CHAPTER

Interpolation-Based Animation 4

This chapter presents methods for precisely specifying the motion of objects. While the previous chapter presented basics of interpolating values, this chapter addresses how to use those basics to facilitate the production of computer animation. With these techniques, there is little uncertainty about the positions, orientations, and shapes to be produced. The computer is used only to calculate the actual values. Procedures and algorithms are used, to be sure, but in a very direct manner in which the animator has very specific expectations about the motion that will be produced on a frame-by-frame basis. These techniques are typified by the user setting precise conditions, such as beginning and ending values for position and orientation, and having the intermediate values filled in by some interpolative procedure. Later chapters cover "higher-level" algorithms in which the animator gives up some precision to produce motion with certain desired qualities. The techniques of this chapter deal with low-level, more precise, control of motion that is highly constrained by the animator.

This chapter starts off with a discussion of key-frame animation systems. These are the three-dimensional computer animation version of the main hand-drawn animation technique. The next section discusses animation languages. While animation languages can be used to control high-level algorithms, they are historically associated with their use in specifying the interpolation of values. Indeed, many of the language constructs particular to the animation domain deal with setting transformations at key frames and interpolating between them. The final sections cover techniques to deform an object, interpolating between three-dimensional object shapes, and two-dimensional morphing.

4.1 Key-frame systems

Many of the early computer animation systems were key-frame systems (e.g., [3] [4] [5] [20]). Most of these were two-dimensional systems based on the standard procedure used for hand-drawn animation, in which master animators define and draw the key frames of the sequence to be animated. In hand-drawn animation, assistant animators have the task of drawing the intermediate frames by mentally inferring the action between the keys. The key frames occur often enough in the sequence so that the intermediate action is reasonably well defined, or the keys are accompanied by additional information to indicate placement of the intermediate frames. In computer animation, the term key frame has been generalized to apply to any variable whose value is set at specific key frames and from which values for the intermediate frames are interpolated according to some prescribed procedure. These variables have been referred to in the literature as *articulation variables* (*avars*) [24], and the systems are sometimes referred to as *track based*. It is common for such systems to provide an interactive interface

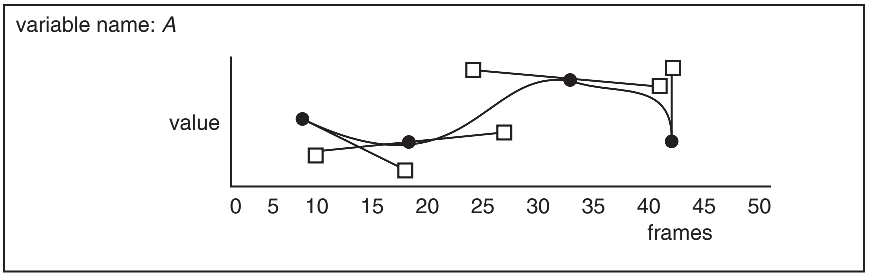


FIGURE 4.1

Sample interface for specifying interpolation of key values and tangents at segment boundaries.

with which the animator can specify the key values and can control the interpolation curve by manipulating tangents or interior control points. For example, see Figure 4.1. Early three-dimensional keyframe systems include TWIXT [15] and BBOP [33].

Because these animation systems keep the hand-drawn strategy of interpolating two-dimensional shapes, the basic operation is that of interpolating one (possibly closed) curve into another (possibly closed) curve. The interpolation is straightforward if the correspondence between lines in the frames is known in enough detail so that each pair of lines can be interpolated on a point-by-point basis to produce lines in the intermediate frames. This interpolation requires that for each pair of curves in the key frames the curves have the same number of points and that for each curve, whether open or closed, the correspondence between the points can be established.

Of course, the simplest way to interpolate the points is using linear interpolation between each pair of keys. For example, see Figure 4.2. Moving in arcs and allowing for ease-in/ease-out can be accommodated by applying any of the interpolation techniques discussed in Appendix B.5, providing that a point can be identified over several key frames.

Point-by-point correspondence information is usually not known, and even if it is, the resulting interpolation is not necessarily what the user wants. The best one can expect is for the curve-to-curve correspondence to be given. The problem is, given two arbitrary curves in key frames, to interpolate a curve as it “should” appear in intermediate frames. For example, observe the egg splatting against the wall in Figure 4.3.

For illustrative purposes, consider the simple case in which the key frames $f1$ and $f2$ consist of a single curve (Figure 4.4). The curve in frame $f1$ is referred to as $\boldsymbol{P}(u)$, and the curve in frame $f2$ is referred to as $\boldsymbol{Q}(v)$. This single curve must be interpolated for each frame between the key frames in which it is defined. In order to simplify the discussion, but without loss of generality, it is assumed that the curve, while it may wiggle some, is a generally vertical line in both key frames.

Some basic assumptions are used about what constitutes reasonable interpolation, such as the fact that if the curve is a single continuous open segment in frames $f1$ and $f2$, then it should remain a single continuous open segment in all the intermediate frames. Also assumed is that the top point of P, $\boldsymbol{P}(0)$, should interpolate to the top point in Q, $\boldsymbol{Q}(0)$, and, similarly, the bottom points should interpolate. However, what happens at intermediate points along the curve is so far left undefined (other than for the obvious assumption that the entire curve P should interpolate to the entire curve Q; that is, the mapping should be one-to-one and onto, also known as a *bijection*).

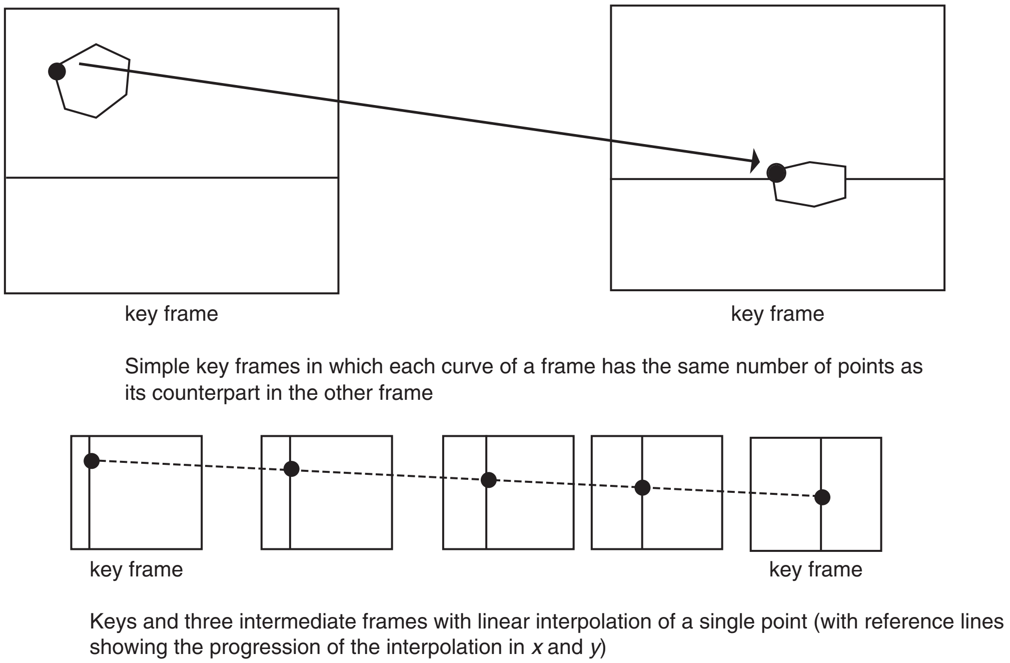

key frame
key frame
Simple key frames in which each curve of a frame has the same number of points as its counterpart in the other frame
key frame
key frame
Keys and three intermediate frames with linear interpolation of a single point (with reference lines showing the progression of the interpolation in *x* and *y*)

FIGURE 4.2

Simple key and intermediate frames. A specific point is identified in both key frames; the interpolation of the point over intermediate frames is shown with auxiliary lines showing its relative placement in the frames.

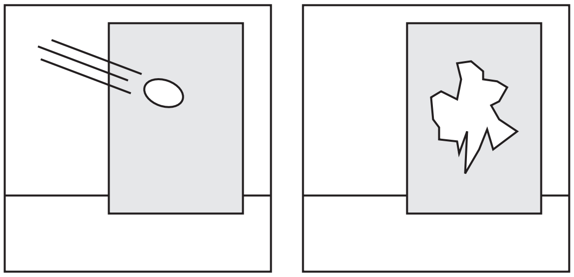


FIGURE 4.3

Object splatting against a wall. The shape must be interpolated from the initial egg shape to the splattered shape.

If both curves were generated with the same type of interpolation information, for example, each is a single, cubic Bezier curve, then intermediate curves could be generated by interpolating the control points and reapplying the Bezier interpolation. Another alternative would be to use interpolating functions to generate the same number of points on both curves. These points could then be interpolated on a point-by-point basis. Although these interpolations get the job done, they might not provide sufficient control to a user who has specific ideas about how the intermediate curves should look.

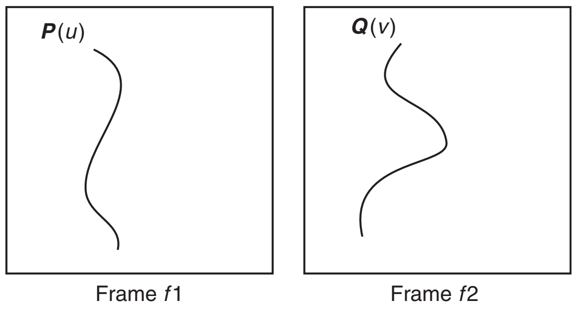


FIGURE 4.4

Two key frames showing a curve to be interpolated.

Reeves [29] proposes a method of computing intermediate curves using *moving point constraints*, which allows the user to specify more information concerning the correspondence of points along the curve and the speed of interpolation of those points. The basic approach is to use surface patch technology (two spatial dimensions) to solve the problem of interpolating a line in time (one spatial dimension, one temporal dimension).

The curve to be interpolated is defined in several key frames. Interpolation information, such as the path and speed of interpolation defined over two or more of the keys for one or more points, is also given. See Figure 4.5.

The first step is to define a segment of the curve to interpolate, bounded on top and bottom by interpolation constraints. Linear interpolation of the very top and very bottom of the curve, if not specified by a moving point constraint, is used to bind the top and bottom segments. Once a bounded segment has been formed, the task is to define an intermediate curve based on the constraints (see Figure 4.6). Various strategies can be used to define the intermediate curve segment, $C_t(u)$ in Figure 4.6, and are typically applications of surface patch techniques. For example, tangent information along the curves can be extracted from the curve definitions. The endpoint and tangent information can then be interpolated along the top and bottom interpolation boundaries. These can then be used to define an intermediate curve, $\boldsymbol{C}_t(u)$, using Hermite interpolation.

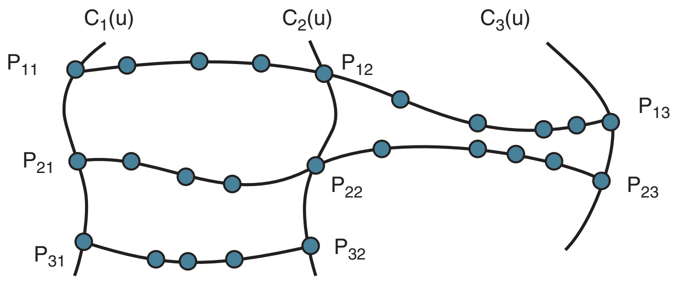


FIGURE 4.5

Moving point constraints showing the key points specified on the key curves as well as the intermediate points.

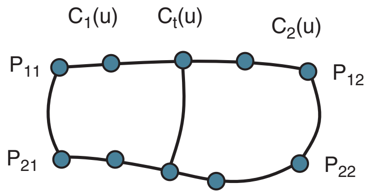


FIGURE 4.6

Patch defined by interpolation constraints made up of the boundary curve segments C_1 and C_{2q}.

4.2 Animation languages

An *animation language* is made up of structured commands that can be used to encode the information necessary to produce animations. If well-designed, an animation language can have enough expressive power to describe a wide variety of motions. Most languages are *script-based*, which means they are composed of text instructions; the instructions are recorded for later evaluation. Script-based languages can also be *interpreted* in which the animation is modified as new instructions are input by the user. Some languages are graphical in which, for example, flowchart-like diagrams encode relationships between objects and procedures. Nadia Magnenat-Thalmann and Daniel Thalmann present an excellent survey that includes many early script-based animation systems [22], and Steve May presents an excellent overview of animation languages from the perspective of his interest in procedural representations and encapsulated models [24]. Much of the discussion here is taken from these sources.

The first animation systems used general-purpose programming languages (e.g., Fortran) to produce the motion sequences. The language was used to compute whatever input values were needed by the rendering system to produce a frame of the animation. In these early systems, there were no special facilities in the language itself that were designed to support animation. Therefore, each time an animation was to be produced, there was overhead in defining graphical primitives, object data structures, transformations, and renderer output.

A language that does provide special facilities to support animation can use special libraries developed to augment a general-purpose programming language (e.g., an implementation of an applied programming interface) or it can be a specialized language designed from the ground up specifically for animation. In either case, typical features of an animation language include built-in input/output operations for graphical objects, data structures to represent objects and to support the hierarchical composition of objects, a time variable, interpolation functions, functions to animate object hierarchies, affine transformations, rendering-specific parameters, parameters for specifying camera attributes and defining the view frustum, and the ability to direct the rendering, viewing, and storing of one or more frames of animation. The program written in an animation language is often referred to as a *script*.

The advantage of using an animation language is twofold. First, the specification of an animation written in the language, the script, is a hard-coded record of the animation that can be used at any time to regenerate it. The script can be copied and transmitted easily. The script also allows the animation to be iteratively refined because it can be incrementally changed and a new animation generated. Second, the availability of programming constructs allows an algorithmic approach to motion control. The animation language, if sufficiently powerful, can interpolate values, compute arbitrarily complex numerical quantities, implement behavioral rules, and so on. The disadvantage of using an animation language is that it is, fundamentally, a programming language and thus requires the user (animator) to be a programmer. The animator must be trained not only in the arts but also in the art of programming.

4.2.1 Artist-oriented animation languages

To accommodate animators not trained in the art of computer programming, several simple animation languages were designed from the ground up with the intention of keeping the syntax simple and the semantics easy to understand (e.g., [16] [17] [23]). In the early days of computer graphics, almost all animation systems were based on such languages because there were few artists with technical backgrounds who were able to effectively program in a full-fledged programming language. These systems, as well as some of those with graphical user interfaces (e.g., [13] [26] [34]), were more accessible to artists but also had limited capabilities.

In these simple animation languages, typical statements referred to named objects, a transformation, and a time at which (or a span of time over which) to apply the transformation to the object. They also tended to have a syntax that was easy to parse (for the interpreter/compiler) and easy to read (for the animator). The following examples are from ANIMA II [16].

```
set position <name> <x> <y> <z> at frame <number>
set rotation <name> [X,Y,Z] to <angle> at frame <number>
change position <name> to <x> <y> <z> from frame <number> to frame <number>
change rotation <name> [X,Y,Z] by <angle> from frame <number> to frame <number>
```

Specific values or variable names could be placed in the statements, which included an indicator as to whether the transformation was relative to the object's current position or absolute in global coordinates. The instructions operated in parallel; the script was not a traditional programming language in that sense.

As animators became more adept at writing animation scripts in such languages, they tended to demand that more capability be incorporated into the language. As Steve May [24] points out, "By eliminating the language constructs that make learning animation languages difficult, command [artist-oriented] languages give up the mechanisms that make animation languages powerful." The developers found themselves adding looping constructs, conditional control statements, random variables, procedural calls, and data structure support. An alternative to developing a full featured animation programming language from scratch was to add support for graphical objects and operations to an existing language such as C/C++, Python, Java, or LISP. For example, Steve May developed a scheme-based animation language called AL [25].

4.2.2 Full-featured programming languages for animation

Languages have been developed with simplified programming constructs, but such simplification usually reduces the algorithmic power of the language. As users become more familiar with a language, the tendency is to want more of the algorithmic power of a general-purpose programming language put back into the animation language. More recently, animation systems have been developed that are essentially user interfaces implemented by a scripting language processor. Users can use the system strictly from the interface provided, or they can write their own scripts using the scripting language.

One example of such a system is AutoDesk's Maya. The scripting language is called MEL and there is an associated animation language whose scripts are referred to as expressions [36]. Maya's interface is written in MEL so anything that can be done using the interface can be done in a MEL script (with a few exceptions). MEL scripts are typically used for scene setup. The scene is represented as a network of nodes that is traversed each time the frame is advanced in a timeline.

Expressions create nodes in the network that are therefore executed each frame. The input to, and output from, an expression node link to the attributes of other nodes in the scene. Object nodes have attributes such as geometric parameters and normals. Transformation nodes have attributes such as translation, rotation, and scale. A transformation node is typically linked to an object node in order to transform the object data into the world coordinate system. Using expression nodes, the translation of one object could, for example, affect the rotation of other objects in the scene. The sample script below creates an expression that controls the *z*-rotation of the object named "slab" based on the *z*-translation of a sphere named "sphere1." During the first frame, translation values for the objects are initialized. For each subsequent frame, two of the sphere's translation values are set as a function of the frame number and the *z*-rotation of slab is set as a function of the sphere's *z*-translation value.

```
if (frame == 1) {
   slab1.translateY = 0;
   sphere1.translateX = -25;
   sphere1.translateY = 15;
}
else {
   sphere1.translateX = 25*sin(frame/100);
   sphere1.translateZ = 25*cos(frame/100) - 25;
   slab1.rotateZ = 10*sphere1.translateZ + 90;
}
```

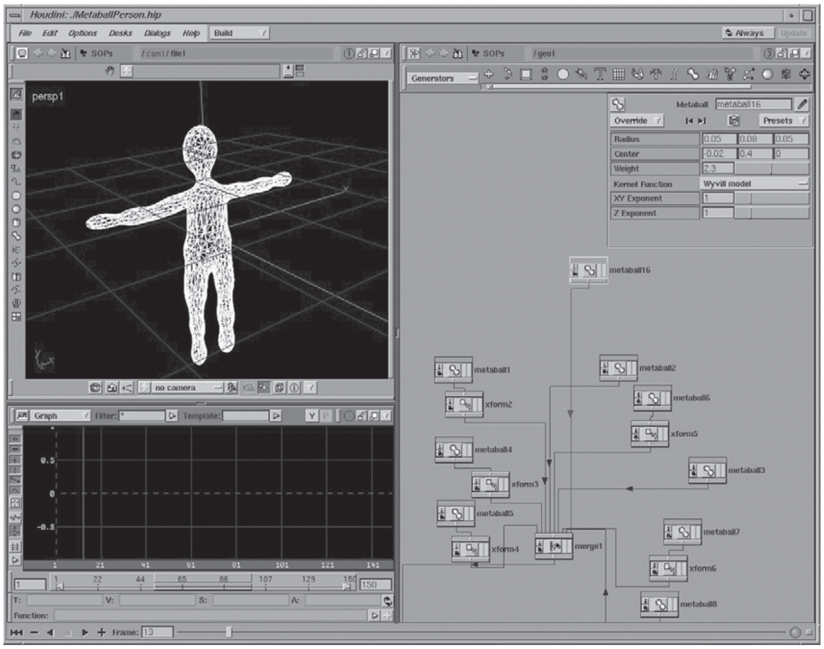
4.2.3 Articulation variables

A feature used by several languages is associating the value of a variable with a function, most notably of time. The function is specified procedurally or designed interactively using interpolating functions. Then a script, or other type of animation system, when it needs a value of the variable in its computation, passes the time variable to the function, which returns its value for that time to the computation. This technique goes by a variety of names, such as *track*, *channel*, or *articulation variable*. The term articulation variable, often shortened to avar, stems from its common use in some systems to control the articulation of linked appendages.

The use of avars, when an interactive system is provided to the user for inputting values or designing the functions, allows a script-based animation language to incorporate some interaction during the development of the animation. It also allows digitized data as well as arbitrarily complex functions to be easily incorporated into an animation by associating these with a particular avar.

4.2.4 Graphical languages

Graphical representations, such as those used in the commercial Houdini system [32], represent an animation by a dataflow network (see Figure 4.7). An acyclic graph is used to represent objects, operations, and the relationships among them. Nodes of the graph have inputs and outputs that connect to



Metaball metaball16

Radius	0.05	0.08	0.05
Center	-0.02	0.4	0
Weight	2.3		
Kernel Function	Wyvill model		
XY Exponent	1		
Z Exponent	1		

FIGURE 4.7

Sample Houdini dataflow network and the object it generates.

other nodes, and data are passed from node to node by arcs of the graphics that are interactively specified by the user. A node represents an operation to perform on the data being passed into the node, such as an object description. A transformation node will operate on an object description passed into the node and will produce a transformed object representation as output. Inputs to a node can also be used to parameterize a transformation or a data generation technique and can be set interactively, set according to an articulation variable, or set according to an arbitrary user-supplied procedure. This is a very effective way to visually design and represent the dependencies among various computations, functions, and values.

4.2.5 Actor-based animation languages

Actor-based languages are an object-oriented approach to animation in which an actor (encapsulated model [24]) is a graphical object with its associated data and procedures, including geometric description, display attributes, and motion control. Reynolds [30] popularized the use of the term *actor* in reference to the encapsulated models he uses in his ASAS system.

The basic idea is that the data associated with a graphical object should not only specify its geometry and display characteristics but also describe its motion. Thus the encapsulated model of a car includes how the doors open, how the windows roll down, and, possibly, the internal workings of the engine. Communication with the actor takes place by way of message passing; animating the object is carried out by passing requests to the actor for certain motions. The current status of the actor can be extracted by sending a request for information to the actor and receiving a message from the actor.

Actor-based systems provide a convenient way of communicating the time-varying information pertaining to a model. However, the encapsulated nature of actors with the associated message passing can result in inefficiencies when they are used with simulation systems in which all objects have the potential to affect all others.

4.3 Deforming objects

Deforming an object shape and transforming one shape into another is a visually powerful animation technique. It adds the notion of malleability and density. Flexible body animation makes the objects in an animation seem much more expressive and alive. There are physically based approaches that simulate the reaction of objects undergoing forces. However, many animators want more precise control over the shape of an object than that provided by simulations and/or do not want the computational expense of the simulating physical processes. In such cases, the animator wants to deform the object directly and define key shapes. Shape definitions that share the same edge connectivity can be interpolated on a vertex-to-vertex basis in order to smoothly change from one shape to the other. A sequence of key shapes can be interpolated over time to produce flexible body animation. Multivariate interpolation can be used to blend among a number of different shapes. The various shapes are referred to as *blend shapes* or *morph targets* and multivariate interpolation a commonly used technique in facial animation.

An immediate question that arises is "what is a shape?" Or "when are two shapes different?" It can probably be agreed that uniform scale does not change the shape of an object, but what about nonuniform scale? Does a rectangle have the same shape as a square? Most would say no. Most would agree that shearing changes the shape of an object. Elementary schools often teach that a square and a diamond are different shapes even though they may differ by only a rotation. Affine transformations are the simplest type of transformation that (sometimes) change the shape of an object. Affine transformations are defined by a 3×3 matrix followed by a translation. Affine transformations can be used to model the squash and stretch of an object, the jiggling of a block of Jello, and the shearing effect of an exaggerated stopping motion. While nonuniform scale can be used for simple squash and stretch, more interesting shape distortions are possible with nonaffine transformations. User-defined distortions are discussed in the following section; physically based approaches are discussed in Chapter 7.

4.3.1 Picking and pulling

A particularly simple way to modify the shape of an object is to displace one or more of its vertices. To do this on a vertex-by-vertex basis can be tedious for a large number of vertices. Simply grouping a number of vertices together and displacing them uniformly can be effective in modifying the shape of an object but is too restrictive in the shapes that can easily be created. An effective improvement to this approach is to allow the user to displace a vertex (the seed vertex) or group of vertices of the object and propagate the displacement to adjacent vertices along the surface while attenuating the amount of displacement. Displacement can be attenuated as a function of the distance between the seed vertex and the vertex to be displaced (see Figure 4.8). A *distance function* can be chosen to trade off quality of the results with computational complexity. One simple function uses the minimum number of edges connecting the seed vertex with the vertex to be displaced. A more accurate, but more computationally

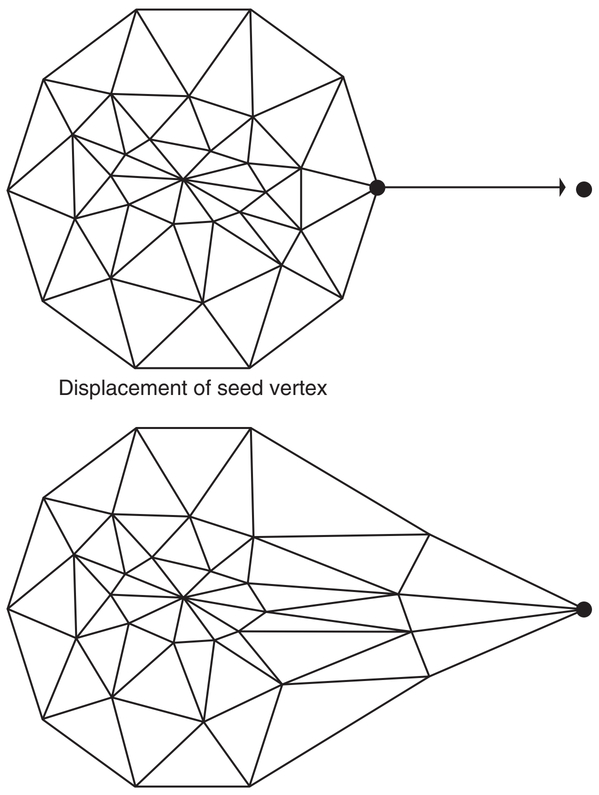

Displacement of seed vertex
Attenuated displacement propagated to adjacent vertices

FIGURE 4.8

Warping of object vertices.

expensive, function uses the minimum distance traveled over the surface of the object between the seed vertex and the vertex to be displaced.

Attenuation is typically a function of the distance metric. For example, the user can select a function from a family of power functions to control the amount of attenuation [28]. In this case, the minimum of connecting edges is used for the distance metric and the user specifies the maximum range of effect to be vertices within n edges of the seed vertex. A scale factor is applied to the displacement vector according to the user-selected integer value of k as shown in Equation 4.1.

$$\begin{aligned} S(i) &= 1 - \left(\frac{i}{n+1}\right)^{k+1} && k \geq 0 \\ &= \left(1 - \left(\frac{i}{n+1}\right)\right)^{-k+1} && k < 0 \end{aligned} \tag{4.1}$$

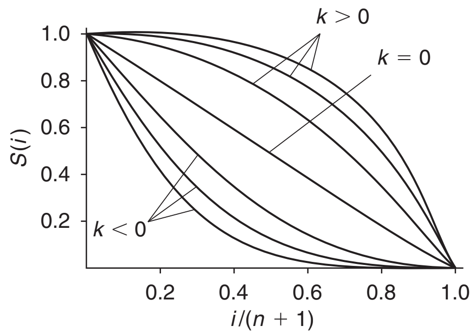


FIGURE 4.9

Power functions of Equation 4.1 for various values of k.

These attenuation functions are easy to compute and provide sufficient flexibility for many desired effects. When $k =$ zero it corresponds to a linear attenuation, while values of $k < 0$ create a more elastic impression. Values of $k > 0$ create the effect of more rigid displacements (Figure 4.9).

4.3.2 Deforming an embedding space

A popular technique for modifying the shape of an object is credited to Sederberg [31] and is called *free-form deformation* (FFD). FFD is only one of a number of techniques that share a similar approach, establishing a local coordinate system that encases the area of the object to be distorted. The initial configuration of the local coordinate system is such that the determination of the local coordinates of a vertex is a simple process. Typically, the initial configuration has orthogonal axes. The object to be distorted, whose vertices are defined in global coordinates, is then placed in this local coordinate space by determining the local coordinates of its vertices. The local coordinate system is distorted by the user in some way, and the local coordinates of the vertices are used to map their positions in global space. The idea behind these techniques is that it is easier or more intuitive for the user to manipulate the local coordinate system than to manipulate the vertices of the object. Of course, the trade-off is that the manipulation of the object is restricted to possible distortions of the coordinate system. The local coordinate grid is usually distorted so that the mapping is continuous (space is not torn apart), but it can be nonlinear, making this technique more powerful than affine transformations.

Two-dimensional grid deformation

Before discussing three-dimensional deformation techniques, simpler two-dimensional schemes are presented. Flexible body animation is demonstrated in the 1974 film *Hunger* [4]. In this seminal work, Peter Foldes, Nestor Burtnyk, and Marceli Wein used a two-dimensional technique that allowed for shape deformation. In this technique, the local coordinate system is a two-dimensional grid in which an object is placed. The grid is initially aligned with the global axes so that the mapping from local to global coordinates consists of a scale and a translation. For example, in Figure 4.10, assume that the local grid vertices are defined at global integer values from 20 to 28 in x and from

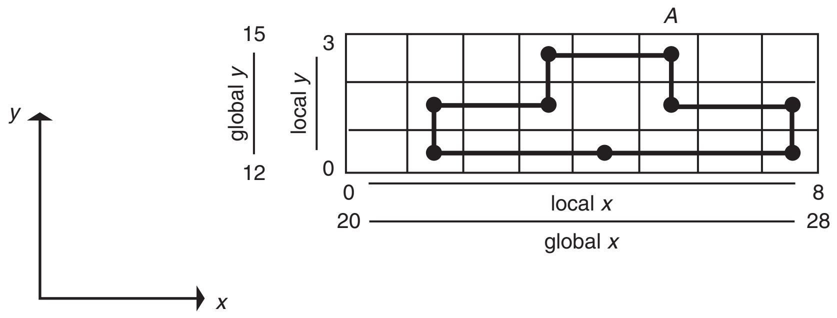


FIGURE 4.10

Two-dimensional coordinate grid used to locate initial position of vertices.

12 to 15 in *y*. Vertex *A* in the figure has global coordinates of (25.6, 14.7). The local coordinates of vertex *A* would be (5.6, 2.7).

The grid is then distorted by the user moving the vertices of the grid so that the local space is distorted. The vertices of the object are then relocated in the distorted grid by bilinear interpolation relative to the cell of the grid in which the vertex is located (Figure 4.11).

The bilinear interpolants used for vertex A would be 0.6 and 0.7. The positions of vertices of cell (5, 2) would be used in the interpolation. Assume the cell's vertices are named P_{00}, P_{01}, P_{10}, and P_{11}. Bilinear interpolation results in Equation 4.2.

$$P = (0.6)(0.7)P_{00} + (0.6)(1.0 - 0.7)P_{01} + (1.0 - 0.6)(0.7)P_{10} + (1.0 - 0.6)(1.0 - 0.7)P_{11} \quad (4.2)$$

(1–u)

$$\begin{aligned} P_{u0} &= (1-u)P_{00} + uP_{10} \\ P_{u1} &= (1-u)P_{01} + uP_{11} \\ P_{uv} &= (1-v)P_{u0} + vP_{u1} \\ &= (1-u)(1-v)P_{00} + (1-u)vP_{01} + u(1-v)P_{10} + uvP_{11} \end{aligned}$$

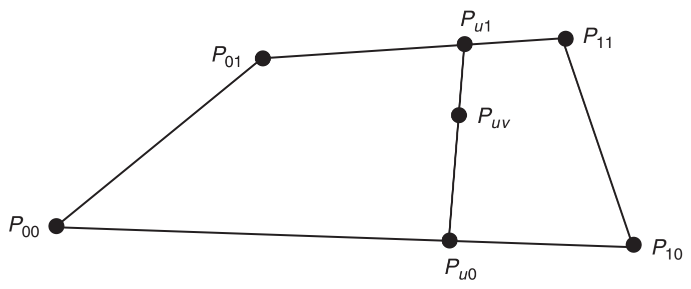


FIGURE 4.11

Bilinear interpolation used to locate vertex within quadrilateral grid.

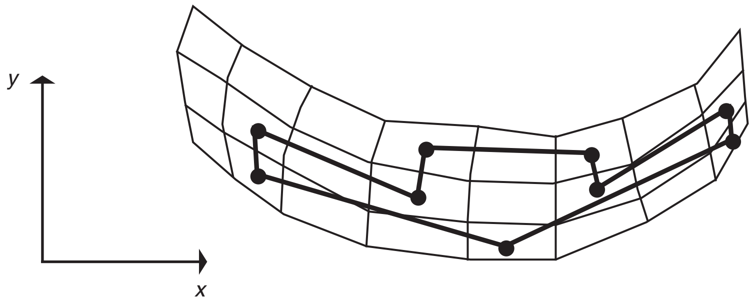


FIGURE 4.12

Two-dimensional grid deformation relocates vertices in space.

Once this is done for all vertices of the object, the object is distorted according to the distortion of the local grid (see Figure 4.12). For objects that contain hundreds or thousands of vertices, the grid distortion is much more efficient than individually repositioning each vertex. In addition, it is more intuitive for the user to specify a deformation.

Polyline deformation

A two-dimensional technique that is similar to the grid approach but lends itself to serpentine objects is based on a simple polyline (linear sequence of connected line segments) drawn by the user through the object to be deformed [28]. Polyline deformation is similar to the grid approach in that the object vertices are mapped to the polyline, the polyline is then modified by the user, and the object vertices are then mapped to the same relative location on the polyline.

The mapping to the polyline is performed by first locating the most relevant line segment for each object vertex. To do this, intersecting lines are formed at the junction of adjacent segments, and perpendicular lines are formed at the extreme ends of the polyline. These lines will be referred to as the boundary lines; each polyline segment has two boundary lines. For each object vertex, the closest polyline segment that contains the object vertex between its boundary lines is selected (Figure 4.13).

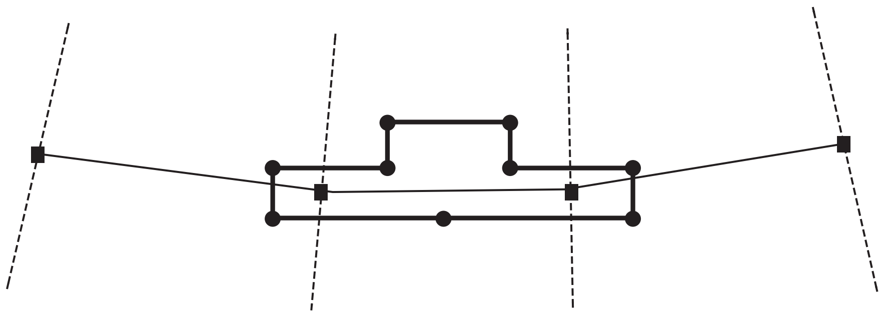


FIGURE 4.13

Polyline drawn through object; bisectors and perpendiculars are drawn as dashed lines.

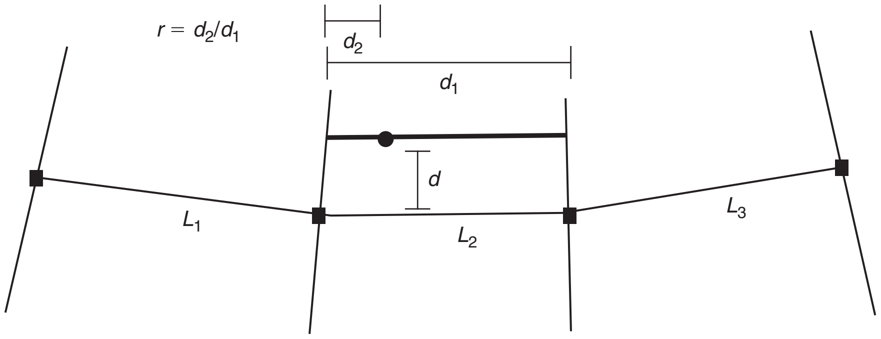


FIGURE 4.14

Measurements used to map an object vertex to a polyline.

Next, each object vertex is mapped to its corresponding polyline segment. A line segment is constructed through the object vertex parallel to the polyline segment and between the boundary lines. For a given object vertex, the following information is recorded (Figure 4.14): the closest line segment (L_2); the line segment's distance to the polyline segment (d); and the object vertex's relative position on this line segment, that is, the ratio r of the length of the line segment (d_1) and the distance from one end of the line segment to the object vertex (d_2).

The polyline is then repositioned by the user and each object vertex is repositioned relative to the polyline using the information previously recorded for that vertex. A line parallel to the newly positioned segment is constructed d units away and the vertex's new position is the same fraction along this line that it was in the original configuration (see Figure 4.15).

Global deformation

Alan Barr [1] presents a method of globally deforming the space in which an object is defined. Essentially, he applies a 3×3 transformation matrix, M, which is a function of the point being transformed, that is, $p' = M(p)\,p$, where $M(p)$ indicates the dependence of M on p. For example, Figure 4.16 shows a

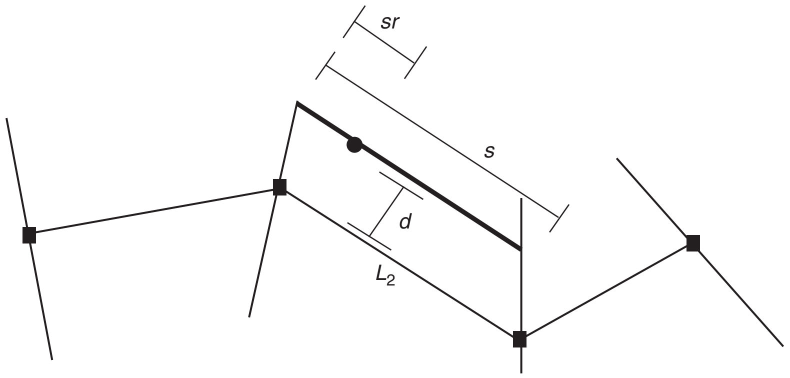
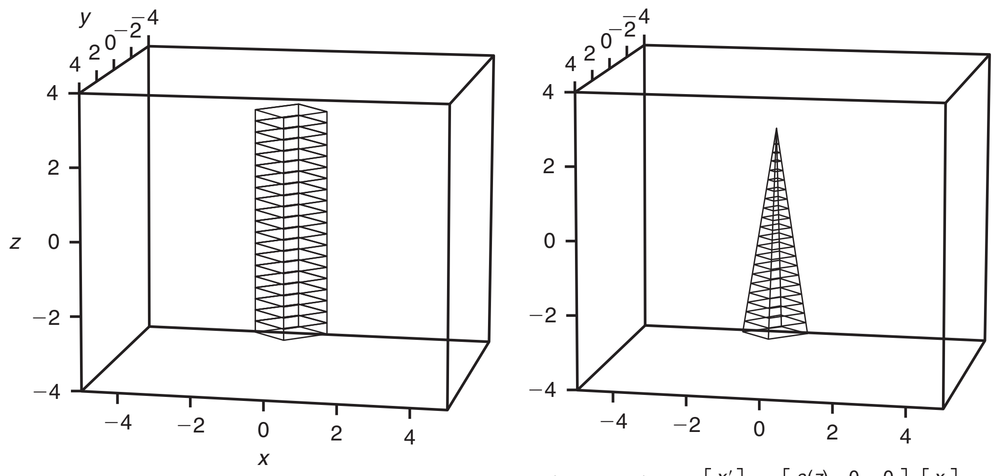


FIGURE 4.15

Remapping of an object vertex relative to a deformed polyline (see Figure 4.14).



$$s(z) = \frac{(\max z - z)}{(\max z - \min z)} \qquad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s(z) & 0 & 0 \\ 0 & s(z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$x' = s(z)x$$
$$y' = s(z)y \qquad P' = M(p)p$$
$$z' = z$$

Original object Tapered object

FIGURE 4.16

Global tapering.

simple linear two-dimensional tapering operation. There is no reason why the function ($s(z)$ in Figure 4.16) needs to be linear; it can be whatever function produces the desired effect. Other global deformations are possible. In addition to the taper operation, twists (Figure 4.17), bends (Figure 4.18), and various combinations of these (Figure 4.19) are possible.

FFD

An FFD is essentially a three-dimensional extension of Burtnyk's technique that incorporates higher-order interpolation. In both cases, a localized coordinate grid, in a standard configuration, is superimposed over an object. For each vertex of the object, coordinates relative to this local grid are determined that register the vertex to the grid. The grid is then manipulated by the user. Using its relative coordinates, each vertex is then mapped back into the modified grid, which relocates that vertex in global space. Instead of the linear interpolation used by Burtnyk, cubic interpolation is typically used with FFDs. In Sederberg's original paper [31], Bezier interpolation is suggested as the interpolating function, but any interpolation technique could be used.

In the first step of the FFD, vertices of an object are located in a three-dimensional rectilinear grid. Initially, the local coordinate system is defined by a not necessarily orthogonal set of three vectors (S, T, U). A vertex P is registered in the local coordinate system by determining its trilinear interpolants, as shown in Figure 4.20 and by Equations 4.3–4.5:

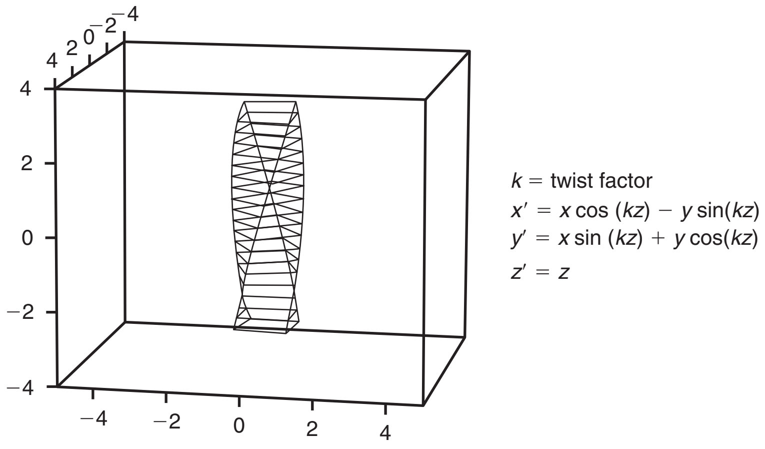


FIGURE 4.17

Twist about an axis.

$$s = (T \times U)\cdot(P - P_0)/((T \times U)\cdot S) \tag{4.3}$$

$$t = (U \times S)\cdot(P - P_0)/((U \times S)\cdot T) \tag{4.4}$$

$$u = (S \times T)\cdot(P - P_0)/((S \times T)\cdot U) \tag{4.5}$$

In these equations, the cross-product of two vectors forms a third vector that is orthogonal to the first two. The denominator normalizes the value being computed. In the first equation, for example, the projection of S onto $T \times U$ determines the distance within which points will map into the range $0 < s < 1$.

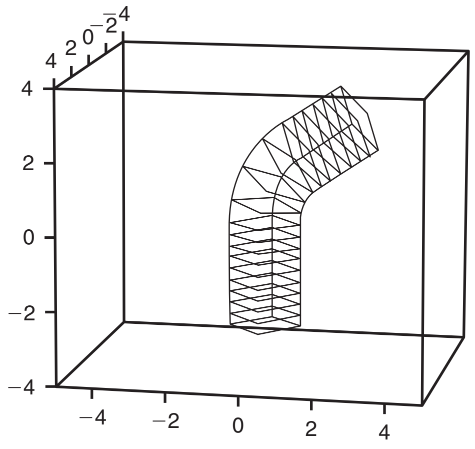
Given the local coordinates (s, t, u) of a point and the unmodified local coordinate grid, a point's position can be reconstructed in global space by simply moving in the direction of the local coordinate axes according to its local coordinates (Eq. 4.6):

$$P = P_0 + sS + tT + uU \tag{4.6}$$

To facilitate the modification of the local coordinate system, a grid of control points is created in the parallelepiped defined by the S, T, U axes. There can be an unequal number of points in the three directions. For example, in Figure 4.21, there are three in the S direction, two in the T direction, and one in the U direction.

If, not counting the origin, there are l points in the S direction, m points in the T direction, and n points in the U direction, the control points are located according to Equation 4.7.

$$P_{ijk} = P_0 + \frac{i}{l}S + \frac{j}{m}T + \frac{k}{n}U \quad \text{for} \begin{pmatrix} 0 \le i \le l \\ 0 \le j \le m \\ 0 \le k \le n \end{pmatrix} \tag{4.7}$$



$(z_{min} : z_{max})$ – *bend region*
(y_0, z_{min}) – *center of bend*

$$x' = x$$

$$y' = \begin{cases} y & z < z_{max} \\ y_0 - (RC_\theta) & z_{min} \le z \le z_{max} \\ y_0 - (RC_\theta) + (z - z_{max})S_\theta & z > z_{max} \end{cases}$$

$$z' = \begin{cases} z & z < z_{max} \\ z_{min} - (RS_\theta) & z_{min} \le z \le z_{max} \\ z_{min} - (RS_\theta) + (z - z_{max})C_\theta & z > z_{max} \end{cases}$$

$$\theta = \begin{cases} 0 & z < z_{min} \\ z_{max} - z_{min} & z > z_{max} \\ z - z_{min} & otherwise \end{cases}$$

$$C_\theta = \cos\theta$$
$$S_\theta = \sin\theta$$
$$R = y_0 - y$$

FIGURE 4.18

Global bend operation.

The deformations are specified by moving the control points from their initial positions. The function that effects the deformation is a trivariate Bezier interpolating function. The deformed position of a point P_{stu} is determined by using its (s, t, u) local coordinates, as defined by Equations 4.3–4.5, in the Bezier interpolating function shown in Equation 4.8. In Equation 4.8, $P(s, t, u)$ represents the global coordinates of the deformed point, and P_{ijk} represents the global coordinates of the control points.

$$P(s,t,u) = \sum_{i=0}^{l} \sum_{j=0}^{m} \sum_{k=0}^{n} \binom{l}{i} \binom{m}{j} \binom{n}{k} \; (1-s)^{l-i} s^i (1-t)^{m-j} t^j (1-u)^{n-k} u^k \tag{4.8}$$

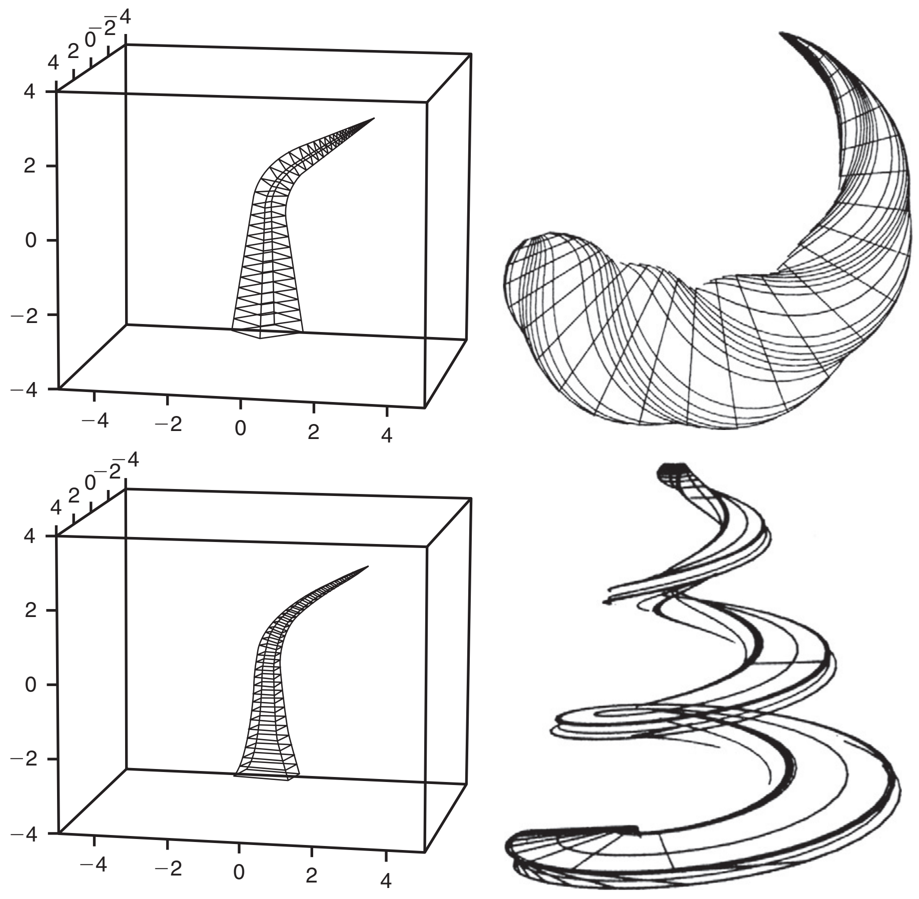


FIGURE 4.19

Examples of global deformations.

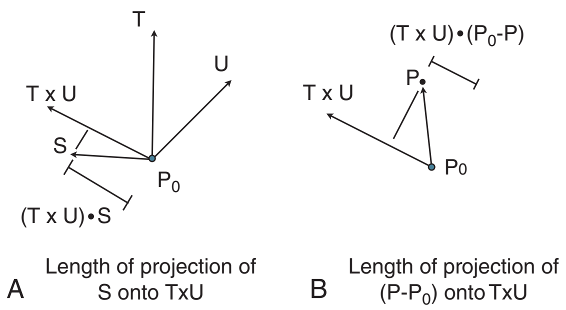

T
U
T x U
S
P_0
(T x U)•S
A
Length of projection of S onto TxU
(T x U)•(P_0-P)
P
T x U
P_0
B
Length of projection of (P-P_0) onto TxU

FIGURE 4.20

Computations forming the local coordinate *s* using an FFD coordinate system.

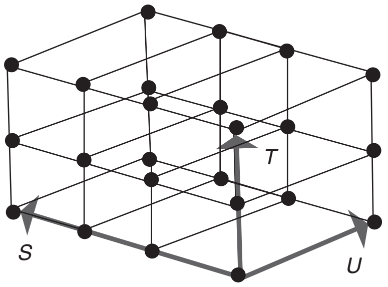


FIGURE 4.21

Grid of control points.

This interpolation function of Equation 4.8 is an example of trivariate interpolation. Just as the Bezier formulation can be used to interpolate a one-dimensional curve or a two-dimensional surface, the Bezier formulation is used here to interpolate a three-dimensional solid space.

Like Bezier curves and surfaces, multiple Bezier solids can be joined with continuity constraints across the boundaries. Of course, to enforce positional continuity, adjacent control lattices must share the control points along the boundary plane. As with Bezier curves, C^1 continuity can be ensured between two FFD control grids by enforcing colinearity and equal distances between adjacent control points across the common boundary (Figure 4.22).

Higher-order continuity can be maintained by constraining more of the control points on either side of the common boundary plane. However, for most applications, C^1 continuity is sufficient. One possibly useful feature of the Bezier formulation is that a bound on the change in volume induced by FFD can be analytically computed. See Sederberg [31] for details.

FFDs have been extended to include initial grids that are something other than a parallelepiped [11]. For example, a cylindrical lattice can be formed from the standard parallelepiped by merging the opposite boundary planes in one direction and then merging all the points along the cylindrical axis, as in Figure 4.23.

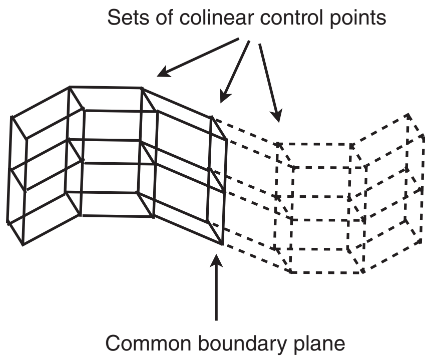


FIGURE 4.22

C^1 continuity between adjacent control grids.

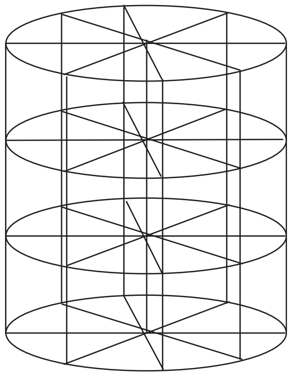


FIGURE 4.23

Cylindrical grid.

Composite FFDs—sequential and hierarchical

FFDs can be composed sequentially or hierarchically. In a sequential composition an object is modeled by progressing through a sequence of FFDs, each of which imparts a particular feature to the object. In this way, various detail elements can be added to an object in stages as opposed to trying to create one mammoth, complex FFD designed to do everything at once. For example, if a bulge is desired on a bent tube, then one FFD can be used to impart the bulge on the surface while a second is designed to bend the object (Figure 4.24).

Organizing FFDs hierarchically allows the user to work at various levels of detail. Finer-resolution FFDs, usually localized, are embedded inside FFDs higher in the hierarchy.[1] As a coarser-level FFD is used to modify the object's vertices, it also modifies the control points of any of its children FFDs that are within the space affected by the deformation. A modification made at a finer level in the hierarchy will remain well defined even as the animator works at a coarser level by modifying an FFD grid higher up in the hierarchy [18] (Figure 4.25).

If an FFD encases only part of an object, then the default assumption is that only those object vertices that are inside the initial FFD grid are changed by the modified FFD grid. Because the finer-level FFDs are typically used to work on a local area of an object, it is useful for the animator to be able to specify the part of the object that is subject to modification by a particular FFD. Otherwise, the rectangular nature of the FFD's grid can make it difficult to delimit the area that the animator actually wants to affect.

Animated FFDs

Thus far FFDs have been considered as a method to modify the shape of an object by repositioning its vertices. Animation would be performed by, for example, linear interpolation of the object's vertices on a vertex-by-vertex basis. However, FFDs can be used to control the animation in a more direct

[1]For this discussion, the hierarchy is conceptualized with the root node at the top, representing the coarsest level. More localized nodes with finer resolution are found lower in the hierarchy.

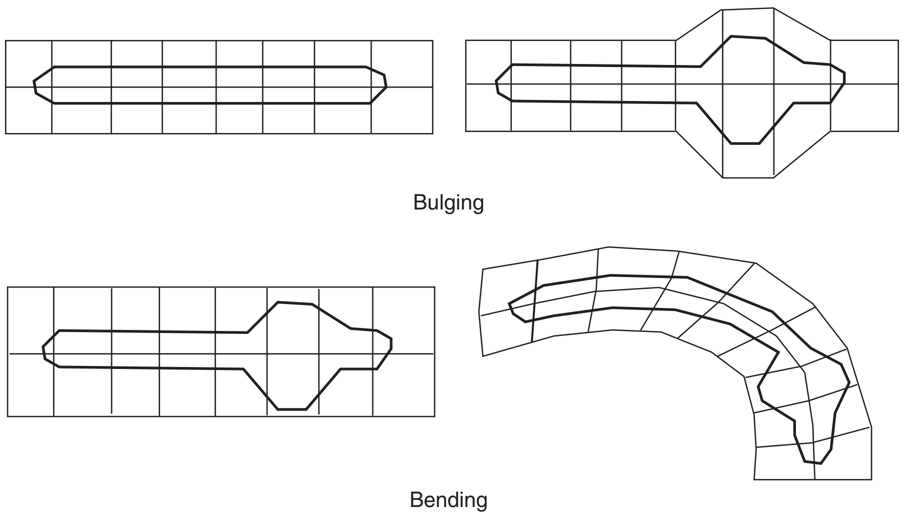


FIGURE 4.24

Sequential FFDs.

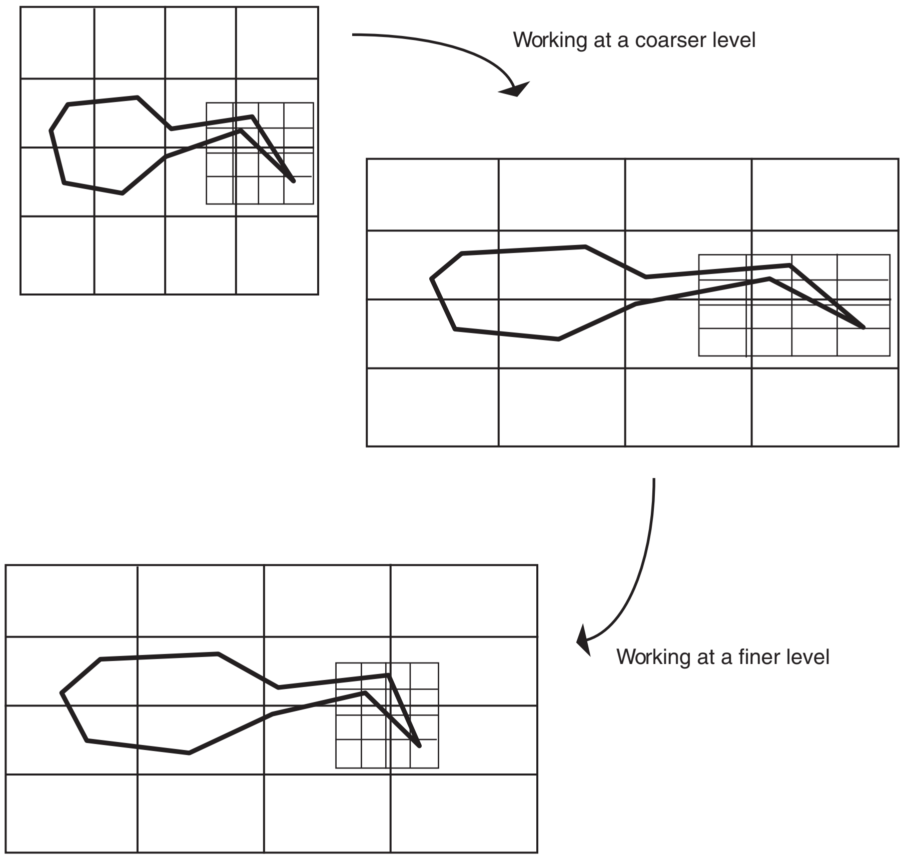


FIGURE 4.25

Simple example of hierarchical FFDs.

manner in one of two ways. The FFD can be constructed so that traversal of an object through the FFD space results in a continuous transformation of its shape [12]. Alternatively, the control points of an FFD can be animated, which results in a time-varying deformation that then animates the object's shape.

Deformation tools

As discussed by Coquillart [12], a *deformation tool* is defined as a composition of an initial lattice and a final lattice. The initial lattice is user defined and is embedded in the region of the model to be animated. The final lattice is a copy of the initial lattice that has been deformed by the user. While the deformation tool may be defined in reference to a particular object, the tool itself is represented in an object-independent manner. This allows for a particular deformation tool to be easily applied to any object (Figure 4.26). To deform an object, the deformation tool must be associated with the object, thus forming what Coquillart calls an *AFFD object*.

Moving the tool

A way to animate the object is to specify the motion of the deformation tool relative to the object. In the example of Figure 4.26, the deformation tool could be translated along the object over time. Thus a sequence of object deformations would be generated. This type of animated FFD works effectively when a particular deformation, such as a bulge, progresses across an object (Figure 4.27).

Moving the object

Alternatively, the object can translate through the local deformation space of the FFD and, as it does, be deformed by the progression through the FFD grid. The object can be animated independently in global world space while the transformation through the local coordinate grid controls the change in shape of the object. This type of animation works effectively for changing the shape of an object to move along a certain path (e.g., Figure 4.28).

Animating the FFD control points

Another way to animate an object using FFDs is to animate the control points of the FFD. For example, the FFD control points can be animated explicitly using key-frame animation, or their movement can be the result of physically based simulation. As the FFD grid points move, they define a changing deformation to be applied to the object's vertices (see Figure 4.29).

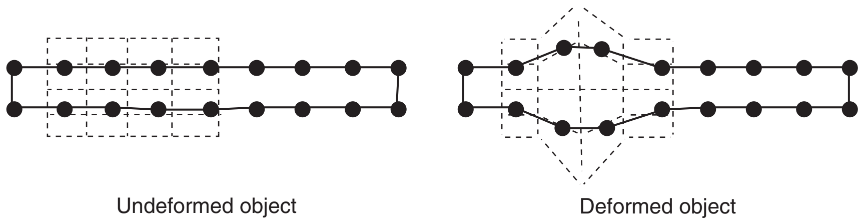


FIGURE 4.26

Deformation tool applied to an object.

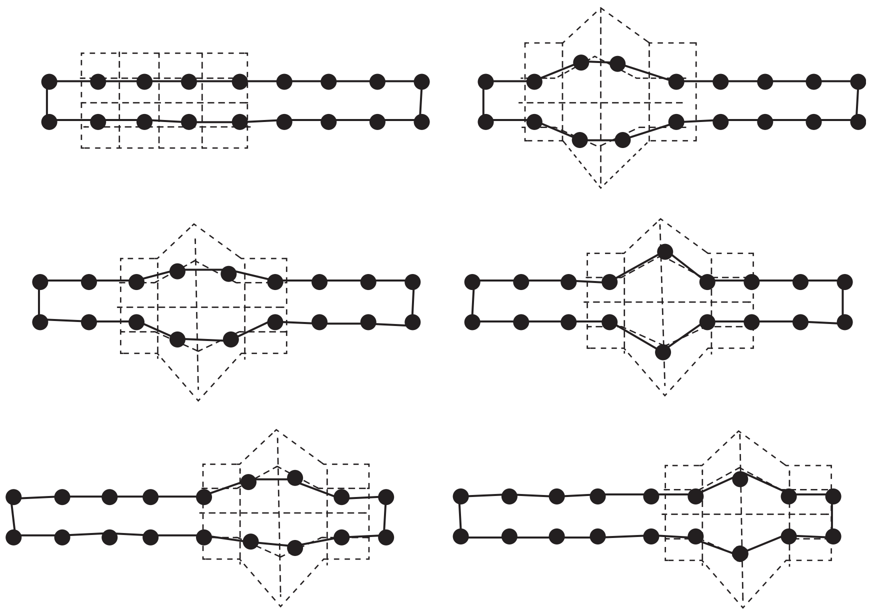


FIGURE 4.27

Deformation by translating the deformation tool relative to an object.

Chadwick et al. [8] described a layered approach to animation in which FFDs are used to animate a human form. The FFDs are animated in two ways. In the first technique, the positions of the FFD grid vertices are located relative to a wire skeleton the animator uses to move a figure. As the skeleton is manipulated, the grid vertices are repositioned relative to the skeleton automatically. The skin of the figure is then located relative to this local FFD coordinate grid. The FFDs thus play the role of muscular deformation of the skin. Joint articulation modifies the FFD grid, which in turn modifies the surface of the figure. The FFD grid is a mechanism external to the skin that effects the muscle deformation. The "muscles" in this case are meant not to be anatomical representations of real muscles but to provide for a more artistic style.

As a simple example, a hinge joint with adjacent links is shown in Figure 4.30; this is the object to be manipulated by the animator by specifying a joint angle. There is a surface associated with this structure that is intended to represent the skin. There are three FFDs: one for each of the two links and one for the joint. The FFDs associated with the links will deform the skin according to a stylized muscle, and the purpose of the FFD associated with the joint is to prevent interpenetration of the skin surface in highly bent configurations. As the joint bends, the central points of the link FFDs will displace upward and the interior panels of the joint FFD will rotate toward each other at the concave end in order to squeeze the skin together without letting it penetrate itself. Each of the grids is 5×4, and the joint grid is shown using dotted lines so that the three grids can be distinguished. Notice that the grids share a common set of control points where they meet.

Moving the FFD lattice points based on joint angle is strictly a kinematic method. The second technique employed by Chadwick [8] uses physically based animation of the FFD lattice points to animate

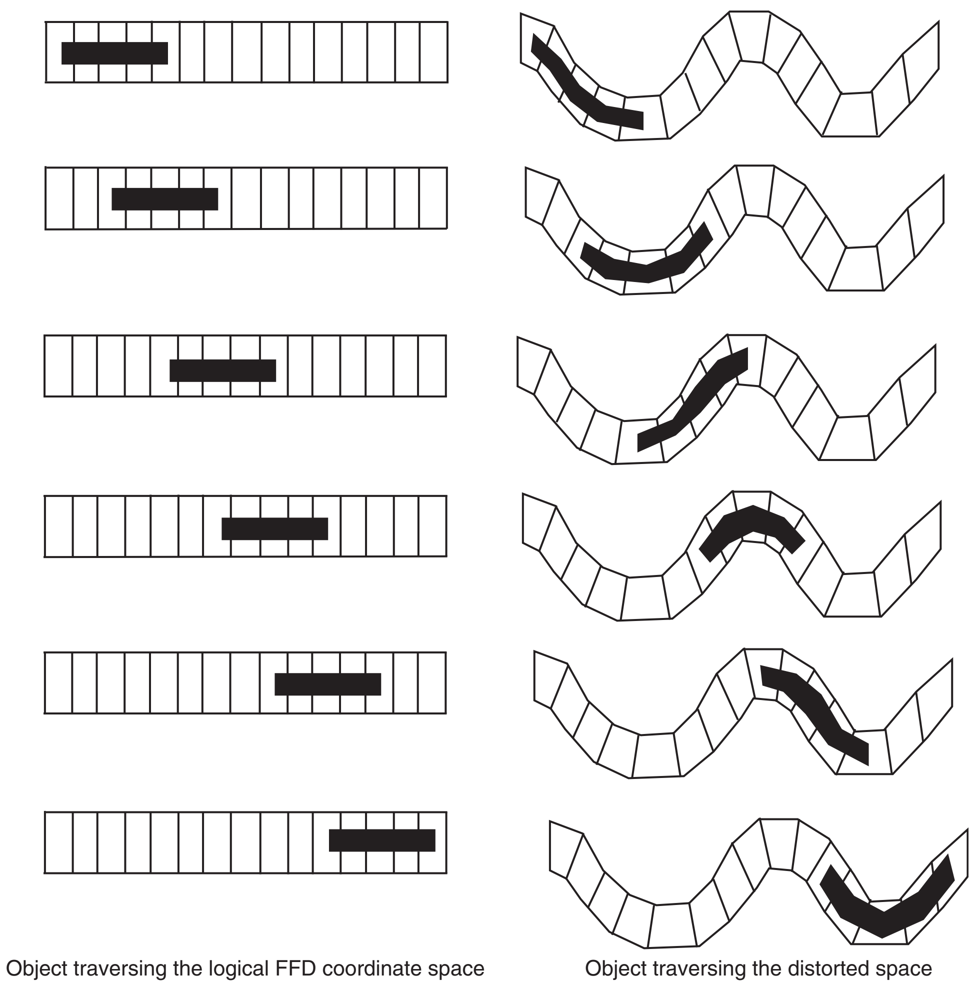

Object traversing the logical FFD coordinate space
Object traversing the distorted space

FIGURE 4.28

Deformation of an object by passing through FFD space.

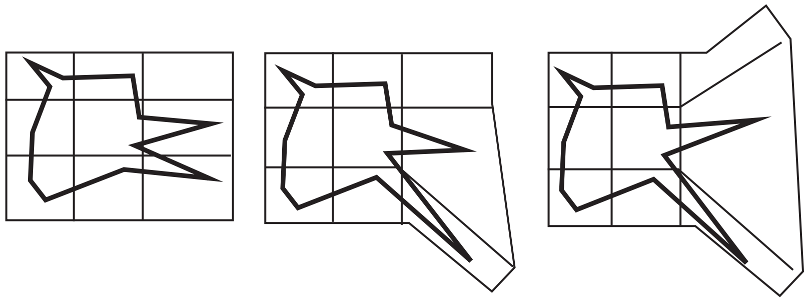


FIGURE 4.29

Using an FFD to animate a figure's head.

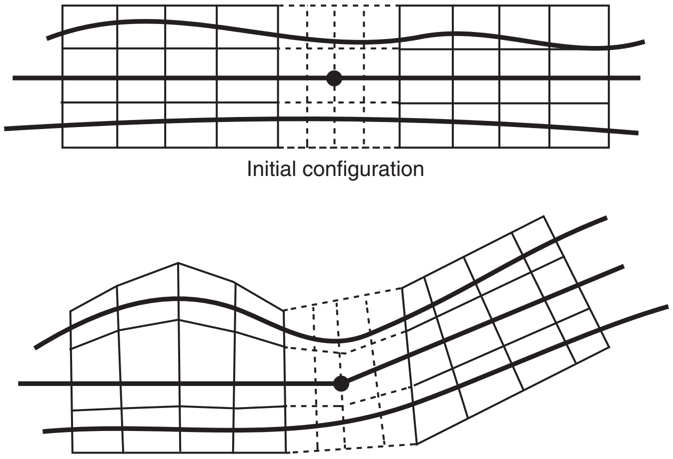


FIGURE 4.30

Using FFD to deform a surface around an articulated joint.

the figure. Animation of the FFD control points is produced by modeling the lattice with springs, dampers, and mass points. The control points of the lattice can then respond to gravity as well as kinematic motion of the figure. To respond to kinematic motion, the center of the FFD is fixed relative to the figure's skeleton. The user kinematically controls the skeleton, and the motion of the skeleton moves the center point of the FFD lattice. The rest of the FFD lattice points react to the movement of this center point via the spring-mass model, and the new positions of the FFD lattice points induce movement in the surface of the figure. This approach animates the clothes and facial tissue of a figure in animations produced by Chadwick [6] [7].

4.4 Three-dimensional shape interpolation

Changing one three-dimensional shape into another three-dimensional shape is a useful effect, but techniques are still being developed for arbitrary shapes. Several solutions exist that have various limitations. The techniques fall into one of two categories: surface based or volume based. The surface-based techniques use the boundary representation of the objects and modify one or both of them so that the vertex-edge topologies of the two objects match. Once this is done, the vertices of the object can be interpolated on a vertex-by-vertex basis. Surface-based techniques usually have some restriction on the types of objects they can handle, especially objects with holes through them. The number of holes through an object is an important attribute of an object's structure, or *topology*. The volume-based techniques consider the volume contained within the objects and blend one volume into the other. These techniques have the advantage of being less sensitive to different object topologies. However, volume-based techniques usually require volume representations of the objects and therefore tend to be more computationally intensive than surface-based approaches. In addition, the connectivity of the original shapes is typically not considered in volume-based approaches. Thus, some information, often important in animation, is lost. Volume-based approaches will not be discussed further.

The terms used in this discussion are defined by Kent et al. [19] and Weiler [35]. *Object* refers to an entity that has a three-dimensional surface geometry; the *shape* of an object refers to the set of points in object space that make up the object's surface; and *model* refers to any complete description of the shape of an object. Thus, a single object may have several different models that describe its shape. The term *topology* has two meanings, which can be distinguished by the context in which they are used. The first meaning, from traditional mathematics, is the connectivity of the surface of an object. For present purposes, this use of topology is taken to mean the number of holes an object has and the number of separate bodies represented. A doughnut and a teacup have the same topology and are said to be topologically equivalent. A beach ball and a blanket have the same topology. Two objects are said to be homeomorphic (or topologically equivalent) if there exists a continuous, invertible, one-to-one mapping between the points on the surfaces of the two objects. The *genus* of an object refers to how many holes, or passageways, there are through it. A beach ball is a genus 0 object; a teacup is a genus 1 object. The second meaning of the term topology, popular in the computer graphics literature, refers to the vertex/edge/face connectivity of a polyhedron; objects that are equivalent in this form of topology are the same except for the x-, y-, z-coordinate definitions of their vertices (the *geometry* of the object).

For most approaches, the shape transformation problem can be discussed in terms of the two subproblems: (1) the *correspondence problem*, or establishing the mapping from a vertex (or other geometric element) on one object to a vertex (or geometric element) on the other object, and (2) the *interpolation problem*, or creating a sequence of intermediate objects that visually represent the transformation of one object into the other. The two problems are related because the elements that are interpolated are typically the elements between which correspondences are established.

In general, it is not enough to merely come up with a scheme that transforms one object into another. An animation tool must give the user some control over mapping particular areas of one object to particular areas of the other object. This control mechanism can be as simple as aligning the object using affine transformations, or it can be as complex as allowing the user to specify an unlimited number of point correspondences between the two objects. A notable characteristic of the various algorithms for shape interpolation is the use of topological information versus geometric information. Topological information considers the logical construction of the objects and, when used, tends to minimize the number of new vertices and edges generated in the process. Geometric information considers the spatial extent of the object and is useful for relating the position in space of one object to the position in space of the other object.

While many of the techniques discussed in the following sections are applicable to surfaces defined by higher order patches, they are discussed here in terms of planar polyhedra to simplify the presentation.

4.4.1 Matching topology

The simplest case of transforming one object into another is when the two shapes to be interpolated share the same vertex-edge topology. Here, the objects are transformed by merely interpolating the positions of vertices on a vertex-by-vertex basis. For example, this case arises when one of the previous shape-modification techniques, such as FFD, has been used to modify one object, without changing the vertex-edge connectivity, to produce the second object. The correspondence between the two shapes is established by the vertex-edge connectivity structure shared by the two objects. The interpolation problem is solved, as in the majority of techniques presented here, by interpolating three-dimensional vertex positions.

4.4.2 Star-shaped polyhedra

If the two objects are both *star-shaped*[2] polyhedra, then polar coordinates can be used to induce a two-dimensional mapping between the two shapes. See Figure 4.31 for a two-dimensional example. The object surfaces are sampled by a regular distribution of rays emanating from a central point in the *kernel* of the object, and vertices of an intermediate object are constructed by interpolating between the intersection points along a ray. A surface is then constructed from the interpolated vertices by forming triangles from adjacent rays. Taken together, these triangles define the surface of the polyhedron. The vertices making up each surface triangle can be determined as a preprocessing step and are only dependent on how the rays are distributed in polar space. Figure 4.32 illustrates the sampling and interpolation for objects in two dimensions. The extension to interpolating in three dimensions is straightforward. In the three-dimensional case, polygon definitions on the surface of the object must then be formed.

4.4.3 Axial slices

Chen [9] interpolates objects that are star shaped with respect to a central axis. For each object, the user defines an axis that runs through the middle of the object. At regular intervals along this axis, perpendicular slices are taken of the object. These slices must be star shaped with respect to the point of intersection between the axis and the slice. This central axis is defined for both objects, and the part of each axis interior to its respective object is parameterized from 0 to 1. In addition, the user defines an orientation vector (or a default direction is used) that is perpendicular to the axis (see Figure 4.33).

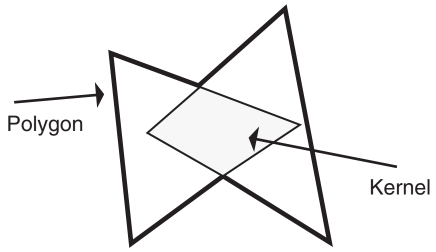


FIGURE 4.31

Star-shaped polygon and corresponding kernel from which all interior points are visible.

[2]A star-shaped (two-dimensional) polygon is one in which there is at least one point from which a line can be drawn to any point on the boundary of the polygon without intersecting the boundary; a star-shaped (three-dimensional) polyhedron is similarly defined. The set of points from which the entire boundary can be seen is referred to as the *kernel* of the polygon (in the two-dimensional case) or polyhedron (in the three-dimensional case).

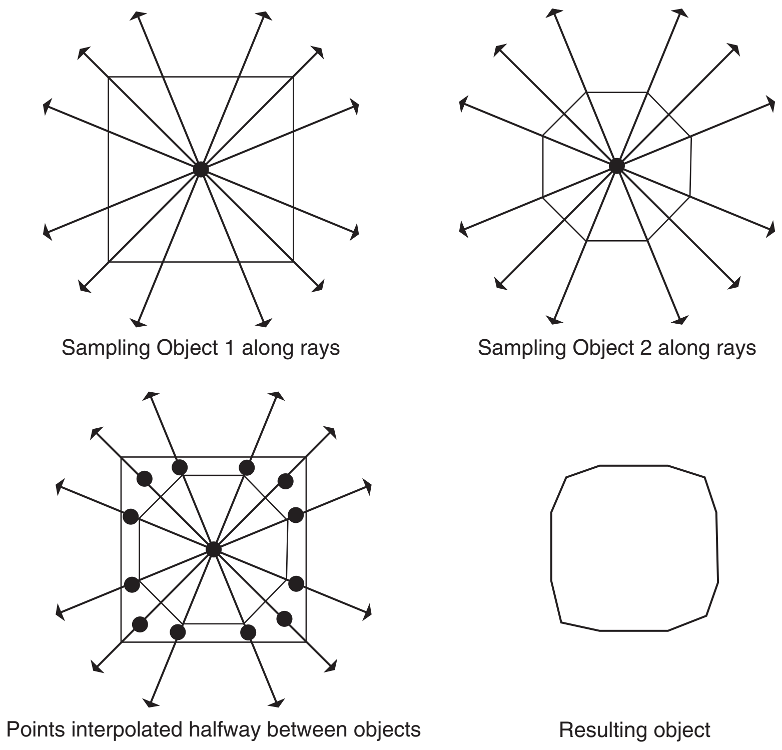

Sampling Object 1 along rays
Sampling Object 2 along rays
Points interpolated halfway between objects
Resulting object

FIGURE 4.32

Star-shaped polyhedral shape interpolation.

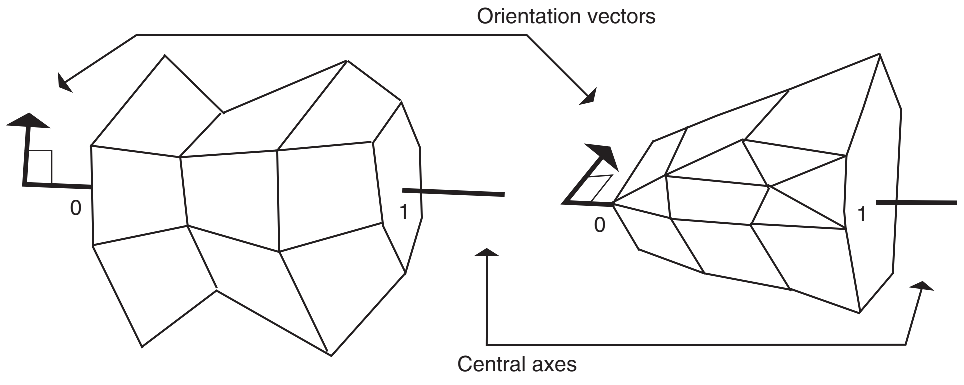


FIGURE 4.33

Object coordinate system suitable for creating axial slices.

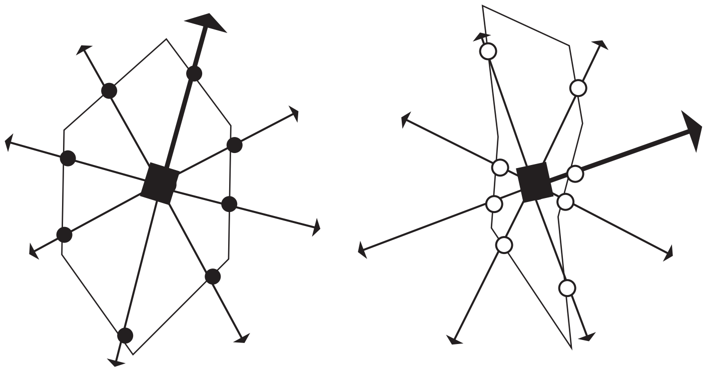


FIGURE 4.34

Projection lines for star-shaped polygons from objects in Figure 4.33.

Corresponding slices (corresponding in the sense that they use the same axis parameter to define the plane of intersection) are taken from each object. All of the slices from one object can be used to reconstruct an approximation to the original object using one of the contour-lofting techniques (e.g., [10] [14]). The two-dimensional slices can be interpolated pairwise (one from each object) by constructing rays that emanate from the center point and sample the boundary at regular intervals with respect to the orientation vector (Figure 4.34).

The parameterization along the axis and the radial parameterization with respect to the orientation vector together establish a two-dimensional coordinate system on the surface of the object. Corresponding points on the surface of the object are located in three-space. The denser the sampling, the more accurate is the approximation to the original object. The corresponding points can then be interpolated in three-space. Each set of ray–polygon intersection points from the pair of corresponding slices is used to generate an intermediate slice based on an interpolation parameter (Figure 4.35). Linear interpolation is often used, although higher-order interpolations are certainly useful. See Figure 4.36 for an example from Chen [9].

This approach can also be generalized somewhat to allow for a segmented central axis, consisting of a linear sequence of adjacent line segments. The approach may be used as long as the star-shaped restriction of any slice is maintained. The parameterization along the central axis is the same as before, except this time the central axis consists of multiple line segments.

4.4.4 Map to sphere

Even among genus 0 objects, more complex polyhedra may not be star shaped or allow an internal axis (single- or multi-segment) to define star-shaped slices. A more complicated mapping procedure may be required to establish the two-dimensional parameterization of objects' surfaces. One approach is to map both objects onto a common surface such as a unit sphere [19]. The mapping must be such that the entire object surface maps to the entire sphere with no overlap (i.e., it must be one-to-one and onto). Once both objects have been mapped onto the sphere, a union of their vertex-edge topologies can be constructed and then inversely mapped back onto each original object. This results in a new model for

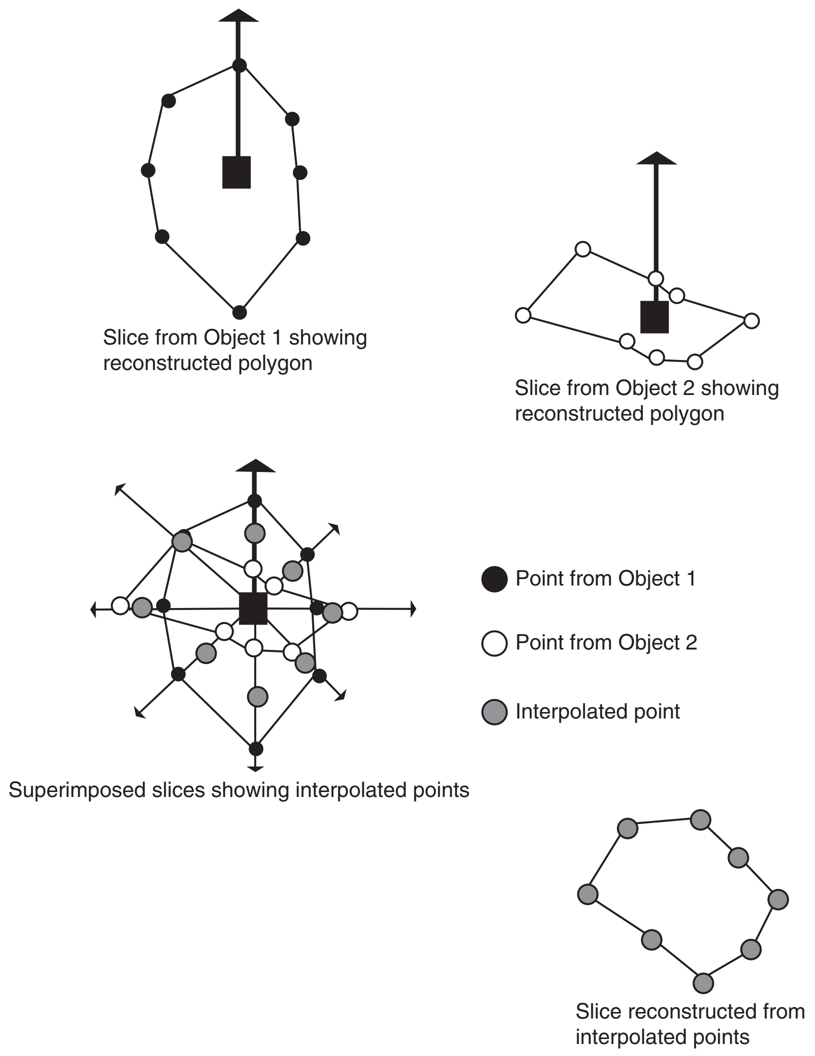

Slice from Object 1 showing reconstructed polygon
Slice from Object 2 showing reconstructed polygon
Point from Object 1
Point from Object 2
Interpolated point
Superimposed slices showing interpolated points
Slice reconstructed from interpolated points

FIGURE 4.35

Interpolating slices taken from two objects along each respective central axis.

each of the original shapes, but the new models now have the same topologies. These new definitions for the objects can then be transformed by a vertex-by-vertex interpolation.

There are several different ways to map an object to a sphere. No one way has been found to work for all objects, but, taken together, most of the genus 0 objects can be successfully mapped to a sphere. The most obvious way is to project each vertex and edge of the object away from a center point of the object onto the sphere. This, of course, works fine for star-shaped polyhedra but fails for others.

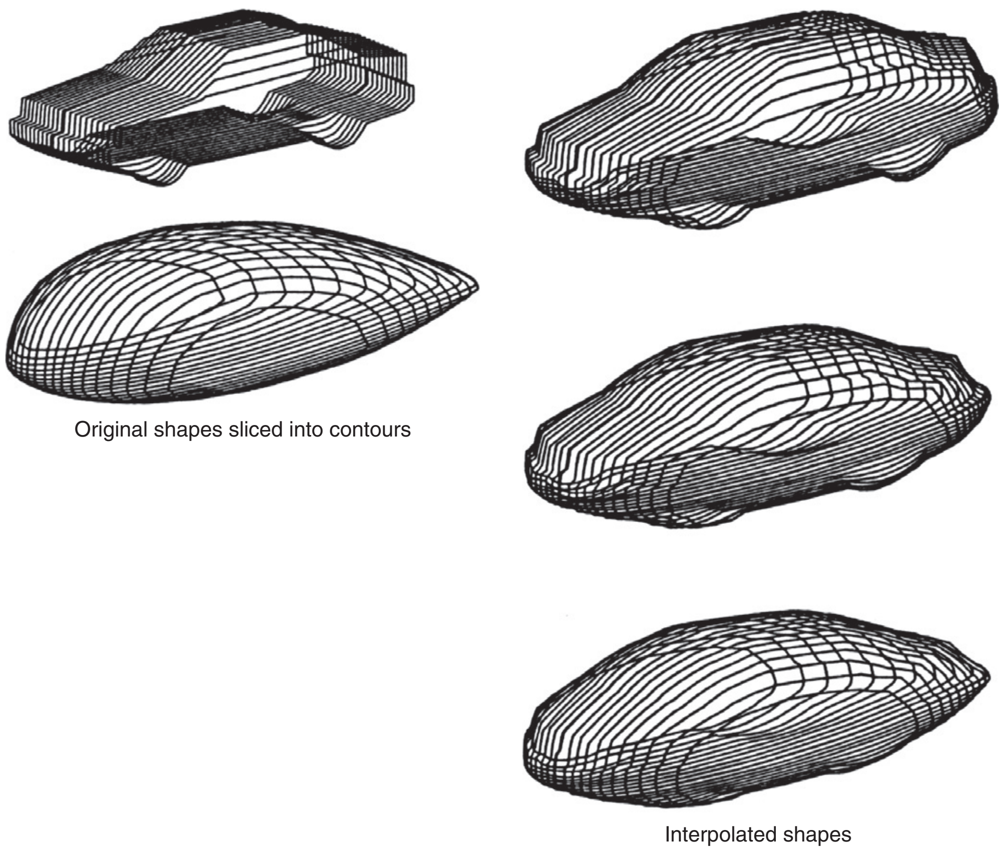


FIGURE 4.36

Three-dimensional shape interpolation from multiple two-dimensional slices [9] (© IEEE Computer Society).

Another approach fixes key vertices to the sphere. These vertices are either selected by the user or automatically picked by being topmost, bottommost, leftmost, and so on. A spring-damper model is then used to force the remaining vertices to the surface of the sphere as well as to encourage uniform edge lengths.

If both objects are successfully mapped to the sphere (i.e., no self-overlap), the projected edges are intersected and merged into one topology. The new vertices and edges are a superset of both object topologies. They are then projected back onto both object surfaces. This produces two new object definitions, identical in shape to the original objects but now having the same vertex-edge topology, allowing for a vertex-by-vertex interpolation to transform one object into the other.

Once the vertices of both objects have been mapped onto a unit sphere, edges map to circular arcs on the sphere. The following description of the algorithm to merge the topologies follows that found in Kent et al. [19]. It assumes that the faces of the models have been triangulated prior to projection onto the sphere and that degenerate cases, such as the vertex of one object projecting onto the vertex or edge of the other object, do not occur; these can be handled by relatively simple extensions to the algorithm.

To efficiently intersect each edge of one object with edges of the second object, it is necessary to avoid a brute force edge-edge comparison for all pairs of edges. While this approach would work theoretically, it would, as noted by Kent, be very time-consuming and subject to numerical inaccuracies that may result in intersection points erroneously ordered along an edge. Correct ordering of intersections along an edge is required by the algorithm.

In the following discussion on merging the topologies of the two objects, all references to vertices, edges, and faces of the two objects refer to their projection on the unit sphere. The two objects are referred to as Object *A* and Object *B*. Subscripts on vertex, edge, and face labels indicate which object they come from. Each edge will have several lists associated with it: an intersection list, a face list, and an intersection-candidate list.

The algorithm starts by considering one vertex, V_A, of Object *A* and finding the face, F_B, of Object *B* that contains vertex V_A (see Figure 4.37). Taking into account that it is operating within the two-dimensional space of the surface of the unit sphere, the algorithm can achieve this result quite easily and quickly.

The edges emanating from V_A are added to the work list. Face F_B becomes the current face, and all edges of face F_B are put on each edge's intersection-candidate list. This phase of the algorithm has finished when the work list has been emptied of all edges.

An edge, $\mathrm{E_A}$, and its associated intersection-candidate list are taken from the work list. The edge $\mathrm{E_A}$ is tested for any intersection with the edges on its intersection-candidate list. If no intersections are found, intersection processing for edge $\mathrm{E_A}$ is complete and the algorithm proceeds to the intersection-ordering phase. If an intersection, *I*, is found with one of the edges, $\mathrm{E_B}$, then the following steps are done: *I* is added to the final model; *I* is added to both edge E_A's intersection list and edge E_B's intersection list; the face, G_B, on the other side of edge E_B becomes the current face; and the other edges of face G_B (the edges not involved in the intersection) replace the edges in edge E_A's intersection-candidate list. In addition, to facilitate the ordering of intersections along an edge, pointers to the two faces from Object *A* that share E_A are associated with *I*. This phase of the algorithm then repeats by considering the edges on the intersection-candidate list for possible intersections and, if any are

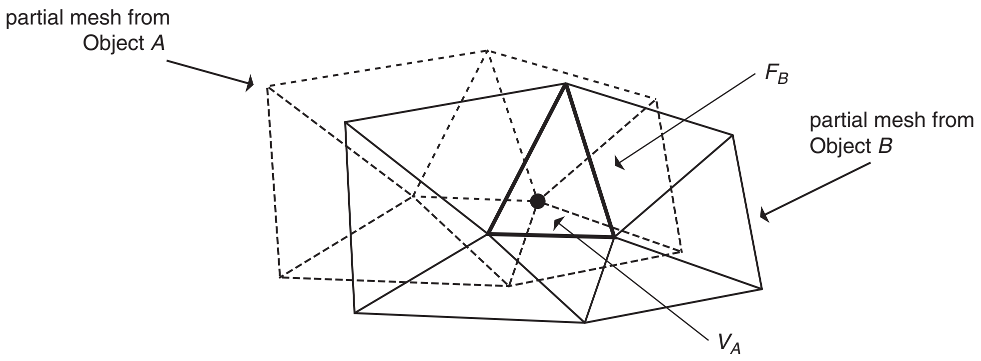


FIGURE 4.37

Locating initial vertex of Object *A* in the face of Object *B*.

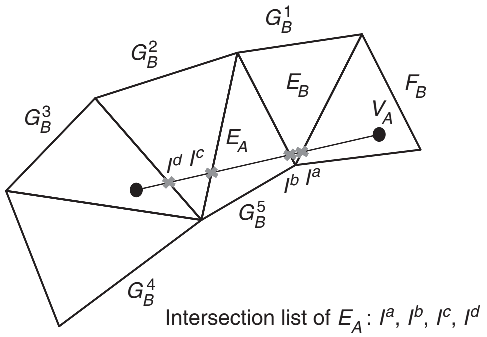


FIGURE 4.38

The intersection list for edge E_A.

found, processes subsequent intersections. When this phase is complete all edge-edge intersections have been found (see Figure 4.38).

For Object A, the intersections have been generated in sorted order along the edge. However, for Object B, the intersections have been associated with the edges but have been recorded in essentially a random order along the edge. The intersection-ordering phase uses the faces associated with intersection points and the list of intersections that have been associated with an edge of Object B to sort the intersection points along the edge. The face information is used because numerical inaccuracies can result in erroneous orderings if the numeric values of only parameters or intersection coordinates are used to order the intersections along the edge.

One of the defining vertices, V_B, for an edge, E_B, from Object B, is located within a face, F_A, of Object A. Initially, face F_A is referred to as the *current face*. As a result of the first phase, the edge already has all of its intersections recorded on its intersection list. Associated with each intersection point are the faces of Object A that share the edge E_B intersected to produce the point. The intersection points are searched to find the one that lists the current face, F_A, as one of its faces; one and only one intersection point will list F_A. This intersection point is the first intersection along the edge, and the other face associated with the intersection point will become the current face. The remaining intersection points are searched to see which lists this new current face; it becomes the next intersection point, and the other face associated with it becomes the new current face. This process continues until all intersection points have been ordered along edge E_B.

All of the information necessary for the combined models has been generated. It needs to be mapped back to each original object, and new polyhedral definitions need to be generated. The intersections along edges, kept as parametric values, can be used on the original models. The vertices of Object A, mapped to the sphere, need to be mapped onto the original Object B and vice versa for vertices of Object B. This can be done by computing the barycentric coordinates of the vertex with respect to the triangle that contains it (see Appendix B.2.9 for details). These barycentric coordinates can be used to locate the point on the original object surface.

Now all of the vertices, edges, and intersection points from both objects have been mapped back onto each object. New face definitions need to be constructed for each object. Because both models started out as triangulated meshes, there are only a limited number of configurations possible when

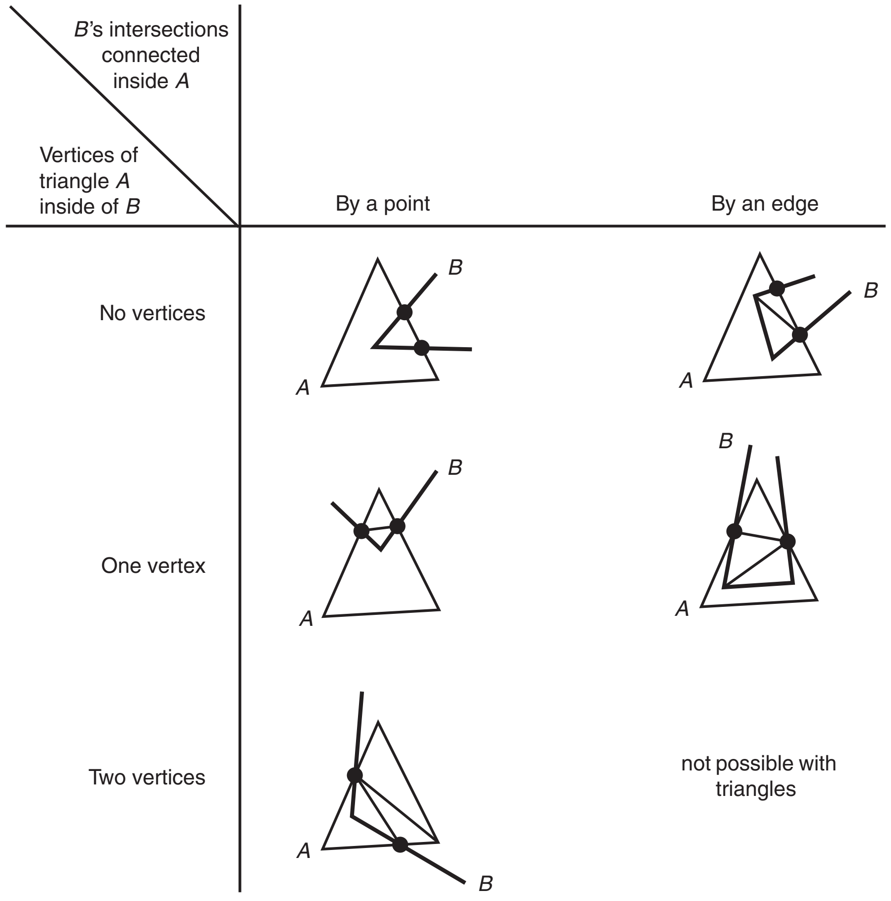

B's intersections connected inside A
Vertices of triangle A inside of B
By a point
By an edge
No vertices
One vertex
Two vertices
A
B
not possible with triangles

FIGURE 4.39

Configurations possible with overlapping triangles and possible triangulations.

one considers the triangulation required from the combined models (Figure 4.39). Processing can proceed by considering all of the original triangles from one of the models. For each triangle, use the intersection points along its edges and the vertices of the other object that are contained in it (along with any corresponding edge definitions) and construct a new triangulation of the triangle. When this is finished, repeat the process with the next triangle from the object.

The process of triangulation proceeds as follows. First, output any complete triangles from the other object contained in this object. Second, triangulate the fragments that include an edge or edge segment of the original triangle. Start at one of the vertices of the triangle and go to the first intersection encountered along the boundary of the triangle. The procedure progresses around the boundary of the triangle and ends when it returns to this intersection point. The next intersection along the boundary is also obtained. These two intersections are considered and the configuration identified by noting whether zero, one, or two original vertices are contained in the boundary between the two intersection points. The element inside the triangle that connects the two vertices/intersections is a vertex or an edge of the other object (along with the two edge segments involved in the intersections) (see Figure 4.39). Once the configuration is determined, triangulating the region is a simple task. The procedure then continues with the succeeding vertices or intersections in the order that they appear around the boundary of the original triangle.

This completes the triangulation of the combined topologies on the sphere. The resulting mesh can then be mapped back onto the surfaces of both objects, which establishes new definitions of the original objects but with the same vertex-edge connectivity. Notice that geometric information is used in the various mapping procedures in an attempt to map similarly oriented regions of the objects onto one another, thus giving the user some control over how corresponding parts of the objects map to one another.

4.4.5 Recursive subdivision

The main problem with the procedure above is that many new edges are created as a result of the merging operation. There is no attempt to map existing edges into one another. To avoid a plethora of new edges, a recursive approach can be taken in which each object is reduced to two-dimensional polygonal meshes [27]. Meshes from each object are matched by associating the boundary vertices and adding new ones when necessary. The meshes are then similarly split and the procedure is recursively applied until everything has been reduced to triangles. During the splitting process, existing edges are used whenever possible to reduce the number of new edges created. Edges and faces are added during subdivision to maintain topological equivalence. A data structure must be used that supports a closed, oriented path of edges along the surface of an object. Each mesh is defined by being on a particular side (e.g., right side) of such a path, and each section of a path will be shared by two and only two meshes.

The initial objects are divided into an initial number of polygonal meshes. Each mesh is associated with a mesh from the other object so that adjacency relationships are maintained by the mapping. The simplest way to do this is merely to break each object into two meshes—a front mesh and a back mesh. A front and back mesh can be constructed by searching for the shortest paths between the topmost, bottommost, leftmost, and rightmost vertices of the object and then appending these paths (Figure 4.40). On particularly simple objects, care must be taken so that these paths do not touch except at the extreme points.

This is the only place where geometric information is used. If the user wants certain areas of the objects to map to each other during the transformation process, then those areas should be the initial meshes associated with each other, providing the adjacency relationships are maintained by the associations.

When a mesh is associated with another mesh, a one-to-one mapping must be established between the vertices on the boundary of the two meshes. If one of the meshes has fewer boundary vertices than the other, then new vertices must be introduced along its boundary to make up for the difference. There are various ways to do this, and the success of the algorithm is not dependent on the method. A suggested method is to compute the normalized distance of each vertex along the boundary as measured from the first vertex of the boundary (the topmost vertex can be used as the first vertex of the boundaries of the initial meshes). For the boundary with fewer vertices, new vertices can be added one at a time by searching for the largest gap in normalized distances for successive vertices in the boundary (Figure 4.41). These vertices must be added to the original object definition. When the boundaries have the same number of vertices, a vertex on one boundary is said to be associated with the vertex on the other boundary at the same relative location.

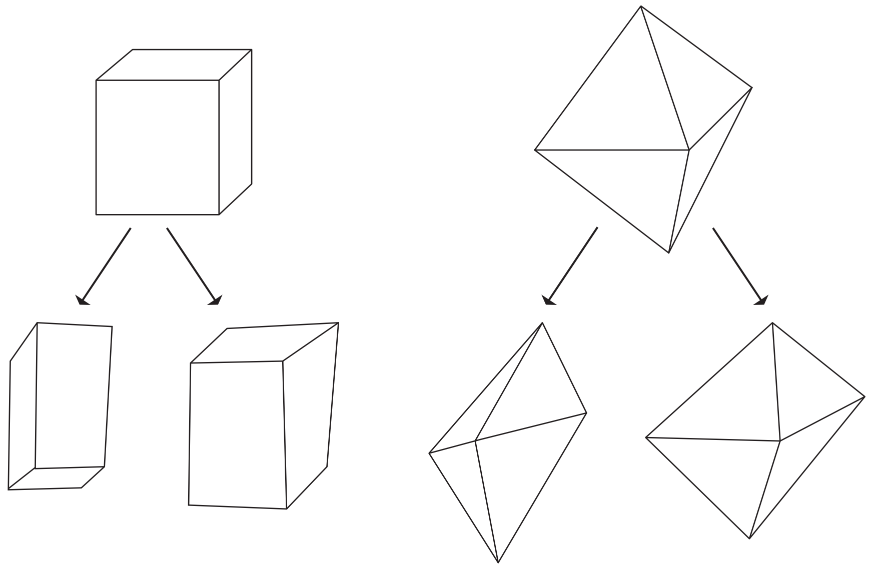
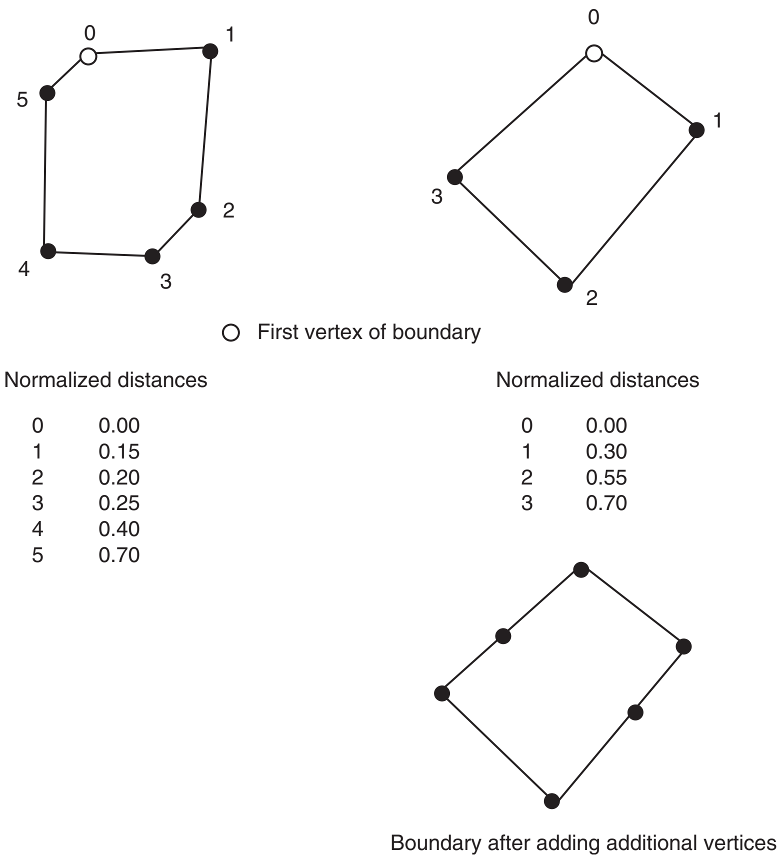


FIGURE 4.40

Splitting objects into initial front and back meshes.



Normalized distances

0	0.00
1	0.15
2	0.20
3	0.25
4	0.40
5	0.70

Normalized distances

0	0.00
1	0.30
2	0.55
3	0.70

FIGURE 4.41

Associating vertices of boundaries.

Once the meshes have been associated, each mesh is recursively divided. One mesh is chosen for division, and a path of edges is found across it. Again, there are various ways to do this and the procedure is not dependent on the method chosen for its success. However, the results will have a slightly different quality depending on the method used. One good approach is to choose two vertices across the boundary from each other and try to find an existing path of edges between them. An iterative procedure can be easily implemented that tries all pairs halfway around and then tries all pairs one less than halfway around, then two less, and so on. There will be no path only if the "mesh" is a single triangle—in which case the other mesh must be tried. There will be some path that exists on one of the two meshes unless both meshes are a single triangle, which is the terminating criterion for the recursion (and would have been previously tested for).

Once a path has been found across one mesh, then the path across the mesh it is associated with must be established between corresponding vertices. This may require creating new vertices and new edges (and, therefore, new faces) and is the trickiest part of the implementation, because minimizing the number of new edges will help reduce the complexity of the resulting topologies. When these paths (one on each mesh) have been created, the meshes can be divided along these paths, creating two pairs of new meshes. The boundary association, finding a path of edges, and mesh splitting are recursively applied to each new mesh until all of the meshes have been reduced to single triangles. At this point the new vertices and new edges have been added to one or both objects so that both objects have the same topology. Vertex-to-vertex interpolation of vertices can take place at this point in order to carry out the object interpolation.

4.5 Morphing (two-dimensional)

Two-dimensional image metamorphosis has come to be known as *morphing*. Although really an image postprocessing technique, and thus not central to the theme of this book, it has become so well known and has generated so much interest that it demands attention. Typically, the user is interested in transforming one image, called the source image, into the other image, called the destination image. There have been several techniques proposed in the literature for specifying and effecting the transformation. The main task is for the user to specify corresponding elements in the two images; these correspondences are used to control the transformation. Here, two approaches are presented. The first technique is based on user-defined coordinate grids superimposed on each image [37]. These grids impose a coordinate space to relate one image to the other. The second technique is based on pairs of user-defined feature lines. For each pair, one line in each image is marking corresponding features in the two images.

4.5.1 Coordinate grid approach

To transform one image into another, the user defines a curvilinear grid over each of the two images to be morphed. It is the user's responsibility to define the grids so that corresponding elements in the images are in the corresponding cells of the grids. The user defines the grid by locating the same number of grid intersection points in both images; the grid must be defined at the borders of the images in order to include the entire image (Figure 4.42). A curved mesh is then generated using the grid intersection points as control points for an interpolation scheme such as Catmull-Rom splines.

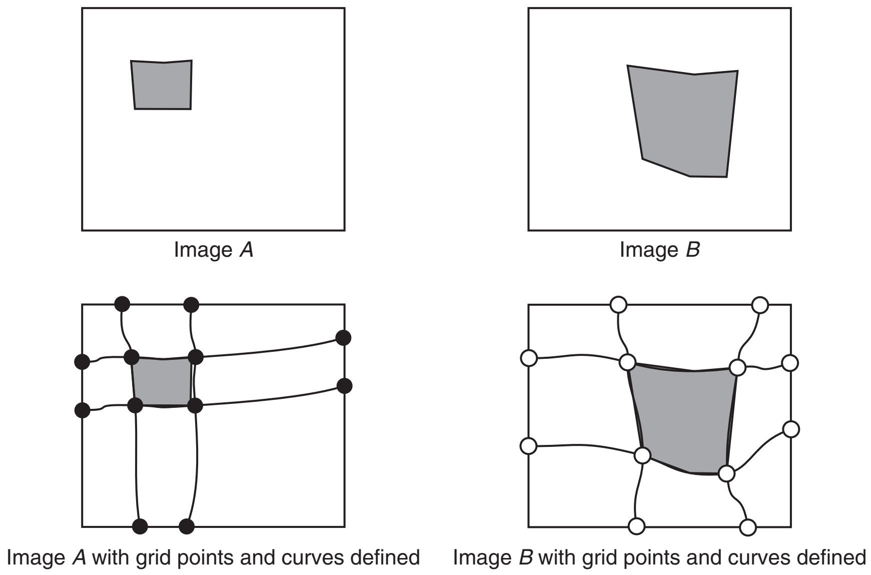


Image *A*
Image *B*
Image *A* with grid points and curves defined
Image *B* with grid points and curves defined

FIGURE 4.42

Sample grid definitions.

To generate an intermediate image, say t ($0 < t < 1.0$) along the way from the source image to the destination image, the vertices (points of intersection of the curves) of the source and destination grids are interpolated to form an intermediate grid. This interpolation can be done linearly, or grids from adjacent key frames can be used to perform higher-order interpolation. Pixels from the source and destination images are stretched and compressed according to the intermediate grid so that warped versions of both the source image and the destination grid are generated. A two-pass procedure is used to accomplish this (described in the following section). A cross-dissolve is then performed on a pixel-by-pixel basis between the two warped images to generate the final image (see Figure 4.43).

For purposes of explaining the two-pass procedure, it will be assumed that it is the source image to be warped to the intermediate grid, but the same procedure is used to warp the destination image to the intermediate grid.

First, the pixels from the source image are stretched and compressed in the x-direction to fit the interpolated grid. These pixels are then stretched and compressed in the y-direction to fit the intermediate grid. To carry this out, an auxiliary grid is computed that, for each grid point, uses the y-coordinate from the corresponding grid point of the source image grid and the x-coordinate from the corresponding point of the intermediate grid. The source image is stretched/compressed in x by mapping it to the auxiliary grid, and then the auxiliary grid is used to stretch/compress pixels in y to map them to the intermediate grid. In the following discussion, it is assumed the curves are numbered left to right and bottom to top; a curve's number is referred to as its *index*.

Figure 4.44 illustrates the formation of the auxiliary grid from the source image grid and the intermediate grid. Once the auxiliary grid is defined, the first pass uses the source image and auxiliary grids to distort the source pixels in the x-direction. For each row of grid points in both the source and the

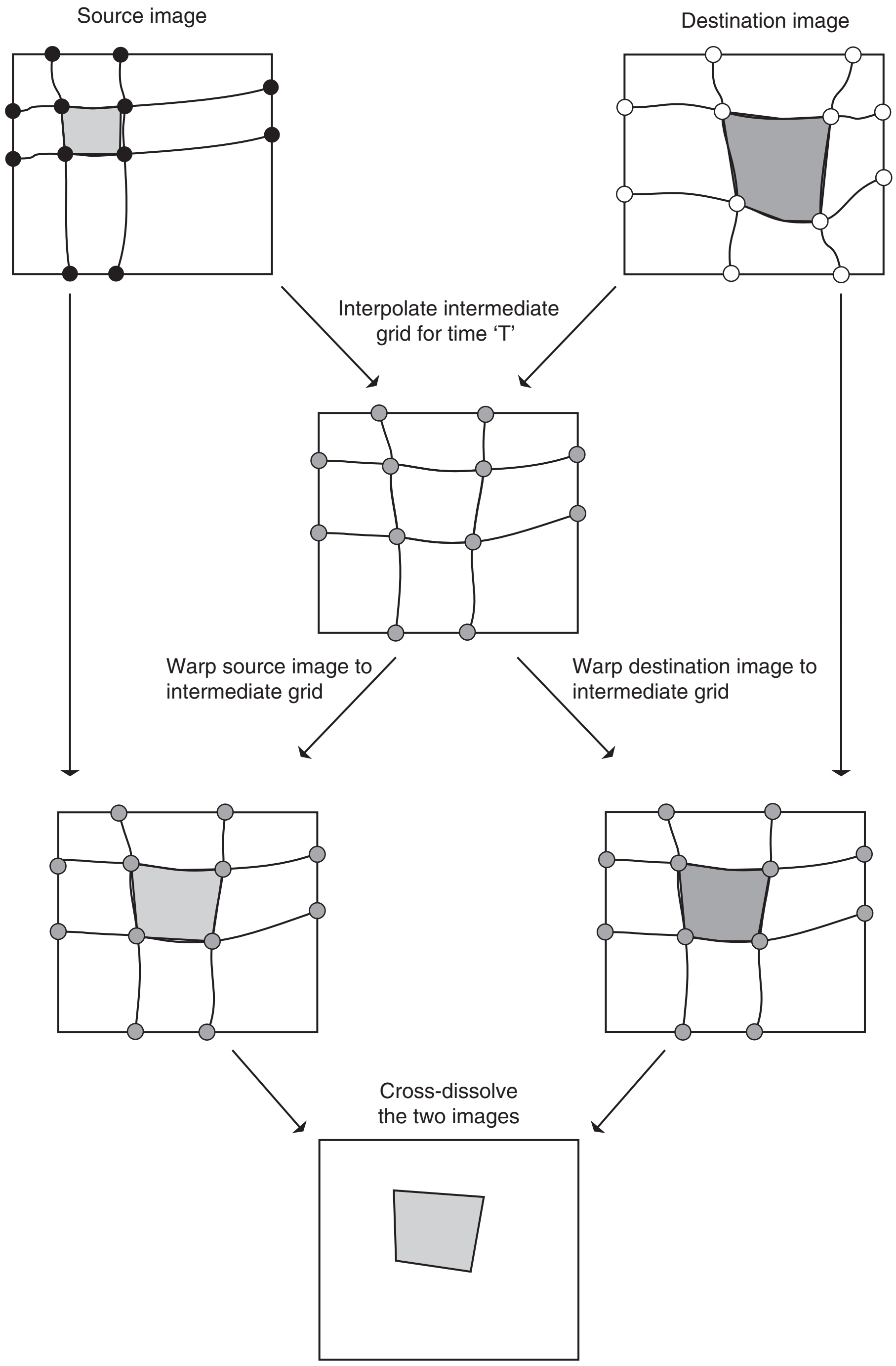

Source image
Destination image
Interpolate intermediate grid for time 'T'
Warp source image to intermediate grid
Warp destination image to intermediate grid
Cross-dissolve the two images

FIGURE 4.43

Interpolating to intermediate grid and cross-dissolve.

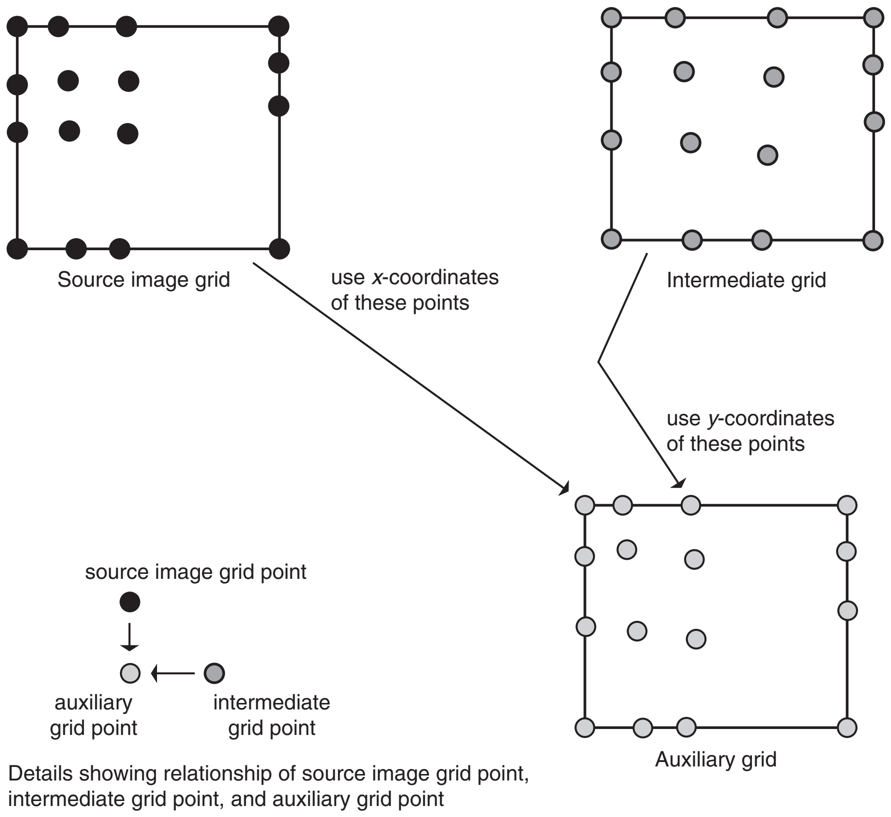

Source image grid
use *x*-coordinates of these points
Intermediate grid
use *y*-coordinates of these points
source image grid point
auxiliary grid point
intermediate grid point
Details showing relationship of source image grid point, intermediate grid point, and auxiliary grid point
Auxiliary grid

FIGURE 4.44

Formation of auxiliary grid for two-pass warping of source image to intermediate grid.

auxiliary grid, a cubic Catmull-Rom spline is defined in pixel coordinates. The leftmost and rightmost columns define straight lines down the sides of the images; this is necessary to include the entire image in the warping process. See the top of Figure 4.45. For each scanline, the x-intercepts of the curves with the scanline are computed. These define a grid coordinate system on the scanline. The position of each pixel on the scanline in the source image is determined relative to the x-intercepts by computing the Catmull-Rom spline passing through the two-dimensional space of the (grid index, x-intercept) pairs. See the middle of Figure 4.45. The integer values of $x \pm 1/2$, representing the pixel boundaries, can then be located in this space and the fractional index value recorded. In the auxiliary image, the x-intercepts of the curves with the scanline are also computed, and for the corresponding scanline, the source image pixel boundaries can be mapped into the intermediate image by using their fractional indices and locating their x-positions in the auxiliary scanline. See the bottom of Figure 4.45. Once this is complete, the color of the source image pixel can be used to color in auxiliary pixels by using fractional coverage to affect anti-aliasing.

The result of the first phase generates colored pixels of an auxiliary image by averaging source image pixel colors on a scanline-by-scanline basis. The second phase repeats the same process on a

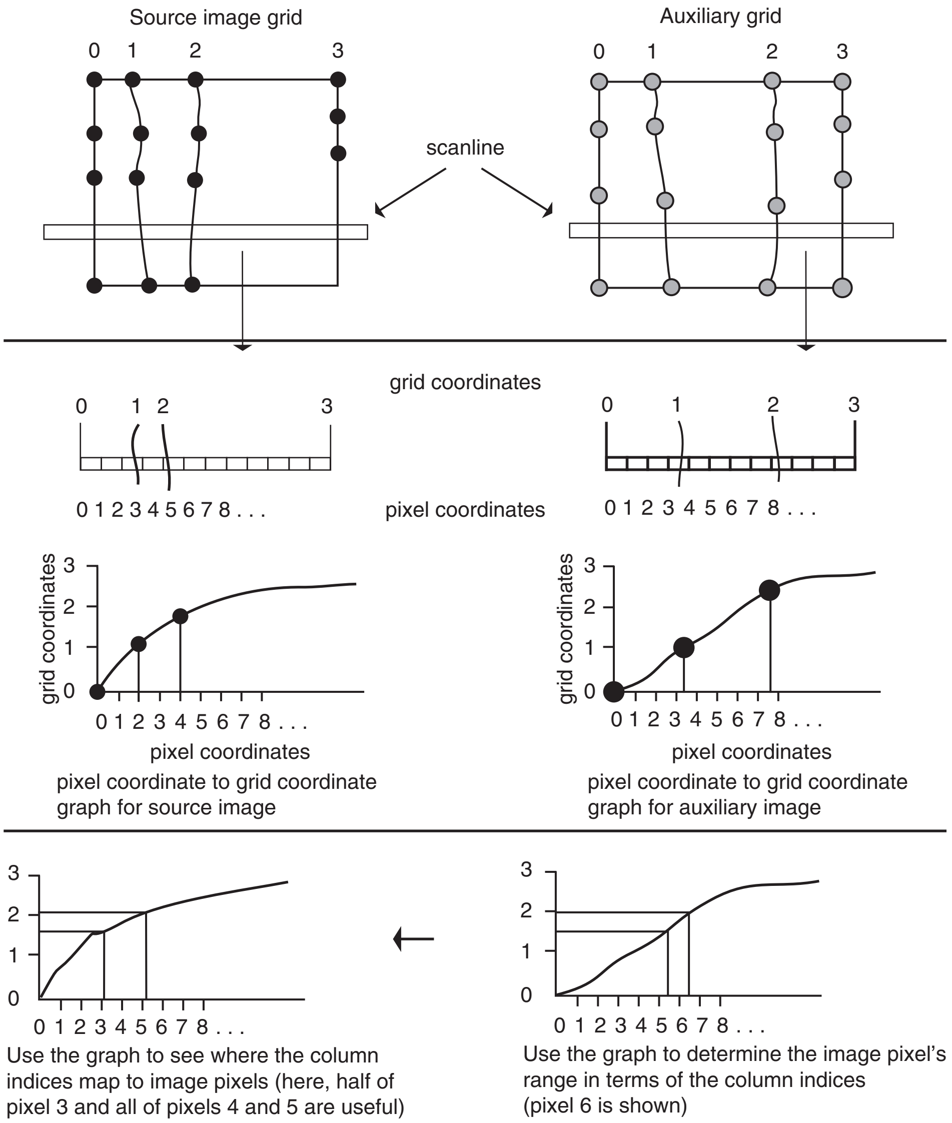

Source image grid
Auxiliary grid
0 1 2 3
scanline
grid coordinates
pixel coordinates
0 1 2 3 4 5 6 7 8 . . .
grid coordinates
pixel coordinates
pixel coordinate to grid coordinate graph for source image
pixel coordinate to grid coordinate graph for auxiliary image
Use the graph to see where the column indices map to image pixels (here, half of pixel 3 and all of pixels 4 and 5 are useful)
Use the graph to determine the image pixel's range in terms of the column indices (pixel 6 is shown)

FIGURE 4.45

For a given pixel in the auxiliary image, determine the range of pixel coordinates in the source image (e.g., pixel 6 of auxiliary grid maps to pixel coordinates 3.5 to 5 of the source image).

column-by-column basis by averaging auxiliary image pixel colors to form the intermediate image. The columns are processed by using the horizontal grid curves to establish a common coordinate system between the two images (see Figure 4.46).

This two-pass procedure is applied to both the source and the destination images with respect to the intermediate grid. Once both images have been warped to the same intermediate grid, the

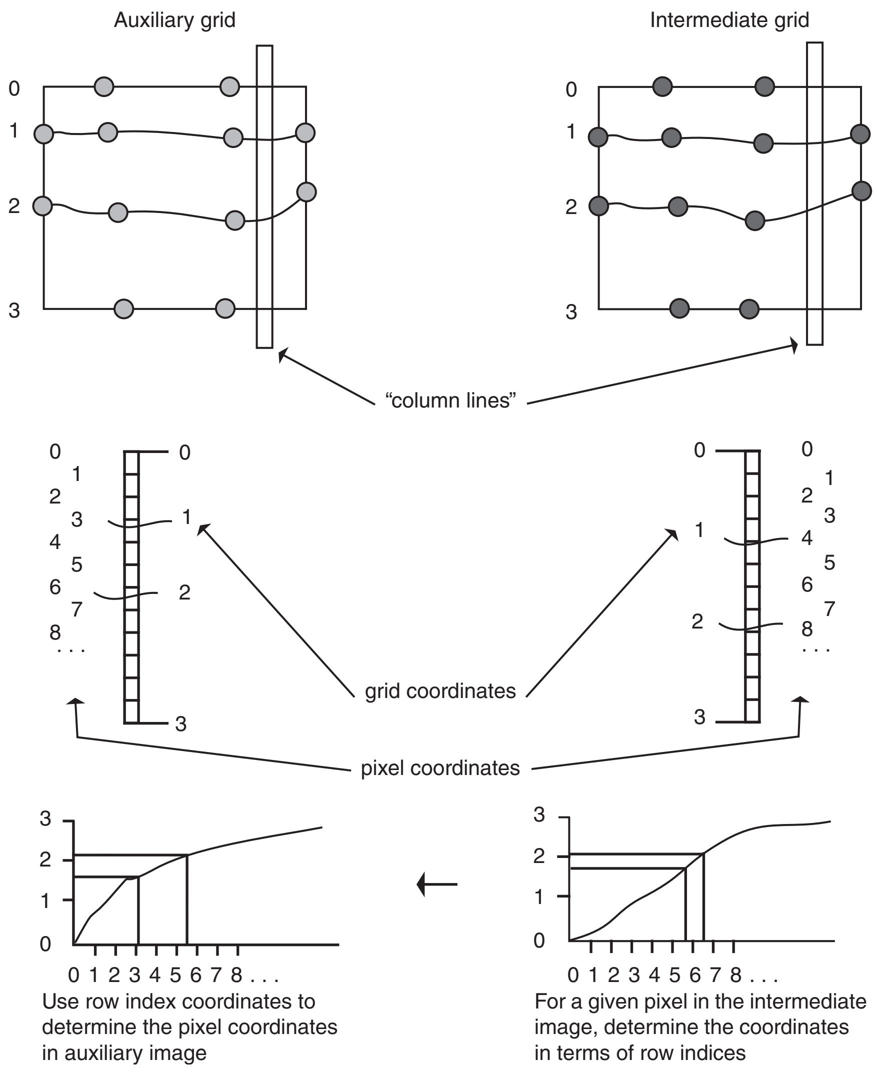

Auxiliary grid
Intermediate grid
"column lines"
grid coordinates
pixel coordinates
Use row index coordinates to determine the pixel coordinates in auxiliary image
For a given pixel in the intermediate image, determine the coordinates in terms of row indices

FIGURE 4.46

Establishing the auxiliary pixel range for a pixel of the intermediate image (e.g., pixel 6 of the intermediate grid maps to pixel coordinates 3.5 to 5 of the auxiliary image).

important features are presumably (if the user has done a good job of establishing the grids on the two images) in similar positions. At this point the images can be cross-dissolved on a pixel-by-pixel basis. The cross-dissolve is merely a blend of the two colors from corresponding pixels (Eq. 4.9).

$$C[i][j] = \alpha C_1[i][j] + (1-\alpha)C_2[i][j] \tag{4.9}$$

In the simplest case, α might merely be a linear function in terms of the current frame number and the range of frame numbers over which the morph is to take place. However, as Wolberg [37] points out, a nonlinear blend is often more visually appealing. It is also useful to be able to locally control the cross-dissolve rates based on aesthetic concerns. For example, in one of the first popular commercials to use morphing, in which a car morphs into a tiger, the front of the car is morphed into the head of the tiger at a faster rate than the tail to add to the dynamic quality of the animated morph.

Animated images are morphed by the user-defined coordinate grids for various key images in each of two animation sequences. The coordinate grids for a sequence are then interpolated over time so that at any one frame in the sequence a coordinate grid can be produced for that frame. The interpolation is carried out on the *x-y* positions of the grid intersection points; cubic interpolation such as Catmull-Rom is typically used. Once a coordinate grid has been produced for corresponding images in the animated sequences, the morphing procedure reduces to the static image case and proceeds according to the previous description (see Figure 4.47).

4.5.2 Feature-based morphing

Instead of using a coordinate grid, the user can establish the correspondence between images by using feature lines [2]. Lines are drawn on the two images to identify features that correspond to one another and feature lines are interpolated to form an intermediate feature line set. The interpolation can be based on interpolating endpoints or on interpolating center points and orientation. In either case, a mapping for each pixel in the intermediate image is established to each interpolated feature line, and a

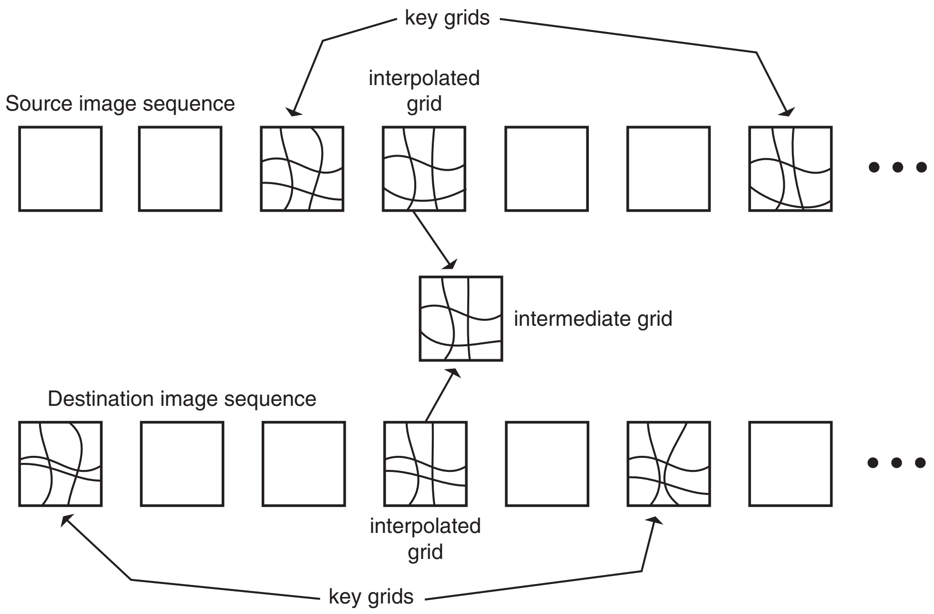


FIGURE 4.47

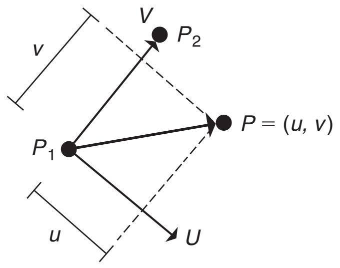
Morphing of animated sequences.

relative weight is computed that indicates the amount of influence that feature lines should have on the pixel. The mapping is used in the source image to locate the source image pixel that corresponds to the intermediate image pixel. The relative weight is used to average the source image locations generated by multiple feature lines into a final source image location. This location is used to determine the color of the intermediate image pixel. This same procedure is used on the destination image to form its intermediate image. These intermediate images are then cross-dissolved to form the final intermediate image.

Consider the mapping established by a single feature line, defined by two endpoints and oriented from P_1 to P_2. In effect, the feature line establishes a local two-dimensional coordinate system (U, V) over the image. For example, the first point of the line can be considered the origin. The second point of the line establishes the unit distance in the positive V-axis direction and scale. A line perpendicular to this line and of unit length extending to its right (as one stands on the first point and looks toward the second point) establishes the U-axis direction and scale. The coordinates (u, v) of a pixel relative to this feature line can be found by simply computing the pixel's position relative to the U- and V-axes of a local coordinate system defined by that feature line. Variable v is the projection of $(P - P_1)$ onto the direction of $(P_2 - P_1)$, normalized to the length of $(P_2 - P_1)$. u is calculated similarly (see Figure 4.48).

Assume that the points P_1 and P_2 are selected in the intermediate image and used to determine the (u, v) coordinates of a pixel. Given the corresponding feature line in the source image defined by the points Q_1 and Q_2, a similar two-dimensional coordinate system, (S, T), is established. Using the intermediate pixel's u-, v-coordinates relative to the feature line, one can compute its corresponding location in the source image (Figure 4.49).

To transform an image by a single feature line, each pixel of the intermediate image is mapped back to a source image position according to the equations above. The colors of the source image pixels in the neighborhood of that position are then used to color in the pixel of the intermediate image (see Figure 4.50).

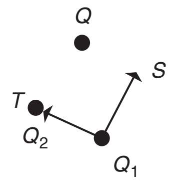


$$v = (P - P_1) \cdot \frac{(P_2 - P_1)}{|P_2 - P_1|^2}$$

$$u = \left| (P - P_1) \times \frac{(P_2 - P_1)}{|P_2 - P_1|^2} \right|$$

FIGURE 4.48

Local coordinate system of a feature in the intermediate image.



$$T = Q_2 - Q_1$$
$$S = \left(T_y - T_x\right)$$
$$Q = Q_1 + uS + vT$$

FIGURE 4.49

Relocating a point's position using local coordinates in the source image.

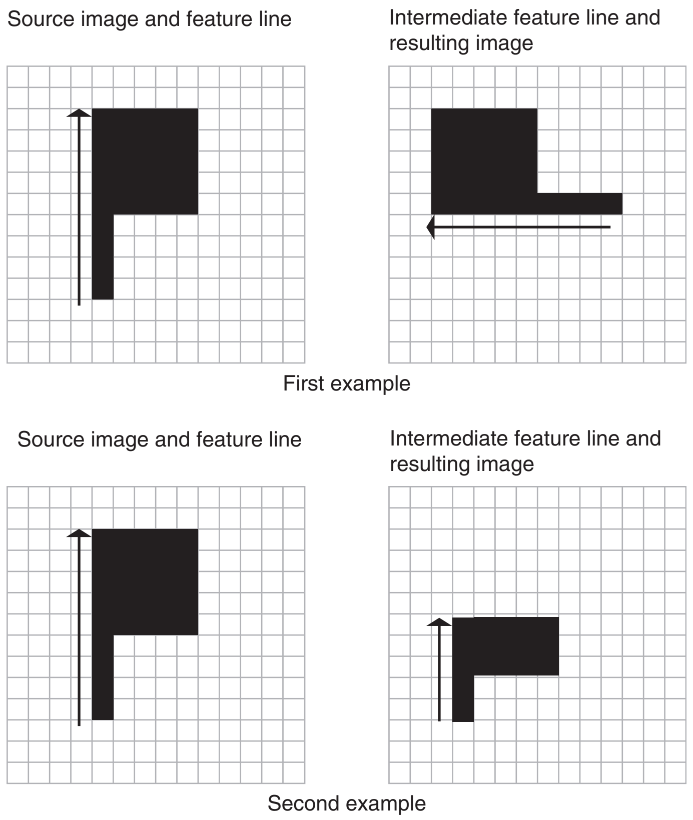

Source image and feature line
Intermediate feature line and resulting image
First example
Source image and feature line
Intermediate feature line and resulting image
Second example

FIGURE 4.50

Two examples of single-feature line morphing.

Of course, the mapping does not typically transform an intermediate image pixel back to the center of a source image pixel. The floating point coordinates of the source image location could be rounded to the nearest pixel coordinates, which would introduce aliasing artifacts. To avoid such artifacts in the intermediate image, the corner points of the intermediate pixel could be mapped back to the source image, which would produce a quadrilateral area in the source image; the pixels, wholly or partially contained in this quadrilateral area, would contribute to the color of the destination pixel.

The mapping described so far is an affine transformation. For image pairs to be really interesting and useful, multiple line pairs must be used to establish correspondences between multiple features in the images. For a pair of images with multiple feature lines, each feature line pair produces a displacement vector from an intermediate image pixel to its source image position. Associated with this displacement is a weight based on the pixel's position relative to the feature line in the intermediate image. The weight presented by Beier and Neely [2] is shown in Equation 4.10.

$$\mathrm{W} = \left(\frac{|Q_2 - Q_1|^{\mathrm{p}}}{a + dist}\right)^b \tag{4.10}$$

The line is defined by points Q_1 and Q_2, and dist is the distance that the pixel is from the line. The distance is measured from the finite line segment defined by Q_1 and Q_2 so that if the perpendicular projection of P onto the infinite line defined by Q_1 and Q_2 falls beyond the finite line segment, then the distance is taken to be the distance to the closer of the two endpoints. Otherwise, the distance is the perpendicular distance to the finite line segment. User-supplied parameters (a, b, p in Eq. 4.10) control the overall character of the mapping. As dist increases, w decreases but never goes to zero; as a practical matter, a lower limit could be set below which w is clamped to 0 and the feature line's effect on the point is ignored above a certain distance. If a is nearly 0, then pixels on the line are rigidly transformed with the line. Increasing a makes the effect of lines over the image smoother. Increasing p increases the effect of longer lines. Increasing b makes the effect of a line fall off more rapidly. As presented here, these parameters are global; for more precise control these parameters could be set on a feature-line-by-feature-line basis. For a given pixel in the intermediate image, the displacement indicated by each feature line pair is scaled by its weight. The weights and the weighted displacements are accumulated. The final accumulated displacement is then divided by the accumulated weights. This gives the displacement from the intermediate pixel to its corresponding position in the source image. See the code segment in Figure 4.51.

When morphing between two images, the feature lines are interpolated over some number of frames. For any one of the intermediate frames, the feature line induces a mapping back to the source image and forward to the destination image. The corresponding pixels from both images are identified and their colors blended to produce a pixel of the intermediate frame. In this way, feature-based morphing produces a sequence of images that transform from the source image to the destination image.

The transformations implied by the feature lines are fairly intuitive, but some care must be taken in defining transformations with multiple line pairs. Pixels that lie on a feature line are mapped onto that feature line in another image. If feature lines cross in one image, pixels at the intersection of the feature lines are mapped to both feature lines in the other image. This situation essentially tries to pull apart the image and can produce unwanted results. Also, some configurations of feature lines can produce nonintuitive results. Other techniques in the literature (e.g., [21]) have suggested algorithms to alleviate these shortcomings.

```
//====================================================================

//XY structure
typedef struct xy_struct {
   float x,y;
}xy_td;


//FEATURE
// line in image1: p1,p2;
// line in image2: q1,q2
// weights used in mapping: a,b,p
// length of line in image2
typedef struct feature_struct {
   xy_td p1,p2,q1,q2;
   float a,b,p;
   float plength,qlength;
}feature_td;


//FEATURE LIST
typedef struct featureList_struct {
   int num;
   feature_td *features;
}featureList_td;

//□---------------------------------------------------------------
//□□□MORPH
//□---------------------------------------------------------------
void morph(featureList_td *featureList)
{
   Float a,b,p,plength,qlength;
   xy_td p1,p2,q1,q2;
   xy_td vp,wp,vq,wq,v,qq;
   int   ii,jj,indexI,indexD;
   float idisp,jdisp;
   float t,s,vx,vy;
   float weight;
   char  background[3];

   background[0] = background[1] = background[2] = 120;
   for (int i=0; i<HEIGHT; i++) {
      for (int j=0; j<WIDTH; j++) {
         weight = 0;
         idsp = jdsp = 0.0;
         for (int k=0; k<featureList->num; k++) {
            // get info about kth feature line
            a = featureList->features[k].a;
            b = featureList->features[k].b;
            p = featureList->features[k].p;
            p1 = featureList->features[k].p1;
            p2 = featureList->features[k].p2;
            q1 = featureList->features[k].q1;
            q2 = featureList->features[k].q2;
            plength = featureList->features[k].plength;
            qlength = featureList->features[k].qlength;
```

FIGURE 4.51

Code using feature lines to morph from source to destination image.

Continued

```
                // get local feature coordinate system in image1
                vp.x = p2.x-p1.x;
                vp.y = p2.y-p1.y;
                wp.x = vp.y;
                wp.y = -vp.x;
                // compute local coordinates of pixel (i,j)
                v.x = j-q1.x;
                v.y = i-q1.y;
                s = (v.x*vp.x + v.y*vp.y)/plength;
                t = (v.x*wp.y - v.y*wp.x)/plength;
                // map the pixel to the source image
                jj = (int)(q1.x + s*v1.x + t*wp.x);
                ii = (int)(q1.y + s*v1.y + t*wp.y);
                // compute distance of pixel (Iii,jj) from line segment q1q2
                If (s<0.0) {
                      v.x = jj - q2.x;
                      v.y = ii - w2.y;
                      t = sqrt(v.x*v.x + v.y*v.y);
                }
                else t = fabs(t*qlength);
                t = pow(pow(qlength,p)/(a+t,b);
                jdisp + = (jj-j)*t;
                idisp + = (ii-i)*t;
                weight + = t;
             }
             jdisp /= weight;
             idisp /= weight;
             ii = (int)(i+idisp);
             jj = (int)(j+jdisp);
             indexI = (WIDTH*i+j)*3;
             if ((ii<0) || (ii>=HEIGHT) || (jj<0) || (jj>=WIDTH) ) {
                image2[indexD] = background[0];
                image2[indexD+1] = background[1];
                image2[indexD+2] = background[2];
             }
             else {
                indexS = (WIDTH*ii+jj)*3;
                imageI[indexI] = image1[indexS];
                imageI[indexI+1] = image1[indexS+1];
                imageI[indexI+2] = image1[indexS+2];
             }
          }
       }

    }
```

FIGURE 4.51—Cont'd

4.6 Chapter summary

Interpolation-based techniques form the foundation for much of computer animation. Whether interpolating images, object shapes, or parameter values, interpolation is both powerful and flexible. While this chapter has often discussed techniques in terms of linear interpolation, the reader should keep in mind that higher level techniques, such as piecewise cubic interpolation, can be much more expressive and relatively easy to implement.

References

[1] Barr A. Global and Local Deformations of Solid Primitives. In: Computer Graphics. Proceedings of SIGGRAPH 84, vol. 18(3). Minneapolis, Minn.; July 1984. p. 21–30.

[2] Beier T, Neely S. Feature Based Image Metamorphosis. In: Catmull EE, editor. Computer Graphics. Proceedings of SIGGRAPH 92, vol. 26(2). Chicago, Ill.; July 1992. p. 253–4. ISBN 0-201-51585-7.

[3] Burtnyk N, Wein M. Computer Generated Key Frame Animation. Journal of the Society of Motion Picture and Television Engineers 1971;8(3):149–53.

[4] Burtnyk N, Wein M. Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Frame Animation. Communications of the ACM October 1976;19(10):564–9.

[5] Catmull E. The Problems of Computer-Assisted Animation. In: Computer Graphics. Proceedings of SIGGRAPH 78, vol. 12(3). Atlanta, Ga.; August 1978. p. 348–53.

[6] Chadwick J. Bragger. Columbus: Ohio State University; 1989.

[7] Chadwick J. City Savvy. Columbus: Ohio State University; 1988.

[8] Chadwick J, Haumann D, Parent R. Layered Construction for Deformable Animated Characters. In: Computer Graphics. Proceedings of SIGGRAPH 89, vol. 23(3). Boston, Mass.; August 1989. p. 243–52.

[9] Chen E, Parent R. Shape Averaging and Its Application to Industrial Design. IEEE Comput Graph Appl January 1989;9(1):47–54.

[10] Christiansen HN, Sederberg TW. Conversion of Complex Contour Line Definitions into Polygonal Element Mosaics. In: Computer Graphics. Proceedings of SIGGRAPH 78, vol. 12(3). Atlanta, Ga.; August 1978. p. 187–912.

[11] Coquillart S. Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling. In: Baskett F, editor. Computer Graphics. Proceedings of SIGGRAPH 90, vol. 24(4) Dallas, Tex.; August 1990. p. 187–96. ISBN 0-201-50933-4.

[12] Coquillart S, Jancene P. Animated Free-Form Deformation: An Interactive Animation Technique. In: Sederberg TW, editor. Computer Graphics. Proceedings of SIGGRAPH 91, vol. 25(4). Las Vegas, Nev.; July 1991. p. 41–50. ISBN 0-201-56291-X.

[13] Defanti T. The Digital Component of the Circle Graphics Habitat. In: Proceedings of the National Computer Conference 76, vol. 45. New York; June 7–10. 1976. p. 195–203 AFIPS Press, Montvale, New Jersey.

[14] Ganapathy S, Denny TG. A New General Triangulation Method for Planar Contours. In: Computer Graphics. Proceedings of SIGGRAPH 82, vol. 16(3). Boston, Mass.; July 1982. p. 69–75.

[15] Gomez J. Twixt: A 3D Animation System. Computer and Graphics 1985;9(3):291–8.

[16] Hackathorn R. ANIMA II: A 3D Color Animation System. In: George J, editor. Computer Graphics. Proceedings of SIGGRAPH 77, vol. 11(2). San Jose, Calif.11.; July 1977. p. 54–64.

[17] Hackathorn R, Parent R, Marshall B, Howard M. An Interactive Micro-Computer Based 3D Animation System. In: Proceedings of Canadian Man-Computer Communications Society Conference 81. Waterloo, Ontario; June 10–12, 1981. p. 181–91.

[18] Hofer E. A Sculpting Based Solution for Three-Dimensional Computer Character Facial Animation. Master's thesis. Ohio State University; 1993.

[19] Kent J, Carlson W, Parent R. Shape Transformation for Polyhedral Objects. In: Catmull EE, editor. Computer Graphics. Proceedings of SIGGRAPH 92, vol. 26(2). Chicago, Ill.; July 1992. p. 47–54. ISBN 0-201-51585-7.

[20] Litwinowicz P. Inkwell: A $2^1/_2$-D Animation System. In: Sederberg TW, editor. Computer Graphics. Proceedings of SIGGRAPH 91, vol. 25(4). Las Vegas, Nev.; July 1991. p. 113–22.

[21] Litwinowicz P, Williams L. Animating Images with Drawings. In: Glassner A, editor. Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series. Orlando, Fla.: ACM Press; July 1994. p. 409–12. ISBN 0-89791-667-0.

[22] Magnenat-Thalmann N, Thalmann D. Computer Animation: Theory and Practice. New York: Springer-Verlag; 1985.

[23] Magnenat-Thalmann N, Thalmann D, Fortin M. Miranim: An Extensible Director-Oriented System for the Animation of Realistic Images. IEEE Comput Graph Appl March 1985;5(3):61–73.

[24] May S. Encapsulated Models: Procedural Representations for Computer Animation. Ph.D. dissertation. Ohio State University; 1998.

[25] May S. AL: The Animation Language. http://accad.osu.edu/~smay/AL/scheme.html**; July 2005.**

[26] Mezei L, Zivian A. ARTA: An Interactive Animation System. In: Proceedings of Information Processing 71. Amsterdam: North-Holland; 1971. p. 429–34.

[27] Parent R. Shape Transformation by Boundary Representation Interpolation: A Recursive Approach to Establishing Face Correspondences, OSU Technical Report OSU-CISRC-2–91-TR7. Journal of Visualization and Computer Animation January 1992;3:219–39.

[28] Parent R. A System for Generating Three-Dimensional Data for Computer Graphics. Ph.D. dissertation. Ohio State University; 1977.

[29] Reeves W. In-betweening for Computer Animation Utilizing Moving Point Constraints. In: Computer Graphics. Proceedings of SIGGRAPH 81, vol. 15(3). Dallas, Tex.; August 1981. p. 263–70.

[30] Reynolds C. Computer Animation with Scripts and Actors. In: Computer Graphics. Proceedings of SIGGRAPH 82, vol. 16(3). Boston, Mass.; July 1982. p. 289–96.

[31] Sederberg T. Free-Form Deformation of Solid Geometric Models. In: Evans DC, Athay RJ, editors. Computer Graphics. Proceedings of SIGGRAPH 86, vol. 20(4). Dallas, Tex.; August 1986. p. 151–60.

[32] Houdini 2.0: User Guide. Toronto: Side Effects Software; September 1997.

[33] Stern G. BBOP—A program for 3-dimensional Animation. In: Nicograph 83 proceedings. Tokyo, Dec '83. p. 403–404.

[34] Talbot P, Carr III J, Coulter Jr R, Hwang R. Animator: An On-line Two-Dimensional Film Animation System. Communications of the ACM 1971;14(4):251–9.

[35] Weiler K. Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments. IEEE Comput Graph Appl January 1985;5(1):21–40.

[36] Wilkins M, Kazmier C. MEL Scripting for Maya Animators. San Francisco, California: Morgan Kaufmann; 2003.

[37] Wolberg G. Digital Image Warping. Los Alamitos, California: IEEE Computer Society Press; 1988.