The material system also needs to ensure good performance. Besides specialized compilation of shading variants, there are a few other common optimizations the material system can perform. The *Destiny* shader system and the Unreal Engine automatically detect computations that are constant across a draw call (such as the warm and cool color computation in the earlier implementation example) and move it outside of the shader. Another example is the scoping system used in *Destiny* to differentiate between constants that are updated at different frequencies (e.g., once per frame, once per light, once per object) and update each set of constants at the appropriate times to reduce API overhead.

As we have seen, implementing a shading equation is a matter of deciding what parts can be simplified, how frequently to compute various expressions, and how the user is able to modify and control the appearance. The ultimate output of the rendering pipeline is a color and blend value. The remaining sections on antialiasing, transparency, and image display detail how these values are combined and modified for display.

5.4 Aliasing and Antialiasing

Imagine a large black triangle moving slowly across a white background. As a screen grid cell is covered by the triangle, the pixel value representing this cell should smoothly drop in intensity. What typically happens in basic renderers of all sorts is that the moment the grid cell's center is covered, the pixel color immediately goes from white to black. Standard GPU rendering is no exception. See the leftmost column of Figure 5.14.

Triangles show up in pixels as either there or not there. Lines drawn have a similar problem. The edges have a jagged look because of this, and so this visual artifact is called "the jaggies," which turn into "the crawlies" when animated. More formally, this problem is called *aliasing*, and efforts to avoid it are called *antialiasing* techniques.

The subject of sampling theory and digital filtering is large enough to fill its own book [559, 1447, 1729]. As this is a key area of rendering, the basic theory of sampling and filtering will be presented. We will then focus on what currently can be done in real time to alleviate aliasing artifacts.

5.4.1 Sampling and Filtering Theory

The process of rendering images is inherently a sampling task. This is so since the generation of an image is the process of sampling a three-dimensional scene in order to obtain color values for each pixel in the image (an array of discrete pixels). To use texture mapping (Chapter 6), texels have to be resampled to get good results under varying conditions. To generate a sequence of images in an animation, the animation is often sampled at uniform time intervals. This section is an introduction to the topic of sampling, reconstruction, and filtering. For simplicity, most material will be

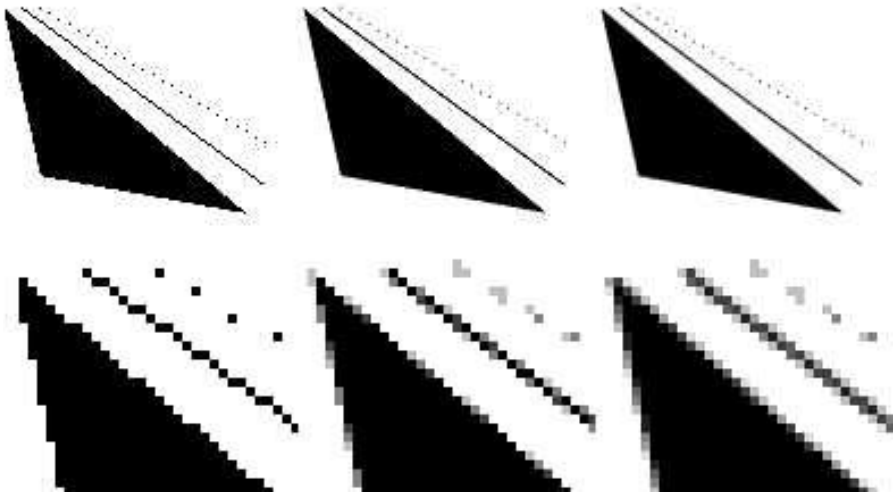

Figure 5.14. The upper row shows three images with different levels of antialiasing of a triangle, a line, and some points. The lower row images are magnifications of the upper row. The leftmost column uses only one sample per pixel, which means that no antialiasing is used. The middle column images were rendered with four samples per pixel (in a grid pattern), and the right column used eight samples per pixel (in a 4×4 checkerboard, half the squares sampled).

presented in one dimension. These concepts extend naturally to two dimensions as well, and can thus be used when handling two-dimensional images.

Figure 5.15 shows how a continuous signal is being sampled at uniformly spaced intervals, that is, discretized. The goal of this *sampling* process is

to represent information digitally. In doing so, the amount of information is reduced. However, the sampled signal needs to be *reconstructed* to recover the original signal. This is done by *filtering* the sampled signal.

Whenever sampling is done, aliasing may occur. This is an unwanted artifact, and we need to battle aliasing to generate pleasing images. A classic example of aliasing seen in old Westerns is a spinning wagon wheel filmed by a movie camera. Because

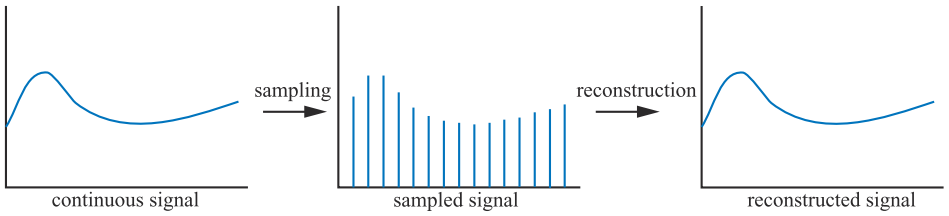

Figure 5.15. A continuous signal (left) is sampled (middle), and then the original signal is recovered by reconstruction (right).

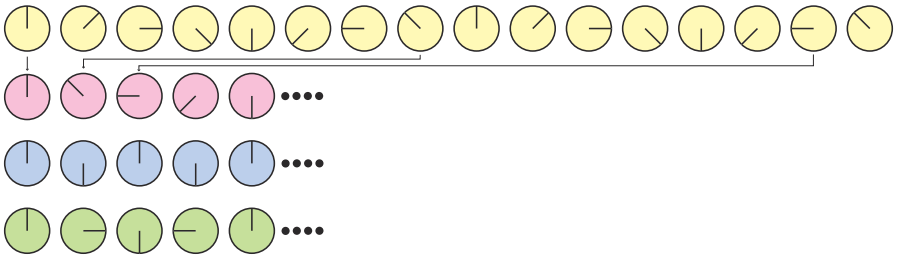

Figure 5.16. The top row shows a spinning wheel (original signal). This is inadequately sampled in second row, making it appear to move in the opposite direction. This is an example of aliasing due to a too low sampling rate. In the third row, the sampling rate is exactly two samples per revolution, and we cannot determine in which direction the wheel is spinning. This is the Nyquist limit. In the fourth row, the sampling rate is higher than two samples per revolution, and we suddenly can see that the wheel spins in the right direction.

the spokes move much faster than the camera records images, the wheel may appear to be spinning slowly (backward or forward), or may even look like it is not rotating at all. This can be seen in Figure 5.16. The effect occurs because the images of the wheel are taken in a series of time steps, and is called *temporal aliasing*.

Common examples of aliasing in computer graphics are the "jaggies" of a rasterized line or triangle edge, flickering highlights known as "fireflies", and when a texture with a checker pattern is minified (Section 6.2.2).

Aliasing occurs when a signal is being sampled at too low a frequency. The sampled signal then appears to be a signal of lower frequency than the original. This is illustrated in Figure 5.17. For a signal to be sampled properly (i.e., so that it is possible to reconstruct the original signal from the samples), the sampling frequency

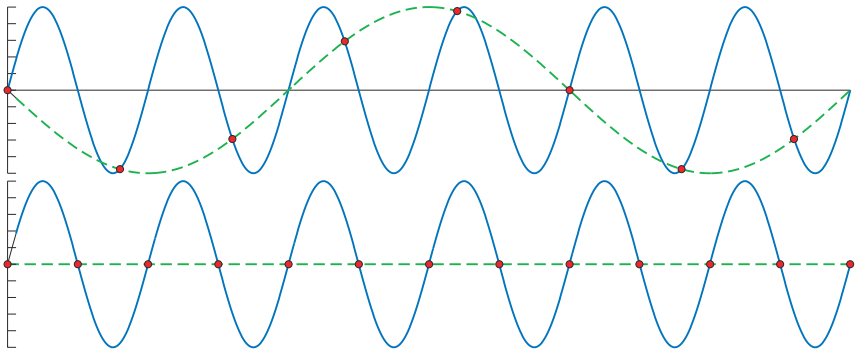

Figure 5.17. The solid blue line is the original signal, the red circles indicate uniformly spaced sample points, and the green dashed line is the reconstructed signal. The top figure shows a too low sample rate. Therefore, the reconstructed signal appears to be of lower frequency, i.e., an alias of the original signal. The bottom shows a sampling rate of exactly twice the frequency of the original signal, and the reconstructed signal is here a horizontal line. It can be proven that if the sampling rate is increased ever so slightly, perfect reconstruction is possible.

has to be more than twice the maximum frequency of the signal to be sampled. This is often called the *sampling theorem*, and the sampling frequency is called the *Nyquist rate* [1447] or *Nyquist limit*, after Harry Nyquist (1889–1976), a Swedish scientist who discovered this in 1928. The Nyquist limit is also illustrated in Figure 5.16. The fact that the theorem uses the term "maximum frequency" implies that the signal has to be *band-limited*, which just means that there are not any frequencies above a certain limit. Put another way, the signal has to be smooth enough relative to the spacing between neighboring samples.

A three-dimensional scene is normally never band-limited when rendered with point samples. Edges of triangles, shadow boundaries, and other phenomena produce a signal that changes discontinuously and so produces frequencies that are infinite [252]. Also, no matter how closely packed the samples are, objects can still be small enough that they do not get sampled at all. Thus, it is impossible to entirely avoid aliasing problems when using point samples to render a scene, and we almost always use point sampling. However, at times it is possible to know when a signal is band-limited. One example is when a texture is applied to a surface. It is possible to compute the frequency of the texture samples compared to the sampling rate of the pixel. If this frequency is lower than the Nyquist limit, then no special action is needed to properly sample the texture. If the frequency is too high, then a variety of algorithms are used to band-limit the texture (Section 6.2.2).

Reconstruction

Given a band-limited sampled signal, we will now discuss how the original signal can be reconstructed from the sampled signal. To do this, a filter must be used. Three commonly used filters are shown in Figure 5.18. Note that the area of the filter should always be one, otherwise the reconstructed signal can appear to grow or shrink.

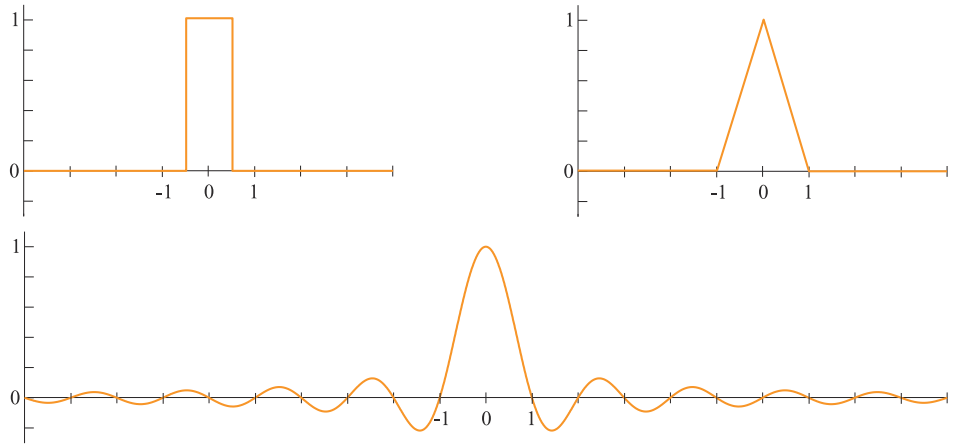

Figure 5.18. The top left shows the box filter, and the top right the tent filter. The bottom shows the sinc filter (which has been clamped on the x-axis here).

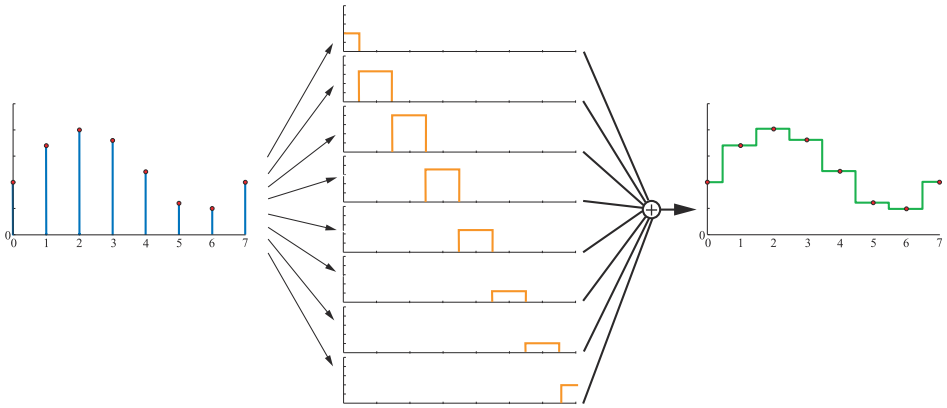

Figure 5.19. The sampled signal (left) is reconstructed using the box filter. This is done by placing the box filter over each sample point, and scaling it in the y-direction so that the height of the filter is the same as the sample point. The sum is the reconstruction signal (right).

In Figure 5.19, the box filter (nearest neighbor) is used to reconstruct a sampled signal. This is the worst filter to use, as the resulting signal is a noncontinuous stair case. Still, it is often used in computer graphics because of its simplicity. As can be seen in the illustration, the box filter is placed over each sample point, and then scaled so that the topmost point of the filter coincides with the sample point. The sum of all these scaled and translated box functions is the reconstructed signal shown to the right.

The box filter can be replaced with any other filter. In Figure 5.20, the tent filter, also called the triangle filter, is used to reconstruct a sampled signal. Note that this

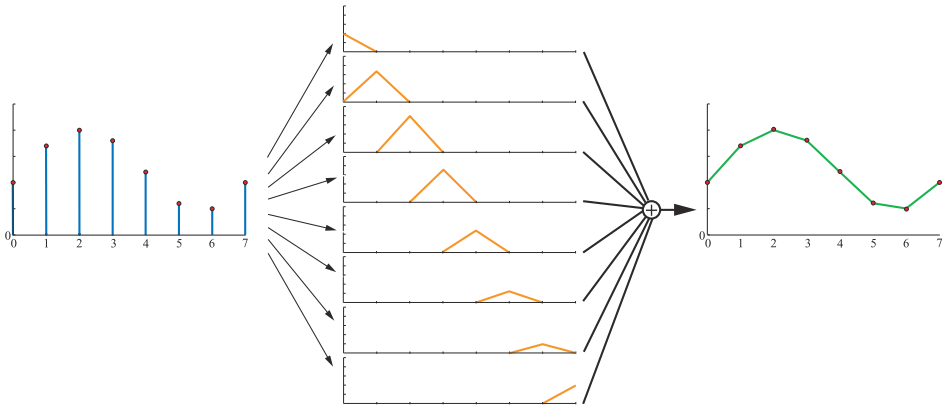

Figure 5.20. The sampled signal (left) is reconstructed using the tent filter. The reconstructed signal is shown to the right.

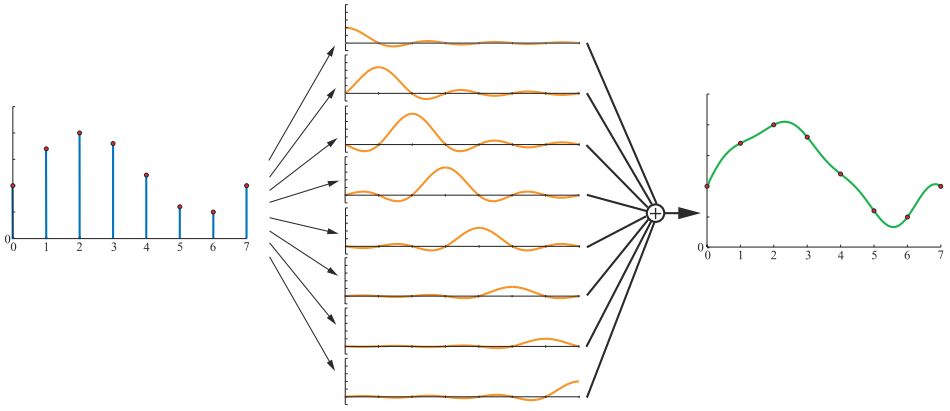

Figure 5.21. Here, the sinc filter is used to reconstruct the signal. The sinc filter is the ideal low-pass filter.

filter implements linear interpolation between neighboring sample points, and so it is better than the box filter, as the reconstructed signal now is continuous.

However, the smoothness of the reconstructed signal using a tent filter is poor; there are sudden slope changes at the sample points. This has to do with the fact that the tent filter is not a perfect reconstruction filter. To get perfect reconstruction the ideal *low-pass filter* has to be used. A frequency component of a signal is a sine wave: $\sin(2\pi f)$, where f is the frequency of that component. Given this, a low-pass filter removes all frequency components with frequencies higher than a certain frequency defined by the filter. Intuitively, the low-pass filter removes sharp features of the signal, i.e., the filter blurs it. The ideal low-pass filter is the sinc filter (Figure 5.18 bottom):

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}. \tag{5.22}$$

The theory of Fourier analysis [1447] explains why the sinc filter is the ideal low-pass filter. Briefly, the reasoning is as follows. The ideal low-pass filter is a box filter in the frequency domain, which removes all frequencies above the filter width when it is multiplied with the signal. Transforming the box filter from the frequency domain to the spatial domain gives a sinc function. At the same time, the multiplication operation is transformed into the *convolution* function, which is what we have been using in this section, without actually describing the term.

Using the sinc filter to reconstruct the signal gives a smoother result, as shown in Figure 5.21. The sampling process introduces high-frequency components (abrupt changes) in the signal, and the task of the low-pass filter is to remove these. In fact, the sinc filter eliminates all sine waves with frequencies higher than 1/2 the sampling rate. The sinc function, as presented in Equation 5.22, is the perfect reconstruction filter when the sampling frequency is 1.0 (i.e., the maximum frequency of the sampled

signal must be smaller than 1/2). More generally, assume the sampling frequency is f_s, that is, the interval between neighboring samples is $1/f_s$. For such a case, the perfect reconstruction filter is $\text{sinc}(f_s x)$, and it eliminates all frequencies higher than $f_s/2$. This is useful when resampling the signal (next section). However, the filter width of the sinc is infinite and is negative in some areas, so it is rarely useful in practice.

There is a useful middle ground between the low-quality box and tent filters on one hand, and the impractical sinc filter on the other. Most widely used filter functions [1214, 1289, 1413, 1793] are between these extremes. All these filter functions have some approximation to the sinc function, but with a limit on how many pixels they influence. The filters that most closely approximate the sinc function have negative values over part of their domain. For applications where negative filter values are undesirable or impractical, filters with no negative lobes (often referred to generically as Gaussian filters, since they either derive from or resemble a Gaussian curve) are typically used [1402]. Section 12.1 discusses filter functions and their use in more detail.

After using any filter, a continuous signal is obtained. However, in computer graphics we cannot display continuous signals directly, but we can use them to resample the continuous signal to another size, i.e., either enlarging the signal, or diminishing it. This topic is discussed next.

Resampling

Resampling is used to magnify or minify a sampled signal. Assume that the original sample points are located at integer coordinates $(0, 1, 2, \dots)$, that is, with unit intervals between samples. Furthermore, assume that after resampling we want the new sample points to be located uniformly with an interval a between samples. For $a > 1$, minification (downsampling) takes place, and for $a < 1$, magnification (upsampling) occurs.

Magnification is the simpler case of the two, so let us start with that. Assume the sampled signal is reconstructed as shown in the previous section. Intuitively, since the signal now is perfectly reconstructed and continuous, all that is needed is to resample the reconstructed signal at the desired intervals. This process can be seen in Figure 5.22.

However, this technique does not work when minification occurs. The frequency of the original signal is too high for the sampling rate to avoid aliasing. Instead it has been shown that a filter using $\text{sinc}(x/a)$ should be used to create a continuous signal from the sampled one [1447, 1661]. After that, resampling at the desired intervals can take place. This can be seen in Figure 5.23. Said another way, by using $\text{sinc}(x/a)$ as a filter here, the width of the low-pass filter is increased, so that more of the signal's higher frequency content is removed. As shown in the figure, the filter width (of the individual sinc's) is doubled to decrease the resampling rate to half the original sampling rate. Relating this to a digital image, this is similar to first blurring it (to remove high frequencies) and then resampling the image at a lower resolution.

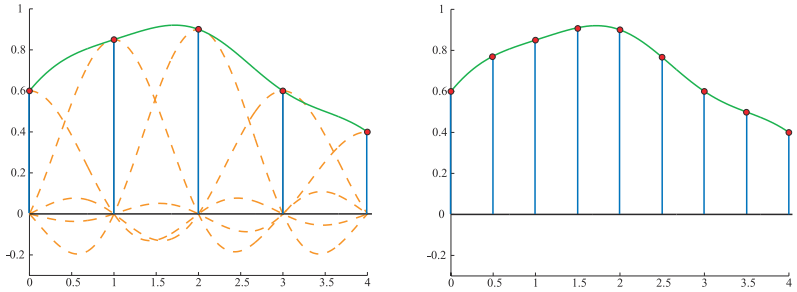

Figure 5.22. On the left is the sampled signal, and the reconstructed signal. On the right, the reconstructed signal has been resampled at double the sample rate, that is, magnification has taken place.

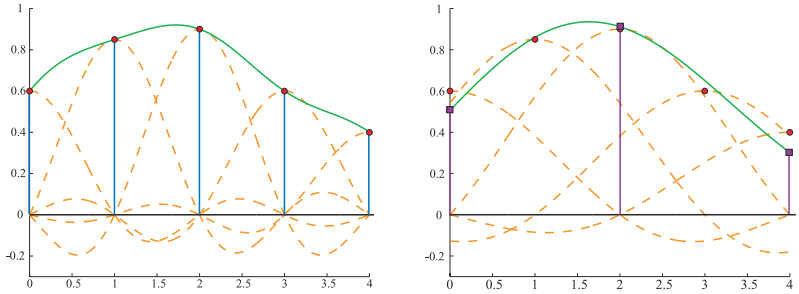

Figure 5.23. On the left is the sampled signal, and the reconstructed signal. On the right, the filter width has doubled in order to double the interval between the samples, that is, minification has taken place.

With the theory of sampling and filtering available as a framework, the various algorithms used in real-time rendering to reduce aliasing are now discussed.

5.4.2 Screen-Based Antialiasing

Edges of triangles produce noticeable artifacts if not sampled and filtered well. Shadow boundaries, specular highlights, and other phenomena where the color is changing rapidly can cause similar problems. The algorithms discussed in this section help improve the rendering quality for these cases. They have the common thread that they are screen based, i.e., that they operate only on the output samples of the pipeline. There is no one best antialiasing technique, as each has different advantages in terms of quality, ability to capture sharp details or other phenomena, appearance during movement, memory cost, GPU requirements, and speed.

In the black triangle example in Figure 5.14, one problem is the low sampling rate. A single sample is taken at the center of each pixel's grid cell, so the most that is

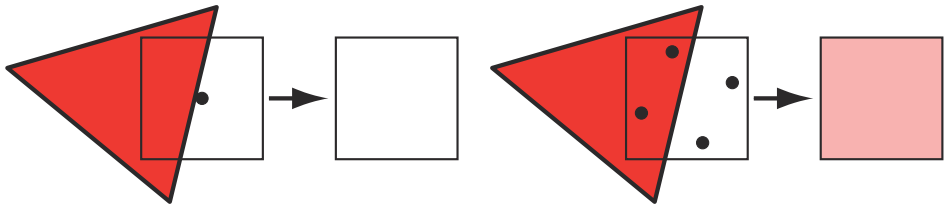

Figure 5.24. On the left, a red triangle is rendered with one sample at the center of the pixel. Since the triangle does not cover the sample, the pixel will be white, even though a substantial part of the pixel is covered by the red triangle. On the right, four samples are used per pixel, and as can be seen, two of these are covered by the red triangle, which results in a pink pixel color.

known about the cell is whether or not the center is covered by the triangle. By using more samples per screen grid cell and blending these in some fashion, a better pixel color can be computed. This is illustrated in Figure 5.24.

The general strategy of screen-based antialiasing schemes is to use a sampling pattern for the screen and then weight and sum the samples to produce a pixel color, $\mathbf{p}$:

$$\mathbf{p}(x,y) = \sum_{i=1}^{n} w_i \mathbf{c}(i,x,y), \tag{5.23}$$

where n is the number of samples taken for a pixel. The function $\mathbf{c}(i,x,y)$ is a sample color and w_i is a weight, in the range $[0,1]$, that the sample will contribute to the overall pixel color. The sample position is taken based on which sample it is in the series $1,\ldots,n$, and the function optionally also uses the integer part of the pixel location (x,y). In other words, where the sample is taken on the screen grid is different for each sample, and optionally the sampling pattern can vary from pixel to pixel. Samples are normally point samples in real-time rendering systems (and most other rendering systems, for that matter). So, the function $\mathbf{c}$ can be thought of as two functions. First, a function $\mathbf{f}(i,n)$ retrieves the floating point (x_f, y_f) location on the screen where a sample is needed. This location on the screen is then sampled, i.e., the color at that precise point is retrieved. The sampling scheme is chosen and the rendering pipeline configured to compute the samples at particular subpixel locations, typically based on a per-frame (or per-application) setting.

The other variable in antialiasing is w_i, the weight of each sample. These weights sum to one. Most methods used in real-time rendering systems give a uniform weight to their samples, i.e., $w_i = \frac{1}{n}$. The default mode for graphics hardware, a single sample at the center of the pixel, is the simplest case of the antialiasing equation above. There is only one term, the weight of this term is one, and the sampling function $\mathbf{f}$ always returns the center of the pixel being sampled.

Antialiasing algorithms that compute more than one full sample per pixel are called *supersampling* (or *oversampling*) methods. Conceptually simplest, *full-scene antialiasing* (FSAA), also known as "supersampling antialiasing" (SSAA), renders the

scene at a higher resolution and then filters neighboring samples to create an image. For example, say an image of 1280×1024 pixels is desired. If you render an image of 2560×2048 offscreen and then average each 2×2 pixel area on the screen, the desired image is generated with four samples per pixel, filtered using a box filter. Note that this corresponds to 2×2 grid sampling in Figure 5.25. This method is costly, as all subsamples must be fully shaded and filled, with a z-buffer depth per sample. FSAA's main advantage is simplicity. Other, lower-quality versions of this method sample at twice the rate on only one screen axis, and so are called 1×2 or 2×1 supersampling. Typically, powers-of-two resolution and a box filter are used for simplicity. NVIDIA's *dynamic super resolution* feature is a more elaborate form of supersampling, where the scene is rendered at some higher resolution and a 13-sample Gaussian filter is used to generate the displayed image [1848].

A sampling method related to supersampling is based on the idea of the *accumulation buffer* [637, 1115]. Instead of one large offscreen buffer, this method uses a buffer that has the same resolution as the desired image, but with more bits of color per channel. To obtain a 2×2 sampling of a scene, four images are generated, with the view moved half a pixel in the screen x- or y-direction as needed. Each image generated is based on a different sample position within the grid cell. The additional costs of having to re-render the scene a few times per frame and copy the result to the screen makes this algorithm costly for real-time rendering systems. It is useful for generating higher-quality images when performance is not critical, since any number of samples, placed anywhere, can be used per pixel [1679]. The accumulation buffer used to be a separate piece of hardware. It was supported directly in the OpenGL API, but was deprecated in version 3.0. On modern GPUs the accumulation buffer concept can be implemented in a pixel shader by using a higher-precision color format for the output buffer.

Additional samples are needed when phenomena such as object edges, specular highlights, and sharp shadows cause abrupt color changes. Shadows can often be made softer and highlights smoother to avoid aliasing. Particular object types can be increased in size, such as electrical wires, so that they are guaranteed to cover at least one pixel at each location along their length [1384]. Aliasing of object edges still remains as a major sampling problem. It is possible to use analytical methods, where object edges are detected during rendering and their influence is factored in, but these are often more expensive and less robust than simply taking more samples. However, GPU features such as conservative rasterization and rasterizer order views have opened up new possibilities [327].

Techniques such as supersampling and accumulation buffering work by generating samples that are fully specified with individually computed shades and depths. The overall gains are relatively low and the cost is high, as each sample has to run through a pixel shader.

Multisampling antialiasing (MSAA) lessens the high computational costs by computing the surface's shade once per pixel and sharing this result among the samples. Pixels may have, say, four (x, y) sample locations per fragment, each with their own

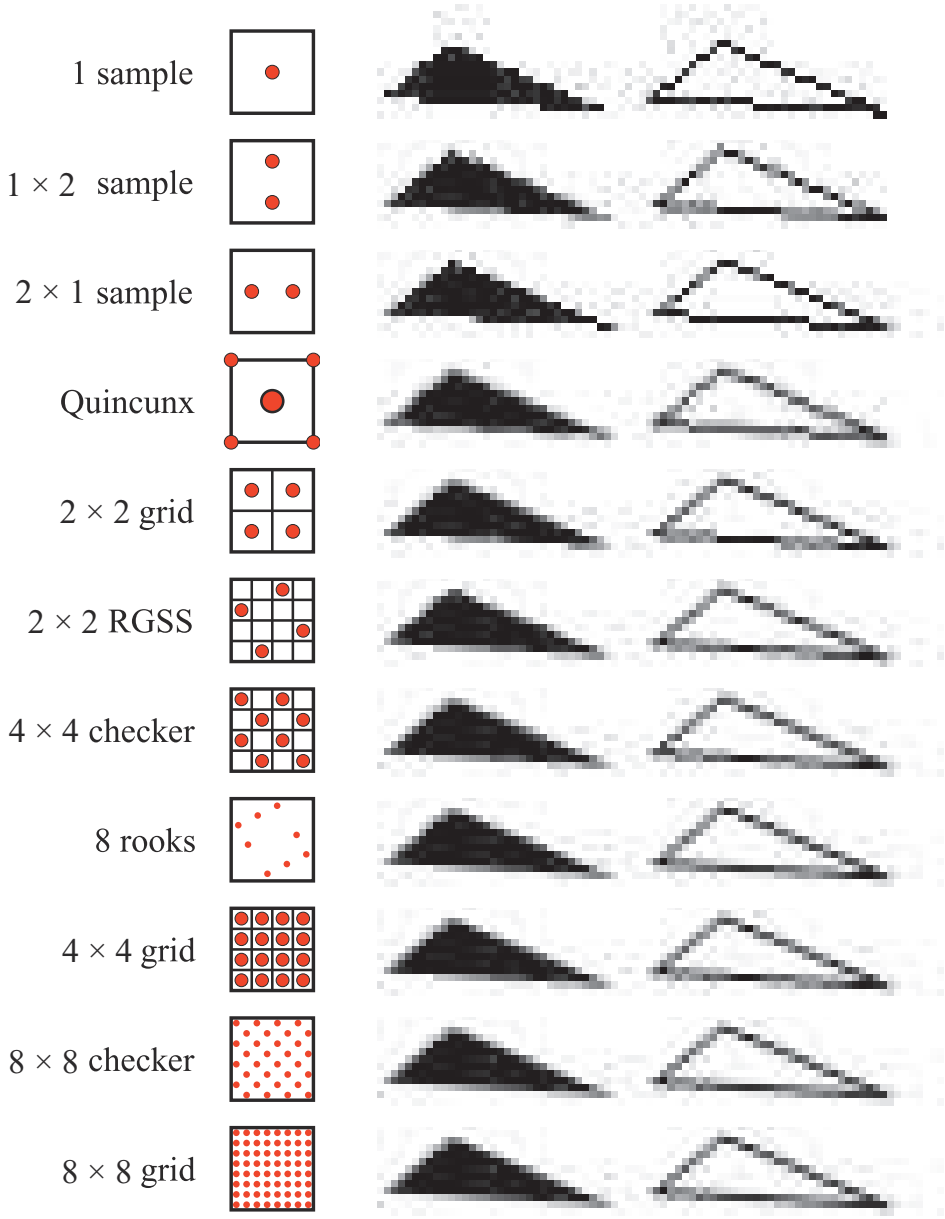

Figure 5.25. A comparison of some pixel sampling schemes, ranging from least to most samples per pixel. Quincunx shares the corner samples and weights its center sample to be worth half of the pixel's final color. The 2×2 rotated grid captures more gray levels for the nearly horizontal edge than a straight 2×2 grid. Similarly, the 8 rooks pattern captures more gray levels for such lines than a 4×4 grid, despite using fewer samples.

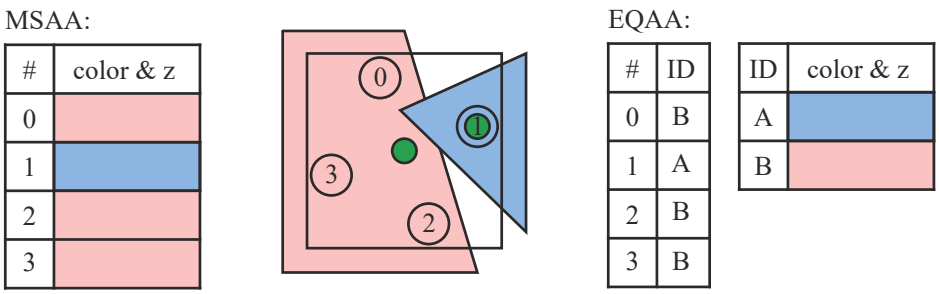

Figure 5.26. In the middle, a pixel with two objects overlapping it. The red object covers three samples, the blue just one. Pixel shader evaluation locations are shown in green. Since the red triangle covers the center of the pixel, this location is used for shader evaluation. The pixel shader for the blue object is evaluated at the sample's location. For MSAA, a separate color and depth is stored at all four locations. On the right the 2f4x mode for EQAA is shown. The four samples now have four ID values, which index a table of the two colors and depths stored.

color and z-depth, but the pixel shader is evaluated only once for each object fragment applied to the pixel. If all MSAA positional samples are covered by the fragment, the shading sample is evaluated at the center of the pixel. If instead the fragment covers fewer positional samples, the shading sample's position can be shifted to better represent the positions covered. Doing so avoids shade sampling off the edge of a texture, for example. This position adjustment is called *centroid sampling* or *centroid interpolation* and is done automatically by the GPU, if enabled. Centroid sampling avoids off-triangle problems but can cause derivative computations to return incorrect values [530, 1041]. See Figure 5.26.

MSAA is faster than a pure supersampling scheme because the fragment is shaded only once. It focuses effort on sampling the fragment's pixel coverage at a higher rate and sharing the computed shade. It is possible to save more memory by further decoupling sampling and coverage, which in turn can make antialiasing faster still—the less memory touched, the quicker the render. NVIDIA introduced *coverage sampling antialiasing* (CSAA) in 2006, and AMD followed suit with *enhanced quality antialiasing* (EQAA). These techniques work by storing only the coverage for the fragment at a higher sampling rate. For example, EQAA's "2f4x" mode stores two color and depth values, shared among four sample locations. The colors and depths are no longer stored for particular locations but rather saved in a table. Each of the four samples then needs just one bit to specify which of the two stored values is associated with its location. See Figure 5.26. The coverage samples specify the contribution of each fragment to the final pixel color. If the number of colors stored is exceeded, a stored color is evicted and its samples are marked as unknown. These samples do not contribute to the final color [382, 383]. For most scenes there are relatively few pixels containing three or more visible opaque fragments that are radically different in shade, so this scheme performs well in practice [1405]. However, for highest quality, the game *Forza Horizon 2* went with 4× MSAA, though EQAA had a performance benefit [1002].

Once all geometry has been rendered to a multiple-sample buffer, a *resolve* operation is then performed. This procedure averages the sample colors together to determine the color for the pixel. It is worth noting that a problem can arise when using multisampling with high dynamic range color values. In such cases, to avoid artifacts you normally need to tone-map the values before the resolve [1375]. This can be expensive, so a simpler approximation to the tone map function or other methods can be used [862, 1405].

By default, MSAA is resolved with a box filter. In 2007 ATI introduced *custom filter antialiasing* (CFAA) [1625], with the capabilities of using narrow and wide tent filters that extend slightly into other pixel cells. This mode has since been supplanted by EQAA support. On modern GPUs pixel or compute shaders can access the MSAA samples and use whatever reconstruction filter is desired, including one that samples from the surrounding pixels' samples. A wider filter can reduce aliasing, though at the loss of sharp details. Pettineo [1402, 1405] found that the cubic smoothstep and B-spline filters with a filter width of 2 or 3 pixels gave the best results overall. There is also a performance cost, as even emulating the default box filter resolve will take longer with a custom shader, and a wider filter kernel means increased sample access costs.

NVIDIA's built-in TXAA support similarly uses a better reconstruction filter over a wider area than a single pixel to give a better result. It and the newer MFAA (multi-frame antialiasing) scheme both also use *temporal antialiasing* (TAA), a general class of techniques that use results from previous frames to improve the image. In part such techniques are made possible due to functionality that lets the programmer set the MSAA sampling pattern per frame [1406]. Such techniques can attack aliasing problems such as the spinning wagon wheel and can also improve edge rendering quality.

Imagine performing a sampling pattern "manually" by generating a series of images where each render uses a different location within the pixel for where the sample is taken. This offsetting is done by appending a tiny translation on to the projection matrix [1938]. The more images that are generated and averaged together, the better the result. This concept of using multiple offset images is used in temporal antialiasing algorithms. A single image is generated, possibly with MSAA or another method, and the previous images are blended in. Usually just two to four frames are used [382, 836, 1405]. Older images may be given exponentially less weight [862], though this can have the effect of the frame shimmering if the viewer and scene do not move, so often equal weighting of just the last and current frame is done. With each frame's samples in a different subpixel location, the weighted sum of these samples gives a better coverage estimate of the edge than a single frame does. So, a system using the latest two frames averaged together can give a better result. No additional samples are needed for each frame, which is what makes this type of approach so appealing. It is even possible to use temporal sampling to allow generation of a lower-resolution image that is upscaled to the display's resolution [1110]. In addition, illumination methods or other techniques that require many samples for a good result can instead use fewer samples each frame, since the results will be blended over several frames [1938].

While providing antialiasing for static scenes at no additional sampling cost, this type of algorithm has a few problems when used for temporal antialiasing. If the frames are not weighted equally, objects in a static scene can exhibit a shimmer. Rapidly moving objects or quick camera moves can cause ghosting, i.e., trails left behind the object due to the contributions of previous frames. One solution to ghosting is to perform such antialiasing on only slow-moving objects [1110]. Another important approach is to use *reprojection* (Section 12.2) to better correlate the previous and current frames' objects. In such schemes, objects generate motion vectors that are stored in a separate "velocity buffer" (Section 12.5). These vectors are used to correlate the previous frame with the current one, i.e., the vector is subtracted from the current pixel location to find the previous frame's color pixel for that object's surface location. Samples unlikely to be part of the surface in the current frame are discarded [1912]. Because no extra samples, and so relatively little extra work, are needed for temporal antialiasing, there has been a strong interest and wider adoption of this type of algorithm in recent years. Some of this attention has been because deferred shading techniques (Section 20.1) are not compatible with MSAA and other multisampling support [1486]. Approaches vary and, depending on the application's content and goals, a range of techniques for avoiding artifacts and improving quality have been developed [836, 1154, 1405, 1533, 1938]. Wihlidal's presentation [1885], for example, shows how EQAA, temporal antialiasing, and various filtering techniques applied to a checkerboard sampling pattern can combine to maintain quality while lowering the number of pixel shader invocations. Iglesias-Guitian et al. [796] summarize previous work and present their scheme to use pixel history and prediction to minimize filtering artifacts. Patney et al. [1357] extend TAA work by Karis and Lottes on the Unreal Engine 4 implementation [862] for use in virtual reality applications, adding variable-sized sampling along with compensation for eye movement (Section 21.3.2).

Sampling Patterns

Effective sampling patterns are a key element in reducing aliasing, temporal and otherwise. Naiman [1257] shows that humans are most disturbed by aliasing on near-horizontal and near-vertical edges. Edges with near 45 degrees slope are next most disturbing. *Rotated grid supersampling* (RGSS) uses a rotated square pattern to give more vertical and horizontal resolution within the pixel. Figure 5.25 shows an example of this pattern.

The RGSS pattern is a form of *Latin hypercube* or *N-rooks sampling*, in which n samples are placed in an $n \times n$ grid, with one sample per row and column [1626]. With RGSS, the four samples are each in a separate row and column of the 4×4 subpixel grid. Such patterns are particularly good for capturing nearly horizontal and vertical edges compared to a regular 2×2 sampling pattern, where such edges are likely to cover an even number of samples, so giving fewer effective levels.

N-rooks is a start at creating a good sampling pattern, but it is not sufficient. For example, the samples could all be places along the diagonal of a subpixel grid and so give a poor result for edges that are nearly parallel to this diagonal. See Figure 5.27.

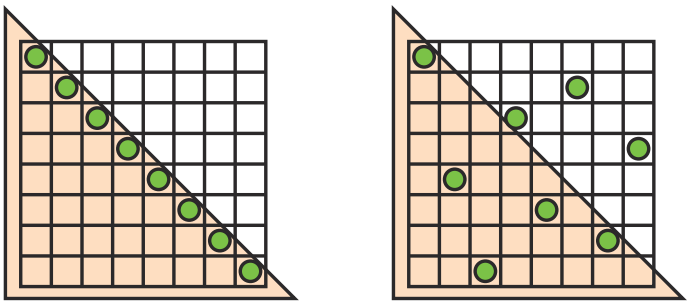

Figure 5.27. N-rooks sampling. On the left is a legal N-rooks pattern, but it performs poorly in capturing triangle edges that are diagonal along its line, as all sample locations will be either inside or outside the triangle as this triangle shifts. On the right is a pattern that will capture this and other edges more effectively.

For better sampling we want to avoid putting two samples near each other. We also want a uniform distribution, spreading samples evenly over the area. To form such patterns, *stratified sampling* techniques such as Latin hypercube sampling are combined with other methods such as jittering, Halton sequences, and Poisson disk sampling [1413, 1758].

In practice GPU manufacturers usually hard-wire such sampling patterns into their hardware for multisampling antialiasing. Figure 5.28 shows some MSAA patterns used in practice. For temporal antialiasing, the coverage pattern is whatever the programmer wants, as the sample locations can be varied frame to frame. For example, Karis [862] finds that a basic *Halton sequence* works better than any MSAA pattern provided by the GPU. A Halton sequence generates samples in space that appear random but have low discrepancy, that is, they are well distributed over the space and none are clustered [1413, 1938].

While a subpixel grid pattern results in a better approximation of how each triangle covers a grid cell, it is not ideal. A scene can be made of objects that are arbitrarily

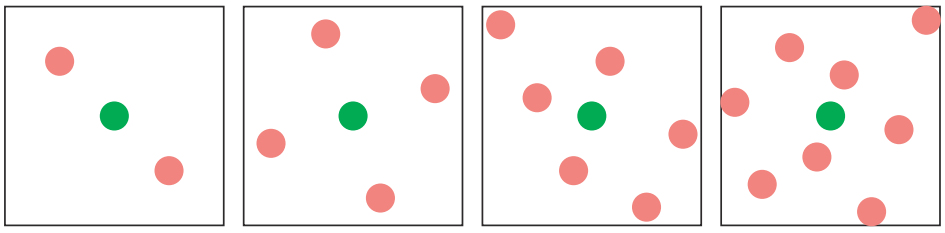

Figure 5.28. MSAA sampling patterns for AMD and NVIDIA graphics accelerators. The green square is the location of the shading sample, and the red squares are the positional samples computed and saved. From left to right: 2×, 4×, 6× (AMD), and 8× (NVIDIA) sampling. *(Generated by the D3D FSAA Viewer.)*

small on the screen, meaning that no sampling rate can ever perfectly capture them. If these tiny objects or features form a pattern, sampling at constant intervals can result in Moiré fringes and other interference patterns. The grid pattern used in supersampling is particularly likely to alias.

One solution is to use *stochastic sampling*, which gives a more randomized pattern. Patterns such as those in Figure 5.28 certainly qualify. Imagine a fine-toothed comb at a distance, with a few teeth covering each pixel. A regular pattern can give severe artifacts as the sampling pattern goes in and out of phase with the tooth frequency. Having a less ordered sampling pattern can break up these patterns. The randomization tends to replace repetitive aliasing effects with noise, to which the human visual system is much more forgiving [1413]. A pattern with less structure helps, but it can still exhibit aliasing when repeated pixel to pixel. One solution is use a different sampling pattern at each pixel, or to change each sampling location over time. *Interleaved sampling*indexsampling!interleaved, where each pixel of a set has a different sampling pattern, has occasionally been supported in hardware over the past decades. For example, ATI's SMOOTHVISION allowed up to 16 samples per pixel and up to 16 different user-defined sampling patterns that could be intermingled in a repeating pattern (e.g., in a 4×4 pixel tile). Molnar [1234], as well as Keller and Heidrich [880], found that using interleaved stochastic sampling minimizes the aliasing artifacts formed when using the same pattern for every pixel.

A few other GPU-supported algorithms are worth noting. One real-time antialiasing scheme that lets samples affect more than one pixel is NVIDIA's older Quincunx method [365]. "Quincunx" means an arrangement of five objects, four in a square and the fifth in the center, such as the pattern of five dots on a six-sided die. Quincunx multisampling antialiasing uses this pattern, putting the four outer samples at the corners of the pixel. See Figure 5.25. Each corner sample value is distributed to its four neighboring pixels. Instead of weighting each sample equally (as most other real-time schemes do), the center sample is given a weight of $\frac{1}{2}$, and each corner sample has a weight of $\frac{1}{8}$. Because of this sharing, an average of only two samples are needed per pixel, and the results are considerably better than two-sample FSAA methods [1678]. This pattern approximates a two-dimensional tent filter, which, as discussed in the previous section, is superior to the box filter.

Quincunx sampling can also be applied to temporal antialiasing by using a single sample per pixel [836, 1677]. Each frame is offset half a pixel in each axis from the frame before, with the offset direction alternating between frames. The previous frame provides the pixel corner samples, and bilinear interpolation is used to rapidly compute the contribution per pixel. The result is averaged with the current frame. Equal weighting of each frame means there are no shimmer artifacts for a static view. The issue of aligning moving objects is still present, but the scheme itself is simple to code and gives a much better look while using only one sample per pixel per frame.

When used in a single frame, Quincunx has a low cost of only two samples by sharing samples at the pixel boundaries. The RGSS pattern is better at capturing more gradations of nearly horizontal and vertical edges. First developed for mobile

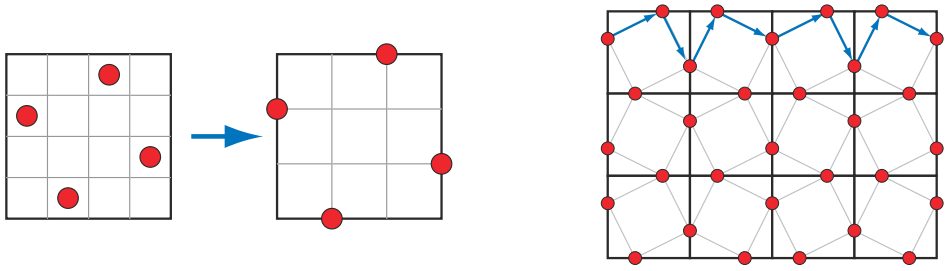

Figure 5.29. To the left, the RGSS sampling pattern is shown. This costs four samples per pixel. By moving these locations out to the pixel edges, sample sharing can occur across edges. However, for this to work out, every other pixel must have a reflected sample pattern, as shown on the right. The resulting sample pattern is called FLIPQUAD and costs two samples per pixel.

graphics, the FLIPQUAD pattern combines both of these desirable features [22]. Its advantages are that the cost is only two samples per pixel, and the quality is similar to RGSS (which costs four samples per pixel). This sampling pattern is shown in Figure 5.29. Other inexpensive sampling patterns that exploit sample sharing are explored by Hasselgren et al. [677].

Like Quincunx, the two-sample FLIPQUAD pattern can also be used with temporal antialiasing and spread over two frames. Drobot [382, 383, 1154] tackles the question of which two-sample pattern is best in his *hybrid reconstruction antialiasing* (HRAA) work. He explores different sampling patterns for temporal antialiasing, finding the FLIPQUAD pattern to be the best of the five tested. A checkerboard pattern has also seen use with temporal antialiasing. El Mansouri [415] discusses using two-sample MSAA to create a checkerboard render to reduce shader costs while addressing aliasing issues. Jimenez [836] uses SMAA, temporal antialiasing, and a variety of other techniques to provide a solution where antialiasing quality can be changed in response to rendering engine load. Carpentier and Ishiyama [231] sample on edges, rotating the sampling grid by 45°. They combine this temporal antialiasing scheme with FXAA (discussed later) to efficiently render on higher-resolution displays.

Morphological Methods

Aliasing often results from edges, such as those formed by geometry, sharp shadows, or bright highlights. The knowledge that aliasing has a structure associated with it can be exploited to give a better antialiased result. In 2009 Reshetov [1483] presented an algorithm along these lines, calling it *morphological antialiasing* (MLAA). "Morphological" means "relating to structure or shape." Earlier work had been done in this area [830], as far back as 1983 by Bloomenthal [170]. Reshetov's paper reinvigorated research into alternatives to multisampling approaches, emphasizing searching for and reconstructing edges [1486].

This form of antialiasing is performed as a post-process. That is, rendering is done in the usual fashion, then the results are fed to a process that generates the

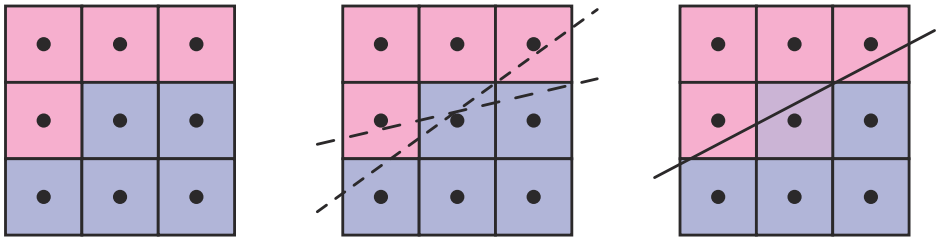

Figure 5.30. Morphological antialiasing. On the left is the aliased image. The goal is to determine the likely orientation of the edge that formed it. In the middle, the algorithm notes the likelihood of an edge by examining neighbors. Given the samples, two possible edge locations are shown. On the right, a best-guess edge is used to blend neighboring colors into the center pixel in proportion to the estimated coverage. This process is repeated for every pixel in the image.

antialiased result. A wide range of techniques have been developed since 2009. Those that rely on additional buffers such as depths and normals can provide better results, such as *subpixel reconstruction antialiasing* (SRAA) [43, 829], but are then applicable for antialiasing only geometric edges. Analytical approaches, such as *geometry buffer antialiasing* (GBAA) and *distance-to-edge antialiasing* (DEAA), have the renderer compute additional information about where triangle edges are located, e.g., how far the edge is from the center of the pixel [829].

The most general schemes need only the color buffer, meaning they can also improve edges from shadows, highlights, or various previously applied post-processing techniques, such as silhouette edge rendering (Section 15.2.3). For example, *directionally localized antialiasing* (DLAA) [52, 829] is based on the observation that an edge which is nearly vertical should be blurred horizontally, and likewise nearly horizontal edges should be blurred vertically with their neighbors.

More elaborate forms of edge detection attempt to find pixels likely to contain an edge at any angle and determine its coverage. The neighborhoods around potential edges are examined, with the goal of reconstructing as possible where the original edge was located. The edge's effect on the pixel can then be used to blend in neighboring pixels' colors. See Figure 5.30 for a conceptual view of the process.

Iourcha et al. [798] improve edge-finding by examine the MSAA samples in pixels to compute a better result. Note that edge prediction and blending can give a higher-precision result than sample-based algorithms. For example, a technique that uses four samples per pixel can give only five levels of blending for an object's edge: no samples covered, one covered, two, three, and four. The estimated edge location can have more locations and so provide better results.

There are several ways image-based algorithms can go astray. First, the edge may not be detected if the color difference between two objects is lower than the algorithm's threshold. Pixels where there are three or more distinct surfaces overlapping are difficult to interpret. Surfaces with high-contrast or high-frequency elements, where the

color is changing rapidly from pixel to pixel, can cause algorithms to miss edges. In particular, text quality usually suffers when morphological antialiasing is applied to it. Object corners can be a challenge, with some algorithms giving them a rounded appearance. Curved lines can also be adversely affected by the assumption that edges are straight. A single pixel change can cause a large shift in how the edge is reconstructed, which can create noticeable artifacts frame to frame. One approach to ameliorate this problem is to use MSAA coverage masks to improve edge determination [1484].

Morphological antialiasing schemes use only the information that is provided. For example, an object thinner than a pixel in width, such as an electrical wire or rope, will have gaps on the screen wherever it does not happen to cover the center location of a pixel. Taking more samples can improve the quality in such situations; image-based antialiasing alone cannot. In addition, execution time can be variable depending on what content is viewed. For example, a view of a field of grass can take three times as long to antialias as a view of the sky [231].

All this said, image-based methods can provide antialiasing support for modest memory and processing costs, so they are used in many applications. The color-only versions are also decoupled from the rendering pipeline, making them easy to modify or disable, and can even be exposed as GPU driver options. The two most popular algorithms are *fast approximate antialiasing* (FXAA) [1079, 1080, 1084], and *subpixel morphological antialiasing* (SMAA) [828, 830, 834], in part because both provide solid (and free) source code implementations for a variety of machines. Both algorithms use color-only input, with SMAA having the advantage of being able to access MSAA samples. Each has its own variety of settings available, trading off between speed and quality. Costs are generally in the range of 1 to 2 milliseconds per frame, mainly because that is what video games are willing to spend. Finally, both algorithms can also take advantage of temporal antialiasing [1812]. Jimenez [836] presents an improved SMAA implementation, faster than FXAA, and describes a temporal antialiasing scheme. To conclude, we recommend the reader to the wide-ranging review by Reshetov and Jimenez [1486] of morphological techniques and their use in video games.

5.5 Transparency, Alpha, and Compositing

There are many different ways in which semitransparent objects can allow light to pass through them. For rendering algorithms, these can be roughly divided into light-based and view-based effects. Light-based effects are those in which the object causes light to be attenuated or diverted, causing other objects in the scene to be lit and rendered differently. View-based effects are those in which the semitransparent object itself is being rendered.

In this section we will deal with the simplest form of view-based transparency, in which the semitransparent object acts as an attenuator of the colors of the objects behind it. More elaborate view- and light-based effects such as frosted glass, the bending

of light (refraction), attenuation of light due to the thickness of the transparent object, and reflectivity and transmission changes due to the viewing angle are discussed in later chapters.

One method for giving the illusion of transparency is called *screen-door transparency* [1244]. The idea is to render the transparent triangle with a pixel-aligned checkerboard fill pattern. That is, every other pixel of the triangle is rendered, thereby leaving the object behind it partially visible. Usually the pixels on the screen are close enough together that the checkerboard pattern itself is not visible. A major drawback of this method is that often only one transparent object can be convincingly rendered on one area of the screen. For example, if a transparent red object and transparent green object are rendered atop a blue object, only two of the three colors can appear on the checkerboard pattern. Also, the 50% checkerboard is limiting. Other larger pixel masks could be used to give other percentages, but these tend to create detectable patterns [1245].

That said, one advantage of this technique is its simplicity. Transparent objects can be rendered at any time, in any order, and no special hardware is needed. The transparency problem goes away by making all objects opaque at the pixels they cover. This same idea is used for antialiasing edges of cutout textures, but at a subpixel level, using a feature called *alpha to coverage* (Section 6.6).

Introduced by Enderton et al. [423], *stochastic transparency* uses subpixel screen-door masks combined with stochastic sampling. A reasonable, though noisy, image is created by using random stipple patterns to represent the alpha coverage of the fragment. See Figure 5.31. A large number of samples per pixel is needed for the result to look reasonable, as well as a sizable amount of memory for all the subpixel samples. What is appealing is that no blending is needed, and antialiasing, transparency, and any other phenomena that creates partially covered pixels are covered by a single mechanism.

Most transparency algorithms blend the transparent object's color with the color of the object behind it. For this, the concept of *alpha blending* is needed [199, 387, 1429]. When an object is rendered on the screen, an RGB color and a z-buffer depth are associated with each pixel. Another component, called alpha (α), can also be defined for each pixel the object covers. Alpha is a value describing the degree of opacity and coverage of an object fragment for a given pixel. An alpha of 1.0 means the object is opaque and entirely covers the pixel's area of interest; 0.0 means the pixel is not obscured at all, i.e., the fragment is entirely transparent.

A pixel's alpha can represent either opacity, coverage, or both, depending on the circumstances. For example, the edge of a soap bubble may cover three-quarters of the pixel, 0.75, and may be nearly transparent, letting nine-tenths of the light through to the eye, so it is one-tenth opaque, 0.1. Its alpha would then be $0.75 \times 0.1 = 0.075$. However, if we were using MSAA or similar antialiasing schemes, the coverage would be taken into account by the samples themselves. Three-quarters of the samples would be affected by the soap bubble. At each of these samples we would then use the 0.1 opacity value as the alpha.

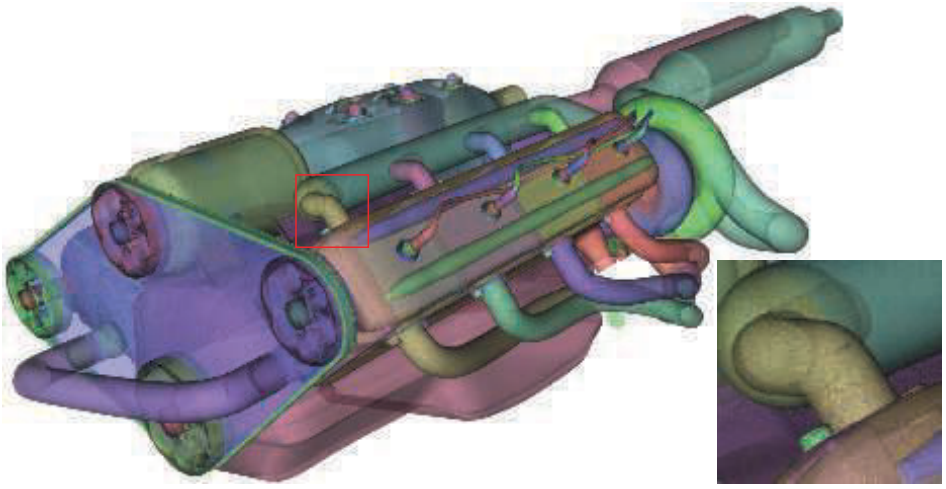

Figure 5.31. Stochastic transparency. The noise produced is displayed in the magnified area. *(Images from NVIDIA SDK 11 [1301] samples, courtesy of NVIDIA Corporation.)*

5.5.1 Blending Order

To make an object appear transparent, it is rendered on top of the existing scene with an alpha of less than 1.0. Each pixel covered by the object will receive a resulting RGBα (also called RGBA) from the pixel shader. Blending this fragment's value with the original pixel color is usually done using the **over** operator, as follows:

$$\mathbf{c}_o = \alpha_s \mathbf{c}_s + (1 - \alpha_s)\mathbf{c}_d \quad [\textbf{over operator}], \tag{5.24}$$

where $\mathbf{c}_s$ is the color of the transparent object (called the *source*), α_s is the object's alpha, $\mathbf{c}_d$ is the pixel color before blending (called the *destination*), and $\mathbf{c}_o$ is the resulting color due to placing the transparent object **over** the existing scene. In the case of the rendering pipeline sending in $\mathbf{c}_s$ and α_s, the pixel's original color $\mathbf{c}_d$ gets replaced by the result $\mathbf{c}_o$. If the incoming RGBα is, in fact, opaque ($\alpha_s = 1.0$), the equation simplifies to the full replacement of the pixel's color by the object's color.

Example: Blending. A red semitransparent object is rendered onto a blue background. Say that at some pixel the RGB shade of the object is $(0.9, 0.2, 0.1)$, the background is $(0.1, 0.1, 0.9)$, and the object's opacity is set at 0.6. The blend of these two colors is then

$$0.6(0.9, 0.2, 0.1) + (1 - 0.6)(0.1, 0.1, 0.9),$$

which gives a color of $(0.58, 0.16, 0.42)$. □

The **over** operator gives a semitransparent look to the object being rendered. Transparency done this way works, in the sense that we perceive something as transparent whenever the objects behind can be seen through it [754]. Using **over** simulates

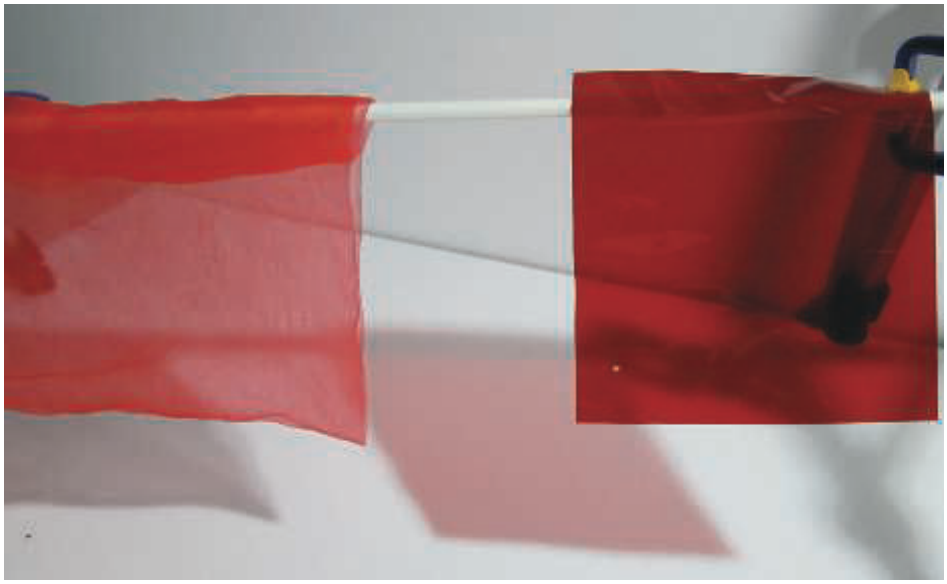

Figure 5.32. A red gauzy square of fabric and a red plastic filter, giving different transparency effects. Note how the shadows also differ. *(Photograph courtesy of Morgan McGuire.)*

the real-world effect of a gauzy fabric. The view of the objects behind the fabric are partially obscured—the fabric's threads are opaque. In practice, loose fabric has an alpha coverage that varies with angle [386]. Our point here is that alpha simulates how much the material covers the pixel.

The **over** operator is less convincing simulating other transparent effects, most notably viewing through colored glass or plastic. A red filter held in front of a blue object in the real world usually makes the blue object look dark, as this object reflects little light that can pass through the red filter. See Figure 5.32. When **over** is used for blending, the result is a portion of the red and the blue added together. It would be better to multiply the two colors together, as well as adding in any reflection off the transparent object itself. This type of physical transmittance is discussed in Sections 14.5.1 and 14.5.2.

Of the basic blend stage operators, **over** is the one commonly used for a transparency effect [199, 1429]. Another operation that sees some use is *additive blending*, where pixel values are simply summed. That is,

$$\mathbf{c}_o = \alpha_s \mathbf{c}_s + \mathbf{c}_d. \tag{5.25}$$

This blending mode can work well for glowing effects such as lightning or sparks that do not attenuate the pixels behind but instead only brighten them [1813]. However, this mode does not look correct for transparency, as the opaque surfaces do not appear

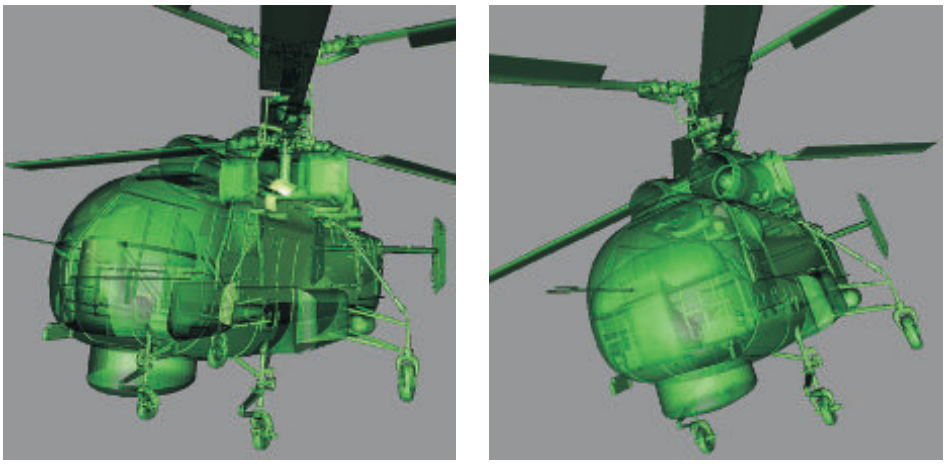

Figure 5.33. On the left the model is rendered with transparency using the z-buffer. Rendering the mesh in an arbitrary order creates serious errors. On the right, depth peeling provides the correct appearance, at the cost of additional passes. *(Images courtesy of NVIDIA Corporation.)*

filtered [1192]. For several layered semitransparent surfaces, such as smoke or fire, additive blending has the effect of saturating the colors of the phenomenon [1273].

To render transparent objects properly, we need to draw them after the opaque objects. This is done by rendering all opaque objects first with blending off, then rendering the transparent objects with **over** turned on. In theory we could always have **over** on, since an opaque alpha of 1.0 would give the source color and hide the destination color, but doing so is more expensive, for no real gain.

A limitation of the z-buffer is that only one object is stored per pixel. If several transparent objects overlap the same pixel, the z-buffer alone cannot hold and later resolve the effect of all the visible objects. When using **over** the transparent surfaces at any given pixel generally need to be rendered in back-to-front order. Not doing so can give incorrect perceptual cues. One way to achieve this ordering is to sort individual objects by, say, the distance of their centroids along the view direction. This rough sorting can work reasonably well, but has a number of problems under various circumstances. First, the order is just an approximation, so objects classified as more distant may be in front of objects considered nearer. Objects that interpenetrate are impossible to resolve on a per-mesh basis for all view angles, short of breaking each mesh into separate pieces. See the left image in Figure 5.33 for an example. Even a single mesh with concavities can exhibit sorting problems for view directions where it overlaps itself on the screen.

Nonetheless, because of its simplicity and speed, as well as needing no additional memory or special GPU support, performing a rough sort for transparency is still commonly used. If implemented, it is usually best to turn off z-depth replacement

when performing transparency. That is, the z-buffer is still tested normally, but surviving surfaces do not change the z-depth stored; the closest opaque surface's depth is left intact. In this way, all transparent objects will at least appear in some form, versus suddenly appearing or disappearing when a camera rotation changes the sort order. Other techniques can also help improve the appearance, such as drawing each transparent mesh twice as you go, first rendering backfaces and then frontfaces [1192, 1255].

The **over** equation can also be modified so that blending front to back gives the same result. This blending mode is called the **under** operator:

$$\begin{aligned} \mathbf{c}_o &= \alpha_d \mathbf{c}_d + (1-\alpha_d)\alpha_s \mathbf{c}_s \quad [\textbf{under operator}], \\ \mathbf{a}_o &= \alpha_s(1-\alpha_d) + \alpha_d = \alpha_s - \alpha_s\alpha_d + \alpha_d. \end{aligned} \tag{5.26}$$

Note that **under** requires the destination to maintain an alpha value, which **over** does not. In other words, the destination—the closer transparent surface being blended under—is not opaque and so needs to have an alpha value. The **under** formulation is like **over**, but with source and destination swapped. Also, notice that the formula for computing alpha is order-independent, in that the source and destination alphas can be swapped, with the same final alpha being the result.

The equation for alpha comes from considering the fragment's alphas as coverages. Porter and Duff [1429] note that since we do not know the shape of the coverage area for either fragment, we assume that each fragment covers the other in proportion to its alpha. For example, if $\alpha_s = 0.7$, the pixel is somehow divided into two areas, with 0.7 covered by the source fragment and 0.3 not. Barring any other knowledge, the destination fragment covering, say, $\alpha_d = 0.6$ will be proportionally overlapped by the source fragment. This formula has a geometric interpretation, shown in Figure 5.34.

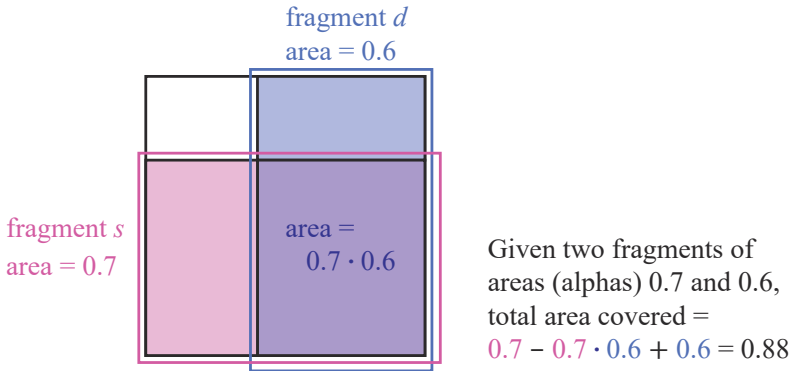

Figure 5.34. A pixel and two fragments, s and d. By aligning the two fragments along different axes, each fragment covers a proportional amount of the other, i.e., they are uncorrelated. The area covered by the two fragments is equivalent to the **under** output alpha value $\alpha_s - \alpha_s\alpha_d + \alpha_d$. This translates to adding the two areas, then subtracting the area where they overlap.

Figure 5.35. Each depth peel pass draws one of the transparent layers. On the left is the first pass, showing the layer directly visible to the eye. The second layer, shown in the middle, displays the second-closest transparent surface at each pixel, in this case the backfaces of objects. The third layer, on the right, is the set of third-closest transparent surfaces. Final results can be found in Figure 14.33 on page 624. *(Images courtesy of Louis Bavoil.)*

5.5.2 Order-Independent Transparency

The **under** equations are used by drawing all transparent objects to a separate color buffer, then merging this color buffer atop the opaque view of the scene using **over**. Another use of the **under** operator is for performing an *order-independent transparency* (OIT) algorithm known as *depth peeling* [449, 1115]. Order-independent means that the application does not need to perform sorting. The idea behind depth peeling is to use two z-buffers and multiple passes. First, a rendering pass is made so that all surfaces' z-depths, including transparent surfaces, are in the first z-buffer. In the second pass all transparent objects are rendered. If the z-depth of an object matches the value in the first z-buffer, we know this is the closest transparent object and save its RGBα to a separate color buffer. We also "peel" this layer away by saving the z-depth of whichever transparent object, if any, is beyond the first z-depth and is closest. This z-depth is the distance of the second-closest transparent object. Successive passes continue to peel and add transparent layers using **under**. We stop after some number of passes and then blend the transparent image atop the opaque image. See Figure 5.35.

Several variants on this scheme have been developed. For example, Thibieroz [1763] gives an algorithm that works back to front, which has the advantage of being able to blend the transparent values immediately, meaning that no separate alpha channel is needed. One problem with depth peeling is knowing how many passes are sufficient to capture all the transparent layers. One hardware solution is to provide a pixel draw counter, which tells how many pixels were written during rendering; when no pixels are rendered by a pass, rendering is done. The advantage of using **under** is that the most important transparent layers—those the eye first sees—are rendered early on. Each transparent surface always increases the alpha value of the pixel it covers. If the

alpha value for a pixel nears 1.0, the blended contributions have made the pixel almost opaque, and so more distant objects will have a negligible effect [394]. Front-to-back peeling can be cut short when the number of pixels rendered by a pass falls below some minimum, or a fixed number of passes can be specified. This does not work as well with back-to-front peeling, as the closest (and usually most important) layers are drawn last and so may be lost by early termination.

While depth peeling is effective, it can be slow, as each layer peeled is a separate rendering pass of all transparent objects. Bavoil and Myers [118] presented dual depth peeling, where two depth peel layers, the closest and the farthest remaining, are stripped off in each pass, thus cutting the number of rendering passes in half. Liu et al. [1056] explore a bucket sort method that captures up to 32 layers in a single pass. One drawback of this type of approach is that it needs considerable memory to keep a sorted order for all layers. Antialiasing via MSAA or similar would increase the costs astronomically.

The problem of blending transparent objects together properly at interactive rates is not one in which we are lacking algorithms, it is one of efficiently mapping those algorithms to the GPU. In 1984 Carpenter presented the *A*-buffer [230], another form of multisampling. In the *A*-buffer, each triangle rendered creates a *coverage mask* for each screen grid cell it fully or partially covers. Each pixel stores a list of all relevant fragments. Opaque fragments can cull out fragments behind them, similar to the z-buffer. All the fragments are stored for transparent surfaces. Once all lists are formed, a final result is produced by walking through the fragments and resolving each sample.

The idea of creating linked lists of fragments on the GPU was made possible through new functionality exposed in DirectX 11 [611, 1765]. The features used include unordered access views (UAVs) and atomic operations, described in Section 3.8. Antialiasing via MSAA is enabled by the ability to access the coverage mask and to evaluate the pixel shader at every sample. This algorithm works by rasterizing each transparent surface and inserting the fragments generated in a long array. Along with the colors and depths, a separate pointer structure is generated that links each fragment to the previous fragment stored for the pixel. A separate pass is then performed, where a screen-filling quadrilateral is rendered so that a pixel shader is evaluated at every pixel. This shader retrieves all the transparent fragments at each pixel by following the links. Each fragment retrieved is sorted in turn with the previous fragments. This sorted list is then blended back to front to give the final pixel color. Because blending is performed by the pixel shader, different blend modes can be specified per pixel, if desired. Continuing evolution of the GPU and APIs have improved performance by reducing the cost of using atomic operators [914].

The *A*-buffer has the advantage that only the fragments needed for each pixel are allocated, as does the linked list implementation on the GPU. This in a sense can also be a disadvantage, as the amount of storage required is not known before rendering of a frame begins. A scene with hair, smoke, or other objects with a potential for many overlapping transparent surfaces can produce a huge number of fragments.

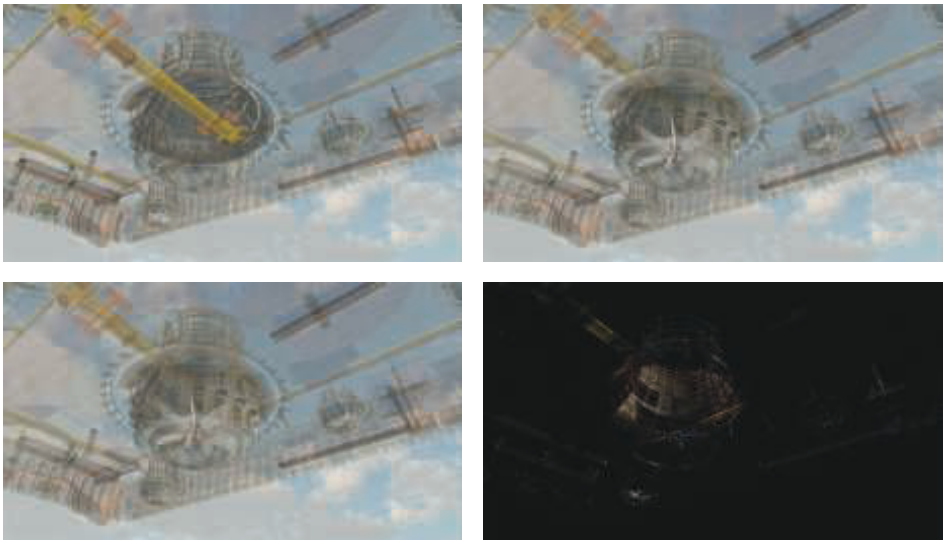

Figure 5.36. In the upper left, traditional back-to-front alpha blending is performed, leading to rendering errors due to incorrect sort order. In the upper right, the *A*-buffer is used to give a perfect, non-interactive result. The lower left presents the rendering with multi-layer alpha blending. The lower right shows the differences between the *A*-buffer and multi-layer images, multiplied by 4 for visibility [1532]. *(Images courtesy of Marco Salvi and Karthik Vaidyanathan, Intel Corporation.)*

Andersson [46] notes that, for complex game scenes, up to 50 transparent meshes of objects such as foliage and up to 200 semitransparent particles may overlap.

GPUs normally have memory resources such as buffers and arrays allocated in advance, and linked-list approaches are no exception. Users need to decide how much memory is enough, and running out of memory causes noticeable artifacts. Salvi and Vaidyanathan [1532] present an approach tackling this problem, *multi-layer alpha blending*, using a GPU feature introduced by Intel called pixel synchronization. See Figure 5.36. This capability provides programmable blending with less overhead than atomics. Their approach reformulates storage and blending so that it gracefully degrades if memory runs out. A rough sort order can benefit their scheme. DirectX 11.3 introduced rasterizer order views (Section 3.8), a type of buffer that allows this transparency method to be implemented on any GPU supporting this feature [327, 328]. Mobile devices have a similar technology called *tile local storage* that permits them to implement multi-layer alpha blending [153]. Such mechanisms have a performance cost, however, so this type of algorithm can be expensive [1931].

This approach builds on the idea of the k-buffer, introduced by Bavoil et al. [115], where the first few visible layers are saved and sorted as possible, with deeper layers discarded and merged as possible. Maule et al. [1142] use a k-buffer and account for these more distant deep layers by using *weighted averaging*. Weighted sum [1202] and

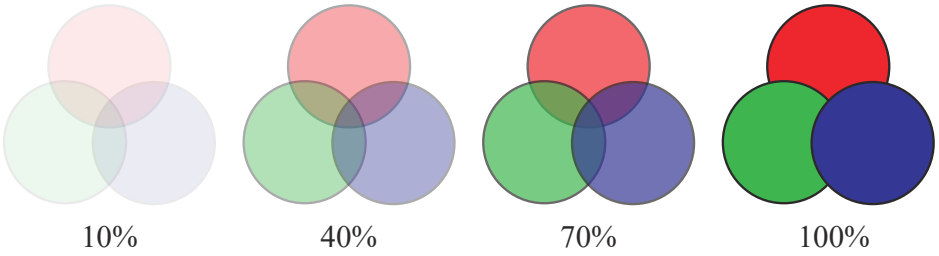

Figure 5.37. The object order becomes more important as opacity increases. *(Images after Dunn [394].)*

weighted average [118] transparency techniques are order-independent, are single-pass, and run on almost every GPU. The problem is that they do not take into account the ordering of the objects. So, for example, using alpha to represent coverage, a gauzy red scarf atop a gauzy blue scarf gives a violet color, versus properly seeing a red scarf with a little blue showing through. While nearly opaque objects give poor results, this class of algorithms is useful for visualization and works well for highly transparent surfaces and particles. See Figure 5.37.

In weighted sum transparency the formula is

$$\mathbf{c}_o = \sum_{i=1}^{n} (\alpha_i \mathbf{c}_i) + \mathbf{c}_d (1 - \sum_{i=1}^{n} \alpha_i), \tag{5.27}$$

where n is the number of transparent surfaces, $\mathbf{c}_i$ and α_i represent the set of transparency values, and $\mathbf{c}_d$ is the color of the opaque portion of the scene. The two sums are accumulated and stored separately as transparent surfaces are rendered, and at the end of the transparency pass, the equation is evaluated at each pixel. Problems with this method are that the first sum saturates, i.e., generates color values greater than $(1.0, 1.0, 1.0)$, and that the background color can have a negative effect, since the sum of the alphas can surpass 1.0.

The weighted average equation is usually preferred because it avoids these problems:

$$\begin{aligned} \mathbf{c}_{\text{sum}} &= \sum_{i=1}^{n} (\alpha_i \mathbf{c}_i), \quad \alpha_{\text{sum}} = \sum_{i=1}^{n} \alpha_i, \\ \mathbf{c}_{\text{wavg}} &= \frac{\mathbf{c}_{\text{sum}}}{\alpha_{\text{sum}}}, \quad \alpha_{\text{avg}} = \frac{\alpha_{\text{sum}}}{n}, \\ u &= (1 - \alpha_{\text{avg}})^n, \\ \mathbf{c}_o &= (1 - u)\mathbf{c}_{\text{wavg}} + u\mathbf{c}_d. \end{aligned} \tag{5.28}$$

The first line represents the results in the two separate buffers generated during transparency rendering. Each surface contributing to $\mathbf{c}_{\text{sum}}$ is given an influence weighted by

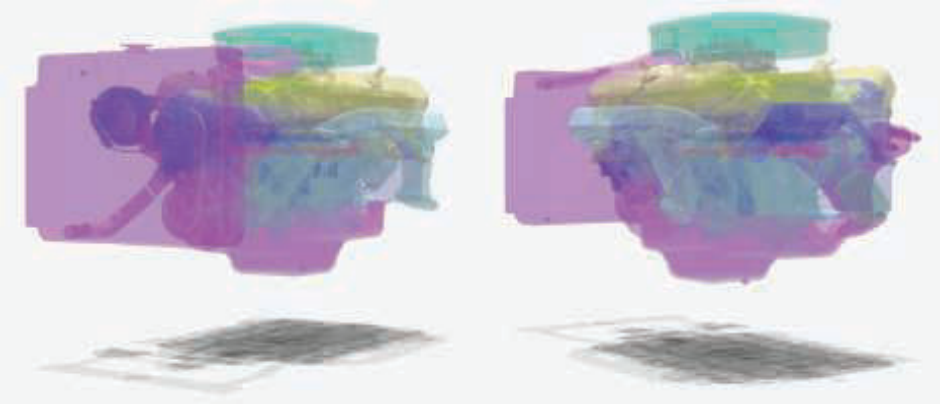

Figure 5.38. Two different camera locations viewing the same engine model, both rendered with weighted blended order-independent transparency. Weighting by distance helps clarify which surfaces are closer to the viewer [1185]. *(Images courtesy of Morgan McGuire.)*

its alpha; nearly opaque surfaces contribute more of their color, and nearly transparent surfaces have little influence. By dividing $\mathbf{c}_{\text{sum}}$ by α_{sum} we get a weighted average transparency color. The value α_{avg} is the average of all alpha values. The value u is the estimated visibility of the destination (the opaque scene) after this average alpha is applied n times, for n transparent surfaces. The final line is effectively the **over** operator, with $(1 - u)$ representing the source's alpha.

One limitation with weighted average is that, for identical alphas, it blends all colors equally, regardless of order. McGuire and Bavoil [1176, 1180] introduced weighted blended order-independent transparency to give a more convincing result. In their formulation, the distance to the surface also affects the weight, with closer surfaces given more influence. Also, rather than averaging the alphas, u is computed by multiplying the terms $(1 - \alpha_i)$ together and subtracting from one, giving the true alpha coverage of the set of surfaces. This method produces more visually convincing results, as seen in Figure 5.38.

A drawback is that objects close to one another in a large environment can have nearly equal weightings from distance, making the result little different than the weighted average. Also, as the camera's distance to the transparent objects changes, the depth weightings may then vary in effect, but this change is gradual.

McGuire and Mara [1181, 1185] extend this method to include a plausible transmission color effect. As noted earlier, all the transparency algorithms discussed in this section blend various colors instead of filtering them, mimicking pixel coverage. To give a color filter effect, the opaque scene is read by the pixel shader and each transparent surface multiplies the pixels it covers in this scene by its color, saving the result to a third buffer. This buffer, in which the opaque objects are now tinted by the transparent ones, is then used in place of the opaque scene when resolving the trans-

parency buffers. This method works because, unlike transparency due to coverage, colored transmission is order-independent.

There are yet other algorithms that use elements from several of the techniques presented here. For example, Wyman [1931] categorizes previous work by memory requirements, insertion and merge methods, whether alpha or geometric coverage is used, and how discarded fragments are treated. He presents two new methods found by looking for gaps in previous research. His stochastic layered alpha blending method uses k-buffers, weighted average, and stochastic transparency. His other algorithm is a variant on Salvi and Vaidyanathan's method, using coverage masks instead of alpha.

Given the wide variety of types of transparent content, rendering methods, and GPU capabilities, there is no perfect solution for rendering transparent objects. We refer the interested reader to Wyman's paper [1931] and Maule et al.'s more detailed survey [1141] of algorithms for interactive transparency. McGuire's presentation [1182] gives a wider view of the field, running through other related phenomena such as volumetric lighting, colored transmission, and refraction, which are discussed in greater depth later in this book.

5.5.3 Premultiplied Alphas and Compositing

The **over** operator is also used for blending together photographs or synthetic renderings of objects. This process is called *compositing* [199, 1662]. In such cases, the alpha value at each pixel is stored along with the RGB color value for the object. The image formed by the alpha channel is sometimes called the *matte*. It shows the silhouette shape of the object. See Figure 6.27 on page 203 for an example. This RGBα image can then be used to blend it with other such elements or against a background.

One way to use synthetic RGBα data is with *premultiplied alphas* (also known as *associated alphas*). That is, the RGB values are multiplied by the alpha value before being used. This makes the compositing **over** equation more efficient:

$$\mathbf{c}_o = \mathbf{c}'_s + (1 - \alpha_s)\mathbf{c}_d, \tag{5.29}$$

where $\mathbf{c}'_s$ is the premultiplied source channel, replacing $\alpha_s\mathbf{c}_s$ in Equation 5.25. Premultiplied alpha also makes it possible to use **over** and additive blending without changing the blend state, since the source color is now added in during blending [394]. Note that with premultiplied RGBα values, the RGB components are normally not greater than the alpha value, though they can be made so to create a particularly bright semitransparent value.

Rendering synthetic images dovetails naturally with premultiplied alphas. An antialiased opaque object rendered over a black background provides premultiplied values by default. Say a white $(1, 1, 1)$ triangle covers 40% of some pixel along its edge. With (extremely precise) antialiasing, the pixel value would be set to a gray of 0.4, i.e., we would save the color $(0.4, 0.4, 0.4)$ for this pixel. The alpha value, if

stored, would also be 0.4, since this is the area the triangle covered. The RGBα value would be $(0.4, 0.4, 0.4, 0.4)$, which is a premultiplied value.

Another way images are stored is with *unmultiplied alphas*, also known as *unassociated alphas* or even as the mind-bending term *nonpremultiplied alphas*. An unmultiplied alpha is just what it says: The RGB value is not multiplied by the alpha value. For the white triangle example, the unmultiplied color would be $(1, 1, 1, 0.4)$. This representation has the advantage of storing the triangle's original color, but this color always needs to be multiplied by the stored alpha before being display. It is best to use premultiplied data whenever filtering and blending is performed, as operations such as linear interpolation do not work correctly using unmultiplied alphas [108, 164]. Artifacts such as black fringes around the edges of objects can result [295, 648]. See the end of Section 6.6 for further discussion. Premultiplied alphas also allow cleaner theoretical treatment [1662].

For image-manipulation applications, an unassociated alpha is useful to mask a photograph without affecting the underlying image's original data. Also, an unassociated alpha means that the full precision range of the color channels can be used. That said, care must be taken to properly convert unmultiplied RGBα values to and from the linear space used for computer graphics computations. For example, no browsers do this properly, nor are they ever likely to do so, since the incorrect behavior is now expected [649]. Image file formats that support alpha include PNG (unassociated alpha only), OpenEXR (associated only), and TIFF (both types of alpha).

A concept related to the alpha channel is *chroma-keying* [199]. This is a term from video production, in which actors are filmed against a green or blue screen and blended with a background. In the film industry this process is called *green-screening* or *blue-screening*. The idea here is that a particular color hue (for film work) or precise value (for computer graphics) is designated to be considered transparent; the background is displayed whenever it is detected. This allows images to be given an outline shape by using just RGB colors; no alpha needs to be stored. One drawback of this scheme is that the object is either entirely opaque or transparent at any pixel, i.e., alpha is effectively only 1.0 or 0.0. As an example, the GIF format allows one color to be designated as transparent.

5.6 Display Encoding

When we calculate the effect of lighting, texturing, or other operations, the values used are assumed to be *linear*. Informally, this means that addition and multiplication work as expected. However, to avoid a variety of visual artifacts, display buffers and textures use nonlinear encodings that we must take into account. The short and sloppy answer is as follows: Take shader output colors in the range $[0, 1]$ and raise them by a power of $1/2.2$, performing what is called *gamma correction*. Do the opposite for incoming textures and colors. In most cases you can tell the GPU to do these things for you. This section explains the how and why of that quick summary.

We begin with the *cathode-ray tube* (CRT). In the early years of digital imaging, CRT displays were the norm. These devices exhibit a power law relationship between input voltage and display radiance. As the energy level applied to a pixel is increased, the radiance emitted does not grow linearly but (surprisingly) rises proportional to that level raised to a power greater than one. For example, imagine the power is 2. A pixel set to 50% will emit a quarter the amount of light, $0.5^2 = 0.25$, as a pixel that is set to 1.0 [607]. Although LCDs and other display technologies have different intrinsic tone response curves than CRTs, they are manufactured with conversion circuitry that causes them to mimic the CRT response.

This power function nearly matches the inverse of the lightness sensitivity of human vision [1431]. The consequence of this fortunate coincidence is that the encoding is roughly *perceptually uniform*. That is, the perceived difference between a pair of encoded values N and $N+1$ is roughly constant over the displayable range. Measured as *threshold contrast*, we can detect a difference in lightness of about 1% over a wide range of conditions. This near-optimal distribution of values minimizes *banding* artifacts when colors are stored in limited-precision display buffers (Section 23.6). The same benefit also applies to textures, which commonly use the same encoding.

The *display transfer function* describes the relationship between the digital values in the display buffer and the radiance levels emitted from the display. For this reason it is also called the *electrical optical transfer function* (EOTF). The display transfer function is part of the hardware, and there are different standards for computer monitors, televisions, and film projectors. There is also a standard transfer function for the other end of the process, image and video capture devices, called the *optical electric transfer function* (OETF) [672].

When encoding linear color values for display, our goal is to cancel out the effect of the display transfer function, so that whatever value we compute will emit a corresponding radiance level. For example, if our computed value is doubled, we want the output radiance to be doubled. To maintain this connection, we apply the inverse of the display transfer function to cancel out its nonlinear effect. This process of nullifying the display's response curve is also called *gamma correction*, for reasons that will become clear shortly. When decoding texture values, we need to apply the display transfer function to generate a linear value for use in shading. Figure 5.39 shows the use of decoding and encoding in the display process.

The standard transfer function for personal computer displays is defined by a color-space specification called *sRGB*. Most APIs controlling GPUs can be set to automatically apply the proper sRGB conversion when values are read from textures or written to the color buffer [491]. As discussed in Section 6.2.2, mipmap generation will also take sRGB encoding into account. Bilinear interpolation among texture values will work correctly, by first converting to linear values and then performing the interpolation. Alpha blending is done correctly by decoding the stored value back into linear values, blending in the new value, and then encoding the result.

It is important to apply the conversion at the final stage of rendering, when the values are written to the framebuffer for the display. If post-processing is applied after

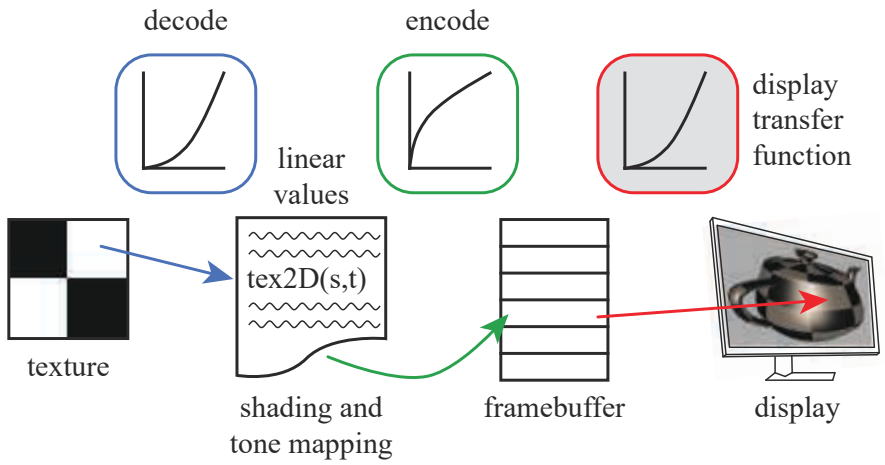

Figure 5.39. On the left, a PNG color texture is accessed by a GPU shader, and its nonlinearly encoded value is converted (blue) to a linear value. After shading and tone mapping (Section 8.2.2), the final computed value is encoded (green) and stored in the framebuffer. This value and the display transfer function determine the amount of radiance emitted (red). The green and red functions combined cancel out, so that the radiance emitted is proportional to the linear computed value.

display encoding, such effects will be computed on nonlinear values, which is usually incorrect and will often cause artifacts. Display encoding can be thought of as a form of compression, one that best preserves the value's perceptual effect [491]. A good way to think about this area is that there are linear values that we use to perform physical computations, and whenever we want to display results or access displayable images such as color textures, we need to move data to or from its display-encoded form, using the proper encode or decode transform.

If you do need to apply sRGB manually, there is a standard conversion equation or a few simplified versions that can be used. In practical terms the display is controlled by a number of bits per color channel, e.g., 8 for consumer-level monitors, giving a set of levels in the range $[0, 255]$. Here we express the display-encoded levels as a range $[0.0, 1.0]$, ignoring the number of bits. The linear values are also in the range $[0.0, 1.0]$, representing floating point numbers. We denote these linear values by x and the nonlinearly encoded values stored in the framebuffer by y. To convert linear values to sRGB nonlinear encoded values, we apply the inverse of the sRGB display transfer function:

$$y = f_{\text{sRGB}}^{-1}(x) = \begin{cases} 1.055x^{1/2.4} - 0.055, & \text{where } x > 0.0031308, \\ 12.92x, & \text{where } x \leq 0.0031308, \end{cases} \tag{5.30}$$

with x representing a channel of the linear RGB triplet. The equation is applied to each channel, and these three generated values drive the display. Be careful if you

apply conversion functions manually. One source of error is using an encoded color instead of its linear form, and another is decoding or encoding a color twice.

The bottom of the two transform expressions is a simple multiply, which arises from a need by digital hardware to make the transform perfectly invertible [1431]. The top expression, involving raising the value to a power, applies to almost the whole range $[0.0, 1.0]$ of input values x. With the offset and scale taken into account, this function closely approximates a simpler formula [491]:

$$y = f_{\text{display}}^{-1}(x) = x^{1/\gamma}, \tag{5.31}$$

with $\gamma = 2.2$. The Greek letter γ is the basis for the name "gamma correction."

Just as computed values must be encoded for display, images captured by still or video cameras must be converted to linear values before being used in calculations. Any color you see on a monitor or television has some display-encoded RGB triplet that you can obtain from a screen capture or color picker. These values are what are stored in file formats such as PNG, JPEG, and GIF, formats that can be directly sent to a framebuffer for display on the screen without conversion. In other words, whatever you see on the screen is by definition display-encoded data. Before using these colors in shading calculations, we must convert from this encoded form back to linear values. The sRGB transformation we need from display encoding to linear values is

$$x = f_{\text{sRGB}}(y) = \begin{cases} \left(\dfrac{y + 0.055}{1.055}\right)^{2.4}, & \text{where } y > 0.04045, \\ \dfrac{y}{12.92}, & \text{where } y \leq 0.04045, \end{cases} \tag{5.32}$$

with y representing a normalized displayed channel value, i.e., what is stored in an image or framebuffer, expressed as a value in the range $[0.0, 1.0]$. This decode function is the inverse of our previous sRGB formula. This means that if a texture is accessed by a shader and output without change, it will appear the same as before being processed, as expected. The decode function is the same as the display transfer function because the values stored in a texture have been encoded to display correctly. Instead of converting to give a linear-response display, we are converting to give linear values.

The simpler gamma display transfer function is the inverse of Equation 5.31:

$$x = f_{\text{display}}(y) = y^{\gamma}. \tag{5.33}$$

Sometimes you will see a conversion pair that is simpler still, particularly on mobile and browser apps [1666]:

$$\begin{aligned} y &= f_{\text{simpl}}^{-1}(x) = \sqrt{x}, \\ x &= f_{\text{simpl}}(y) = y^2; \end{aligned} \tag{5.34}$$

that is, take the square root of the linear value for conversion for display, and just multiply the value by itself for the inverse. While a rough approximation, this conversion is better than ignoring the problem altogether.

Figure 5.40. Two overlapping spotlights illuminating a plane. In the left image, gamma correction is not performed after adding the light values of 0.6 and 0.4. The addition is effectively performed on nonlinear values, causing errors. Note how the left light looks considerably brighter than the right, and the overlap appears unrealistically bright. In the right image, the values are gamma corrected after addition. The lights themselves are proportionally brighter, and they combine properly where they overlap.

If we do not pay attention to gamma, lower linear values will appear too dim on the screen. A related error is that the hue of some colors can shift if no gamma correction is performed. Say our $\gamma = 2.2$. We want to emit a radiance from the displayed pixel proportional to the linear, computed value, which means that we must raise the linear value to the $(1/2.2)$ power. A linear value of 0.1 gives 0.351, 0.2 gives 0.481, and 0.5 gives 0.730. If not encoded, these values used as is will cause the display to emit less radiance than needed. Note that 0.0 and 1.0 are always unchanged by any of these transforms. Before gamma correction was used, dark surface colors would often be artificially boosted by the person modeling the scene, folding in the inverse display transform.

Another problem with neglecting gamma correction is that shading computations that are correct for physically linear radiance values are performed on nonlinear values. An example of this can be seen in Figure 5.40.

Ignoring gamma correction also affects the quality of antialiased edges. For example, say a triangle edge covers four screen grid cells (Figure 5.41). The triangle's

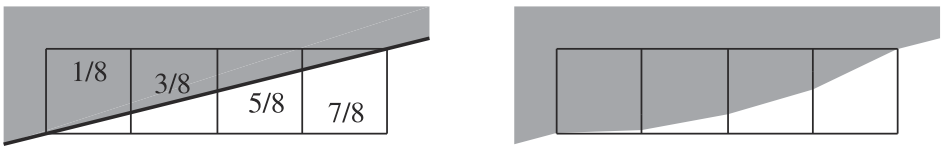

Figure 5.41. On the left, four pixels covered by the edge of a white triangle on a black (shown as gray) background, with true area coverage shown. If gamma correction is not performed, the darkening of midtones will cause the perception of the edge to be distorted, as seen on the right.

Figure 5.42. On the left, the set of antialiased lines are gamma-corrected; in the middle, the set is partially corrected; on the right, there is no gamma correction. *(Images courtesy of Scott R. Nelson.)*

normalized radiance is 1 (white); the background's is 0 (black). Left to right, the cells are covered $\frac{1}{8}$, $\frac{3}{8}$, $\frac{5}{8}$, and $\frac{7}{8}$. So, if we are using a box filter, we want to represent the normalized linear radiance of the pixels as 0.125, 0.375, 0.625, and 0.875. The correct approach is to perform antialiasing on linear values, applying the encoding function to the four resulting values. If this is not done, the represented radiance for the pixels will be too dark, resulting in a perceived deformation of the edge as seen in the right side of the figure. This artifact is called *roping*, because the edge looks somewhat like a twisted rope [167, 1265]. Figure 5.42 shows this effect.

The sRGB standard was created in 1996 and has become the norm for most computer monitors. However, display technology has evolved since that time. Monitors that are brighter and that can display a wider range of colors have been developed. Color display and brightness are discussed in Section 8.1.3, and display encoding for high dynamic range displays is presented in Section 8.2.1. Hart's article [672] is a particularly thorough source for more information about advanced displays.

Further Reading and Resources

Pharr et al. [1413] discuss sampling patterns and antialiasing in more depth. Teschner's course notes [1758] show various sampling pattern generation methods. Drobot [382, 383] runs through previous research on real-time antialiasing, explaining the attributes and performance of a variety of techniques. Information on a wide variety of morphological antialiasing methods can be found in the notes for the related SIGGRAPH course [829]. Reshetov and Jimenez [1486] provide an updated retrospective of morphological and related temporal antialiasing work used in games.

For transparency research we again refer the interested reader to McGuire's presentation [1182] and Wyman's work [1931]. Blinn's article "What Is a Pixel?" [169] provides an excellent tour of several areas of computer graphics while discussing different definitions. Blinn's *Dirty Pixels* and *Notation, Notation, Notation* books [166, 168] include some introductory articles on filtering and antialiasing, as well as articles on

alpha, compositing, and gamma correction. Jimenez's presentation [836] gives a detailed treatment of state-of-the-art techniques used for antialiasing.

Gritz and d'Eon [607] have an excellent summary of gamma correction issues. Poynton's book [1431] gives solid coverage of gamma correction in various media, as well as other color-related topics. Selan's white paper [1602] is a newer source, explaining display encoding and its use in the film industry, along with much other related information.

Chapter 6
Texturing

"All it takes is for the rendered image to look right."
—Jim Blinn

A surface's texture is its look and feel—just think of the texture of an oil painting. In computer graphics, texturing is a process that takes a surface and modifies its appearance at each location using some image, function, or other data source. As an example, instead of precisely representing the geometry of a brick wall, a color image of a brick wall is applied to a rectangle, consisting of two triangles. When the rectangle is viewed, the color image appears where the rectangle is located. Unless the viewer gets close to the wall, the lack of geometric detail will not be noticeable.

However, some textured brick walls can be unconvincing for reasons other than lack of geometry. For example, if the mortar is supposed to be matte, whereas the bricks are glossy, the viewer will notice that the roughness is the same for both materials. To produce a more convincing experience, a second image texture can be applied to the surface. Instead of changing the surface's color, this texture changes the wall's roughness, depending on location on the surface. Now the bricks and mortar have a color from the image texture and a roughness value from this new texture.

The viewer may see that now all the bricks are glossy and the mortar is not, but notice that each brick face appears to be perfectly flat. This does not look right, as bricks normally have some irregularity to their surfaces. By applying *bump mapping*, the shading normals of the bricks may be varied so that when they are rendered, they do not appear to be perfectly smooth. This sort of texture wobbles the direction of the rectangle's original surface normal for purposes of computing lighting.

From a shallow viewing angle, this illusion of bumpiness can break down. The bricks should stick out above the mortar, obscuring it from view. Even from a straight-on view, the bricks should cast shadows onto the mortar. *Parallax mapping* uses a texture to appear to deform a flat surface when rendering it, and *parallax occlusion mapping* casts rays against a heightfield texture for improved realism. *Displacement mapping* truly displaces the surface by modifying triangle heights forming the model. Figure 6.1 shows an example with color texturing and bump mapping.

Figure 6.1. Texturing. Color and bump maps were applied to this fish to increase its visual level of detail. *(Image courtesy of Elinor Quittner.)*

These are examples of the types of problems that can be solved with textures, using more and more elaborate algorithms. In this chapter, texturing techniques are covered in detail. First, a general framework of the texturing process is presented. Next, we focus on using images to texture surfaces, since this is the most popular form of texturing used in real-time work. Procedural textures are briefly discussed, and then some common methods of having textures affect the surface are explained.

6.1 The Texturing Pipeline

Texturing is a technique for efficiently modeling variations in a surface's material and finish. One way to think about texturing is to consider what happens for a single shaded pixel. As seen in the previous chapter, the shade is computed by taking into account the color of the material and the lights, among other factors. If present, transparency also affects the sample. Texturing works by modifying the values used in the shading equation. The way these values are changed is normally based on the position on the surface. So, for the brick wall example, the color at any point on the surface is replaced by a corresponding color in the image of a brick wall, based on the surface location. The pixels in the image texture are often called *texels*, to differentiate them from the pixels on the screen. The roughness texture modifies the roughness value, and the bump texture changes the direction of the shading normal, so each of these change the result of the shading equation.

Texturing can be described by a generalized texture pipeline. Much terminology will be introduced in a moment, but take heart: Each piece of the pipeline will be described in detail.

A location in space is the starting point for the texturing process. This location can be in world space, but is more often in the model's frame of reference, so that as the model moves, the texture moves along with it. Using Kershaw's terminology [884], this point in space then has a *projector* function applied to it to obtain a set of numbers, called *texture coordinates*, that will be used for accessing the texture. This process is called *mapping*, which leads to the phrase *texture mapping*. Sometimes the texture image itself is called the *texture map*, though this is not strictly correct.

Before these new values may be used to access the texture, one or more *corresponder* functions can be used to transform the texture coordinates to texture space. These texture-space locations are used to obtain values from the texture, e.g., they may be array indices into an image texture to retrieve a pixel. The retrieved values are then potentially transformed yet again by a *value transform* function, and finally these new values are used to modify some property of the surface, such as the material or shading normal. Figure 6.2 shows this process in detail for the application of a single texture. The reason for the complexity of the pipeline is that each step provides the user with a useful control. It should be noted that not all steps need to be activated at all times.

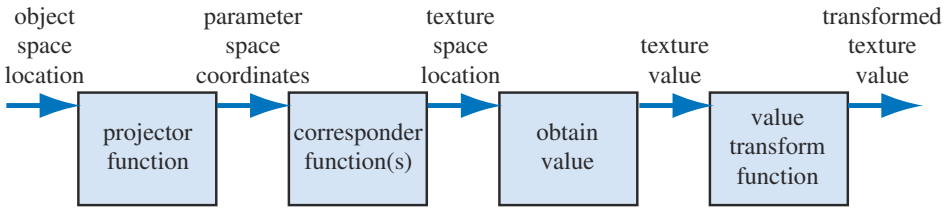

Figure 6.2. The generalized texture pipeline for a single texture.

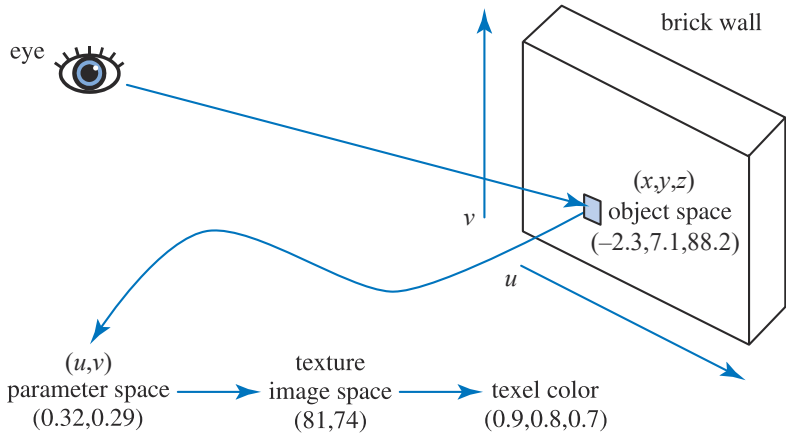

Figure 6.3. Pipeline for a brick wall.

Using this pipeline, this is what happens when a triangle has a brick wall texture and a sample is generated on its surface (see Figure 6.3). The (x, y, z) position in the object's local frame of reference is found; say it is $(-2.3, 7.1, 88.2)$. A projector function is then applied to this position. Just as a map of the world is a projection of a three-dimensional object into two dimensions, the projector function here typically changes the (x, y, z) vector into a two-element vector (u, v). The projector function used for this example is equivalent to an orthographic projection (Section 2.3.1), acting something like a slide projector shining the brick wall image onto the triangle's surface. To return to the wall, a point on its surface could be transformed into a pair of values ranging from 0 to 1. Say the values obtained are $(0.32, 0.29)$. These texture coordinates are to be used to find what the color of the image is at this location. The resolution of our brick texture is, say, 256×256, so the corresponder function multiplies the (u, v) by 256 each, giving $(81.92, 74.24)$. Dropping the fractions, pixel $(81, 74)$ is found in the brick wall image, and is of color $(0.9, 0.8, 0.7)$. The texture color is in sRGB color space, so if the color is to be used in shading equations, it is converted to linear space, giving $(0.787, 0.604, 0.448)$ (Section 5.6).

6.1.1 The Projector Function

The first step in the texture process is obtaining the surface's location and projecting it into texture coordinate space, usually two-dimensional (u, v) space. Modeling packages typically allow artists to define (u, v)-coordinates per vertex. These may be initialized from projector functions or from mesh unwrapping algorithms. Artists can edit (u, v)-coordinates in the same way they edit vertex positions. Projector functions typically work by converting a three-dimensional point in space into texture coordinates. Functions commonly used in modeling programs include spherical, cylindrical, and planar projections [141, 884, 970].

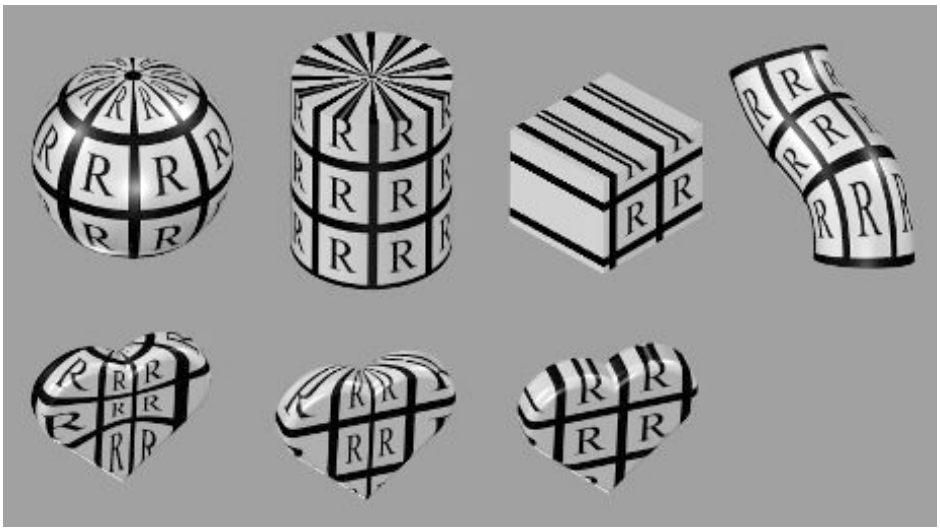

Figure 6.4. Different texture projections. Spherical, cylindrical, planar, and natural (u, v) projections are shown, left to right. The bottom row shows each of these projections applied to a single object (which has no natural projection).

Other inputs can be used to a projector function. For example, the surface normal can be used to choose which of six planar projection directions is used for the surface. Problems in matching textures occur at the seams where the faces meet; Geiss [521, 522] discusses a technique of blending among them. Tarini et al. [1740] describe *polycube maps*, where a model is mapped to a set of cube projections, with different volumes of space mapping to different cubes.

Other projector functions are not projections at all, but are an implicit part of surface creation and tessellation. For example, parametric curved surfaces have a natural set of (u, v) values as part of their definition. See Figure 6.4. The texture coordinates could also be generated from all sorts of different parameters, such as the view direction, temperature of the surface, or anything else imaginable. The goal of the projector function is to generate texture coordinates. Deriving these as a function of position is just one way to do it.

Non-interactive renderers often call these projector functions as part of the rendering process itself. A single projector function may suffice for the whole model, but often the artist has to use tools to subdivide the model and apply various projector functions separately [1345]. See Figure 6.5.

In real-time work, projector functions are usually applied at the modeling stage, and the results of the projection are stored at the vertices. This is not always the case; sometimes it is advantageous to apply the projection function in the vertex or pixel shader. Doing so can increase precision, and helps enable various effects, including animation (Section 6.4). Some rendering methods, such as *environment mapping*

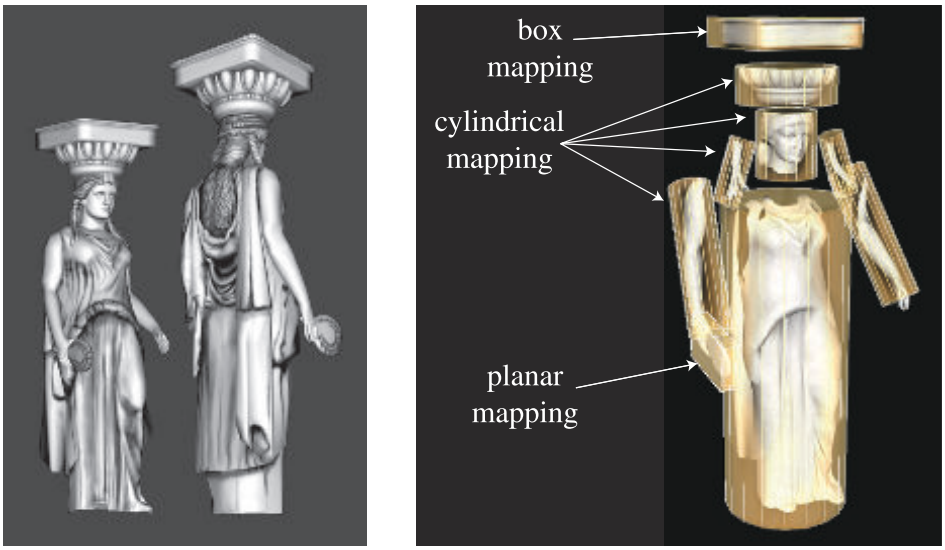

Figure 6.5. How various texture projections are used on a single model. Box mapping consists of six planar mappings, one for each box face. *(Images courtesy of Tito Pagán.)*

(Section 10.4), have specialized projector functions of their own that are evaluated per pixel.

The spherical projection (on the left in Figure 6.4) casts points onto an imaginary sphere centered around some point. This projection is the same as used in Blinn and Newell's environment mapping scheme (Section 10.4.1), so Equation 10.30 on page 407 describes this function. This projection method suffers from the same problems of vertex interpolation described in that section.

Cylindrical projection computes the u texture coordinate the same as spherical projection, with the v texture coordinate computed as the distance along the cylinder's axis. This projection is useful for objects that have a natural axis, such as surfaces of revolution. Distortion occurs when surfaces are near-perpendicular to the cylinder's axis.

The planar projection is like an x-ray beam, projecting in parallel along a direction and applying the texture to all surfaces. It uses orthographic projection (Section 4.7.1). This type of projection is useful for applying decals, for example (Section 20.2).

As there is severe distortion for surfaces that are edge-on to the projection direction, the artist often must manually decompose the model into near-planar pieces. There are also tools that help minimize distortion by unwrapping the mesh, or creating a near-optimal set of planar projections, or that otherwise aid this process. The goal is to have each polygon be given a fairer share of a texture's area, while also maintaining as much mesh connectivity as possible. Connectivity is important in that sampling artifacts can appear along edges where separate parts of a texture meet. A

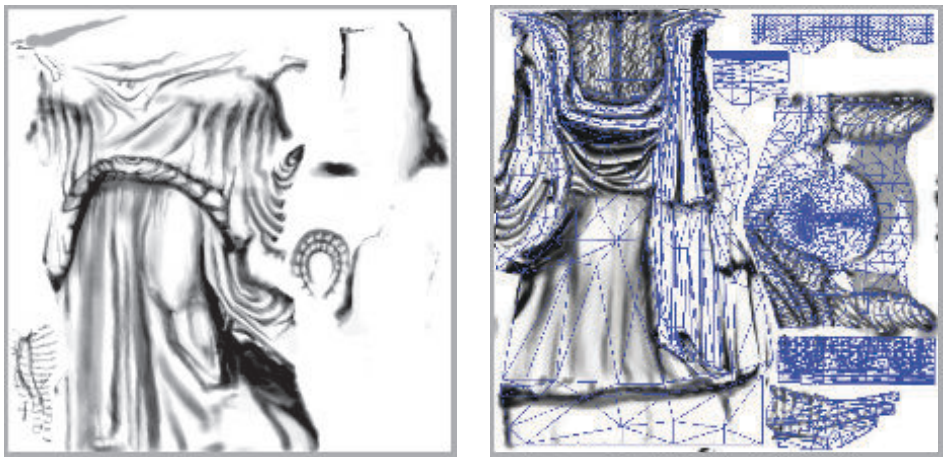

Figure 6.6. Several smaller textures for the statue model, saved in two larger textures. The right figure shows how the triangle mesh is unwrapped and displayed on the texture to aid in its creation. *(Images courtesy of Tito Pagán.)*

mesh with a good unwrapping also eases the artist's work [970, 1345]. Section 16.2.1 discusses how texture distortion can adversely affect rendering. Figure 6.6 shows the workspace used to create the statue in Figure 6.5. This unwrapping process is one facet of a larger field of study, *mesh parameterization*. The interested reader is referred to the SIGGRAPH course notes by Hormann et al. [774].

The texture coordinate space is not always a two-dimensional plane; sometimes it is a three-dimensional volume. In this case, the texture coordinates are presented as a three-element vector, (u, v, w), with w being depth along the projection direction. Other systems use up to four coordinates, often designated (s, t, r, q) [885]; q is used as the fourth value in a homogeneous coordinate. It acts like a movie or slide projector, with the size of the projected texture increasing with distance. As an example, it is useful for projecting a decorative spotlight pattern, called a *gobo*, onto a stage or other surface [1597].

Another important type of texture coordinate space is directional, where each point in the space is accessed by an input direction. One way to visualize such a space is as points on a unit sphere, the normal at each point representing the direction used to access the texture at that location. The most common type of texture using a directional parameterization is the *cube map* (Section 6.2.4).

It is also worth noting that one-dimensional texture images and functions have their uses. For example, on a terrain model the coloration can be determined by altitude, e.g., the lowlands are green; the mountain peaks are white. Lines can also be textured; one use of this is to render rain as a set of long lines textured with a semitransparent image. Such textures are also useful for converting from one value to another, i.e., as a lookup table.

Since multiple textures can be applied to a surface, multiple sets of texture coordinates may need to be defined. However the coordinate values are applied, the idea is the same: These texture coordinates are interpolated across the surface and used to retrieve texture values. Before being interpolated, however, these texture coordinates are transformed by corresponder functions.

6.1.2 The Corresponder Function

Corresponder functions convert texture coordinates to texture-space locations. They provide flexibility in applying textures to surfaces. One example of a corresponder function is to use the API to select a portion of an existing texture for display; only this subimage will be used in subsequent operations.

Another type of corresponder is a matrix transformation, which can be applied in the vertex or pixel shader. This enables to translating, rotating, scaling, shearing, or projecting the texture on the surface. As discussed in Section 4.1.5, the order of transforms matters. Surprisingly, the order of transforms for textures must be the reverse of the order one would expect. This is because texture transforms actually affect the space that determines where the image is seen. The image itself is not an object being transformed; the space defining the image's location is being changed.

Another class of corresponder functions controls the way an image is applied. We know that an image will appear on the surface where (u, v) are in the $[0, 1]$ range. But what happens outside of this range? Corresponder functions determine the behavior. In OpenGL, this type of corresponder function is called the "wrapping mode"; in DirectX, it is called the "texture addressing mode." Common corresponder functions of this type are:

- **wrap** (DirectX), **repeat** (OpenGL), or **tile**—The image repeats itself across the surface; algorithmically, the integer part of the texture coordinates is dropped. This function is useful for having an image of a material repeatedly cover a surface, and is often the default.
- **mirror**—The image repeats itself across the surface, but is mirrored (flipped) on every other repetition. For example, the image appears normally going from 0 to 1, then is reversed between 1 and 2, then is normal between 2 and 3, then is reversed, and so on. This provides some continuity along the edges of the texture.
- **clamp** (DirectX) or **clamp to edge** (OpenGL)—Values outside the range $[0, 1]$ are clamped to this range. This results in the repetition of the edges of the image texture. This function is useful for avoiding accidentally taking samples from the opposite edge of a texture when bilinear interpolation happens near a texture's edge [885].
- **border** (DirectX) or **clamp to border** (OpenGL)—Texture coordinates outside $[0, 1]$ are rendered with a separately defined border color. This function can

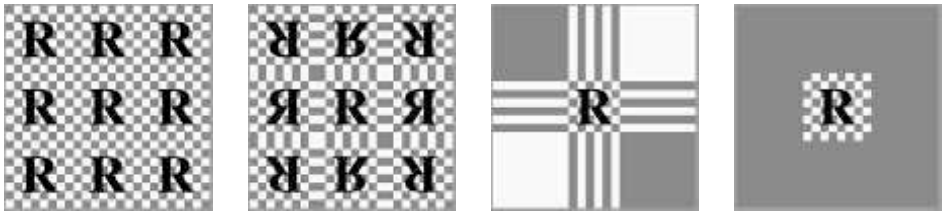

Figure 6.7. Image texture repeat, mirror, clamp, and border functions in action.

be good for rendering decals onto single-color surfaces, for example, as the edge of the texture will blend smoothly with the border color.

See Figure 6.7. These corresponder functions can be assigned differently for each texture axis, e.g., the texture could repeat along the u-axis and be clamped on the v-axis. In DirectX there is also a **mirror once** mode that mirrors a texture once along the zero value for the texture coordinate, then clamps, which is useful for symmetric decals.

Repeated tiling of a texture is an inexpensive way of adding more visual detail to a scene. However, this technique often looks unconvincing after about three repetitions of the texture, as the eye picks out the pattern. A common solution to avoid such *periodicity* problems is to combine the texture values with another, non-tiled, texture. This approach can be considerably extended, as seen in the commercial terrain rendering system described by Andersson [40]. In this system, multiple textures are combined based on terrain type, altitude, slope, and other factors. Texture images are also tied to where geometric models, such as bushes and rocks, are placed within the scene.

Another option to avoid periodicity is to use shader programs to implement specialized corresponder functions that randomly recombine texture patterns or tiles. *Wang tiles* are one example of this approach. A Wang tile set is a small set of square tiles with matching edges. Tiles are selected randomly during the texturing process [1860]. Lefebvre and Neyret [1016] implement a similar type of corresponder function using dependent texture reads and tables to avoid pattern repetition.

The last corresponder function applied is implicit, and is derived from the image's size. A texture is normally applied within the range $[0, 1]$ for u and v. As shown in the brick wall example, by multiplying texture coordinates in this range by the resolution of the image, one may obtain the pixel location. The advantage of being able to specify (u, v) values in a range of $[0, 1]$ is that image textures with different resolutions can be swapped in without having to change the values stored at the vertices of the model.

6.1.3 Texture Values

After the corresponder functions are used to produce texture-space coordinates, the coordinates are used to obtain texture values. For image textures, this is done by

accessing the texture to retrieve texel information from the image. This process is dealt with extensively in Section 6.2. Image texturing constitutes the vast majority of texture use in real-time work, but procedural functions can also be used. In the case of procedural texturing, the process of obtaining a texture value from a texture-space location does not involve a memory lookup, but rather the computation of a function. Procedural texturing is further described in Section 6.3.

The most straightforward texture value is an RGB triplet that is used to replace or modify the surface colors; similarly, a single grayscale value could be returned. Another type of data to return is RGBα, as described in Section 5.5. The α (alpha) value is normally the opacity of the color, which determines the extent to which the color may affect the pixel. That said, any other value could be stored, such as surface roughness. There are many other types of data that can be stored in image textures, as will be seen when bump mapping is discussed in detail (Section 6.7).

The values returned from the texture are optionally transformed before use. These transformations may be performed in the shader program. One common example is the remapping of data from an unsigned range (0.0 to 1.0) to a signed range (-1.0 to 1.0), which is used for shading normals stored in a color texture.

6.2 Image Texturing

In image texturing, a two-dimensional image is effectively glued onto the surface of one or more triangles. We have walked through the process of computing a texture-space location; now we will address the issues and algorithms for obtaining a texture value from the image texture, given that location. For the rest of this chapter, the image texture will be referred to simply as the *texture*. In addition, when we refer to a pixel's *cell* here, we mean the screen grid cell surrounding that pixel. As discussed in Section 5.4.1, a *pixel* is actually a displayed color value that can (and should, for better quality) be affected by samples outside of its associated grid cell.

In this section we particularly focus on methods to rapidly sample and filter textured images. Section 5.4.2 discussed the problem of aliasing, especially with respect to rendering edges of objects. Textures can also have sampling problems, but they occur within the interiors of the triangles being rendered.

The pixel shader accesses textures by passing in texture coordinate values to a call such as `texture2D`. These values are in (u, v) texture coordinates, mapped by a corresponder function to a range $[0.0, 1.0]$. The GPU takes care of converting this value to texel coordinates. There are two main differences among texture coordinate systems in different APIs. In DirectX the upper left corner of the texture is $(0, 0)$ and the lower right is $(1, 1)$. This matches how many image types store their data, the top row being the first one in the file. In OpenGL the texel $(0, 0)$ is located in the lower left, a y-axis flip from DirectX. Texels have integer coordinates, but we often want to access a location between texels and blend among them. This brings up the question of what the floating point coordinates of the center of a pixel are. Heckbert [692] discusses

how there are two systems possible: truncating and rounding. DirectX 9 defined each center at $(0.0, 0.0)$—this uses rounding. This system was somewhat confusing, as the upper left corner of the upper left pixel, at DirectX's origin, then had the value $(-0.5, -0.5)$. DirectX 10 onward changes to OpenGL's system, where the center of a texel has the fractional values $(0.5, 0.5)$—truncation, or more accurately, flooring, where the fraction is dropped. Flooring is a more natural system that maps well to language, in that pixel $(5, 9)$, for example, defines a range from 5.0 to 6.0 for the u-coordinate and 9.0 to 10.0 for the v.

One term worth explaining at this point is *dependent texture read*, which has two definitions. The first applies to mobile devices in particular. When accessing a texture via `texture2D` or similar, a dependent texture read occurs whenever the pixel shader calculates texture coordinates instead of using the unmodified texture coordinates passed in from the vertex shader [66]. Note that this means any change at all to the incoming texture coordinates, even such simple actions as swapping the u and v values. Older mobile GPUs, those that do not support OpenGL ES 3.0, run more efficiently when the shader has no dependent texture reads, as the texel data can then be prefetched. The other, older, definition of this term was particularly important for early desktop GPUs. In this context a dependent texture read occurs when one texture's coordinates are dependent on the result of some previous texture's values. For example, one texture might change the shading normal, which in turn changes the coordinates used to access a cube map. Such functionality was limited or even non-existent on early GPUs. Today such reads can have an impact on performance, depending on the number of pixels being computed in a batch, among other factors. See Section 23.8 for more information.

The texture image size used in GPUs is usually $2^m \times 2^n$ texels, where m and n are non-negative integers. These are referred to as *power-of-two* (POT) textures. Modern GPUs can handle *non-power-of-two* (NPOT) textures of arbitrary size, which allows a generated image to be treated as a texture. However, some older mobile GPUs may not support mipmapping (Section 6.2.2) for NPOT textures. Graphics accelerators have different upper limits on texture size. DirectX 12 allows a maximum of 16384^2 texels, for example.

Assume that we have a texture of size 256×256 texels and that we want to use it as a texture on a square. As long as the projected square on the screen is roughly the same size as the texture, the texture on the square looks almost like the original image. But what happens if the projected square covers ten times as many pixels as the original image contains (called *magnification*), or if the projected square covers only a small part of the screen (*minification*)? The answer is that it depends on what kind of sampling and filtering methods you decide to use for these two separate cases.

The image sampling and filtering methods discussed in this chapter are applied to the values read from each texture. However, the desired result is to prevent aliasing in the final rendered image, which in theory requires sampling and filtering the final pixel colors. The distinction here is between filtering the inputs to the shading equation, or filtering its output. As long as the inputs and output are linearly related (which is true

for inputs such as colors), then filtering the individual texture values is equivalent to filtering the final colors. However, many shader input values stored in textures, such as surface normals and roughness values, have a nonlinear relationship to the output. Standard texture filtering methods may not work well for these textures, resulting in aliasing. Improved methods for filtering such textures are discussed in Section 9.13.

6.2.1 Magnification

In Figure 6.8, a texture of size 48×48 texels is textured onto a square, and the square is viewed rather closely with respect to the texture size, so the underlying graphics system has to magnify the texture. The most common filtering techniques for magnification are *nearest neighbor* (the actual filter is called a box filter—see Section 5.4.1) and *bilinear interpolation*. There is also *cubic convolution*, which uses the weighted sum of a 4×4 or 5×5 array of texels. This enables much higher magnification quality. Although native hardware support for cubic convolution (also called *bicubic interpolation*) is currently not commonly available, it can be performed in a shader program.

In the left part of Figure 6.8, the nearest neighbor method is used. One characteristic of this magnification technique is that the individual texels may become apparent. This effect is called *pixelation* and occurs because the method takes the value of the nearest texel to each pixel center when magnifying, resulting in a blocky appearance. While the quality of this method is sometimes poor, it requires only one texel to be fetched per pixel.

In the middle image of the same figure, bilinear interpolation (sometimes called *linear interpolation*) is used. For each pixel, this kind of filtering finds the four neighboring texels and linearly interpolates in two dimensions to find a blended value for the pixel. The result is blurrier, and much of the jaggedness from using the nearest neighbor method has disappeared. As an experiment, try looking at the left image

Figure 6.8. Texture magnification of a 48×48 image onto 320×320 pixels. Left: nearest neighbor filtering, where the nearest texel is chosen per pixel. Middle: bilinear filtering using a weighted average of the four nearest texels. Right: cubic filtering using a weighted average of the 5×5 nearest texels.

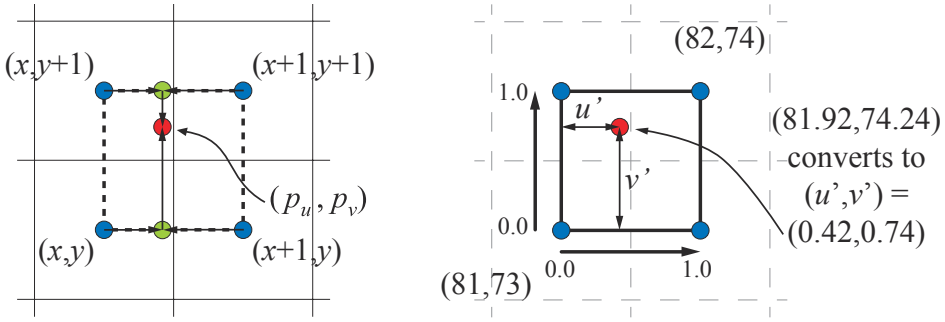

Figure 6.9. Bilinear interpolation. The four texels involved are illustrated by the four squares on the left, texel centers in blue. On the right is the coordinate system formed by the centers of the four texels.

while squinting, as this has approximately the same effect as a low-pass filter and reveals the face a bit more.

Returning to the brick texture example on page 170: Without dropping the fractions, we obtained $(p_u, p_v) = (81.92, 74.24)$. We use OpenGL's lower left origin texel coordinate system here, since it matches the standard Cartesian system. Our goal is to interpolate among the four closest texels, defining a texel-sized coordinate system using their texel centers. See Figure 6.9. To find the four nearest pixels, we subtract the pixel center fraction $(0.5, 0.5)$ from our sample location, giving $(81.42, 73.74)$. Dropping the fractions, the four closest pixels range from $(x, y) = (81, 73)$ to $(x+1, y+1) = (82, 74)$. The fractional part, $(0.42, 0.74)$ for our example, is the location of the sample relative to the coordinate system formed by the four texel centers. We denote this location as (u', v').

Define the texture access function as $\mathbf{t}(x, y)$, where x and y are integers and the color of the texel is returned. The bilinearly interpolated color for any location (u', v') can be computed as a two-step process. First, the bottom texels, $\mathbf{t}(x, y)$ and $\mathbf{t}(x+1, y)$, are interpolated horizontally (using u'), and similarly for the topmost two texels, $\mathbf{t}(x, y+1)$ and $\mathbf{t}(x+1, y+1)$. For the bottom texels, we obtain $(1-u')\mathbf{t}(x, y) + u'\mathbf{t}(x+1, y)$ (bottom green circle in Figure 6.9), and for the top, $(1-u')\mathbf{t}(x, y+1) + u'\mathbf{t}(x+1, y+1)$ (top green circle). These two values are then interpolated vertically (using v'), so the bilinearly interpolated color $\mathbf{b}$ at (p_u, p_v) is

$$\begin{aligned}\mathbf{b}(p_u, p_v) &= (1-v')\big((1-u')\mathbf{t}(x, y) + u'\mathbf{t}(x+1, y)\big) \\ &\quad + v'\big((1-u')\mathbf{t}(x, y+1) + u'\mathbf{t}(x+1, y+1)\big) \\ &= (1-u')(1-v')\mathbf{t}(x, y) + u'(1-v')\mathbf{t}(x+1, y) \\ &\quad + (1-u')v'\mathbf{t}(x, y+1) + u'v'\mathbf{t}(x+1, y+1).\end{aligned} \tag{6.1}$$

Intuitively, a texel closer to our sample location will influence its final value more. This is indeed what we see in this equation. The upper right texel at $(x+1, y+1)$ has