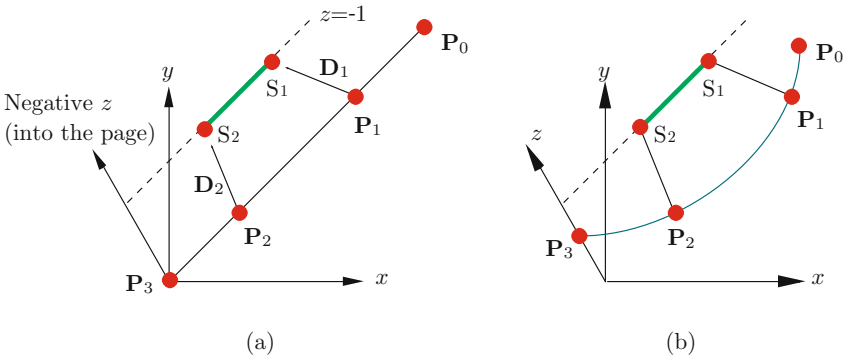


Figure 19.10: An Animation Path with Panning.

The two normalized direction vectors are

$$\begin{aligned}\mathbf{D}_1 &= \mathbf{S}_1 - \mathbf{P}_1 = (1.75, 1.75, -1) - (2, 2, 0)\\ &= (-0.25, -0.25, -1) \text{ normalized to } (-0.2357, -0.2357, -0.9428),\\ \mathbf{D}_2 &= \mathbf{S}_2 - \mathbf{P}_2 = (1.25, 1.25, -1) - (1, 1, 0)\\ &= (0.25, 0.25, -1) \text{ normalized to } (0.2357, 0.2357, -0.9428).\end{aligned}$$

The angle between them is

$$\cos\theta_{12} = (\mathbf{D}_1 \bullet \mathbf{D}_2) = -0.049 - 0.049 + 0.79 = 0.7777,$$

implying $\theta_{12} = 38.94°$. For $t = 0.5$, the position of the camera midway between $\mathbf{P}_1$ and $\mathbf{P}_2$ is given by the linear interpolation $\mathbf{P}_1(0.5) = (2 - 0.5, 2 - 0.5, 0) = (1.5, 1.5, 0)$. Its direction of view is calculated by the spherical interpolation

$$\mathbf{D}_{1+0.5} = \frac{\sin 19.47°}{\sin 38.94°}(\mathbf{D}_1 + \mathbf{D}_2) = 0.5303(0, 0, -1.8856) = (0, 0, -1).$$

Both values, the position and direction of view, are easy to verify visually because of the simple geometry of the problem.

◇ **Exercise 19.8:** Change the camera path from a straight line to an arc (Figure 19.10b) by moving the two extreme guide points $\mathbf{P}_0$ and $\mathbf{P}_3$ to positions $(3, 3, -0.25)$ and $(0, 0, -0.25)$, respectively. Notice that this will not change the interpolated directions of the camera.

19.8 Interpolating Orientations: II

The discussion so far has employed only two direction vectors, $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$, to compute the new camera orientation at each frame by spherical interpolation. This, however, may lead to a sudden change in camera direction at a key frame and thus to nonsmooth, jerky animation. As a simple example, imagine a two-segment camera path with direction vectors at three consecutive key frames pointing, respectively, in the positive x, y, and z directions. When the camera is moved along the first segment, it will change directions from the x to the y axis, so it will always point somewhere in the xy plane. When the camera switches to the second segment, it will start pointing somewhere in the yz plane. Switching directions between the two perpendicular xy and yz planes (Figure 19.11) may cause a jerk in the animation. The usual solution is to define key frames with direction vectors that don't differ by much. An alternative may be to derive a new spherical interpolation function that interpolates several consecutive direction vectors. When the camera moves in segment i, such a function should interpolate $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$ but should also assign weights to $\mathbf{D}_{i-1}$ (mostly at the start of segment i) and to $\mathbf{D}_{i+2}$ (mostly at the end of the segment).

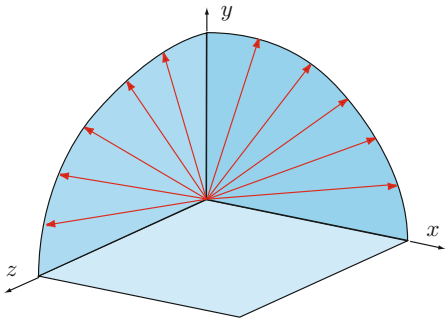


Figure 19.11: Abrupt Change of Direction.

We now show how such interpolation can be achieved. At a certain point, the animator has to input the $n-1$ direction vectors $\mathbf{D}_i$ at the key frames (in practice, the animator may input the coordinates of the points the camera should be looking at in every key frame and the software uses these points to calculate the direction vectors). In addition to these data, the animator should input two more direction vectors $\mathbf{X}_1$ and $\mathbf{Y}_{n-1}$, to provide the software with more direction information at the start and at the end of the path. The animation software then calculates "intermediate" direction vectors $\mathbf{X}_i$ and $\mathbf{Y}_i$ for each segment, in the same way the interior points are defined for the segment, i.e.,

$$\mathbf{X}_i = \mathbf{D}_i + \frac{1}{6}(\mathbf{D}_{i+1} - \mathbf{D}_{i-1}), \quad \mathbf{Y}_i = \mathbf{D}_{i+1} - \frac{1}{6}(\mathbf{D}_{i+2} - \mathbf{D}_i).$$

Once this is done, the software should do the following for each segment i of the camera path. Vary the time parameter t from 0 to 1, calculate spatial camera positions

$\mathbf{P}_i(t)$, and for each value t, use spherical interpolation to interpolate the four direction vectors $\mathbf{D}_i$, $\mathbf{X}_i$, $\mathbf{Y}_i$, and $\mathbf{D}_{i+1}$ of the segment. Initially, when t is close to 0, the interpolation should give more weight to $\mathbf{X}_i$ (and thus include a contribution from $\mathbf{D}_{i-1}$). Toward the end, when t gets close to 1, the same spherical interpolation should assign more weight to $\mathbf{Y}_i$ (and, thus, to $\mathbf{D}_{i+2}$).

Our problem is to find the right way to do this kind of spherical interpolation. The first thing that comes to mind is to define the interpolated camera direction $\mathbf{D}_{i+t}$ as the standard Bézier weighted sum,

$$\mathbf{D}_{i+t} = (1-t)^3\mathbf{D}_i + 3t(1-t)^2\mathbf{X}_i + 3t^2(1-t)\mathbf{Y}_i + t^3\mathbf{D}_{i+1}.$$

This certainly favors $\mathbf{X}_i$ in the early parts of the segment and favors $\mathbf{Y}_i$ in the later parts. However, since the Bézier curve has variable velocity, this kind of interpolation will produce direction vectors that are spread nonuniformly between $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$. What we need in this case is to extend spherical interpolation to four vectors, and we do this by means of the de Casteljau construction of Section 13.6, but with spherical instead of linear mediation. As a reminder, the mediation operator $t[\![\mathbf{P}_0, \mathbf{P}_1]\!]$ between two points $\mathbf{P}_0$ and $\mathbf{P}_1$ is defined as

$$t[\![\mathbf{P}_0, \mathbf{P}_1]\!] = t\mathbf{P}_1 + (1-t)\mathbf{P}_0 = t(\mathbf{P}_1 - \mathbf{P}_0) + \mathbf{P}_0, \quad \text{where} \quad 0 \le t \le 1.$$

We now use our spherical interpolation, Equation (19.2), as a *spherical mediation operator* and apply it to construct the scaffolding of the four direction vectors in the same way it is done for four points (see Page 650 and Figure 13.8). We use the notation $[\mathbf{A}; \mathbf{B}; t]$ to denote the spherical interpolation of vectors $\mathbf{A}$ and $\mathbf{B}$ (Equation (19.3)) and we construct the scaffold in three steps.

1. Calculate the three interpolated direction vectors

$$\mathbf{P}_{01} = [\mathbf{D}_i; \mathbf{X}_i; t], \quad \mathbf{P}_{02} = [\mathbf{X}_i; \mathbf{Y}_i; t], \quad \text{and} \quad \mathbf{P}_{03} = [\mathbf{Y}_i; \mathbf{D}_{i+1}; t].$$

2. Calculate the two interpolated direction vectors $\mathbf{P}_{11} = [\mathbf{P}_{01}; \mathbf{P}_{02}; t]$ and $\mathbf{P}_{12} = [\mathbf{P}_{02}; \mathbf{P}_{03}; t]$.

3. Compute the final interpolated direction vector $\mathbf{D}_{i+t} = [\mathbf{P}_{11}; \mathbf{P}_{12}; t]$. This becomes the direction of the camera at point $\mathbf{P}_i(t)$.

When the camera is moved to point $\mathbf{P}_i(t)$ and is oriented there, pointing in direction $\mathbf{D}_{i+t}$, we can expect smooth animation since direction $\mathbf{D}_{i+t}$ not only takes into account $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$ but also "remembers" the past direction $\mathbf{D}_{i-1}$ and "anticipates" the future direction $\mathbf{D}_{i+2}$. Notice that four such direction vectors are available at every key frame, including the first and last ones, since the animator inputs the two extra direction vectors $\mathbf{X}_1$ and $\mathbf{Y}_{n-1}$ explicitly.

19.8.1 Example 4

The same points and direction vectors of Example 1 of Section 19.7.1 are used. The normalized direction vectors are

$$\mathbf{D}_1 = -\mathbf{P}_1 = (0,1,0), \quad \mathbf{D}_2 = -\mathbf{P}_2 = (1,0,0), \quad \mathbf{D}_3 = -\mathbf{P}_3 = (-3/5, -4/5, 0),$$
$$\mathbf{D}_4 = -\mathbf{P}_4 = (-1,0,0), \quad \mathbf{D}_5 = -\mathbf{P}_5 = (-0.83, 0.55, 0).$$

We assume that the animator inputs the two extra directions, $\mathbf{X}_1 = (1, 3, 0)$ and $\mathbf{Y}_4 = (-1, 0.5, 0)$. All other "interior" directions are now calculated and normalized:

$$\mathbf{X}_1 = (1, 3, 0) \rightarrow (0.3162, 0.9487, 0),$$
$$\mathbf{Y}_1 = \mathbf{D}_2 - \frac{1}{6}(\mathbf{D}_3 - \mathbf{D}_1) = (1.1, 0.3, 0) \rightarrow (0.964764, 0.263117, 0),$$
$$\mathbf{X}_2 = \mathbf{D}_2 + \frac{1}{6}(\mathbf{D}_3 - \mathbf{D}_1) = (0.9, -0.3, 0) \rightarrow (0.948683, -0.316228, 0),$$
$$\mathbf{Y}_2 = \mathbf{D}_3 - \frac{1}{6}(\mathbf{D}_4 - \mathbf{D}_2) = (-0.2667, -0.8, 0) \rightarrow (-0.316228, -0.948683, 0),$$
$$\mathbf{X}_3 = \mathbf{D}_3 + \frac{1}{6}(\mathbf{D}_4 - \mathbf{D}_2) = (-0.9333, -0.8, 0) \rightarrow (-0.759257, -0.650791, 0),$$
$$\mathbf{Y}_3 = \mathbf{D}_4 - \frac{1}{6}(\mathbf{D}_5 - \mathbf{D}_3) = (-0.9617, -0.225, 0) \rightarrow (-0.973704, -0.227816, 0),$$
$$\mathbf{X}_4 = \mathbf{D}_4 + \frac{1}{6}(\mathbf{D}_5 - \mathbf{D}_3) = (-1.0383, 0.225, 0) \rightarrow (-0.977318, 0.211778, 0),$$
$$\mathbf{Y}_4 = (-1, 0.5, 0) \rightarrow (-0.8944, 0.4472, 0).$$

We now calculate $\mathbf{D}_{1+0.5}$ in three steps.
Step 1: Calculate the three interpolated direction vectors

$$\mathbf{P}_{01} = [\mathbf{D}_1; \mathbf{X}_1; .5] = \frac{\sin 18.43^\circ}{\sin 9.215^\circ}(\mathbf{D}_1 + \mathbf{X}_1) = (0.160167, 0.987089, 0),$$
$$\mathbf{P}_{02} = [\mathbf{X}_1; \mathbf{Y}_1; .5] = \frac{\sin 56.31^\circ}{\sin 28.155^\circ}(\mathbf{X}_1 + \mathbf{Y}_1) = (0.726438, 0.687225, 0),$$
$$\mathbf{P}_{03} = [\mathbf{Y}_1; \mathbf{D}_2; .5] = \frac{\sin 15.255^\circ}{\sin 7.628^\circ}(\mathbf{Y}_1 + \mathbf{D}_2) = (0.991152, 0.132733, 0).$$

Step 2: Calculate the two interpolated direction vectors

$$\mathbf{P}_{11} = [\mathbf{P}_{01}; \mathbf{P}_{02}; .5] = \frac{\sin 37.37^\circ}{\sin 18.69^\circ}(\mathbf{P}_{01} + \mathbf{P}_{02}) = (0.46797, 0.883741, 0),$$
$$\mathbf{P}_{12} = [\mathbf{P}_{02}; \mathbf{P}_{03}; .5] = \frac{\sin 35.78^\circ}{\sin 17.89^\circ}(\mathbf{P}_{02} + \mathbf{P}_{03}) = (0.902439, 0.430814, 0).$$

Step 3: Calculate the final interpolated direction vector

$$\mathbf{D}_{1+0.5} = [\mathbf{P}_{11}; \mathbf{P}_{12}; 0.5] = \frac{\sin 36.58^\circ}{\sin 18.29^\circ}(\mathbf{P}_{11} + \mathbf{P}_{12}) = (0.721658, 0.692245, 0).$$

This becomes the direction of the camera at point $\mathbf{P}_1(0.5) = (-1.3125, -1.125, 0)$. Notice that it differs from the 45° direction calculated in Example 1, since it depends on the choice of the "exterior" direction $\mathbf{X}_1$ that was input by the animator.

19.8.2 Interpolating Orientations: III

This approach to the problem of interpolating orientations uses *quaternions*. These mathematical entities are introduced in Appendix B and their application to general

rotations is discussed in Section 4.4.5. The reader should review these sections prior to reading this section. Quaternions can be used in computer animation to interpolate orientations between key frames because of two facts:

1. A general rotation of θ degrees about an axis $\mathbf{u}$ can be expressed by the unit quaternion $\mathbf{q} = [\cos(\theta/2), \sin(\theta/2)\mathbf{u}]$.

2. When a rigid object is sent flying through space, it may roll and tumble in a complicated way, but at any moment, its position and orientation can be completely described by two transformations—a translation from its initial position to its present position and a rotation of θ degrees about some axis $\mathbf{u}$.

We can imagine all the unit quaternions to form a unit four-dimensional sphere. Spherically interpolating two unit quaternions is therefore equivalent to moving along a great arc on this sphere. The technique is identical to the one discussed in Section 19.5 for vectors. Interpolating camera orientation between two key frames by using quaternions is done in the following steps:

1. The animator inputs the data for all the key frames. The software uses this to calculate the directions of view $\mathbf{D}_i$ for each key frame i.

2. The software "positions" the camera at the preferred point $(0, 0, -k)$, looking in the positive z direction (i.e., in direction $(0, 0, 1)$).

3. A quaternion $\mathbf{q}_i$ is calculated for each key frame i, describing the rotation that would bring the camera *from its initial orientation* $(0, 0, 1)$ to its new orientation in key frame i.

4. The software goes into a loop where it moves the camera along its path, segment by segment. In segment i (the segment between key frames i and $i + 1$), the time parameter t is incremented from 0 to 1 in F steps. In each step t_m, the camera is translated to position $\mathbf{P}_i(t_m)$ and is reoriented by rotating it. The main point is that the translation is done *from the initial position* $(0, 0, -k)$, and the rotation is done *from the initial orientation* $(0, 0, 1)$. The software uses t_m to spherically interpolate the two quaternions $\mathbf{q}_i$ and $\mathbf{q}_{i+1}$ to a quaternion $\mathbf{q}_{i+t_m}$. Quaternion $\mathbf{q}_i$ describes a rotation from $(0, 0, 1)$ to key frame i. Similarly, $\mathbf{q}_{i+1}$ describes a rotation from $(0, 0, 1)$ to key frame $i + 1$. Thus, their interpolation describes a rotation that will bring the camera from its initial orientation $(0, 0, 1)$ to the orientation it should have at point $\mathbf{P}_i(t_m)$.

(The discussion of Section 19.8 suggests that four quaternions, instead of two, should participate in any interpolation. The software should therefore calculate two auxiliary quaternions $\mathbf{X}_i$ and $\mathbf{Y}_i$ for each segment and use the scaffolding construction on $\mathbf{q}_i$, $\mathbf{X}_i$, $\mathbf{Y}_i$, and $\mathbf{q}_{i+1}$ to calculate $\mathbf{q}_{i+t_m}$.)

5. Once $\mathbf{q}_{i+t_m}$ is obtained, the software uses it to generate a rotation matrix $\mathbf{M}$ according to Equation (4.33). This matrix is then used to take a snapshot. The snapshot can be taken with the methods of Section 6.9 or 6.10. An alternative is the technique of Section 6.6, which is the one used here. The principle is the following: We know that the camera had to be translated from its initial position $(0, 0, -k)$ to its present position $\mathbf{P}_i(t_m)$ and rotated according to $\mathbf{q}_{i+t_m}$. The software simply applies the two *reverse* transformations (and in reverse order) to every point of the scene, thus "bringing the scene to the camera" (which remains in its preferred position) instead of bringing the camera to the scene. Once the scene is brought to the camera, any point in the scene can be projected using the standard projection matrix $\mathbf{T}_p$, Equation (6.6).

19.8.3 Example 5

The camera is located at the preferred point $(0, 0, -k)$ (here, we assume that $0 < k < 1$), looking in the preferred direction $\mathbf{D} = (0, 0, 1)$. Three key frames are defined, at points $\mathbf{P}_1 = (-1, 0, 2)$, $\mathbf{P}_2 = (0, 0, 1)$, and $\mathbf{P}_3 = (1, 0, 2.5)$ (Figure 19.12a, where a right-handed coordinate system implies that the y axis should come out of the page). The center of interest (the direction the camera should be looking at) is arbitrarily selected as point $(0, 0, 2)$. The three direction vectors are, thus,

$$\begin{aligned}
\mathbf{D}_1 &= (0, 0, 2) - (-1, 0, 2) = (1, 0, 0), \\
\mathbf{D}_2 &= (0, 0, 2) - (0, 0, 1) = (0, 0, 1), \\
\mathbf{D}_3 &= (0, 0, 2) - (1, 0, 2.5) = (-1, 0, -0.5), \quad \text{normalized to } (-0.8944, 0, -0.4472).
\end{aligned}$$

The angles between each direction vector and the original direction $\mathbf{D}$ are (Figure 19.12b)

$$\begin{aligned}
\cos\theta_1 &= \mathbf{D} \bullet \mathbf{D}_1 = (0, 0, 1) \bullet (1, 0, 0) = 0 \rightarrow \theta_1 = 90^\circ, \\
\cos\theta_2 &= \mathbf{D} \bullet \mathbf{D}_2 = (0, 0, 1) \bullet (0, 0, 1) = 1 \rightarrow \theta_2 = 0^\circ, \\
\cos\theta_3 &= \mathbf{D} \bullet \mathbf{D}_3 = (0, 0, 1) \bullet (-.8944, 0, -.4472) = -.4472 \rightarrow \theta_3 = 116.56^\circ.
\end{aligned}$$

The quaternions for the three key frames can now be calculated:

$$\begin{aligned}
\mathbf{q}_1 &= [\cos(\theta_1/2), \sin(\theta_1/2)(0, 1, 0)] = (0.7071, 0, 0.7071, 0), \\
\mathbf{q}_2 &= [\cos(\theta_2/2), \sin(\theta_2/2)(0, 1, 0)] = (1, 0, 0, 0), \\
\mathbf{q}_3 &= [\cos(\theta_3/2), \sin(\theta_3/2)(0, -1, 0)] = (0.5258, 0, -0.8506, 0).
\end{aligned}$$

Notice that $\mathbf{q}_1$ corresponds to a clockwise rotation about the positive y axis, whereas $\mathbf{q}_3$ corresponds to a clockwise rotation about the *negative* y axis (Figure 19.12b). This is the reason for using direction $(0, -1, 0)$ as the rotation axis for the latter. The axis of rotation for quaternion $\mathbf{q}_i$ is simply the cross-product $\mathbf{D} \times \mathbf{D}_i$, where the unnormalized form of $\mathbf{D}_i$ is used.

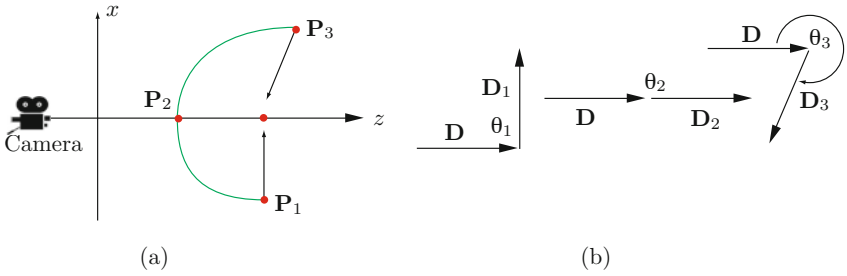


Figure 19.12: A Three-Point Animation Path.

The quaternions are now used to calculate, as an example, the two interpolations $\mathbf{q}_{1+0.5}$ and $\mathbf{q}_{2+0.5}$. From $\mathbf{q}_1 \bullet \mathbf{q}_2 = 0.7071$, we find that the angle between $\mathbf{q}_1$ and $\mathbf{q}_2$ is

45°. Similarly, $\mathbf{q}_2 \bullet \mathbf{q}_3 = 0.5258$ implies that the angle between them is 58.28°. (Notice that these are angles between quaternions, not between the direction vectors.) Using these angles, we get

$$\begin{aligned}
\mathbf{q}_{1+0.5} &= \frac{\sin(45°/2)}{\sin 45°}\,[(0.7071, 0, 0.7071, 0) + (1, 0, 0, 0)] \\
&= \frac{0.3829}{0.7071}(1.7071, 0, 0.7071, 0) \\
&= (0.9239, 0, 0.3829, 0) = (\cos 22.5°, 0, \sin 22.5°, 0), \\
\mathbf{q}_{2+0.5} &= \frac{\sin(58.28°/2)}{\sin 58.28°}\,[(1, 0, 0, 0) + (0.5258, 0, -0.8506, 0)] \\
&= \frac{0.4869}{0.85}(1.5258, 0, 0.8506, 0) \\
&= (0.8734, 0, 0.4869, 0) = (\cos -29.14°, 0, \sin -29.14°, 0).
\end{aligned}$$

Quaternion $\mathbf{q}_{1+0.5}$ thus generates a rotation of $22.5 \times 2 = 45°$ from the initial direction $(0, 0, 1)$ about the y axis. Quaternion $\mathbf{q}_{2+0.5}$ corresponds to a rotation of $-29.14 \times 2 = -58.28°$ *from the same initial direction* about the same axis. (Notice how the camera has to be rotated in opposite directions for $\mathbf{q}_{1+0.5}$ and $\mathbf{q}_{2+0.5}$.)

Here are the details of the snapshots for the first two key frames. At $\mathbf{P}_1$, the camera has to go (Figure 19.13) through the three transformations (1) translate to the origin (k units in the positive z direction), (2) rotate 90° clockwise about the positive y axis, and (3) translate one unit in the negative x and two units in the positive z directions. Since we leave the camera in place, we have to apply the *reverse transformations in reverse order* to the scene: (4) translate one unit in the positive x and two units in the negative z directions, (5) rotate 90° about the origin, counterclockwise about the positive y axis, and (6) translate k units in the negative z direction. Notice how the relative positions of the camera and scene are the same in parts (3) and (6) of the figure.

At $\mathbf{P}_2$, the camera has to go through the two transformations (Figure 19.14): (1) translate to the origin (k units in the positive z direction), and (2) translate one unit in the positive z direction. We again apply the reverse transformations in reverse order to the scene, (3) translate one unit in the negative z direction, and (4) translate k units in the negative z direction.

◇ **Exercise 19.9:** Describe the transformations for key frame 3.

Thus, a general snapshot is taken as follows:

1. Determine the camera position $\mathbf{P}_i(t_m)$. Assume that this is point (a, b, c).
2. Calculate $\mathbf{q}_{i+t_m}$ by interpolating $\mathbf{q}_i$ and $\mathbf{q}_{i+1}$. Assume that this is quaternion (w, x, y, z).
3. The camera is translated to the origin. This is done by matrix $\mathbf{T}_1$ below.
4. The camera is rotated by matrix $\mathbf{M}$, Equation (4.33).
5. The camera is translated the rest of the way to point (a, b, c), i.e., by an amount

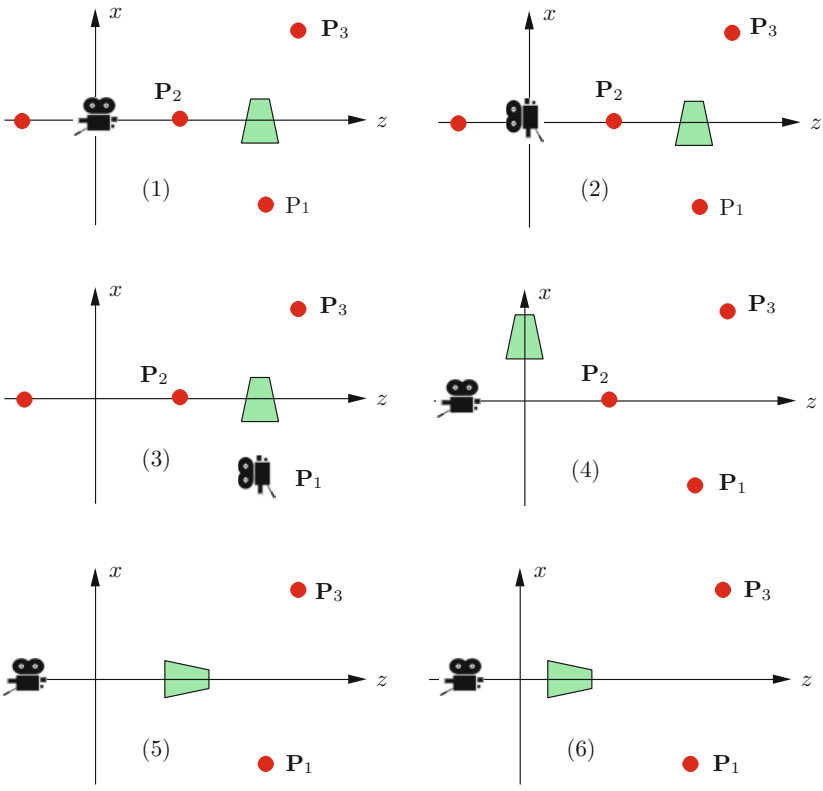


Figure 19.13: Camera (1–3) and Scene (4–6) Transformations for $\mathbf{P}_1$.

$(a, b, c-k)$ using matrix $\mathbf{T}_2$:

$$\mathbf{T}_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & k & 1 \end{pmatrix}, \qquad \mathbf{T}_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c-k & 1 \end{pmatrix},$$

$$\mathbf{M} = \begin{pmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy & 0 \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz-2wx & 0 \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

6. Since we want to apply the reverse transformations to the scene (and in reverse

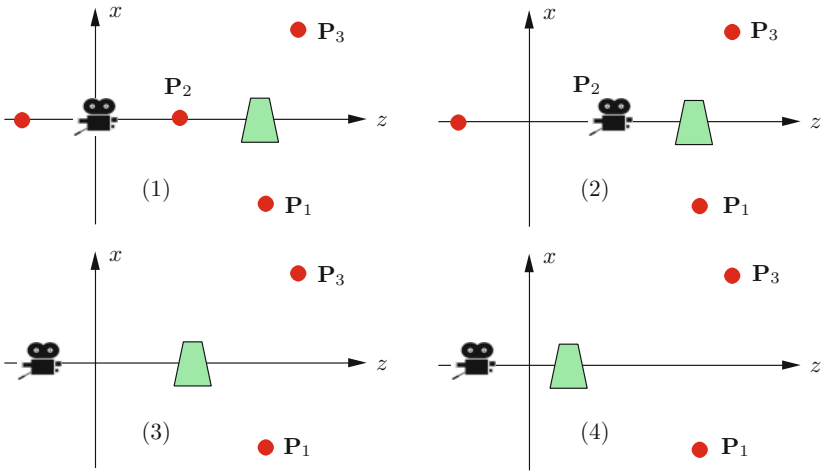


Figure 19.14: Camera (1–2) and Scene (3–4) Transformations for $\mathbf{P}_2$.

order), we multiply each point of the scene by

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & -b & -c+k & 1 \end{pmatrix} \cdot \mathbf{M}^{-1} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -k & 1 \end{pmatrix} \cdot \mathbf{T}_p,$$

where $\mathbf{T}_p$ is the standard projection matrix, Equation (6.6). (The inverse of matrix $\mathbf{M}$ can be calculated in general by appropriate software, but it is too big and complex to list here.)

19.9 Nonuniform Interpolation

Spherical interpolation has been specifically developed for uniform change of orientation. Varying t in equal steps produces direction vectors that are uniformly distributed between directions $\mathbf{D}_i$ and $\mathbf{D}_{i+1}$. Sometimes, however, nonuniform changes of orientation and/or position are required. In such cases, a function $T(t)$ is needed, such that varying t in equal steps will vary $T(t)$ from 0 to 1 in unequal steps. If T is used to position the camera, this will simulate acceleration or deceleration of the animation. If T is used to interpolate camera orientation, this will simulate rotating the camera at nonuniform rates. The methods presented here for nonuniform interpolation are based on the concept of blending (Section 8.5).

19.9.1 Quadratic and Cubic Blending

The well-known expression $P(t) = (1-t)P_0 + tP_1$ (Equation (9.1)) can be considered a blending of the two values P_0 and P_1. It blends a $(1-t)$ fraction of P_0 with a t fraction of P_1. The weights (or fractions) should add up to 1. Since this expression is linear in t, we can call it *linear blending*.

It is possible to blend values (numbers, points, vectors, etc.) in nonlinear ways. Section 13.5.1 is a short discussion of the concepts involved. Nonlinear blending seems the best approach for nonuniform interpolation and we start by exploring quadratic blending, i.e., ways to blend two quantities by using weights that employ t^2. The simplest approach is to generalize the linear expression above by squaring t and $(1-t)$. This results in $(1-t)^2P_1 + t^2P_2$, which varies from P_1 (for $t = 0$) to P_2 (for $t = 1$). However, this expression is clearly wrong since the two weights $(1-t)^2$ and t^2 do not add up to 1 and therefore cannot serve as fractions. It is possible to correct this by artificially adding the missing term $2t(1-t)$ (recall that $(1-t)^2 + 2t(1-t) + t^2 = 1$). We now notice that the term $2t(1-t)$ is zero when $t = 0$ and also when $t = 1$. It therefore does not affect the blending at the extreme values, but it must have an effect on the blending in between. We can therefore multiply this term by any quantity P_w and find out how various values of P_w affect the blending. Figure 19.15a shows a blending of the form $(1-t)^2P_1 + 2t(1-t)P_w + t^2P_2$ created by the *Mathematica* code of Figure 19.15c. The figure shows the results of blending $P_0 = 0$ and $P_1 = 1$ for five values of P_w ranging from 0 to 1. Notice that certain values of P_w create blendings outside the range $[P_0, P_1]$, but, in general, quadratic blending can give satisfactory results in many, perhaps most, practical cases.

Similarly, if we try a cubic blend by simply writing $P(t) = (1-t)^3P_1 + t^3P_2$, we end up with the same problem. Cubic blending can be achieved by adding four terms with weights t^3, $3t^2(1-t)$, $3t(1-t)^2$, and $(1-t)^3$. Figure 19.15b shows the results of cubically blending the two values $P_0 = 0$ and $P_3 = 1$. It calculates

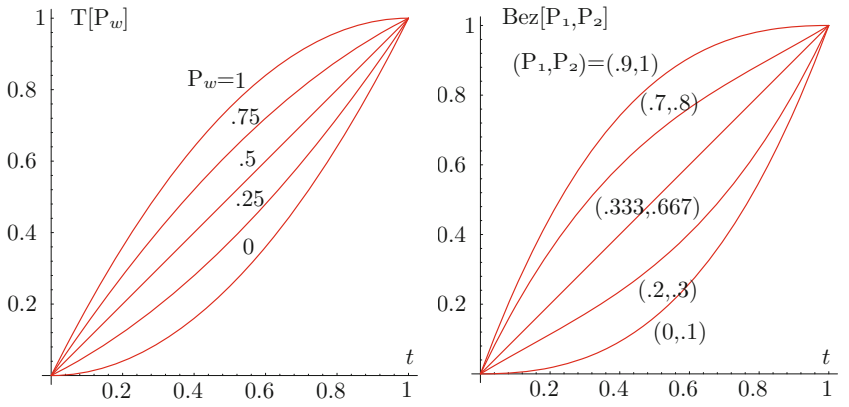
$$t^3P_0 + 3t^2(1-t)P_1 + 3t(1-t)^2P_2 + (1-t)^3P_3,$$

for the five pairs of "interior" weights (P_1, P_2) set to $(0, 0.1)$, $(0.2, 0.3)$, $(0.333, 0.667)$, $(0.7, 0.8)$, and $(0.9, 1)$.

We next notice that the expressions for the linear, quadratic, and cubic blends are identical to the parametric sums used to construct the Bézier curve. This suggests a way to define parametric blends for cases where complex behavior of $T(t)$ is required. In general, a parametric blend $T(t)$ that uses the $n-1$ parameters P_1, P_2,..., P_{n-1} to blend the two quantities P_0 and P_n should have the form

$$T(t) = \sum_{i=0}^{n} P_i B_{ni}(t),$$

where $B_{ni}(t)$ are the Bernstein polynomials of degree n The fact that the Bézier curve is an ideal tool for blending numbers also suggests how to obtain smooth blending across key frames. We know from Section 13.5 how to connect individual Bézier segments smoothly. The same idea can be used when numbers are blended. Suppose, for example,



(a) (b)

```
Clear[T];p1=0;p2=1;(*Quadratic Blending*)
T[pw_]:=Plot[(1-t)^2 p1+2t (1-t)pw+t^2 p2,{t,0,1},
 PlotStyle->{Red, AbsoluteThickness[.5]}];
Show[T[0],T[.25],T[.5],T[.75],T[1],
 PlotRange->All,AspectRatio->Automatic]

Clear[Bez];p0=0;p3=1;(*Bezier Blending*)Bez[p1_,p2_]:=
Plot[(1-t)^3 p0+3t (1-t)^2p1+3t^2(1-t)p2+t^3 p3,{t,0,1},
 AspectRatio->Automatic,PlotStyle->{Red, AbsoluteThickness[.5]}];
Show[Bez[0,.1],Bez[.2,.3],Bez[.333,.667],Bez[.7,.8],Bez[.9,1],
 PlotRange->All]
```

(c)

Figure 19.15: (a) Quadratic Blending. (b) Cubic Blending. (c) Code.

that we use a five-point (i.e., $n = 4$) Bézier blending to advance $T(t)$ nonuniformly from 0 to 1 in each of our key frames. For each key frame, we therefore have to select $P_0 = 0$, $P_4 = 1$, plus three parameters P_1, P_2, and P_3 to control the precise way T varies. If we want T to have the same speed on both sides of a key frame, we have to make sure that the difference $P_4 - P_3 = 1 - P_3$ in the segment to the left of the key frame equals the difference $P_1 - P_0 = P_1 - 0$ in the segment to the right of the same key frame. If we select, for example, $P_3 = 0.9$ in one key frame, then we should select $P_1 = 0.1$ in the next key frame.

It is also possible to use the Hermite interpolation (Section 11.1) to blend two values v_1 and v_2 in different proportions by using two parameters s and e.

Hermite interpolation has been developed to construct a curve by blending two points and two tangent vectors. It can also be applied to blend any two numbers v_1 and v_2, by using two user-defined "rates of change" s and e. Equation (11.7) can be modified by substituting v_1 for $\mathbf{P}_1$, v_2 for $\mathbf{P}_2$, and s and e for $\mathbf{P}_1^t$ and $\mathbf{P}_2^t$, respectively.

The result is

$$T(t) = (t^3, t^2, t, 1) \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ s \\ e \end{pmatrix}.$$

The quantities s and e can be considered the start and end "slopes" or rates of change of the blending. Figure 19.16a illustrates the results of the five Hermite interpolations of $v_1 = 0$ and $v_2 = 1$ for $e = 0$ and s values ranging from 0 to 4. It is easy to see how s affects the start slope of the interpolation. Figure 19.16b shows the results of similar interpolations for identical s and e values ranging from 0 to 4. Notice that both the start and end slopes are affected.

Ease-in/Ease-out: This term refers to nonuniform velocity that starts with acceleration, gradually changes to constant speed, then decelerates. Figure 19.17 shows a typical example. One way to achieve this effect is to set parameters $0 \le a \le b \le 1$ and use the sine function to interpolate and define a parameter $T(t)$ that accelerates when t varies from 0 to a, decelerates when t varies from b to 1, and is linear in the range $[a, b]$. Mathematically, this is expressed by Equation (19.4) (where the second line scales $T(t)$ to the range $[0, 1]$). Figure 19.17 illustrates the result. Notice that the precise shape of $T(t)$ depends on the values of a and b.

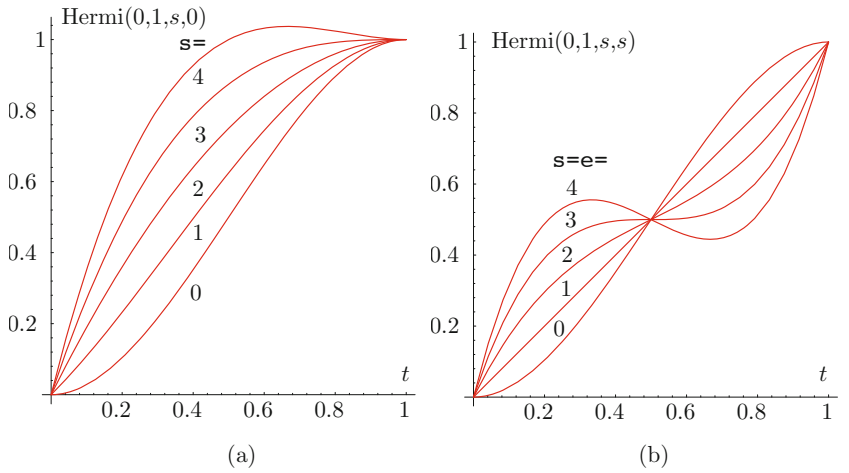
$$T_0(t) = \begin{cases} \dfrac{2a}{\pi} \sin\left(\dfrac{\pi}{2} \cdot \dfrac{t-a}{a}\right), & t < a, \\ \dfrac{2(1-b)}{\pi} \sin\left(\dfrac{\pi}{2} \cdot \dfrac{t-b}{1-b}\right) + \dfrac{2a}{\pi} + b - a, & t > b, \\ \dfrac{2a}{\pi} + t - a, & a \le t \le b. \end{cases}$$

$$T(t) = T_0(t) / \left(\frac{2a}{\pi} + \frac{2(1-b)}{\pi} + b - a\right). \tag{19.4}$$

◇ **Exercise 19.10:** Calculate the acceleration of $T(t)$ in the initial interval $[0, a]$ and its deceleration in the final interval $[b, 1]$.

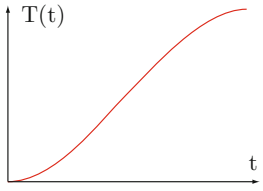
The same effect of ease-in/ease-out can be obtained from physical considerations, without the use of the sine function, by integrating speed to obtain position. We first decide what speed $v(t)$ we want in each subrange of $[0, 1]$, then integrate $v(t)$ to obtain the position $T(t)$ as a function of t in each subrange. Equation (19.5) describes a speed $v(t)$ that increases from zero to a certain value V in subrange $[0, a)$, decreases from V to zero in subrange $(b, 1]$, and is constant in between:

$$v(t) = \begin{cases} V \cdot \dfrac{t}{a}, & t < a, \\ V, & a \le t \le b, \\ V - V \cdot \dfrac{t-b}{1-b} = V \cdot \dfrac{1-t}{1-b}, & t > b. \end{cases} \tag{19.5}$$




```
Clear[T,H,Hermi]; (* Hermite Interpolation *)
T={t^3,t^2,t,1};
H={{2,-2,1,1},{-3,3,-2,-1},{0,0,1,0},{1,0,0,0}};
(*B={0,1,0,0};*)
Hermi[v1_,v2_,s_,e_]:=Plot[T.H.{v1,v2,s,e},{t,0,1},
 AspectRatio->Automatic, Prolog->AbsoluteThickness[.4]];
Show[Hermi[0,1,0,0], Hermi[0,1,1,1], Hermi[0,1,2,2],
 Hermi[0,1,3,3], Hermi[0,1,4,4]]
```

Figure 19.16: Hermite Interpolation.




```
Clear[fa,fb,fm,den,a,b];
a=.1; b=.3;
fa:=2a(Sin[Pi(t-a)/(2a)]+1)/Pi;
fb:=Sin[Pi(t-b)/(2(1-b))]2(1-b)/Pi+2a/Pi+b-a;
fm:=2a/Pi+t-a;
den=2a/Pi+2(1-b)/Pi+b-a;
T:=If[t<a,fa/den,If[t>b,fb/den,fm/den]];
Plot[T, {t,0,1}, AspectRatio->1]
```

Figure 19.17: Ease-in/Ease-out with a Sine Function.

We can find the total distance traveled in each subrange by integrating $v(t)$:

$$\int_0^a \frac{Vt}{a}\,dt = \frac{1}{2}\frac{V}{a}(a^2 - 0),$$
$$\int_a^b V\,dt = V(b-a),$$
$$\int_b^1 V\frac{1-t}{1-b}\,dt = \frac{V(1-b)^2}{2(1-b)} = \frac{1}{2}V(1-b).$$

Thus, the total distance for the three subranges is

$$\frac{1}{2}Va + V(b-a) + \frac{1}{2}V(1-b) = \frac{b-a+1}{2}.$$

If we want this distance to equal one unit, we should select $V = 2/(b-a+1)$. The distance $T(t)$ traveled in the first subrange is

$$\int_0^t \frac{Vt}{a}\,dt = \frac{Vt^2}{2a}.$$

For $a \le t \le b$, the total distance traveled from the start of the curve is

$$\frac{1}{2}Va + \int_a^t V\,dt = \frac{1}{2}Va + V(t-a).$$

(Notice that for $t = a$, this equals the distance traveled in the first subrange.) For $b \le t \le 1$, the total distance traveled from the start of the curve is

$$\frac{V}{2}a + V(b-a) + \int_b^t V\frac{1-t}{1-b}\,dt = \frac{V}{2}a + V(b-a) + \left[-\frac{V(1-t)^2}{2(1-b)} + \frac{V(1-b)^2}{2(1-b)}\right]$$
$$= \frac{V}{2}a + V(b-a) + \frac{V}{2(1-b)}[-2b + b^2 + 2t - t^2].$$

These methods can be generalized to obtain other types of nonuniform speeds.

> When graphing a function, the width of the line should be inversely proportional to the precision of the data.
>
> —Marvin J. Albinak.

19.10 Morphing

The technique of in-betweening is one of the main advantages of computer animation. This technique has been mentioned before, but it can also be implemented by means of *morphing*. The idea in morphing is for an artist or designer to prepare two key frames of animation and use software to generate all the in-between frames automatically.

The *metamorphosis* of an object, a topic that came to be known as *morphing*, is the case where two pictures are painted by an artist and are designated as the first and last frames of a scene. The artist specifies what points on the first and last frames correspond to each other and the computer then creates several intermediate frames by interpolating each point.

The word morphing is derived from the Greek $\mu o \rho \phi \epsilon$, meaning form or shape.

Let's assume that point $\mathbf{P}_1$ in the first frame corresponds to point $\mathbf{P}_2$ in the last frame and that four intermediate frames are needed. The coordinates of the point in the four frames are simply $t[\![\mathbf{P}_1, \mathbf{P}_2]\!]$, where $t = 0.2, 0.4, 0.6, 0.8$. Figure 19.18 is a simple example of morphing (see also Plates K.1 and K.3). To us, it seems that only two objects are involved, a start face and an end face. To the computer, however, each component, such as eye, nose, and mouth, has to be transformed separately.

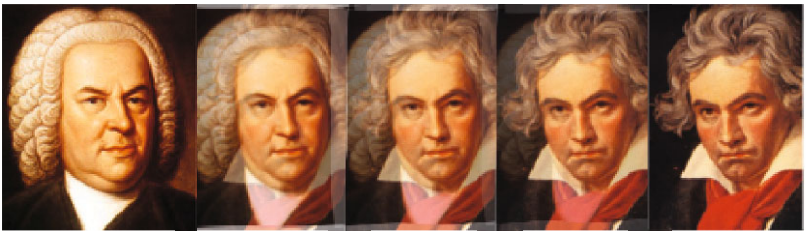


Figure 19.18: Morphing Bach to Beethoven.

You know the funny thing about morphin? You don't appreciate it till you can't do it anymore!

—David Yost (as Billy) in *Mighty Morphin Power Rangers: The Movie* (1995).

19.11 Free-Form Deformations

Free-form deformation is a modern technique based on old concepts. The principle is to employ a grid of control points combined with two-dimensional interpolation to distort an image in a systematic way, in order to achieve special effects.

Figure 19.19 shows objects in grids that are distorted in two different ways (see also Plates E.1, N.3, and R.2). Such effects can be useful in computer animation. The principle is to construct a bounding box around the object and partition it into a regular grid. The box is then deformed in the desired way and enough control points are placed at strategic locations on the grid to fully specify the deformation (points $\mathbf{P}_{11}$ through $\mathbf{P}_{33}$ in Figure 19.19). The image is then displayed, point by point, where each point is transformed from the original bounding box to the deformed box based on its original coordinates and on the control points.

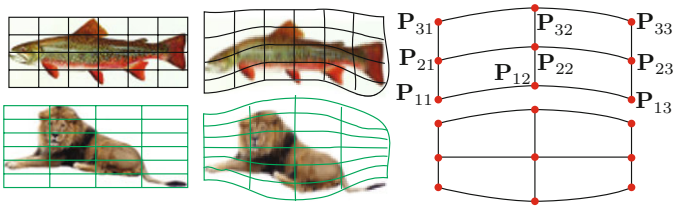


Figure 19.19: Free-Form Deformations.

We denote the coordinates of the bottom-left and top-right corners of the bounding box by (x_{min}, y_{min}) and (x_{max}, y_{max}), respectively. A point $\mathbf{P} = (x, y)$ in the image that is being deformed is transformed to $\mathbf{P}^* = (x^*, y^*)$ in two steps as follows:

1. Its position relative to the two corners of the bounding box is first determined by

$$u = \frac{(x - x_{min})}{(x_{max} - x_{min})} \qquad \text{and} \qquad w = \frac{(y - y_{min})}{(y_{max} - y_{min})}.$$

Notice that u and w are in the interval $[0, 1]$.

2. Its new, deformed coordinates are computed using appropriate two-dimensional interpolation. In our example there are 3×3 control points, so we denote $n = 3$ and use biquadratic interpolation (Section 2.4) to obtain

$$\begin{aligned}(x^*, y^*) &= \left((1-u)^2, 2u(1-u), u^2\right) \begin{pmatrix} \mathbf{P}_{11} & \mathbf{P}_{12} & \mathbf{P}_{13} \\ \mathbf{P}_{21} & \mathbf{P}_{22} & \mathbf{P}_{23} \\ \mathbf{P}_{31} & \mathbf{P}_{32} & \mathbf{P}_{33} \end{pmatrix} \begin{pmatrix} (1-w)^2 \\ 2w(1-w) \\ w^2 \end{pmatrix} \\ &= \left(B_{20}(u), B_{21}(u), B_{22}(u)\right) \begin{pmatrix} \mathbf{P}_{11} & \mathbf{P}_{12} & \mathbf{P}_{13} \\ \mathbf{P}_{21} & \mathbf{P}_{22} & \mathbf{P}_{23} \\ \mathbf{P}_{31} & \mathbf{P}_{32} & \mathbf{P}_{33} \end{pmatrix} \begin{pmatrix} B_{20}(w) \\ B_{21}(w) \\ B_{22}(w) \end{pmatrix}.\end{aligned}$$

This is repeated for every point in the image. If the image consists of straight lines, only the two endpoints of each line have to be transformed.

If the image is complex, or if a complicated deformation is needed, the grid can be made bigger and more control points added. The only difference is that higher-order Bernstein polynomials need to be used.

This technique can also be extended to three-dimensional images. The control points must be arranged in a three-dimensional grid and the process is similar. In each step, three parameters, u, v, and w, are determined and are used to transform a point $\mathbf{P} = (x, y, z)$ to a point $\mathbf{P}^* = (x^*, y^*, z^*)$.

To produce an animation sequence of F frames showing an image being deformed, we start with two sets of control points, an initial and a final. We then calculate $F - 2$ intermediate sets of control points and use the resulting F sets to compute F deformed images which are then displayed at the desired rate (between 18 and 24 frames per second) to animate the image.

Animation can explain whatever the mind of man can conceive.
—Walt Disney