21 Animation

To *animate* is, literally, to bring to life. Although people often think of animation as synonymous with motion, it covers all changes that have a visual effect. It thus includes the time-varying position (*motion dynamics),* shape, color, transparency, structure, and texture of an object (*update dynamics*), and changes in lighting, camera position, orientation, and focus, and even changes of rendering technique.

Animation is used widely in the entertainment industry, and is also being applied in education, in industrial applications such as control systems and heads-up displays and flight simulators for aircraft, and in scientific research. The scientific applications of computer graphics, and especially of animation, have come to be grouped under the heading *scientific visualization*. Visualization is more than the mere application of graphics to science and engineering, however; it can involve other disciplines, such as signal processing, computational geometry, and database theory. Often, the animations in scientific visualization are generated from *simulations* of scientific phenomena. The results of the similations may be large datasets representing 2D or 3D data (e.g., in the case of fluid-flow simulations); these data are converted into images that then constitute the animation. At the other extreme, the simulation may generate positions and locations of physical objects, which must then be rendered in some form to generate the animation. This happens, for example, in chemical simulations, where the positions and orientations of the various atoms in a reaction may be generated by simulation, but the animation may show a ball-and-stick view of each molecule, or may show overlapping smoothly shaded spheres representing each atom. In some cases, the simulation program will contain an embedded animation language, so that the simulation and animation processes are simultaneous.

If some aspect of an animation changes too quickly relative to the number of animated frames displayed per second, *temporal aliasing* occurs. Examples of this are wagon wheels that apparently turn backward and the jerky motion of objects that move through a large field of view in a short time. Videotape is shown at 30 frames per second (fps), and photographic film speed is typically 24 fps, and both of these provide adequate results for many applications. Of course, to take advantage of these rates, we must create a new image for each videotape or film frame. If, instead, the animator records each image on two videotape frames, the result will be an effective 15 fps, and the motion will appear jerkier.[1]

Some of the animation techniques described here have been partially or completely implemented in hardware. Architectures supporting basic animation in real time are essential for building flight simulators and other real-time control systems; some of these architectures were discussed in Chapter 18.

Traditional animation (i.e., noncomputer animation) is a discipline in itself, and we do not discuss all its aspects. Here, we concentrate on the basic concepts of computer-based animation, and also describe some state-of-the-art systems. We begin by discussing conventional animation and the ways in which computers have been used to assist in its creation. We then move on to animation produced principally by computer. Since much of this is 3D animation, many of the techniques from traditional 2D character animation no longer apply directly. Also, controlling the course of an animation is more difficult when the animator is not drawing the animation directly: it is often more difficult to describe *how* to do something than it is to do that action directly. Thus, after describing various animation languages, we examine several animation control techniques. We conclude by discussing a few general rules for animation, and problems peculiar to animation.

21.1 CONVENTIONAL AND COMPUTER-ASSISTED ANIMATION

21.1.1 Conventional Animation

A conventional animation is created in a fairly fixed sequence: The story for the animation is written (or perhaps merely conceived), then a *storyboard* is laid out. A storyboard is an animation in outline form—a high-level sequence of sketches showing the structure and ideas of the animation. Next, the soundtrack (if any) is recorded, a detailed layout is produced (with a drawing for every scene in the animation), and the soundtrack is read—that is, the instants at which significant sounds occur are recorded in order. The detailed layout and the soundtrack are then correlated.[2] Next, certain *key frames* of the animation are drawn—these are the frames in which the entities being animated are at extreme or characteristic positions, from which their intermediate positions can be inferred. The intermediate frames are then filled in (this is called *inbetweening*), and a trial film is made (a *pencil test*). The pencil-test frames are then transferred to *cels* (sheets of acetate

[1]This lets the animator generate only half as many frames, however. In some applications, the time savings may be worth the tradeoff in quality.

[2]The order described here is from conventional studio cartoon animation. In fine-arts animation, the soundtrack may be recorded last; in computer-assisted animation, the process may involve many iterations.

film), either by hand copying in ink or by photocopying directly onto the cels. In multiplane animation, multiple layers of cels are used, some for background that remains constant (except perhaps for a translation), and some for foreground characters that change over time. The cels are colored in or painted, and are assembled into the correct sequence; then, they are filmed. The people producing the animation have quite distinct roles: some design the sequence, others draw key frames, others are strictly inbetweeners, and others work only on painting the final cels. Because of the use of key frames and inbetweening, this type of animation is called *key-frame animation*. The name is also applied to computer-based systems that mimic this process.

The organizational process of an animation is described [CATM78a] by its storyboard; by a *route sheet*, which describes each scene and the people responsible for the various aspects of producing the scene; and by the *exposure sheet*, which is an immensely detailed description of the animation. The exposure sheet has one line of information for each frame of the animation, describing the dialogue, the order of all the figures in the frame, the choice of background, and the camera position within the frame. This level of organization detail is essential in producing a coherent animation. For further information on conventional animation, see [LAYB79; HALA68; HALA73].

The entire process of producing an animation is supposed to be sequential, but is often (especially when done with computers) iterative: the available sound effects may cause the storyboard to be modified slightly, the eventual look of the animation may require that some sequences be expanded, in turn requiring new sound-track segments, and so on.

21.1.2 Computer Assistance

Many stages of conventional animation seem ideally suited to computer assistance, especially inbetweening and coloring, which can be done using the seed-fill techniques described in Section 19.5.2. Before the computer can be used, however, the drawings must be digitized. This can be done by using optical scanning, by tracing the drawings with a data tablet, or by producing the original drawings with a drawing program in the first place. The drawings may need to be postprocessed (e.g., filtered) to clean up any glitches arising from the input process (especially optical scanning), and to smooth the contours somewhat. The composition stage, in which foreground and background figures are combined to generate the individual frames for the final animation, can be done with the image-composition techniques described in Section 17.6.

By placing several small low-resolution frames of an animation in a rectangular array, the equivalent of a pencil test can be generated using the pan-zoom feature available in some frame buffers. The frame buffer can take a particular portion of such an image (the portion consiting of one low-resolution frame), move it to the center of the screen (*panning*), and then enlarge it to fill the entire screen (*zooming*).[3] This process can be repeated on the

[3]The panning and zooming are actually effected by changing the values in frame-buffer registers. One set of registers determines which pixel in the frame-buffer memory corresponds to the upper-left corner of the screen, and another set of registers determines the pixel-replication factors—how many times each pixel is replicated in the horizontal and vertical direction. By adjusting the values in these registers, the user can display each of the frames in sequence, pixel-replicated to fill the entire screen.

several frames of the animation stored in the single image; if done fast enough, it gives the effect of continuity. Since each frame of the animation is reduced to a very small part of the total image (typically one twenty-fifth or one thirty-sixth), and is then expanded to fill the screen, this process effectively lowers the display device's resolution. Nonetheless, these low-resolution sequences can be helpful in giving a sense of an animation, thus acting as a kind of pencil test.

21.1.3 Interpolation

The process of inbetweening is amenable to computer-based methods as well, but many problems arise. Although a human inbetweener can perceive the circumstances of the object being interpolated (is it a falling ball or a rolling ball?), a computer-based system is typically given only the starting and ending positions. The easiest interpolation in such a situation is *linear interpolation*: Given the values, v_s and v_e, of some attribute (position, color, size) in the starting and ending frames, the value v_t at intermediate frames is $v_t = (1 - t)v_s + t\,v_e$; as the value t ranges from 0 to 1, the value of v_t varies smoothly from v_s to v_e. Linear interpolation (sometimes called *lerping*—Linear intERPolation), although adequate in some circumstances, has many limitations. For instance, if lerping is used to compute intermediate positions of a ball that is thrown in the air using the sequence of three key frames shown in Fig. 21.1 (a), the resulting track of the ball shown in Fig. 21.1(b) is entirely unrealistic. Particularly problematic is the sharp corner at the zenith of the trajectory: Although lerping generates continuous motion, it does not generate continuous derivatives, so there may be abrupt changes in velocity when lerping is used to interpolate positions. Even if the positions of the ball in the three key frames all lie in a line, if the distance between the second and third is greater than that between the first and second, then lerping causes a discontinuity in speed at the second key frame. Thus, lerping generates derivative discontinuities in time as well as in space (the time discontinuities are measured by the parametric continuity described in Chapter 11).

Because of these drawbacks of lerping, splines have been used instead to smooth out interpolation between key frames. Splines can be used to vary any parameter smoothly as a function of time. The splines need not be polynomials.[4] For example, to get smooth initiation and termination of changes (called *slow-in* and *slow-out*) and fairly constant rates of change in between, we could use a function such as $f(t)$ in Fig. 21.2. A value can be interpolated by setting $v_t = (1 - f(t))v_s + f(t)v_e$. Since the slope of f is zero at both $t = 0$ and $t = 1$, the change in v begins and ends smoothly. Since the slope of f is constant in the middle of its range, the rate of change of v is constant in the middle time period.

Splines can make individual points (or individual objects) move smoothly in space and time, but this by no means solves the inbetweening problem. Inbetweening also involves interpolating the shapes of objects in the intermediate frames. Of course, we could describe a spline path for the motion of each point of the animation in each frame, but splines give the smoothest motion when they have few control points, in both space and time. Thus, it is preferable to specify the positions of only a few points at only a few times, and somehow to

[4]This is an extension of the notion of spline introduced in Chapter 11, where a spline was defined to be a piecewise cubic curve. Here we use the term in the more general sense of any curve used to approximate a set of control points.

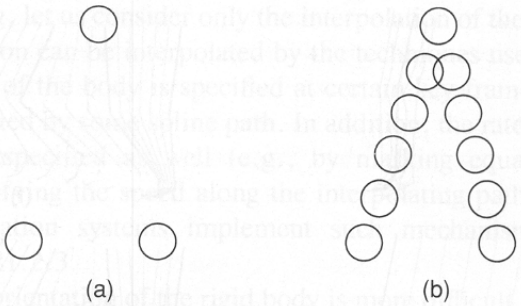


Fig. 21.1 Linear interpolation of the motion of a ball generates unrealistic results. (a) Three key-frame positions for the ball. (b) The resulting interpolated positions.

extend the spline interpolation over intermediate points and times. At least one special case deserves mention: A figure drawn as a polyline can be interpolated between key frames by interpolating each vertex of the polyline from its starting to ending position. As long as the key frames do not differ too much, this is adequate (for examples where this fails, see Exercise 21.1).

Several approaches to this have been developed. Burtnyk and Wein [BURT76] made a *skeleton* for a motion by choosing a polygonal arc describing the basic shape of a 2D figure or portion of a figure, and a neighborhood of this arc (see Fig. 21.3). The figure is represented in a coordinate system based on this skeleton. They then specify the thickness of the arc and positions of the vertices at subsequent key frames and redraw the figure in a new coordinate system based on the deformed arc. Inbetweening is done by interpolating the characteristics of the skeleton between the key frames. (A similar technique can be developed for 3D, using the trivariate Bernstein polynomial deformations or the heirarchical B-splines described in Chapter 20.)

Reeves [REEV81] designed a method in which the intermediate trajectories of particular points on the figures in successive key frames are determined by hand-drawn paths (marked by the animator to indicate constant time intervals). A region bounded by two such *moving-points paths* and an arc of the figure in each of the two key frames

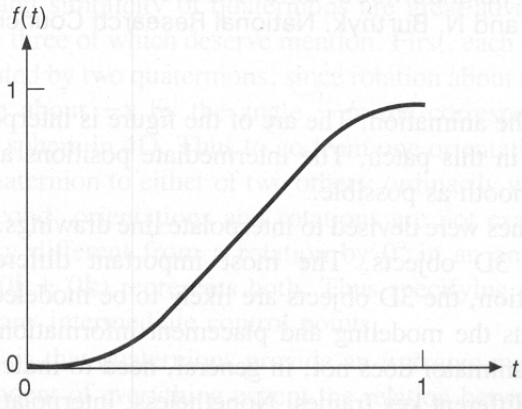


Fig. 21.2 The graph of a function $f(t)$ with zero derivative at its endpoints and constant derivative in its middle section.

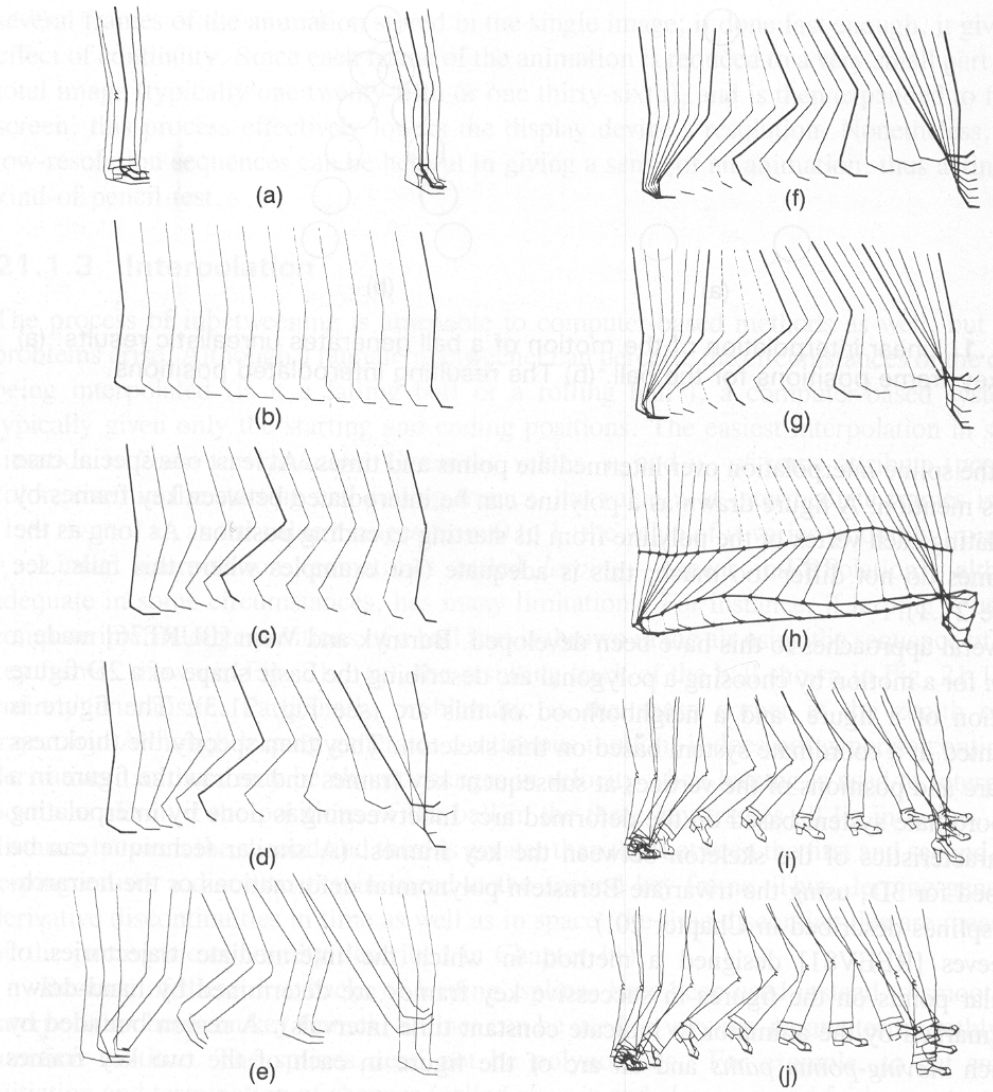


Fig. 21.3 Use of a neighborhood of a skeleton to define interpolated shapes. (Courtesy of M. Wein and N. Burtnyk, National Research Council of Canada.)

determines a *patch* of the animation. The arc of the figure is interpolated by computing its intermediate positions in this patch. The intermediate positions are determined so as to make the motion as smooth as possible.

Both these techniques were devised to interpolate line drawings, but the same problems arise in interpolating 3D objects. The most important difference is that, in most computer-based animation, the 3D objects are likely to be modeled explicitly, rather than drawn in outlines. Thus the modeling and placement information is available for use in interpolation, and the animator does not, in general, need to indicate which points on the objects correspond in different key frames. Nonetheless, interpolation between key frames is a difficult problem.

For the time being, let us consider only the interpolation of the position and orientation of a rigid body. Position can be interpolated by the techniques used in 2D animation: The position of the center of the body is specified at certain key frames, and the intermediate positions are interpolated by some spline path. In addition, the rate at which the spline path is traversed may be specified as well (e.g., by marking equal-time intervals on the trajectory, or by specifying the speed along the interpolating path as a function of time). Many different animation systems implement such mechanisms; some of these are discussed in Section 21.2.3.

Interpolating the orientation of the rigid body is more difficult. In fact, even specifying the orientation is not easy. If we specify orientations by amounts of rotation about the three principal axes (called *Euler angles*), then the order of specification is important. For example, if a book with its spine facing left is rotated 90° about the x axis and then $-90°$ about the y axis, its spine will face you, whereas if the rotations are done in the opposite order, its spine will face down. A subtle consequence of this is that interpolating Euler angles leads to unnatural interpolations of rotations: A rotation of 90° about the z axis and then 90° about the y axis has the effect of a 120° rotation about the axis (1, 1, 1). But rotating 30° about the z axis and 30° about the y axis does not give a rotation of 40° about the axis (1, 1, 1)—it gives approximately a 42° rotation about the axis (1, 0.3, 1)!

The set of all possible rotations fits naturally into a coherent algebraic structure, the *quaternions* [HAMI53]. The rotations are exactly the *unit quaternions*, which are symbols of the form $a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$, where a, b, c, and d are real numbers satisfying $a^2 + b^2 + c^2 + d^2 = 1$; quaternions are multiplied using the distributive law and the rules $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1$, $\mathbf{ij} = \mathbf{k} = -\mathbf{ji}$, $\mathbf{jk} = \mathbf{i} = -\mathbf{kj}$, and $\mathbf{ki} = \mathbf{j} = -\mathbf{ik}$. Rotation by angle ϕ about the unit vector $[b \quad c \quad d]^t$ corresponds to the quaternion $\cos \phi/2 + b \sin \phi/2\ \mathbf{i} + c \sin \phi/2\ \mathbf{j} + d \sin \phi/2\ \mathbf{k}$. Under this correspondence, performing successive rotations corresponds to multiplying quaternions. The inverse correspondence is described in Exercise 21.7.

Since unit quaternions satisfy the condition $a^2 + b^2 + c^2 + d^2 = 1$, they can be thought of as points on the unit sphere in 4D. To interpolate between two quaternions, we simply follow the shortest path between them on this sphere (a *great arc*). This spherical linear interpolation (called *slerp*) is a natural generalization of linear interpolation. Shoemake [SHOE85] proposed the use of quaternions for interpolation in graphics, and developed generalizations of spline interpolants for quaternions.

The compactness and simplicity of quaternions are great advantages, but difficulties arise with them as well, three of which deserve mention. First, each orientation of an object can actually be represented by two quaternions, since rotation about the axis $\mathbf{v}$ by an angle ϕ is the same as rotation about $-\mathbf{v}$ by the angle $-\phi$; the corresponding quaternions are antipodal points on the sphere in 4D. Thus to go from one orientation to another, we may interpolate from one quaternion to either of two others; ordinarily we choose the shorter of the two great arcs. Second, orientations and rotations are not exactly the same thing: a rotation by 360° is very different from a rotation by 0° in an animation, but the same quaternion $(1 + 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k})$ represents both. Thus specifying multiple rotations with quaternions requires many intermediate control points.

The third difficulty is that quaternions provide an *isotropic* method for rotation—the interpolation is independent of everything except the relation between the initial and final rotations. This is ideal for interpolating positions of tumbling bodies, but not for

interpolating the orientation of a camera in a scene: Humans strongly prefer cameras to be held upright (i.e., the horizontal axis of the film plane should lie in the (x, z) plane), and are profoundly disturbed by tilted cameras. Quaternions have no such preferences, and therefore should not be used for camera interpolation. The lack of an adequate method for interpolating complex camera motion has led to many computer animations having static cameras or very limited camera motion.

21.1.4 Simple Animation Effects

In this section, we describe a few simple computer-animation tricks that can all be done in real time. These were some of the first techniques developed, and they are therefore hardware-oriented.

In Section 4.4.1, we discussed the use of color look-up tables (luts) in a frame buffer and the process of double-buffering; and in Section 17.6, we described image compositing by color-table manipulations. Recall that lut animation is generated by manipulating the lut. The simplest method is to cycle the colors in the lut (to replace color i with color $i - 1 \bmod n$, where n is the number of colors in the table), thus changing the colors of the various pieces of the image. Figure 21.4 shows a source, a sink, and a pipe going between them. Each piece of the figure is labeled with its lut index. The lut is shown at the right. By cycling colors 1 through 5, we can generate an animation of material flowing through the pipe.

Using this lut animation is a great deal faster than sending an entire new pixmap to the frame buffer for each frame. Assuming 8 color bits per pixel in a 640 by 512 frame buffer, a single image contains 320 KB of information. Transferring a new image to the frame buffer every thirtieth of a second requires a bandwidth of over 9 MB per second, which is well beyond the capacity of most small computers. On the other hand, new values for the lut can be sent very rapidly, since luts are typically on the order of a few hundred to a few thousand bytes.

Lut animation tends to look jerky, since the colors change suddenly. This effect can be softened somewhat by taking a color to be made visible and changing its lut entry gradually over several frames from the background color to its new color, and then similarly fading it out as the next lut entry is being faded in. Details of this and other tricks are given by Shoup [SHOU79].

Lut animation can be combined with the pan-zoom movie technique described

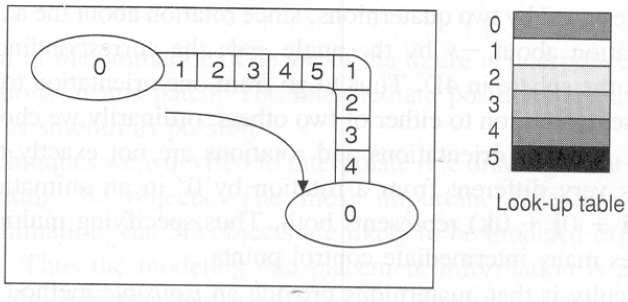


Fig. 21.4 The look-up–table entries can be cycled to give the impression of flow through the pipe.

previously to make longer pan-zoom movies with less color resolution. To make a very long two-color pan-zoom movie on a frame buffer with eight planes of memory, for example, we can generate 200 frames of an animation, each at one-twenty-fifth of full screen resolution. Frames 1 through 25 are arranged in a single image to be used for a pan-zoom movie. The same is done with frames 26 through 50, and so on up to frames 176 through 200, giving a total of eight bitmaps. These are combined, on a pixel-by-pixel basis, into a single 8-bit-deep image, which is then downloaded to the frame buffer. We make all the lut entries black except entry 00000001, which we make white. We then run a 25-frame pan-zoom movie and see the first 25 images of the animation. Then, we set entry 00000001 to black and entry 00000010 to white. Running another 25-frame pan-zoom movie shows us the next 25 images. Continuing in this fashion, we see the full 200-frame animation. By allocating several planes to each image, we can generate shorter pan-zoom movies with additional bits of color.

Finally, let's look at the hardware-based animation technique called *sprites*. A sprite is a small rectangular region of memory that is mixed with the rest of the frame-buffer memory at the video level. The location of the sprite at any time is specified in registers in the frame buffer, so altering the values in these registers causes the sprite to move. The sprites may hide the frame-buffer values at each pixel, or may be blended with them. We can use sprites to implement cursors in frame buffers, and also to generate animations by moving the sprite (or sprites) around on top of a background image. Some frame buffers have been designed to allow several sprites with different priorities, so that some sprites can be "on top of" others.

One of the most popular uses of sprites is in video games, where the animation in the game may consist almost entirely of sprites moving over a fixed background. Since the location and size of each sprite are stored in registers, it is easy to check for collisions between sprites, which further enhances the use of sprites in this application.

21.2 ANIMATION LANGUAGES

There are many different languages for describing animation, and new ones are constantly being developed. They fall into three categories: linear-list notations, general-purpose languages with embedded animation directives, and graphical languages. Here, we briefly describe each type of language and give examples. Many animation languages are mingled with modeling languages, so the descriptions of the objects in an animation and of the animations of these objects are done at the same time.

21.2.1 Linear-List Notations

In linear-list notations for animation such as the one presented in [CATM72], each event in the animation is described by a starting and ending frame number and an action that is to take place (the *event*). The actions typically take parameters, so a statement such as

```
42, 53,B ROTATE "PALM", 1, 30
```

means "between frames 42 and 53, rotate the object called PALM about axis 1 by 30 degrees, determining the amount of rotation at each frame from table B." Thus, the actions

are given interpolation methods to use (in this case a table of values) and objects to act on as well. Since the statements describe individual actions and have frame values associated with them, their order is, for the most part, irrelevant. If two actions are applied to the same object at the same time, however, the order may matter: rotating 90° in x and then 90° in y is different from rotating 90° in y and then 90° in x.

Many other linear-list notations have been developed, and many notations are supersets of the basic linear-list idea. Scefo (SCEne FOrmat) [STRA88], for example, has some aspects of linear-list notation, but also includes a notion of groups and object hierarchy and supports abstractions of changes (called *actions*) and some higher-level programming-language constructs (variables, flow of control, and expression evaluation) distinguishing it from a simple linear list. Scefo also supports a model of animation that differs from many animation languages in that it is renderer-independent. A Scefo script describes only an animation; the individual objects in the script can be rendered with any renderer at all, and new renderers can easily be added to the animation system of which Scefo is the core.

21.2.2 General-Purpose Languages

Another way to describe animations is to embed animation capability within a general-purpose programming language [REYN82; SYMB85; MAGN85]. The values of variables in the language can be used as parameters to whatever routines actually generate animations, so the high-level language can actually be used to generate simulations that then generate animations as a side effect. Such languages have great potential (e.g., they can certainly do everything that linear-list notations do), but most of them require considerable programming expertise on the part of the user.

Such systems can use the constructs of the surrounding language to create concise routines that have complex effects. Of course, these can sometimes be cryptic. ASAS [REYN82] is an example of such a language. It is built on top of LISP, and its primitive entities include vectors, colors, polygons, solids (collections of polygons), groups (collections of objects), points of view, subworlds, and lights. A point of view consists of a location and an orientation for an object or a camera (hence, it corresponds to the cumulative transformation matrix of an object in PHIGS). Subworlds are entities associated with a point of view; the point of view can be used to manipulate the entities in the subworld in relation to the rest of the objects in the animation.

ASAS also includes a wide range of geometric transformations that operate on objects; they take an object as an argument and return a value that is a transformed copy of the object. These transformations include *up*, *down*, *left*, *right*, *zoom-in*, *zoom-out*, *forward*, and *backward*. Here is an ASAS program fragment, describing an animated sequence in which an object called my-cube is spun while the camera pans. Anything following a semicolon is a comment. This fragment is evaluated at each frame in order to generate the entire sequence.

```
(grasp my-cube)          ; The cube becomes the current object
(cw 0.05)                ; Spin it clockwise by a small amount
(grasp camera)           ; Make the camera the current object
(right panning-speed)    ; Move it to the right
```

The advantage of ASAS over linear-list notations is the ability to generate procedural objects and animations within the language. This ability comes at the cost of increased skill required of the animator, who must be an able programmer. Scefo lies in the middle ground, providing some flow-of-control constructs, and the ability to bind dynamically with routines written in a high-level language, while being simple enough for nonprogrammers to learn and use readily.

21.2.3 Graphical Languages

One problem with the textual languages we have described is that it is difficult for an animator to see what will take place in an animation just by looking at the script. Of course, this should not be surprising, since the script is a program, and to the extent that the program's language allows high-level constructs, it encodes complex events in compact form. If a real-time previewer for the animation language is available, this is not a problem; unfortunately the production of real-time animations is still beyond the power of most hardware.

Graphical animation languages describe animation in a more visual way. These languages are used for expressing, editing, and comprehending the simultaneous changes taking place in an animation. The principal notion in such languages is substitution of a visual paradigm for a textual one: rather than explicitly writing out descriptions of actions, the animator provides a picture of the action. Some of the earliest work in this area was done by Baecker [BAEC69], who introduced the notion of P-curves in the GENESYS animation system. A P-curve is a parametric representation of the motion (or any other attribute) of an object or assembly of objects within a scene. The animator describes an object path of motion by graphically specifying its coordinates as a function of time (just as splines do, where functions $X(t)$, $Y(t)$, and $Z(t)$ specify the 3D location of a point on a curve as a function of an independent variable). Figure 21.5(a) shows a motion path in the plane; Fig. 21.5(b) of that figure shows the path's x and y components as functions of time. Notice that the curves in part (b)) uniquely determine the curve in part (a), but the opposite is not true: One can traverse the path in part (a) at different speeds. By marking the path in part (a) to indicate constant time steps, we can convey the time dependence of the path, as shown in part (c), which is what Baecker calls a P-curve. Note that part (c) can be constructed as shown in part (d) by graphing the x and y components as functions of t, on coordinate systems that are rotated 90° from each other, and then drawing lines to connect corresponding time points. Thus, editing the components of a parametric curve induces changes in the P-curve, and editing the placement of the hash marks on the P-curve induces changes in the components.

The diagrammatic animation language DIAL [FEIN82b] retains some of the features of linear-list notations, but displays the sequence of events in an animation as a series of parallel rows of marks: A vertical bar indicates the initiation of an action, and dashes indicate the time during which the action is to take place. The actions are defined in a DIAL script (by statements of the form ''% t1 translate ''block'' 1.0 7.0 15.3,'' which defines action t1 as the translation of an object called ''block'' by the vector (1.0, 7.0, 15.3)), and then the applications of the actions are defined subsequently. The particular instructions that DIAL executes are performed by a user-specified back end given at run time. DIAL

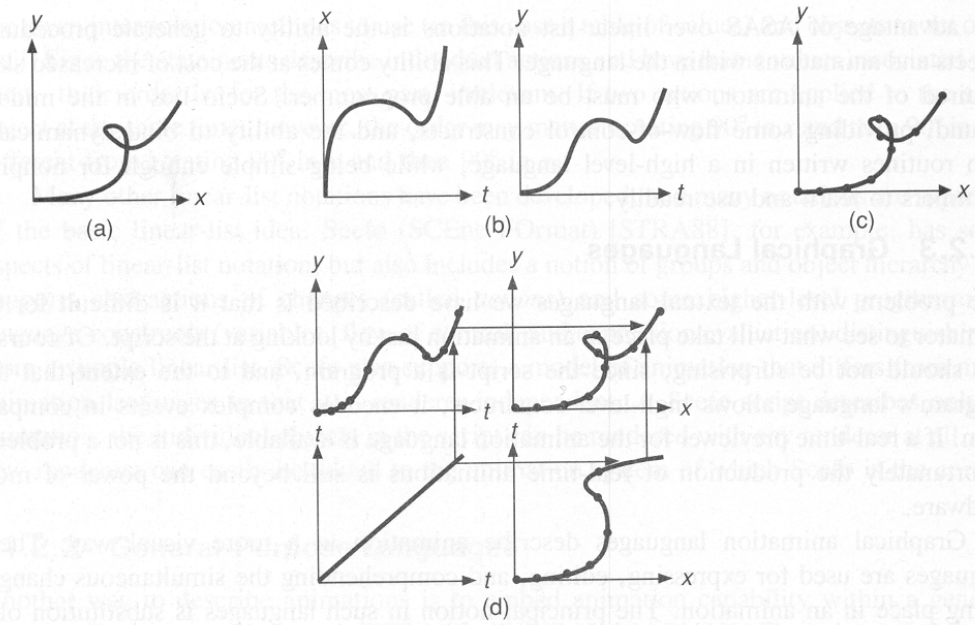


Fig. 21.5 (a) A parametric path in the plane. (b) Its *x* and *y* components as functions of time. (c) The original curve marked to indicate equal time steps. (d) The construction of a P-curve from component functions.

itself knows nothing about animation; it merely provides a description of the sequence in which instructions are to be performed. The following is a typical DIAL script (lines beginning with a blank are comments):

```
     Read in an object from a file, and assign it the name "block"
! getsurf "block.d" 5 5 "block"
     Define a window on the xy plane
! window -20 20 -20 20
     Define two actions, (1) a translation,
% t1 translate "block" 10 0 0
     and (2) a rotation in the xy plane by 360 degrees
% r1 rotate "block" 0 1 360

     Now describe a translation, spin, and a further translation:
t1   |---------      |--------

r1             |--------

r1   |------------------------
```

The line labeled ''t1'' indicates that action t1 is to take place from frames 1 to 10 (and hence is to translate by one unit per frame, since linear interpolation is the default), and then again from frames 17 to 25. At frame 11, the block stops translating and rotates 40° per frame for six frames (the first line labeled ''r1'' indicates this), and then rotates and translates for the next three, and then just translates.

For longer animations, each tick mark can indicate multiple frames, so that animations of several seconds' duration can be specified easily without indicating every frame. Long animations can be described by starting a new sequence of tick marks on a new line, following a completely blank line. (The second line labeled ''r1'' above is an example: it indicates that a 360° turn should take place between frames 26 and 50.) The format is much like that of a conductor's score of a symphony: Each action corresponds to an instrument in the orchestra, and each group of lines corresponds to a staff in the score. DIAL and many linear-list notations have the advantage of being specified entirely in ASCII text, making them portable to different machines (although a back end for the language must be written for each new machine).

The S-Dynamics system [SYMB85] takes this visual paradigm one step further and combines it with parametric descriptions of actions similar to P-curves. To do this, S-Dynamics uses the full power of a bitmapped workstation. Figure 21.6 shows an S-Dynamics window. Just as in DIAL, time runs horizontally across the window. The period during which actions are to take effect is indicated by the width of the region representing the action. Each action (or *sequence* in S-Dynamics terminology) can be shown as a box that indicates the time extent of the action, or the box can be ''opened''—that is, made to show more internal detail. A sequence may be a composite of

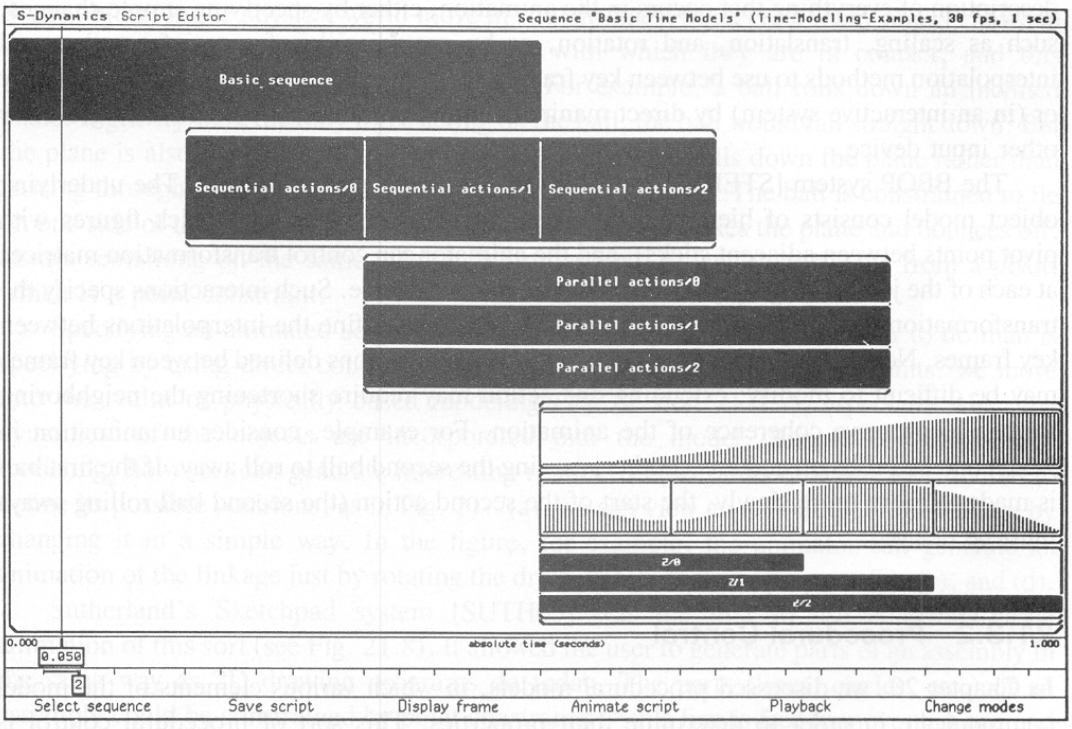


Fig. 21.6 An S-Dynamics window. (Courtesy of Symbolics Graphics Division. The software and the SIGGRAPH paper in which this image first appeared were both written by Craig Reynolds.)

several serial or parallel actions, each of which can be opened to show even more detail, including a graph indicating the time dependence of a parameter of the action.

21.3 METHODS OF CONTROLLING ANIMATION

Controlling an animation is somewhat independent of the language used for describing it—most control mechanisms can be adapted for use with various types of languages. Animation-control mechanisms range from full explicit control, in which the animator explicitly describes the position and attributes of every object in a scene by means of translations, rotations, and other position- and attribute-changing operators, to the highly automated control provided by knowledge-based systems, which take high-level descriptions of an animation (''make the character walk out of the room'') and generate the explicit controls that effect the changes necessary to produce the animation. In this section, we examine some of these techniques, giving examples and evaluating the advantages and disadvantages of each.

21.3.1 Full Explicit Control

Explicit control is the simplest sort of animation control. Here, the animator provides a description of everything that occurs in the animation, either by specifying simple changes, such as scaling, translation, and rotation, or by providing key-frame information and interpolation methods to use between key frames. This interpolation may be given explicitly or (in an interactive system) by direct manipulation with a mouse, joystick, data glove, or other input device.

The BBOP system [STER83] provides this interactive sort of control. The underlying object model consists of hierarchical jointed polyhedral objects (i.e., stick figures with pivot points between adjacent sticks), and the animator can control transformation matrices at each of the joints using a joystick or other interactive device. Such interactions specify the transformations at key frames, and interactive programs define the interpolations between key frames. Notice that, in such a system, a sequence of actions defined between key frames may be difficult to modify; extending one action may require shortening the neighboring actions to preserve coherence of the animation. For example, consider an animation in which one ball rolls up and hits another, causing the second ball to roll away. If the first ball is made to move more slowly, the start of the second action (the second ball rolling away) must be delayed.

21.3.2 Procedural Control

In Chapter 20, we discussed procedural models, in which various elements of the model communicate in order to determine their properties. This sort of procedural control is ideally suited to the control of animation. Reeves and Blau [REEV85] modeled both grass and wind in this way, using a particle system modeling technique (see Section 20.5). The wind particles evolved over time in the production of the animation, and the positions of the

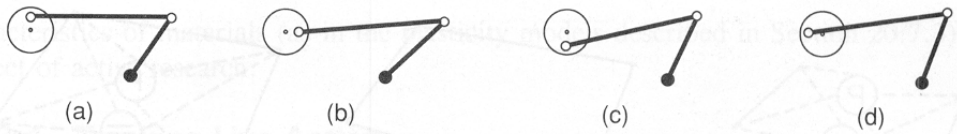


Fig. 21.7 The linkage in (a) is moved by rotating the drive wheel. The constraints generate the motions shown in (b), (c), and (d).

grass blades were then determined by the proximity of wind particles. Thus, the particle system describing the grass was affected by aspects of other objects in the scene. This sort of procedural interaction among objects can be used to generate motions that would be difficult to specify through explicit control. Unfortunately, it also requires that the animator be a programmer.

Procedural control is a significant aspect of several other control mechanisms we discuss. In particular, in physically based systems, the position of one object may influence the motion of another (e.g., balls cannot pass through walls); in actor-based systems, the individual actors may pass their positions to other actors in order to affect the other actors' behaviors.

21.3.3 Constraint-Based Systems

Some objects in the physical world move in straight lines, but a great many objects move in a manner determined by the other objects with which they are in contact, and this compound motion may not be linear at all. For example, a ball rolls down an inclined plane. If gravity were the only force acting on the ball, the ball would fall straight down. But the plane is also pushing up and sideways, and so the ball rolls down the plane rather than passing through it. We can model such motion by constraints. The ball is constrained to lie on one side of the plane. If it is dropped from a height, it strikes the plane and bounces off, always remaining on the same side. In a similar way, a pendulum swings from a pivot, which is a point constraint.

Specifying an animated sequence using constraints is often much easier to do than is specifying by using direct control. When physical forces define the constraints, we move into the realm of physically based modeling (see Section 21.3.7), especially when the dynamics[5] of the objects are incorporated into the model. Simple constraint-based modeling, however, can generate interesting results. If constraints on a linkage are used to define its possible positions, as in Fig. 21.7(a), we can view an animation of the linkage by changing it in a simple way. In the figure, for example, the animator can generate an animation of the linkage just by rotating the drive wheel, as shown in parts (b), (c), and (d).

Sutherland's Sketchpad system [SUTH63] was the first to use constraint-based animation of this sort (see Fig. 21.8). It allowed the user to generate parts of an assembly in the same way as 2D drawing programs do today. The parts (lines, circles, etc.) of an assembly could be constrained by point constraints ("this line is free to move, but one end

[5]Here we use *dynamics* in the sense of physics, to mean the change in position and motion over time, not merely to mean "change," as in earlier chapters.

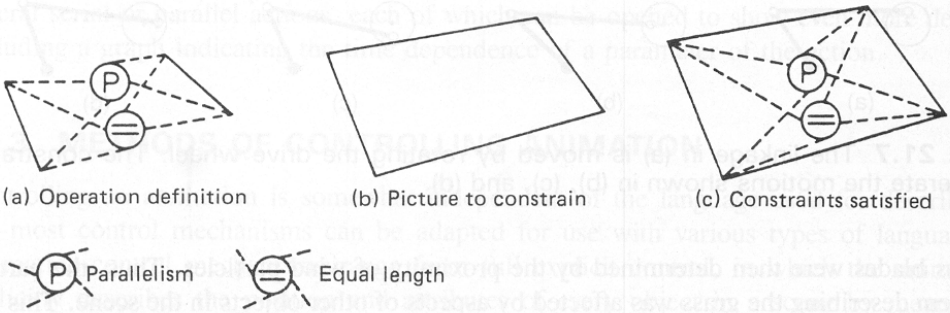


Fig. 21.8 Constraint definition and satisfaction in Sketchpad. (Adapted from [SUTH63].)

is held fixed at this point''), linkage constraints (''these lines must always remain joined end to end''), or angular constraints (''these lines must always be parallel'' or ''these lines must meet at a 60° angle''). This allowed the user to draw four lines in a quadrilateral, put linkage constraints on the corners, to put a point constraint at one corner, and to put angular constraints on opposite sides to make them parallel. This generated a parallelogram with one corner held fixed. Constraints were satisfied by a *relaxation technique* in which the assembly was moved so that the constraints came closer to being satisfied. Thus, the user could watch an assembly move so as to satisfy constraints gradually.[6] Of course, it is possible to overconstrain a system, by requiring, for example, that a line have a length of one unit, but that its ends be joined to two points that are three units apart. The constraints in Sketchpad are described by giving an error function—a function whose value is 0 when a constraint is satisfied, and is positive otherwise. Relaxation attempts to make the sum of these functions 0; when it fails, many constraints may be unsatisfied. Similarly, a system may be underconstrained, and have many solutions that satisfy all the constraints. In this case, the relaxation technique finds one solution that is close to the initial configuration.

Borning's similar ThingLab [BORN79] was really a metasystem: It provided a mechanism for defining systems like Sketchpad, but a user could define a system for modeling electrical circuits in the same framework. This system design was later improved to include a graphical interface [BORN86b]. A system, once designed, provided a world in which a user could build experiments. In a world meant to model geometry, for instance, the user could instantiate lines, point constraints, midpoint constraints, and so on, and then could move the assembly under those constraints. Figure 21.9 shows an example; the user has instantiated four MidpointSegments (segments with midpoints), has constrained their ends to be joined, and has also drawn four lines between adjacent midpoints. The user can vary the outer quadrilateral and observe that the inner quadrilateral always remains a parallelogram. For related work, see [BIER86a].

The extension of constraint-based animation systems to constraint systems supporting hierarchy, and to constraints modeled by the dynamics of physical bodies and the structural

[6]The animations therefore served two purposes: they generated assemblies satsifying the constraints, and they gave a *visualization* of the relaxation technique.

characteristics of materials (as in the plasticity models described in Section 20.7.3), is a subject of active research.

21.3.4 Tracking Live Action

Trajectories of objects in the course of an animation can also be generated by tracking of live action. There are a number of methods for doing tracking. Traditional animation has used *rotoscoping*: A film is made in which people (or animals) act out the parts of the characters in the animation, then animators draw over the film, enhancing the backgrounds and replacing the human actors with their animation equivalents. This technique provides exceptionally realistic motion. Alternatively, key points on an object may be digitized from a series of filmed frames, and then intermediate points may be interpolated to generate similar motion.

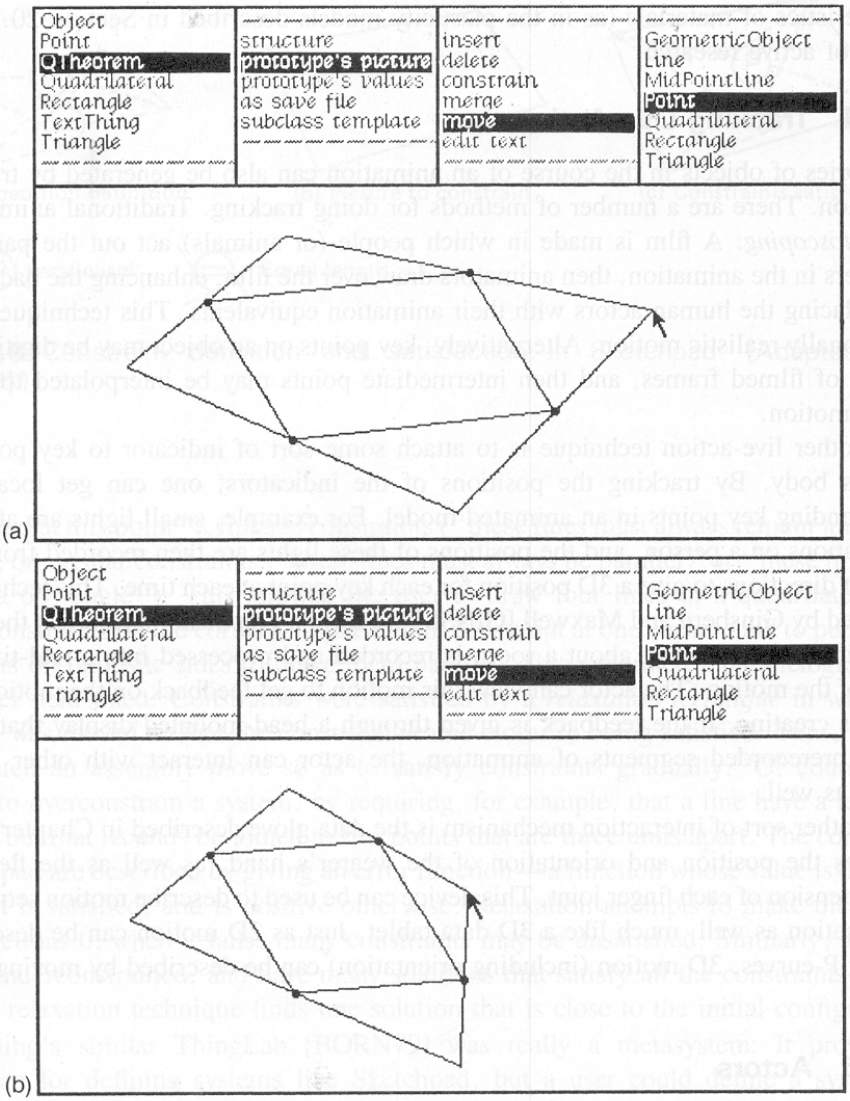
Another live-action technique is to attach some sort of indicator to key points on a person's body. By tracking the positions of the indicators, one can get locations for corresponding key points in an animated model. For example, small lights are attached at key locations on a person, and the positions of these lights are then recorded from several different directions to give a 3D position for each key point at each time. This technique has been used by Ginsberg and Maxwell [GINS83] to form a graphical marionette; the position of a human actor moving about a room is recorded and processed into a real-time video image of the motion. The actor can view this motion to get feedback on the motion that he or she is creating. If the feedback is given through a head-mounted display that can also display prerecorded segments of animation, the actor can interact with other graphical entities as well.

Another sort of interaction mechanism is the data glove described in Chapter 8, which measures the position and orientation of the wearer's hand, as well as the flexion and hyperextension of each finger joint. This device can be used to describe motion sequences in an animation as well, much like a 3D data tablet. Just as 2D motion can be described by drawing P-curves, 3D motion (including orientation) can be described by moving the data glove.

21.3.5 Actors

The use of *actors* is a high-level form of procedural control. An actor in an animation is a small program invoked once per frame to determine the characteristics of some object in the animation. (Thus, an actor corresponds to an ''object'' in the sense of object-oriented programming, as well as in the sense of animation.) An actor, in the course of its once-per-frame execution, may send messages to other actors to control their behaviors. Thus we could construct a train by letting the engine actor respond to some predetermined set of rules (move along the track at a fixed speed), while also sending the second car in the train the message ''place yourself on the track, with your forward end at the back end of the engine.'' Each car would pass a similar message to the next car, and the cars would all follow the engine.

Such actors were originally derived from a similar notion in Smalltalk [GOLD76] and other languages, and were the center of the ASAS animation system described in Section 21.2.2. The concept has been developed further to include actors with wide ranges of ''behaviors'' that they can execute depending on their circumstances.



21.3.6 Kinematics and Dynamics

Kinematics refers to the positions and velocities of points. A kinematic description of a scene, for example, might say, ''The cube is at the origin at time $t = 0$. It moves with a constant acceleration in the direction (1, 1, 5) thereafter.'' By contrast, *dynamics* takes into account the physical laws that govern kinematics (Newton's laws of motion for large bodies, the Euler–Lagrange equations for fluids, etc.). A particle moves with an acceleration proportional to the forces acting on it, and the proportionality constant is the mass of the particle. Thus, a dynamic description of a scene might be, ''At time $t = 0$ seconds the cube is at position (0 meters, 100 meters, 0 meters). The cube has a mass of 100 grams. The

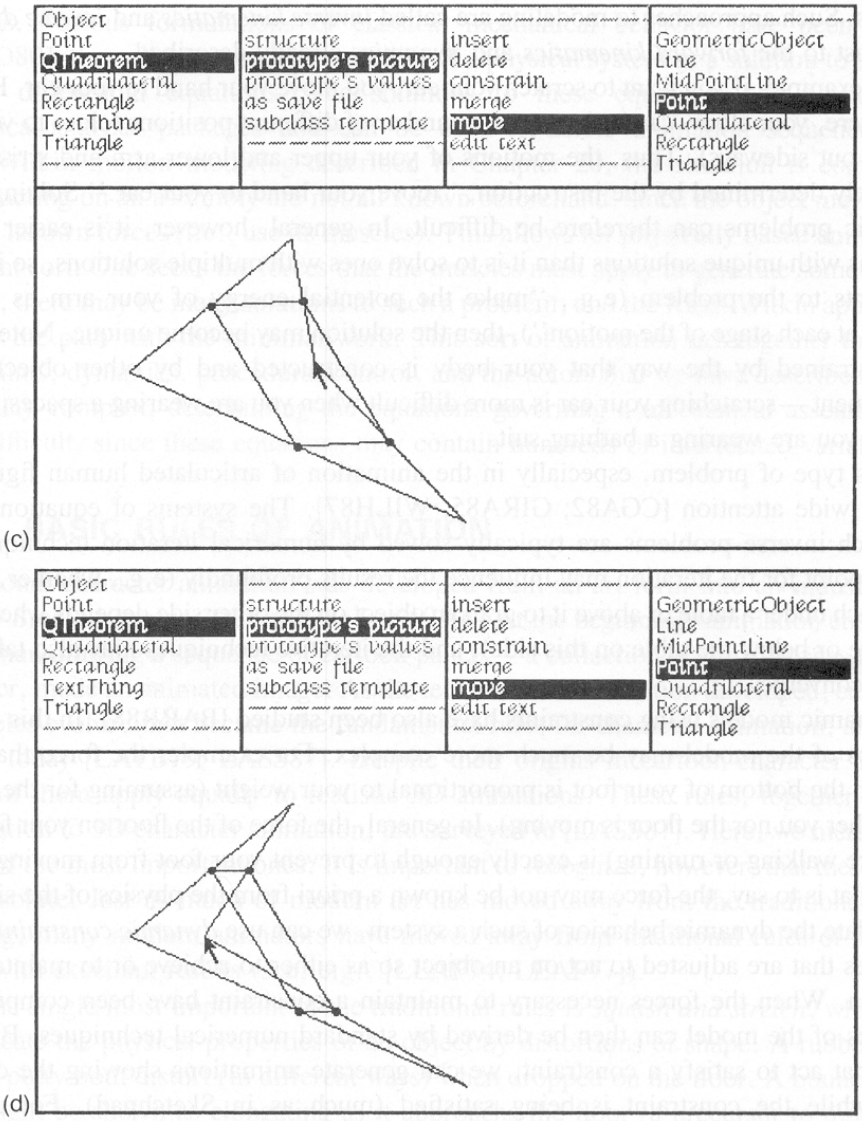


Fig. 21.9 A ThingLab display.(Courtesy of Alan Borning, Xerox PARC and University of Washington.)

force of gravity acts on the cube.'' Naturally, the result of a dynamic simulation[7] of such a model is that the cube falls.

Both kinematics and dynamics can be inverted; that is, we can ask the question, ''What must the (constant) velocity of the cube be for it to reach position (12, 12, 42) in 5 seconds?'' or, ''What force must we apply to the cube to make it get to (12, 12, 42) in 5 seconds?'' For simple systems, these sorts of questions may have unique answers; for more complicated ones, however, especially hierarchical models, there may be large families of

[7]This simulation could be based on either an explicit analytical solution of the equations of motion or a numerical solution provided by a package for solving differential equations.

solutions. Such approaches to modeling are called *inverse kinematics* and *inverse dynamics*, in contrast to the *forward kinematics* and *dynamics* already described.

For example, if you want to scratch your ear, you move your hand to your ear. But when it gets there, your elbow can be in any of a number of different positions (close to your body or stuck out sideways). Thus, the motions of your upper and lower arm and wrist are not completely determined by the instruction, "move your hand to your ear." Solving inverse kinematic problems can therefore be difficult. In general, however, it is easier to solve equations with unique solutions than it is to solve ones with multiple solutions, so if we add constraints to the problem (e.g., "make the potential energy of your arm as small as possible at each stage of the motion"), then the solution may become unique. Note that you are constrained by the way that your body is constructed and by other objects in the environment—scratching your ear is more difficult when you are wearing a spacesuit than it is when you are wearing a bathing suit.

This type of problem, especially in the animation of articulated human figures, has received wide attention [CGA82; GIRA85; WILH87]. The systems of equations arising from such inverse problems are typically solved by numerical iteration techniques. The starting point for the iteration may influence the results profoundly (e.g., whether a robot's arms reach under a table or above it to grab an object on the other side depends whether they are above or below the table on this side), and the iterative techniques may also take a long time to converge.

Dynamic models using constraints have also been studied [BARR88]. In this case, the dynamics of the model may be much more complex. For example, the force that a floor exerts on the bottom of your foot is proportional to your weight (assuming for the moment that neither you nor the floor is moving). In general, the force of the floor on your foot (even if you are walking or running) is exactly enough to prevent your foot from moving into the floor. That is to say, the force may not be known a priori from the physics of the situation.[8] To simulate the dynamic behavior of such a system, we can use *dynamic constraints*, which are forces that are adjusted to act on an object so as either to achieve or to maintain some condition. When the forces necessary to maintain a constraint have been computed, the dynamics of the model can then be derived by standard numerical techniques. By adding forces that act to satisfy a constraint, we can generate animations showing the course of events while the constraint is being satisfied (much as in Sketchpad). For example, constraining the end of a chain to connect to a post makes it move from where it is toward the post. This example was the subject of an animation by Barr and Barzel, one frame of which is shown in Color Plate IV.26.

21.3.7 Physically Based Animation

The dynamics we have described are examples of physically based animations. So, in animated form, are the physically based models of cloth, plasticity, and rigid-body motion described in Chapter 20. These models are based on simulations of the evolution of physical

[8]Of course, the floor actually does move when you step on it, but only a very small amount. We usually want to avoid modeling the floor as a massive object, and instead just model it as a fixed object.

systems. Various formulations of classical mechanical behavior have been developed [GOLD80]; they all represent the evolution of a physical system as a solution to a system of partial differential equations. The solutions to these equations can be found with numerical-analysis packages and can be used to derive animation sequences. In the Kass–Witkin motion modeling described in Chapter 20, the situation is complex. The forces acting on an assembly are not all known beforehand, since the object may be able to supply its own forces (i.e., use its muscles). This allows for physically based animation of a different sort: One seeks the forces that the muscles must apply to generate some action. Of course, there may be many solutions to such a problem, and the Kass–Witkin approach is to choose the path with the minimal work. This sort of animation ties together the work on constraints, dynamics, procedural control, and the actors that we have described. It is also extremely complex; determining the equations governing a mechanical assembly can be very difficult, since these equations may contain hundreds of interrelated variables.

21.4 BASIC RULES OF ANIMATION

Traditional character animation was developed from an art form into an industry at Walt Disney Studio between 1925 and the late 1930s. At the beginning, animation entailed little more than drawing a sequence of cartoon panels—a collection of static images that, taken together, made an animated image. As the techniques of animation developed, certain basic principles evolved that became the fundamental rules for character animation, and are still in use today [LAYB79; LASS87]. Despite their origins in cartoon-character animation, many of them apply equally to realistic 3D animations. These rules, together with their application to 3D character animation, are surveyed in [LASS87]. Here, we merely discuss a few of the most important ones. It is important to recognize, however, that these rules are not absolute. Just as much of modern art has moved away from the traditional rules for drawing, many modern animators have moved away from traditional rules of animation, often with excellent results (see, e.g., [LEAF74; LEAF77]).

The single most important of the traditional rules is *squash and stretch*, which is used to indicate the physical properties of an object by distortions of shape. A rubber ball or a ball of putty both distort (in different ways) when dropped on the floor. A bouncing rubber ball might be shown as elongating as it approachs the floor (a precursor to motion blur), flattening out when it hits, and then elongating again as it rises. By contrast, a metal sphere hitting the floor might distort very little but might wobble after the impact, exhibiting very small, high-frequency distortions. The jump made by Luxo Jr., described in Chapter 20 and simulated by the physically based modeling described in this chapter, is made with a squash and stretch motion: Luxo crouches down, storing potential energy in his muscles; then springs up, stretching out completely and throwing his base forward; and then lands, again crouching to absorb the kinetic energy of the forward motion without toppling over. It is a tribute to the potential of the Kass–Witkin simulation that it generated this motion automatically; it is also a tribute to traditional animators that they are able, in effect, to estimate a solution of a complex partial differential equation.

A second important rule is to use slow-in and slow-out to help smooth interpolations. Sudden, jerky motions are extremely distracting. This is particularly evident in interpolating the *camera position* (the point of view from which the animation is drawn or computed).

An audience viewing an animation identifies with the camera view, so sudden changes in camera position may make the audience feel motion sickness. Thus, camera changes should be as smooth as possible.

A third rule that carries over naturally from the 2D character-animation world to 3D animations, whether they are for the entertainment industry or for scientific visualization, is to *stage* the action properly. This includes choosing a view that conveys the most information about the events taking place in the animation, and (when possible) isolating events so that only one thing at a time occupies the viewer's attention. In the case of animations for scientific visualization, this isolation may not be possible—the events being simulated may be simultaneous—but it may be possible to view the scene from a position in which the different events occupy different portions of the image, and each can be watched individually without visual clutter from the others.

There are many other aspects of the design of animations that are critical. Many of these are matters of "eye" rather than strict rules, although rules of thumb are gradually evolving. The appropriate use of color is too often ignored, and garish animations in which objects are obscured by their colors are the result. The timing of animations is often driven by computing time instead of by final appearance; no time is given to introducing actions, to spacing them adequately, or to terminating them smoothly, and the resulting action seems to fly by. The details of an animation are given too much attention at the cost of the overall feeling, and the result has no aesthetic appeal. When you are planning an animation, consider these difficulties, and allot as much time as possible to aesthetic considerations in the production of the animation.

21.5 PROBLEMS PECULIAR TO ANIMATION

Just as moving from 2D to 3D graphics introduced many new problems and challenges, the change from 3D to 4D (the addition of the time dimension) poses special problems as well. One of these problems is *temporal aliasing*. Just as the aliasing problems in 2D and 3D graphics are partially solved by increasing the screen resolution, the temporal aliasing problems in animation can be partially solved by increasing temporal resolution. Of course, another aspect of the 2D solution is antialiasing; the corresponding solution in 3D is temporal antialiasing.

Another problem in 4D rendering is the requirement that we render many very similar images (the images in an ideal animation do not change much from one frame to the next—if they did, we would get jerky changes from frame to frame). This problem is a lot like that of rendering multiple scan lines in a 2D image: each scan line, on the average, looks a lot like the one above it. Just as scan-line renderers take advantage of this inter–scan-line coherence, it is possible to take advantage of interframe coherence as well. For ray tracing, we do this by thinking of the entire animation as occupying a box in 4D space–time—three spatial directions and one time direction. Each object, as it moves through time, describes a region of 4D space–time. For example, a sphere that does not move at all describes a spherical tube in 4D. The corresponding situation in 3D is shown in Fig. 21.10: If we make the 2D animation of a circle shown in part (a), the corresponding box in 3D space–time is that shown in part (b). The circle sweeps out a circular cylinder in space–time. For the 4D case, each image rendered corresponds to taking a 2D picture of a

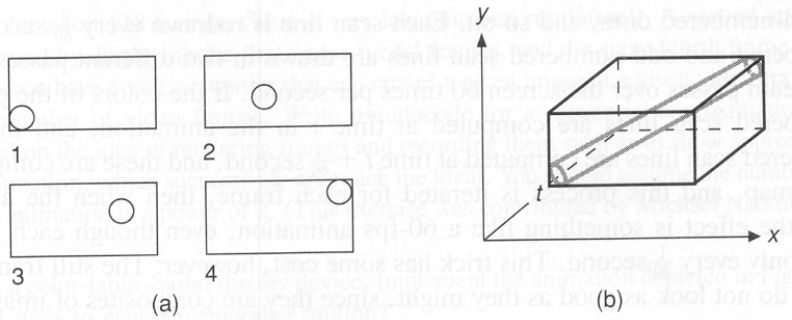


Fig. 21.10 The circle in (a) moves from lower left to upper right. By stacking these pictures along a third axis, we get the space–time animation shown in (b); the set of circles has become a tube in space–time.

3D slice of the 4D space–time. That is to say, we cast rays from a particular space–time point $(x, y, z; t)$ whose direction vectors have a time component of zero, so that all rays hit points whose time coordinate is also t. By applying the usual space-subdivision tricks for ray tracing to this 4D space–time, we can save a lot of time. A single hyperspace subdivision can be used for the entire course of the animation, so the time spent in creating the space subdivision does not need to be repeated once per frame. This idea and other uses of interframe coherence in ray tracing are described in [GLAS88].

High temporal resolution (many frames per second) may seem unnecessary. After all, video motion[9] seems smooth, and it is achieved at only 30 fps. Movies, however, at 24 fps, often have a jerkiness about them, especially when large objects are moving fast close to the viewer, as sometimes happens in a panning action. Also, as noted before, wagon wheels in movies sometimes appear to roll backward because of strobing. Higher temporal resolution helps to solve these problems. Doubling the number of frames per second lets the wagon wheel turn twice as fast before it seems to turn backward, and it certainly helps to smooth out the motion of fast-moving objects on the screen. The new Showscan technology [SHOW89] involves making and showing movies at 60 fps, on 70-millimeter film; this produces a bigger picture, which therefore occupies a larger portion of the visual field, and produces much smoother motion.

Temporal antialiasing can be done by taking multiple samples of a signal and computing their weighted average. In this case, however, the multiple samples must be in the time direction rather than in the spatial direction, so we compute the intensity at a point in the image for several sequential times and weight these to get a value at a particular frame. Many approaches to temporal-aliasing problems have been developed; super-sampling, box filtering in the time domain, and all the other tricks (including postfiltering!) from spatial antialiasing have been applied. One of the most successful is the distributed ray tracing described in Chapter 16 [COOK86].

Another trick for reducing temporal aliasing deserves mention: animation on fields. A conventional video image is traced twice; all the even-numbered scan lines are drawn, then

[9]We mean video motion filmed by a camera, not synthetically generated.

all the odd-numbered ones, and so on. Each scan line is redrawn every $\frac{1}{30}$ second, but the even-numbered and odd-numbered scan lines are drawn in two different passes. Thus, the electron beam passes over the screen 60 times per second. If the colors of the pixels of the even-numbered scan lines are computed at time t in the animation, and those for the odd-numbered scan lines are computed at time $t + \frac{1}{60}$ second, and these are composed into a single pixmap, and this process is iterated for each frame, then when the animation is displayed the effect is something like a 60-fps animation, even though each scan line is refreshed only every $\frac{1}{30}$ second. This trick has some cost, however: The still frames from an animation do not look as good as they might, since they are composites of images taken at two different times, and they thus seem to flicker if shown on an interlaced display. Also, twice as many frames must be rendered, so twice as many interpolated positions of the objects must be computed, and so on. Despite these drawbacks, the technique is widely employed in the computer-animation industry.

At the other extreme in animation is the process of *animating on twos*, or threes, and so on, in which the animation is produced at a temporal resolution lower than the display's refresh rate. Typically, each frame of the animation is displayed for two frames of video (''on twos''), so the effective refresh rate for video becomes 12 fps rather than 24 fps. This approach necessarily produces jerkier images (if no temporal antialiasing is done) or blurrier images (if it is). Animating on multiple frames and then filling in the intermediate ones can be useful in developing an animation, however, since it allows the animator to get a sense of the animation long before the individual frames have all been created (see Exercise 21.2.)

21.6 SUMMARY

Computer animation is a young field, and high-level animation is a recent development. As the computational power available to animators increases and as animation systems become more sophisticated, generating a high-quality computer animation will become simpler. At present, however, many compromises must be accepted. Simulation software is likely to advance rapidly, and the automated generation of graphical simulations is just a step away. On the other hand, until animation software contains knowledge about the tricks of conventional animation, computer character animation will remain as much an art as a science, and the ''eye'' of the animator will continue to have an enormous effect on the quality of the animation.

EXERCISES

21.1 Consider a unit square with corners at (0, 0) and (1, 1). Suppose we have a polygonal path defined by the vertices (0, 1), (1, 1), and (1, 0), in that order, and we wish to transform it to the polygonal path defined by the vertices (1, 0), (0, 0), and (0, 1) (i.e., we want to rotate it by 180°). Draw the intermediate stages that result if we linearly interpolate the positions of the vertices. This shows that strict interpolation of vertices is not adequate for key-frame interpolation unless the key frames are not too far apart.

21.2 Suppose that you are creating an animation, and can generate the frames in any order. If the animation is 128 frames long, a first ''pencil sketch'' can be created by rendering the first frame, and

displaying it for a full 128 frames. (This is very low temporal resolution!). A second approximation can be generated by displaying the first frame for 64 frames, and the sixty-fourth frame for the next 64. Suppose you have a video recorder that can record a given image at a given video-frame number, for a given number of video frames. Write pseudocode for a sequential-approximation recording scheme based on the idea of rendering frames and recording them such as to show approximations of the entire animation, which successively approach the ideal. You should assume the number of frames in the entire animation is a power of 2. (This exercise was contributed by Michael Natkin and Rashid Ahmad.)

21.3 Using a color-table–based display device, implement the animation depicted in Fig. 21.4 . Can you think of ways to generate smoother motion?

21.4 Using a frame-buffer that supports *pan* and *zoom* operations, implement the pan-zoom movie technique described in Section 21.1.2.

21.5 Make an animation of fireworks, using the particle systems of Section 20.5. If you do not have a frame buffer capable of displaying the images, you may instead be able to program the particle systems in POSTSCRIPT, and to display them on a printer. Hold the resulting pictures as a book and riffle through them, making a *flip-book* animation.

21.6 Suppose you were trying to make a 2D animation system that started with scanned-in hand drawings. Suggest techniques for cleaning up the hand drawings automatically, including the closing of nearly closed loops, the smoothing of curved lines, but not of sharp corners, etc. The automation of this process is extremely difficult, and trying to imagine how to automate the process suggests the value of interactive drawing programs as a source for 2D animation material.

21.7 a. Suppose that q and r are quaternions corresponding to rotations of ϕ and θ about the axis $\mathbf{v}$. Explicitly compute the product qr and use trigonometric identities to show that it corresponds to the rotation about $\mathbf{v}$ by angle $\phi + \theta$.

b. Show that the product of two unit quaternions is a unit quaternion.

c. If q is the unit quaternion $a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$, and s is the quaternion $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$, we can form a new quaternion $s' = qsq^{-1}$, where $q^{-1} = a - b\mathbf{i} - c\mathbf{j} - d\mathbf{k}$. If we write $s' = x'\mathbf{i} + y'\mathbf{j} + z'\mathbf{k}$, then the numbers x', y', and z' depend on the numbers x, y, and z. Find a matrix Q such that $[x' \quad y' \quad z']^t = Q[x \quad y \quad z]^t$. When we generate rotations from quaternions, it is this matrix form that we should use, not an explicit computation of the quaternion product.

d. Show that the vector $[b \quad c \quad d]^t$ is left fixed under multiplication by Q, so that Q represents a rotation about the vector $[b \quad c \quad d]$. It actually represents a rotation by angle $2\cos^{-1}(a)$, so that this describes the correspondence between quaternions and rotations.