APPENDIX

Rendering Issues

A

This appendix presents rendering techniques for computing a series of images that are to be played back as an animation sequence. It is assumed that the reader has a solid background in rendering techniques and issues, namely, the use of frame buffers, the *z*-buffer display algorithm, and aliasing. The techniques presented here concern smoothly displaying a sequence of images on a computer monitor (*double buffering*), efficiently computing images of an animated sequence (*compositing, drop shadows, billboarding*), and effectively rendering moving objects (*motion blur*). An understanding of this material is not necessary for understanding the techniques and algorithms covered in the rest of the book, but computer animators should be familiar with these techniques when considering the trade-offs involved in rendering images for animation.

A.1 Double buffering

Not all computer animation is first recorded onto film or video for later viewing. In many cases, image sequences are displayed in real time on the computer monitor. Computer games are a prime example of this, as is Web animation. Real-time display also occurs in simulators and for previewing animation for later high-quality recording. In some of these cases, the motion is computed and images are rendered as the display is updated; sometimes pre-calculated images are read from the disk and loaded into the frame buffer. In either case, the time it takes to paint an image on a computer screen can be significant (for various reasons). To avoid waiting for the image to update, animators often paint the new image into an off-screen buffer. Then a quick operation is performed to change the off-screen buffer to on-screen (often with hardware display support); the previous on-screen buffer becomes off-screen. This is called *double buffering*.

In double buffering, two (or more) buffers are used. One buffer is used to refresh the computer display, while another is used to assemble the next image. When the next image is complete, the two buffers switch roles. The second buffer is used to refresh the display, while the first buffer is used to assemble the next image. The buffers continue exchanging roles to ensure that the buffer used for display is not actively involved in the update operations (see Figure A.1).

Double buffering is often supported by the hardware of a computer graphics system. For example, the display system may have two built-in frame buffers, both of which are accessible to the user's program and are program selectable for driving the display. Alternatively, the graphics system may be designed so that the screen can be refreshed from an arbitrary section of main memory and the buffers are identified by pointers into the memory. Double buffering is also effective when

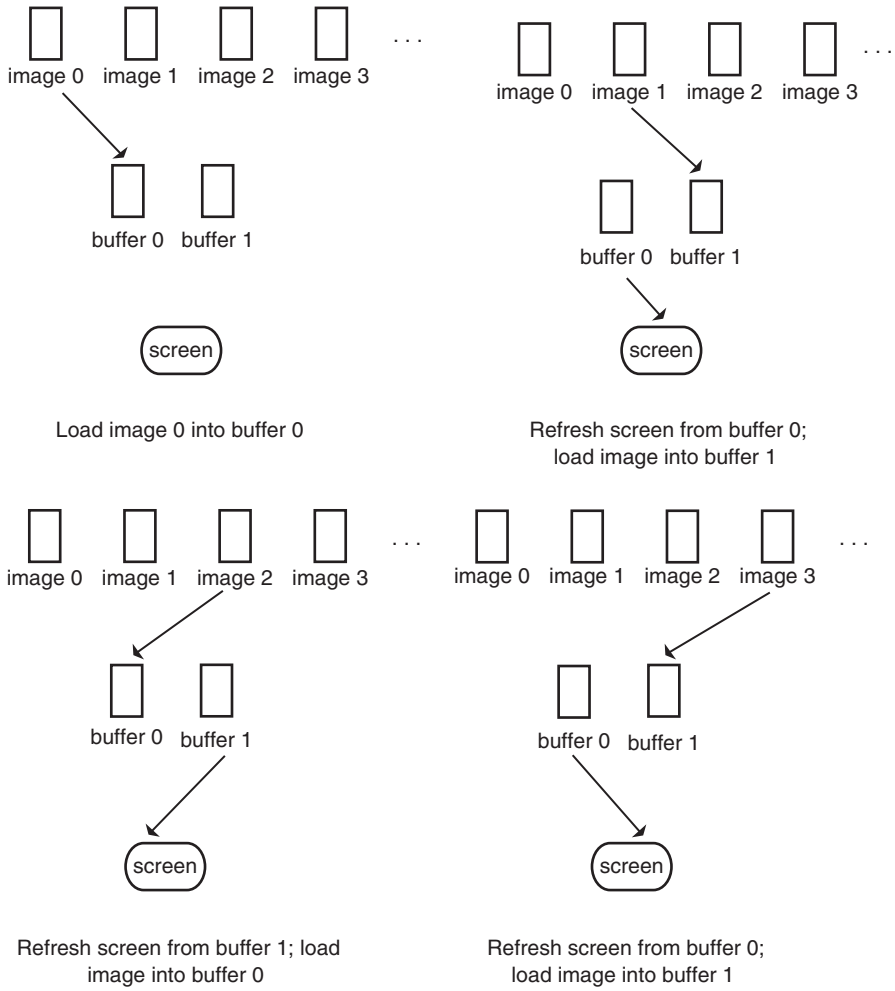

FIGURE A.1

Double buffering.

implemented completely in software, as is popular in JAVA applets. Pseudocode of simple double buffering using two frame buffers is shown in Figure A.2.

A.2 Compositing

Compositing is the act of combining separate image layers into a single picture. It allows the animator, or compositor, to combine elements obtained from different sources to form a single image. It allows artistic decisions to be deferred until all elements are brought together. Individual elements can be

```
open_window (w)
i=0;                // start using buffer 0
j=0;                // start with image 0
do{
 load_image(buffer[i],image[j]);/*load the jth image into the ith frame
                                  buffer*/
 display_in_window(w,buffer[i]);//display the ith buffer
 i=1-i;             // swap which buffer to load image into
 j=j+1;              // advance to the next image
}until (done);
```

FIGURE A.2

Double buffering pseudocode.

selectively manipulated without the user having to regenerate the scene as a whole. As frames of computer animation become more complex, the technique allows layers to be calculated separately and then composited after each has been finalized. Compositing provides great flexibility and many advantages to the animation process. Before digital imagery, compositing was performed optically in much the same way that a multi-plane camera is used in conventional animation. With digital technology, compositing operates on digital image representations.

One common and extremely important use of compositing is that of computing the foreground animation separately from the static background image. If the background remains static it need only be computed and rendered once. Then, as each frame of the foreground image is computed, it is composited with the background to form the final image. The foreground can be further segmented into multiple layers so that each object, or even each object part, has its own layer. This technique allows the user to modify just one of the object parts, re-render it, and then recompose it with the previously rendered layers, thus potentially saving a great deal of rendering time and expense. This is very similar to the two-and-a-half dimensional approach taken in conventional hand-drawn cell animation.

Compositing offers a more interesting advantage when digital frame buffers are used to store depth information in the form of z-values, along with the color information at each pixel. To a certain extent, this approach allows the animator to composite layers that actually interleave each other in depth. The initial discussion focuses on compositing without pixel depth information, which effectively mimics the approach of the multi-plane camera. Following this, compositing that takes the z-values into account is explored.

A.2.1 Compositing without pixel depth information

Digital compositing attempts to combine multiple two-dimensional images of three-dimensional scenes so as to approximate the visibility of the combined scenes. Compositing will combine two images at a time with the desire of maintaining the equilibrium shown in Equation A.1.

$$composite(render(scene1),\ render(scene2)) = render(merge(scene1, scene2)) \qquad \text{(A.1)}$$

The *composite* operation on the left side of the equation refers to compositing the two rendered images; the *merge* operation on the right side of the equation refers to combining the geometries of the two scenes into one. The *render* function represents the process of rendering an image based on the input scene geometries. In the case in which the composite operator is used, the render function must tag pixels not covered by the scene as transparent. The equality will hold if the scenes are disjoint in depth from the observer and the composite operator gives precedence to the image closer to the observer. The visibility between elements from the different scenes can be accurately represented in two-and-a-half dimensional compositing. The visibility between disjoint planes can be accurately resolved by assigning a single visibility priority to all elements within a single plane.

The image-based **over** operator places one image on top of the other. In order for **over** to operate, there must be some assumption or some additional information indicating which part of the closer image (also referred to as the *overlay plane* or *foreground image*) occludes the image behind it (the *background image*). In the simplest case, all of the foreground image occludes the background image. This is useful for the restricted situation in which the foreground image is smaller than the background image. In this case, the smaller foreground image is often referred to as a *sprite*. There are usually two-dimensional coordinates associated with the sprite that locate it relative to the background image (see Figure A.3).

However, for most cases, additional information in the form of an *occlusion mask* (also referred to as a *matte* or *key*) is provided along with the overlay image. A one-bit matte can be used to indicate which pixels of the foreground image should occlude the background during the compositing process (see Figure A.4). In frame buffer displays, this technique is often used to overlay text or a cursor on top of an image.

Compositing is a binary operation, combining two images into a single image. However, any number of images can be composited to form a final image. The images must be ordered by depth and are

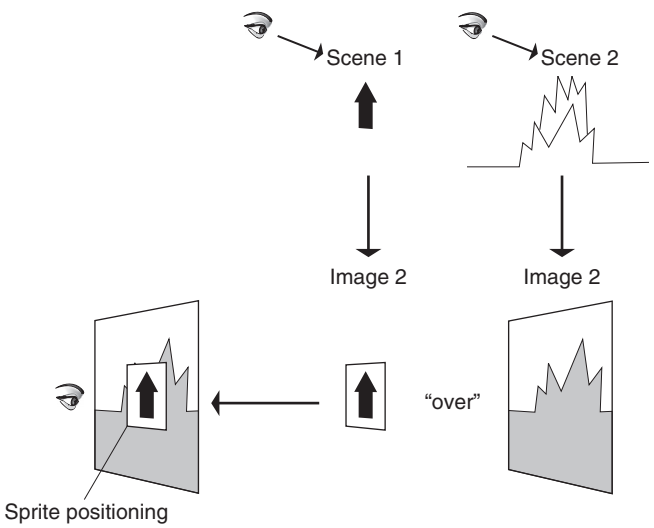

FIGURE A.3

Two-and-a-half dimensional compositing without transparency.

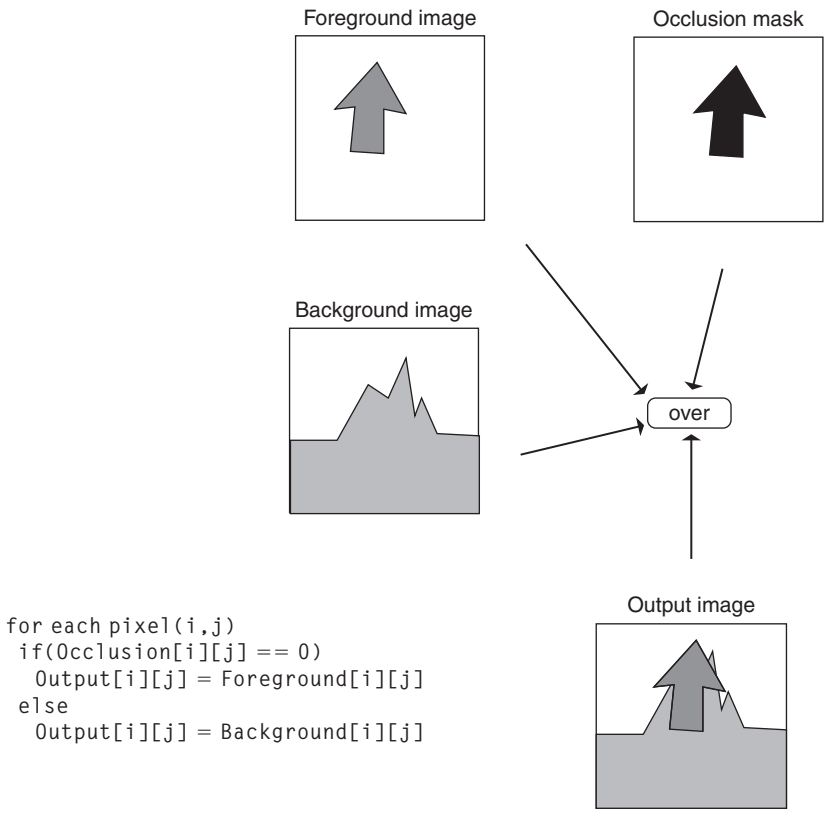

FIGURE A.4

Compositing using a one-bit occlusion mask.

operated on two at a time. Each operation replaces the two input images with the output image. Images can be composited in any order as long as the two composited during any one step are adjacent in their depth ordering.

Compositing the images using a one-bit occlusion mask, that is, using the color of a foreground image pixel in the output image, is an all-or-nothing decision. However, if the foreground image is calculated by considering semitransparent surfaces or partial pixel coverage, then fractional occlusion information must be available and anti-aliasing during the pixel-merging process must be taken into account. Instead of using a one-bit mask for an all-or-nothing decision, using more bits allows a partial decision. This gray-scale mask is called *alpha* and is commonly maintained as a fourth *alpha channel* in addition to the three (red, green, and blue) color channels. The alpha channel is used to hold an opacity factor for each pixel. Even this is a shortcut; to be more accurate, an alpha channel for each of the three colors R, G, and B is required. A typical size for the alpha channel is eight bits.

The fractional occlusion information available in the alpha channel is an approximation used in lieu of detailed knowledge about the three-dimensional geometry of the two scenes to be combined. Ideally,

in the process of determining the color of a pixel, polygons[1] from both scenes are made available to the renderer and visibility is resolved at the sub-pixel level. The combined image is anti-aliased, and a color for each pixel is generated. However, it is often either necessary or more efficient to composite the images made from different scenes. Each image is anti-aliased independently, and, for each pixel, the appropriate color and opacity values are generated in rendering the images. These pixels are then combined using the opacity values (alpha) to form the corresponding pixel in the output image. Note that all geometric relationships are lost between polygons of the two scenes once the image pixels have been generated. Figure A.5(a) shows the combination of two scenes to create complex overlapping geometry that could then be correctly rendered. Figure A.5(b), on the other hand, shows the two scenes rendered independently at which point it's impossible to insert the geometric elements of the one between the geometric elements of the other. The image-based **over** operator, as described above, must give visibility priority to one or the other partial scenes.

The pixel-based compositing operator, **over**, operates on a color value, *RGB*, and an alpha value between 0 and 1 stored at each pixel. The alpha value can be considered either the opacity of the surface that covers the pixel or the fraction of the pixel covered by an opaque surface, or a combination of the two. The alpha channel can be generated by visible surface algorithms that handle transparent surfaces and/or perform some type of anti-aliasing. The alpha value for a given image pixel represents the amount that the pixel's color contributes to the color of the output image when composited with an image behind the given image. To characterize this value as the fraction of the pixel covered by surfaces from the corresponding scene is not entirely accurate. It actually needs to be the coverage of areas contributing to the pixel color weighted by the anti-aliasing filter kernel.[2] In the case of a box filter over non-overlapping pixel areas, the alpha value equates to the fractional coverage.

To composite pixel colors based on the **over** operator, the user computes the new alpha value for the pixel: $\alpha = \alpha_F + (1 - \alpha_F)\,\alpha_B$. The composited pixel color is then computed by Equation A.2.

$$(\alpha_F RGB_F + (1 - \alpha_F)\alpha_B RGB_B)/\alpha \tag{A.2}$$

where RGB_F is the color of the foreground, RGB_B is the color of the background, a_F is the alpha channel of the foreground pixel, and a_B is the alpha channel of the background pixel. The **over** operator is not commutative but is associative (see Equation A.3).

$$F \textbf{ over } B = \begin{cases} \alpha_{F \text{ over } B} = \alpha_F + (1 - \alpha_F)\alpha_B & \text{over operator} \\ RGB_{F \text{ over } B} = (\alpha_F RGB_F + (1 - \alpha_F)\alpha_B RGB_B)/\alpha_{F \text{ over } B} & \\ A \text{ over } B \neq B \text{ over } A & \text{not commutative} \\ (\text{A over B}) \text{over C} = \text{A over}(\text{B over C}) & \text{associative} \end{cases} \tag{A.3}$$

The compositing operator, **over**, assumes that the fragments in the two input images are uncorrelated. The assumption is that the color in the images comes from randomly distributed fragments. For example, if the alpha of the foreground image is 0.5, then the color fragments of what is behind the

[1] To simplify the discussion and diagrams, one must assume that the scene geometry is defined by a collection of polygons. However, any geometric element can be accommodated provided that coverage, occlusion, color, and opacity can be determined on a subpixel basis.

[2] The *filter kernel* is the weighting function used to blend color fragments that partially cover a pixel's area.

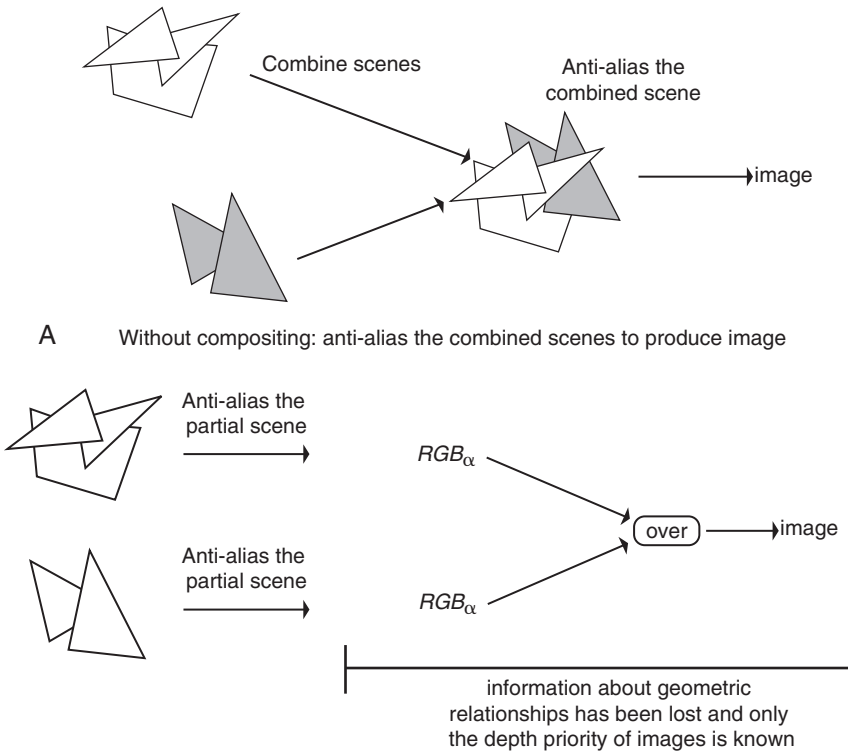

FIGURE A.5

Anti-aliasing combined scenes versus alpha channel compositing.

foreground image will, on average, show through 50 percent of the time. Consider the case of a foreground pixel and middle ground pixel, both with partial coverage in front of a background pixel with full coverage (Figure A.6).

The result of the compositing operation is shown in Equation A.4.

$$\begin{aligned} &RGB_{FMB} = (\alpha_F RGB_F + (1-\alpha_F)\alpha_M RGB_M) + (1-\alpha_{FM})\alpha_B RGB_B \\ &where \quad \alpha_{FM} = (\alpha_F + (1-\alpha_F)\alpha_M) \text{and } \alpha_{FMB} = 1.0 \\ &RGB = 0.5RGB_F + 0.25RGB_M + 0.25RGB_B \end{aligned} \tag{A.4}$$

If the color fragments are correlated, for example, if they share an edge in the image plane, then the result of the compositing operation is incorrect. The computations are the same, but the result does not accurately represent the configuration of the colors in the combined scene. In the example of Figure A.7, none of the background should show through. A similarly erroneous result occurs if the middle ground image has its color completely on the other side of the edge, in which case none of the middle ground color should appear in the composited pixel. Because geometric information has been discarded, compositing fails to handle these cases correctly.

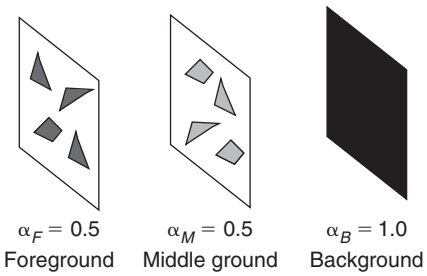

FIGURE A.6

Compositing randomly distributed color fragments for a pixel.

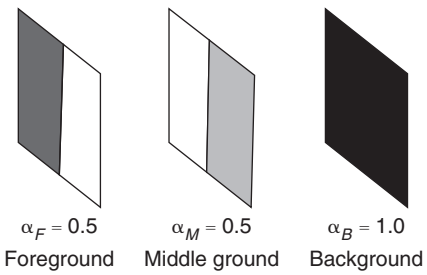

FIGURE A.7

Compositing correlated colors for a pixel.

When α_F and α_B represent full coverage opacities or uncorrelated partial coverage fractions, the **over** operator computes a valid result. However, if the alphas represent partial coverages that share an edge, then the compositing **over** operator does not have enough information to tell whether, for example, the partial coverages overlap in the pixel area or whether the areas of coverage they represent are partially or completely disjoint. Resolution of this ambiguity requires that additional information be stored at each pixel indicating which part of the pixel is covered by a surface fragment. For example, the *A*-buffer[3] algorithm [1] provides this information.

Alpha channel is a term that represents a combination of the partial coverage and the transparency of the surface or surfaces whose color is represented at the pixel. Notice that when compositing colors, a color always appears in the equation multiplied by its alpha value. It is therefore expedient to store the color value already scaled by its alpha value. In the following discussion, lowercase *rgb* refers to a color value that has already been scaled by alpha. Pixels and images whose colors have been scaled by alpha are called *premultiplied*. Uppercase *RGB* refers to a color value that has not been scaled by alpha. In a premultiplied image, the color at a pixel is considered to already be scaled down by its alpha factor, so that if a surface is white with an *RGB* value of (1, 1, 1) and it covers half a pixel as indicated by an alpha value of 0.5, then the *rgb* stored at that pixel will be (0.5, 0.5, 0.5). It is important to recognize that storing premultiplied images is very useful.

[3]The *A*-buffer is a *Z*-buffer in which information recorded at each pixel includes the relative depth and coverage of all fragments, in z -sorted order, which contribute to the pixel's final color.

A.2.2 Compositing with pixel depth information

In compositing, independently generated images may sometimes not be disjoint in depth. In such cases, it is necessary to interleave the images in the compositing process. Duff [5] presents a method for compositing three-dimensional rendered images in which depth separation between images is not assumed. An ***rgb*αz** (premultiplied) representation is used for each pixel that is simply a combination of an *rgb* value, the alpha channel, and the *z*-, or depth, value. The *z*-value associated with a pixel is the depth of the surface visible at that pixel; this value is produced by most rendering algorithms.

Binary operators are defined to operate on a pair of images f and b on a pixel-by-pixel basis to generate a resultant image (Equation A.5). Applying the operators to a sequence of images in an appropriate order will produce a final image.

$$c = f \ \mathbf{op}\ b \tag{A.5}$$

The first operator to define is the **over** operator. Here, it is defined using colors that have been premultiplied by their corresponding alpha values. The **over** operator blends together the color and alpha values of an ordered pair of images on a pixel-by-pixel basis. The first image is assumed to be "over" or "in front of" the second image. The color of the resultant image is the color of the first image plus the product of the color of the second image and the transparency (one minus opacity) of the first image. The alpha value of the resultant image is computed as the alpha value of the first image plus the product of the transparency of the first and the opacity of the second. Values stored at each pixel of the image, resulting from $c = f$ **over** b, are defined as shown in Equation A.6.

$$\begin{aligned} rgb_c &= rgb_f + (1 - \alpha_f) rgb_b \\ \alpha_c &= \alpha_f + (1 - \alpha_f)\alpha_b \end{aligned} \tag{A.6}$$

For a given foreground image with corresponding alpha values, the foreground *rgb*s will be unattenuated during compositing with the **over** operator and the background will show through more as α_f decreases. Notice that when $\alpha_f = 1$, then $rgb_c = rgb_f$ and $\alpha_c = \alpha_f = 1$; when $\alpha_f = 0$ (and therefore $rgb_f = 0, 0, 0$), then $rgb_c = rgb_b$ and $\alpha_c = \alpha_b$. Using **over** with more than two layers requires that their ordering in z be taken into account when compositing. The **over** operator can be successfully used when compositing planes adjacent in z. If non-adjacent planes are composited, a plane lying between these two cannot be accurately composited; the opacity of the closest surface is not separately represented in the composited image. **Over** is not commutative, although it is associative.

The second operator to define is the *z*-depth operator, **zmin**, which operates on the *rgb,* alpha, and *z*-values stored at each pixel. The **zmin** operator simply selects the ***rgb*αz** values of the closer pixel (the one with the minimum z). Values stored at each pixel of the image resulting from $c = f$ **zmin** b are defined by Equation A.7.

$$\begin{aligned} rgb\alpha_c &= \text{if}(z_f < z_b) \quad \text{then}(rgb\alpha_f) \quad \text{else}(rgb\alpha_b) \\ z_c &= \min(z_f, z_b) \end{aligned} \tag{A.7}$$

The order in which the surfaces are processed by **zmin** is irrelevant; it is commutative and associative and can be successfully used on non-adjacent layers.

Comp is an operator that combines the action of **zmin** and **over**. As before, each pixel contains an *rgb* value and an α value. However, for an estimate of relative coverage, each pixel has *z*-values at each of its four corners. Because each *z*-value is shared by four pixels, the upper left *z*-value can be stored at

each pixel location. This requires that an extra row and extra column of pixels be kept in order to provide the z-values for the pixels in the rightmost row and bottommost column of the image; the rgb and α values for these pixels in the extra row and column are never used.

To compute $c = f$ **comp** b at a pixel, one must first compute the z-values at the corners to see which is larger. There are $2^4 = 16$ possible outcomes of the four corner comparisons. If the comparisons are not the same at all four corners, the pixel is referred to as *confused*. This means that within this single pixel, the layers cross each other in z. For any edge of the pixel whose endpoints compare differently, the z-values are interpolated to estimate where along the edge the surfaces actually meet in z. Figure A.8 illustrates the implied division of a pixel into areas of coverage based on the relative z-values at the corners of the foreground and background pixels.

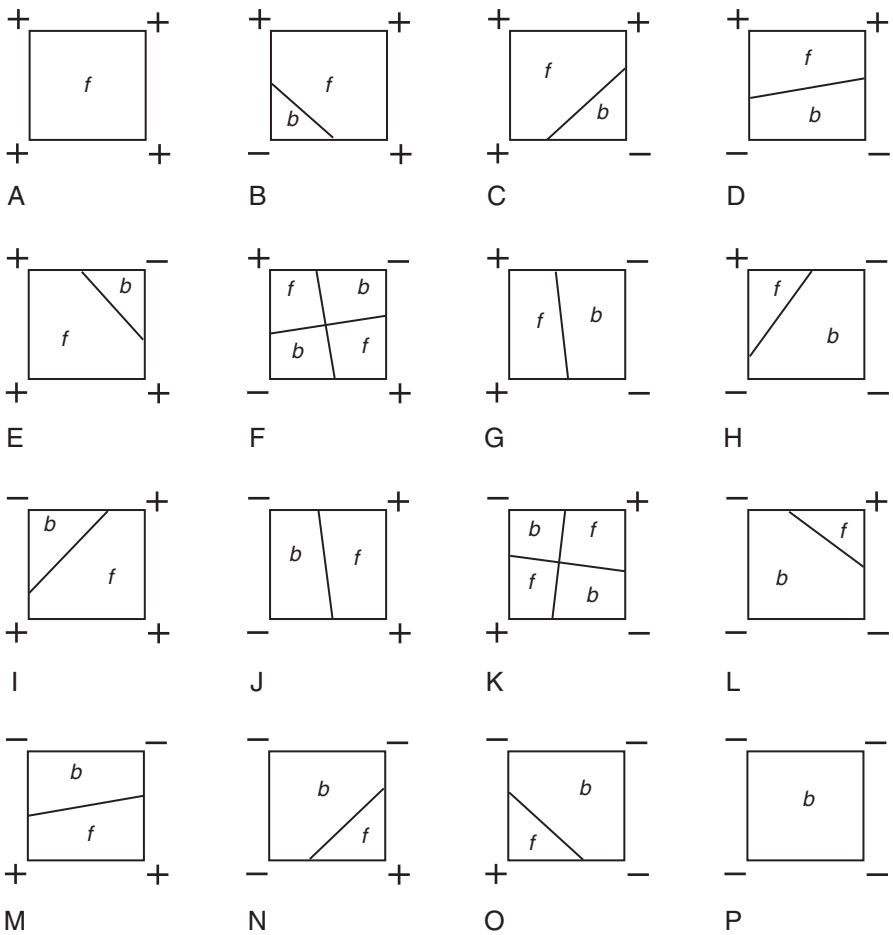

FIGURE A.8

Categories of pixels based on z comparisons at the corners; the label at each corner is sign($z_\mathbf{f}$ - $z_\mathbf{b}$).

Considering symmetry, there are really only four cases to consider in computing β, the coverage fraction for the surface *f:* whole, corner, split, and two-opposite corners. *Whole* refers to the simple cases in which the entire pixel is covered by one surface (cases a and p in Figure A.8); in this case, β equals either 1 or 0. *Corner* refers to the cases in which one surface covers the entire pixel except for one corner (cases b, c, e, h, i, l, n, and o in Figure A.8). If f is the corner surface, then the coverage is $\beta = \frac{s \times t}{2}$, where s and t represent the fraction of the edge indicated by the z-value interpolation as measured from the corner vertex. If b is the corner surface, then $\beta = 1.0 - \frac{s \times t}{2}$. *Split* refers to the situation in which the vertices of opposite edges are tagged differently (cases d, g, j, and m in Figure A.8); the coverage of the surface is $\beta = \frac{s + t}{2}$, where s and t represent the fraction of the edge indicated by the z-value interpolation as measured from the vertices tagged "1" toward the other vertices. The fraction for the other surface would be $\beta = 1.0 - \frac{s + t}{2}$. If each vertex is tagged the same as that diagonally opposite of it, but the diagonally opposite pairs are tagged differently (cases f and k in Figure A.8), the equation for β can be approximated by computing the fractional distance along each pixel edge according to the z-value interpolation from the corners—for opposite pairs, use the same direction to determine the fraction. For the opposite pairs of edges of the pixel, average the two fractional distances. Use these distances as an axis-aligned subdivision of the pixel into four quadrants and compute the areas of f and b.

Once β is computed, then that fraction of the pixel is considered as having the surface f as the front surface and the rest of the pixel is considered as having the surface b as the front surface. The **comp** operator is defined as the linear blend, according to the interpolant β, of two applications of the **over** operator—one with surface f in front and one with surface b in front (Equation A.8).

$$\begin{aligned} rgb_c &= \beta(rgb_f + (1 - \alpha_f)rgb_b) + (1 - \beta)(rgb_b + (1 - \alpha_b)rgb_f) \\ \alpha_c &= \beta(\alpha_f + (1 - \alpha_f)\alpha_b) + (1 - \beta)(\alpha_b + (1 - \alpha_b)\alpha_f) \\ z_c &= \min(z_f, z_b) \end{aligned} \tag{A.8}$$

Comp decides on a pixel-by-pixel basis which image represents a surface in front of the other, including the situation in which the surfaces change relative depth within a single pixel. Thus, images that are not disjoint in depth can be successfully composited using the **comp** operator, which is commutative (with β becoming (**1** – β)) but not associative. See Duff's paper [5] for a discussion.

A.3 Displaying moving objects: motion blur

If it is assumed that objects are moving in time, it is worth addressing the issue of effectively displaying these objects in a frame of animation [6] [8]. In the same way that aliasing is a sampling issue in the spatial domain, it is also a sampling issue in the temporal domain. As frames of animation are calculated, the positions of objects change in the image, and this movement changes the color of a pixel as a function of time. In an animation, the color of a pixel is sampled in the time domain. If the temporal frequency of a pixel's color function is too high, then the temporal sampling can miss important information.

Consider an object moving back and forth in space in front of an observer. Assume the animation is calculated at the rate of thirty frames per second. Now assume that the object starts on the left side of the

screen and moves to the right side in one-sixtieth of a second and moves back to the left side in another one-sixtieth of a second. This means that every thirtieth of a second (the rate at which frames are calculated and therefore the rate at which positions of objects are sampled) the object is on the left side of the screen. The entire motion of the object is missed because the sampling rate is too low to capture the high-frequency motion of the object, resulting in temporal aliasing. Even if the motion is not this contrived, displaying a rapidly moving object by a single instantaneous sample can result in motions that appear jerky and unnatural. As mentioned in Chapter 1, Section 1.1, images of a fast-moving object can appear to be disjointed, resulting in jerky motion similar to that of live action under a strobe light (and this is often called *strobing*).

Conventional animation has developed its own techniques for representing fast motion. Speed lines can be added to moving objects, objects can be stretched in the direction of travel, or both speed lines and object stretching can be used [9] (see Figure A.9).

There is an upper limit on the amount of detail the human eye can resolve when viewing a moving object. If the object moves too fast, mechanical limitations of muscles and joints that control head and eye movement will fail to maintain an accurate track. This will result in an integration of various samples from the environment as the eye tries to keep up. If the eye is not tracking the object and the object moves across the field of view, receptors will again receive various samples from the environment integrated together, forming a cumulative effect in the eye-brain. Similarly, a movie camera will open its shutter for an interval of time, and an object moving across the field of view will create a blurred image on that frame of the film. This will smooth out the apparent motion of the object. In much the same way, the synthetic camera can (and should) consider a frame to be an interval of time instead of an instance in time. Unfortunately, accurately calculating the effect of moving objects in an image requires a nontrivial amount of computation.

To fully consider the effect that moving objects have on an image pixel, one must take into account the area of the image represented by the pixel, the time interval for which a given object is visible in that pixel, the area of the pixel in which the object is visible, and the color variation of the object over that time interval in that area, as well as such dynamic effects as rotating textured objects, shadows, and specular highlights [8].

There are two analytic approaches to calculating motion blur: continuous and discrete. Continuous approaches attempt to be more accurate but are only tractable in limited situations. Discrete approaches, while less accurate, are more generally applicable and more robust; only discrete approaches are considered here. Ray tracing is probably the easiest domain in which to understand the discrete process. It is common in ray tracing to generate more than one ray per pixel in order to

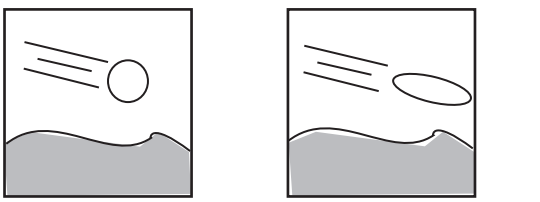

FIGURE A.9

Methods used in conventional animation for displaying speed.

FIGURE A.10

An example of synthetic (calculated) motion blur.

spatially anti-alias the image. These rays can be distributed in a regular pattern or stochastically [2] [3] [10]. To incorporate temporal anti-aliasing, one need only take the rays for a pixel and distribute them in time as well as in space. In this case, the frame is not considered to be an instant in time but rather an interval. The interval can be broken down into subintervals and the rays distributed into these subintervals. The various samples are then filtered to form the final image. See Figure A.10 for an example.

One of the extra costs associated with temporal anti-aliasing is that the motion of the objects must be computed at a higher rate than the frame rate. For example, if a 4×4 grid of subsamples is used for anti-aliasing in ray tracing and these are distributed over separate time subintervals, then the motion of the objects must be calculated at sixteen times the frame rate. If the subframe motion is computed using complex motion control algorithms, this may be a significant computational cost. Linear interpolation is often used to estimate the positions of objects at subframe intervals.

Although discussed above in terms of distributed ray tracing, this same strategy can be used with any display algorithm, as Korein and Badler [6] note. Multiple frame buffers can be used to hold rendered images at subframe intervals. These can then be filtered to form the final output image. The rendering algorithm can be a standard z-buffer, a ray tracer, a scanline algorithm, or any other technique. Because of the discrete sampling, this is still susceptible to temporal aliasing artifacts if the object is moving too fast relative to the size of its features in the direction of travel; instead of motion blur, multiple images of the object may result because intermediate pixels are effectively "jumped over" by the object during the sampling process.

In addition to analytic methods, hand manipulation of the object shape by the animator can reduce the amount of strobing. For example, the animator can stretch the object in the direction of travel. This will tend to reduce or eliminate the amount of separation between images of the object in adjacent frames.

A.4 Drop shadows

The shadow cast by an object onto a surface is an important visual cue in establishing the distance between the two. Contact shadows, or shadows produced by an object contacting the ground, are especially important. Without them, objects appear to float just above the ground plane. In high-quality animation, shadow decisions are prompted by lighting, cinematography, and visual understanding considerations. However, for most other animations, computational expense is an important concern and computing all of the shadows cast by all objects in the scene onto all other objects in the scene is overkill. Much computation can be saved if the display system supports the user specification of which objects in a scene cast shadows onto which other objects in a scene. For example, the self-shadowing[4] of an object is often not important to understanding the scene visually. Shadows cast by moving objects onto other moving objects are also often not of great importance. These principles can be observed in traditional hand-drawn animation in which only a select set of shadows is drawn.

By far the most useful type of shadow in animated sequences is the drop shadow. The drop shadow is the shadow that an object projects to the ground plane. The drop shadow lets the viewer know how far above the ground plane an object is as well as the object's relative depth and therefore relative size (see Figures A.11 and A.12).

Drop shadows can be produced in several different ways. When an object is perspectively projected from the light source to a flat ground plane, an image of the object can be formed (see Figure A.13). If this image is colored dark and displayed on the ground plane (as a texture map, for example), then it is an effective shortcut.

Another inexpensive method for creating a drop shadow is to make a copy of the object, scale it flat vertically, color it black (or make it dark and transparent), and position it just on top of the ground plane

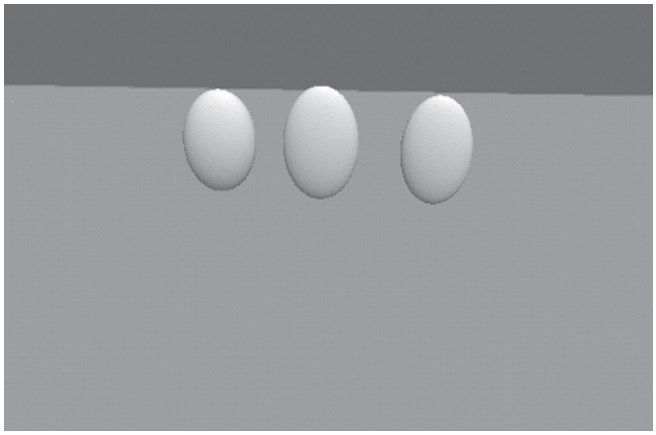

FIGURE A.11

Scene without drop shadows; without shadows, it is nearly impossible to estimate relative heights and distances if the sizes of the objects are not known.

[4] *Self-shadowing* refers to an object casting shadows onto its own surface.

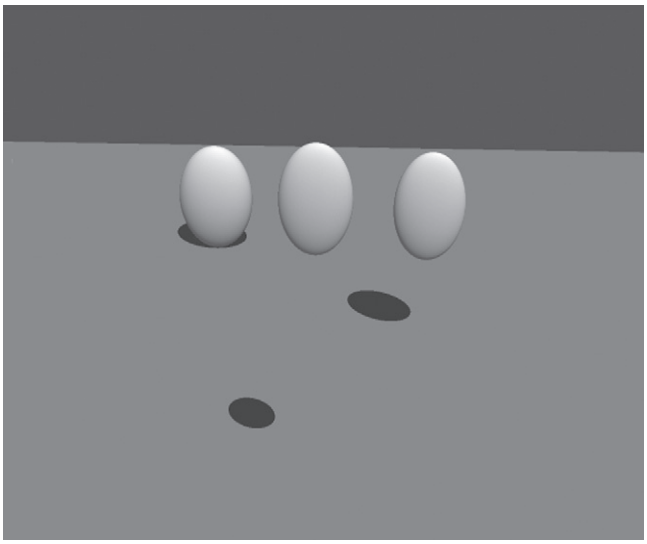

FIGURE A.12

Scene with drop shadows indicating relative depth and, therefore, relative height and size.

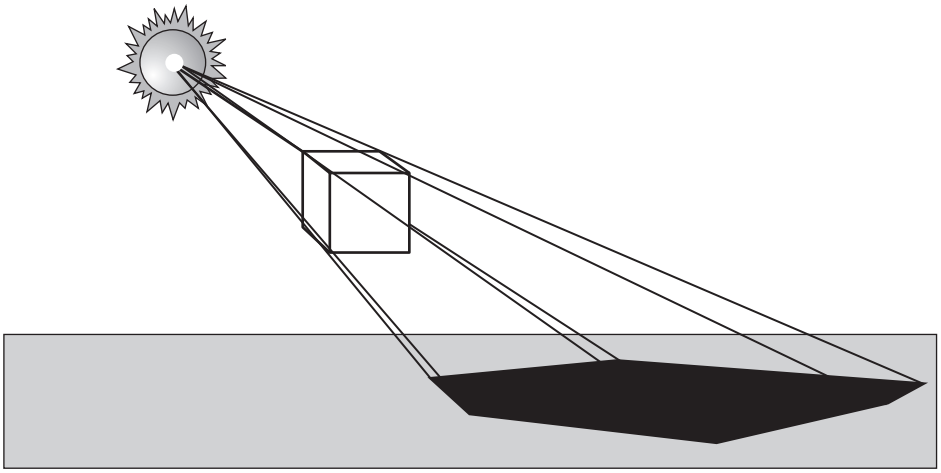

FIGURE A.13

Computing the drop shadow by perspective projection.

(see Figures A.14 and A.15). The drop shadow has the correct silhouette for a light source directly overhead but without the computational expense of the perspective projection method.

The drop shadow can be effective even if it is only an approximation of the real shadow that would be produced by a light source in the environment. The height of the object can be effectively indicated by merely controlling the relative size and softness of a drop shadow, which only suggests the shape of the object that casts it. For simple drop shadows, circles can be used, as in Figure A.16.

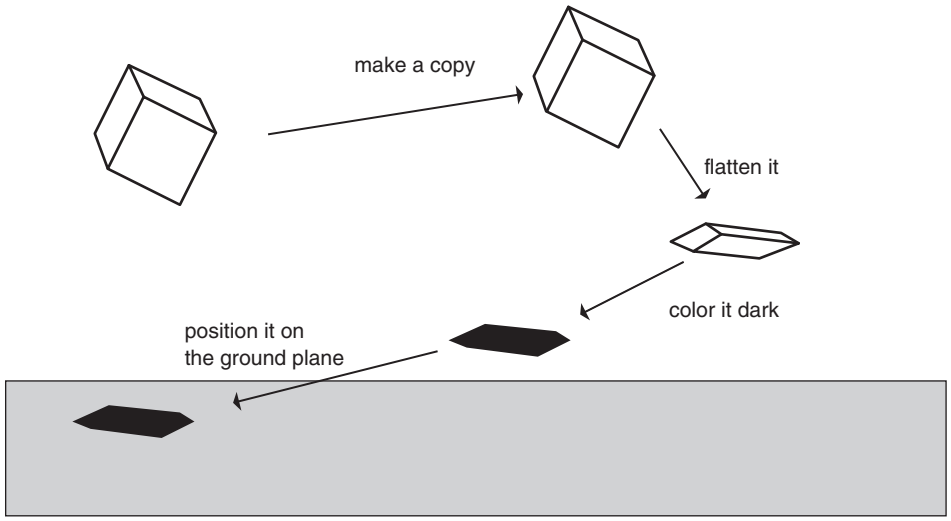

FIGURE A.14

Computing the drop shadow by flattening a copy of the object.

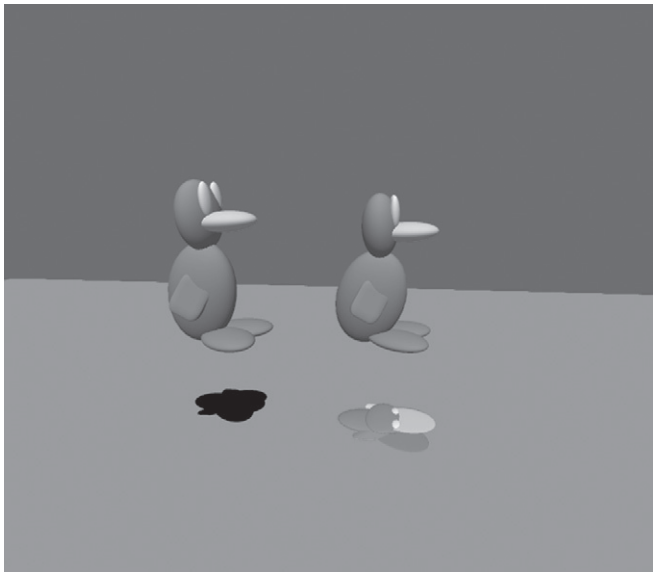

FIGURE A.15

Drop shadow using a flattened copy of the object: right side shows flattened copy with original coloring; left side shows flattened copy as black shadow.

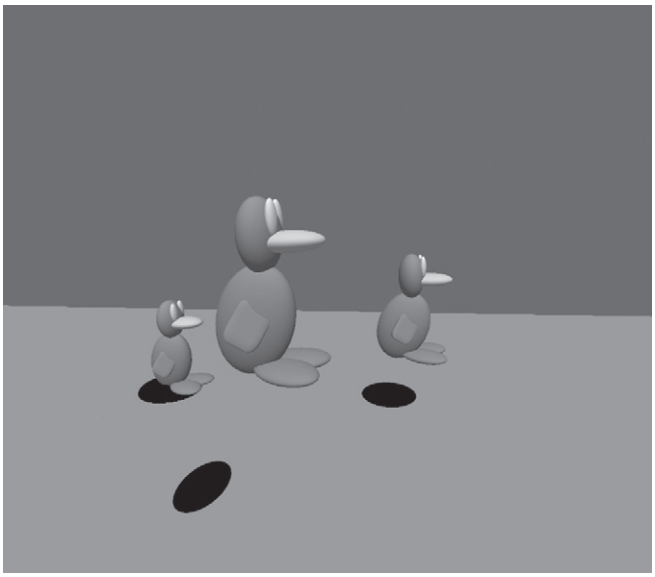

FIGURE A.16

Circular drop shadows.

Globular shapes that may more closely indicate the shapes of the original objects can also be used as shadows. The drop shadow, however it is produced, can be represented on the ground plane in several different ways. It can be colored black and placed just above the ground plane. It can be made into a darkly colored transparent object so that any detail of the ground plane shows through. It can be colored darker in the middle with more transparency toward the edges to simulate a shadow's penumbra[5] (Figure A.17).

When placing a shadow over the ground plane, one must take care to keep it close enough to the ground so that it does not appear as a separate object and at the same time does not conflict with the ground geometry. To avoid the problems of using a separate geometric element to represent the drop shadow, the user can incorporate the shadow directly into the texture map of the ground plane.

A.5 Billboarding and impostors

Billboarding and the use of impostors are a rendering shortcut that reduces the computational complexity of a scene while maintaining its visual complexity (e.g., [4] [7]). These related techniques do this by using a two-dimensional substitute (such as a planar projection) for a complex three-dimensional object that is seamlessly incorporated into a three-dimensional scene. Typically, a billboard is a partially transparent, textured quadrilateral. The texture map is an image of the object represented. The quadrilateral is partially transparent in cases when the object's image does entirely cover the quadrilateral;

[5]The *penumbra* is the area partially shadowed by an opaque body; it receives partial illumination from a light source. The *umbra* is the area completely shadowed by an opaque body; it receives no illumination from the light source.

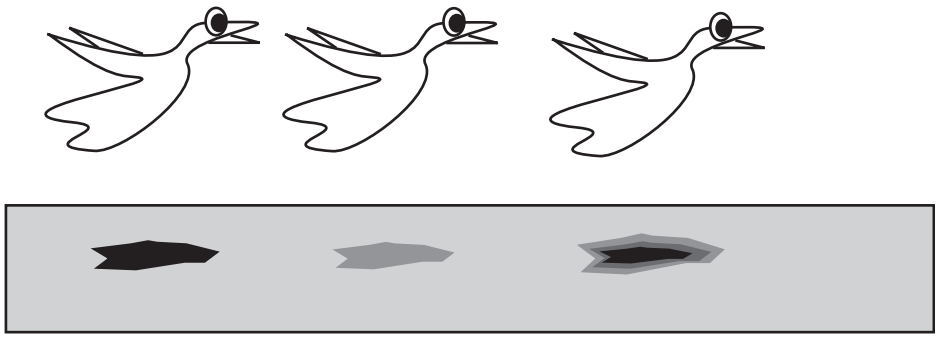

FIGURE A.17

Various ways to render a drop shadow: black, transparent, and black with transparent edges.

transparent sections allow the camera to see objects behind the quadrilateral but not obscured by the billboard image. An impostor usually implies that it is a temporary substitute for a three-dimensional object, whereas billboards usually refer to independent graphical elements.

Billboards are useful when a three-dimensional graphical element is too complex to model and render for its perceptual effect. Billboards are suitable for objects that are to be viewed at a distance, that are spherically or cylindrically perceptually symmetric, and that are just as effectively modeled in two dimensions as in three dimensions. Common uses of billboarding include the representation of trees, clouds, mountains, explosions, fire, and lens flares. See Figure A.18 for an example. Billboard representations can also be incorporated as one level of an object's level of detail representation.

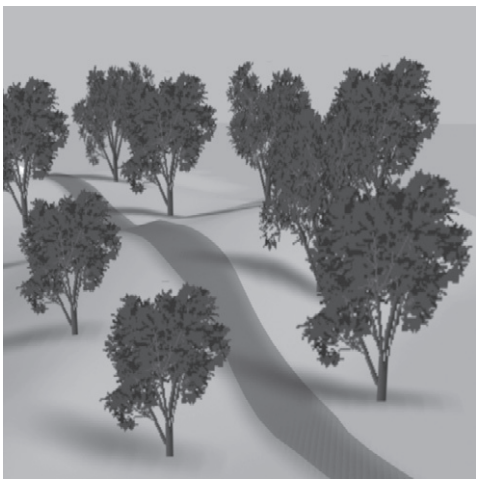

FIGURE A.18

A scene containing several billboards of trees (Image courtesy of Ysheng Chen.)

Background and ground planes could be considered types of billboards, although usually the term is reserved for substitutes of objects that would otherwise be modeled with three-dimensional geometry. However, by using multiple billboards for distance background objects at various depths, a parallax effect, typical of conventional two-dimensional animation, can be created. As an example, consider a passenger looking out the window of a moving train. The passenger may see trees, houses, and mountains moving across the window view at different speeds due to their relative depth. For objects not close enough to the train for their sides to be seen, these objects can be modeled effectively by using two-dimensional elements—billboards.

The two-dimensional background elements discussed above are examples of static billboards. Animation of a billboard's orientation is also useful. When a camera has the ability to travel in and around complex elements, billboard representations can be used that change their orientation depending on the position and orientation of the camera. Tree billboards are good examples. A tree billboard can be procedurally rotated so that its plane is always perpendicular to the direction to the camera. As the camera travels through a terrain environment, for example, the tree billboards are smoothly rotated to always face the camera. In this way, the billboards effectively hide their two dimensionality and always present their visual complexity to the camera. Orienting a billboard toward the camera can create perspective distortion when the billboard is to the side of the center of interest. As an alternative to directly facing the camera, billboards can also be oriented so that they are parallel to the view plane. This produces less perspective distortion when projected to the side of the image. Tree billboards are constrained to only rotate around the y-axis because the tree up vector must be aligned with the world up vector (y-axis). Other billboards, such as representations of clouds, explosions, or snowflakes, can be oriented to always face a certain direction as well but can be free to rotate in their plane about their own center.

Some billboards may contain animated images. These are similar to sprites used in two-and-a-half dimensional graphics common in many early digital arcade games. Animated billboards can be used for fire, explosions, and smoke and typically loop through a sequence of pre-rendered images to create the impression of a dynamic process. A billboard animation loop can be used to create a walk cycle for a figure.

A.6 Summary

Although these techniques are not concerned with specifying or controlling the motion of graphical objects, they are important to the process of generating images for computer animation. Double buffering helps to smoothly update a display of images. Compositing helps to conserve resources and combine elements from different sources. Motion blur prevents fast-moving objects from appearing jerky and distracting to the viewer. Shadows help to locate objects relative to surfaces, but only those shadows that effectively serve such a purpose need to be generated. Billboarding and the use of impostors reduce the computational complexity of the modeling and rendering while maintaining the visual complexity of the scene.

References

[1] Carpenter L. The A-Buffer Hidden Surface Method. In: Computer Graphics. Proceedings of SIGGRAPH 84, vol. 18(3). Minneapolis, Minn.; July 1984. p. 103–8.

[2] Cook R. Stochastic Sampling in Computer Graphics. ACM Transactions on Graphics January 1986;5(1):51–71.

[3] Cook R, Porter T, Carpenter L. Distributed Ray Tracing. In: Computer Graphics. Proceedings of SIGGRAPH 84, vol. 18(3). Minneapolis, Minn.; July 1984. p. 137–46.
[4] Decoret X, Durand F, Sillion F, Dorsey J. Billboard Clouds for Extreme Model Simplification. Transactions on Graphics July 2003;22(3), Special Issue: Proceedings of ACM SIGGRAPH 2003. pp. 689–96.
[5] Duff T. Compositing 3-D Rendered Images. In: Barsky BA., editor. Computer Graphics. Proceedings of SIGGRAPH 85, vol. 19(3). San Francisco, Calif.; August 1985. p. 41–4.
[6] Korein J, Badler N. Temporal Anti-Aliasing in Computer Generated Animation. In: Computer Graphics. Proceedings of SIGGRAPH 83, vol. 17(3). Detroit, Mich.; July 1983. p. 377–88.
[7] Maciel P, Shirley P. Visual Navigation of Large Environments Using Textured Clusters. In: Proc 1995 Symposium on Interactive 3D Graphics. Monterey, California; April 9–12, 1995. p. 95–102.
[8] Potmesil M, Chadkravarty I. Modeling Motion Blur in Computer Generated Images. In: Computer Graphics. Proceedings of SIGGRAPH 83, vol. 17(3). Detroit, Mich.; July 1983. p. 389–400.
[9] Thomas F, Johnson O. The Illusion of Life. New York: Hyperion; 1981.
[10] Whitted T. An Improved Illumination Model for Shaded Display. Communication of the ACM June 1980;23(6):343–9.