CHAPTER SEVEN

07 SAMPLING AND RECONSTRUCTION

Although the final output of a renderer like `pbrt` is a two-dimensional grid of colored pixels, incident radiance is actually a continuous function defined over the film plane. The manner in which the discrete pixel values are computed from this continuous function can noticeably affect the quality of the final image generated by the renderer; if this process is not performed carefully, artifacts will be present. Fortunately, a relatively small amount of additional computation to this end can substantially improve the quality of the rendered images.

This chapter introduces *sampling theory*—the theory of taking discrete sample values from functions defined over continuous domains and then using those samples to reconstruct new functions that are similar to the original. Building on principles of sampling theory, the `Sampler`s in this chapter select sample points on the image plane at which incident radiance will be computed (recall that in the previous chapter `Camera`s used `Sample`s generated by a `Sampler` to construct their camera rays). Five `Sampler` implementations are described in this chapter, spanning a variety of approaches to the sampling problem. This chapter concludes with the `Filter` class and the `Film` class. The `Filter` is used to determine how multiple samples near each pixel are blended together to compute the final pixel value, and `Film` implementations filter image samples and store them in images.

7.1 SAMPLING THEORY

A digital image is represented as a set of pixel values, typically aligned on a rectangular grid. When a digital image is displayed on a physical device, these values are used to set the intensities and colors of pixels on the display. When thinking about digital images, it is important to differentiate between image pixels, which represent the value of a function at a particular sample location, and display pixels, which are physical objects

that emit light with some distribution. For example, in a CRT, each phosphor glows with a distribution that falls off from its center. Pixel intensity has angular distribution as well; it is mostly uniform for CRTs, although it can be quite directional on LCD displays. Displays use the image pixel values to construct a new image function over the display surface. This function is defined at all points on the display, not just the infinitesimal points of the digital image's pixels. This process of taking a collection of sample values and converting them back to a continuous function is called *reconstruction*.

In order to compute the discrete pixel values in the digital image, it is necessary to sample the original continuously defined image function. In `pbrt`, like most other ray-tracing renderers, the only way to get information about the image function is to sample it by tracing rays. For example, there is no general method that can compute bounds on the variation of the image function between two points on the film plane. While an image could be generated by just sampling the function precisely at the pixel positions, a better result can be obtained by taking more samples at different positions and incorporating this additional information about the image function into the final pixel values. Indeed, for the best-quality result, the pixel values should be computed such that the reconstructed image on the display device is as close as possible to the original image of the scene on the virtual camera's film plane. Note that this is a subtly different goal than expecting the display's pixels to take on the image function's actual value at their positions. Handling this difference is the main goal of the algorithms implemented in this chapter.[1]

Because the sampling and reconstruction process involves approximation, it introduces error known as *aliasing,* which can manifest itself in many ways, including jagged edges or flickering in animations. These errors occur because the sampling process is not able to capture all of the information from the continuously defined image function.

As an example of these ideas, consider a one-dimensional function (which we will interchangeably refer to as a signal), given by $f(x)$, where we can evaluate $f(x')$ at any desired location x' in the function's domain. Each such x' is called a *sample position*, and the value of $f(x')$ is the *sample value*. Figure 7.1 shows a set of samples of a smooth 1D function, along with a reconstructed signal $\tilde{f}$ that approximates the original function f. In this example, $\tilde{f}$ is a piecewise linear function that approximates f by linearly interpolating neighboring sample values (readers already familiar with sampling theory will recognize this as reconstruction with a hat function). Because the only information available about f comes from the sample values at the positions x', $\tilde{f}$ is unlikely to match f perfectly since there is no information about f's behavior between the samples.

Fourier analysis can be used to evaluate the quality of the match between the reconstructed function and the original. This section will introduce the main ideas of Fourier analysis with enough detail to work through some parts of the sampling and reconstruction processes, but will omit proofs of many properties and skip details that aren't directly

1 In this book, we will ignore the subtle issues related to the characteristics of the physical display pixels and will work under the assumption that the display performs the ideal reconstruction process described later in this section. This assumption is patently at odds with how actual displays work, but it avoids unnecessary complicating of the analysis here. Chapter 3 of Glassner (1995) has a good treatment of nonidealized display devices and their impact on the image sampling and reconstruction process.

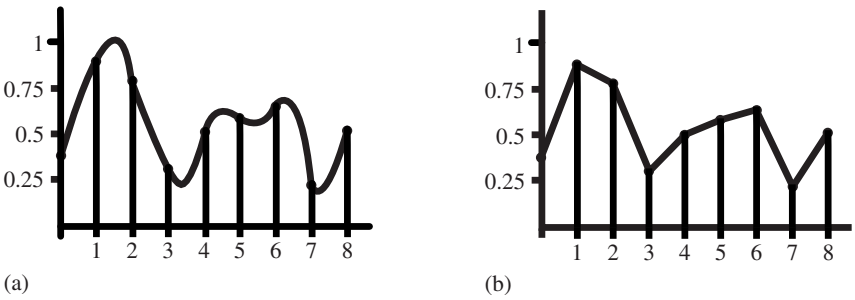


Figure 7.1: (a) By taking a set of *point samples* of $f(x)$ (indicated by black dots), we determine the value of the function at those positions. (b) The sample values can be used to *reconstruct* a function $\tilde{f}(x)$ that is an approximation to $f(x)$. The sampling theorem, introduced in Section 7.1.3, makes a precise statement about the conditions on $f(x)$, the number of samples taken, and the reconstruction technique used under which $\tilde{f}(x)$ is exactly the same as $f(x)$. The fact that the original function can sometimes be reconstructed exactly from point samples alone is remarkable.

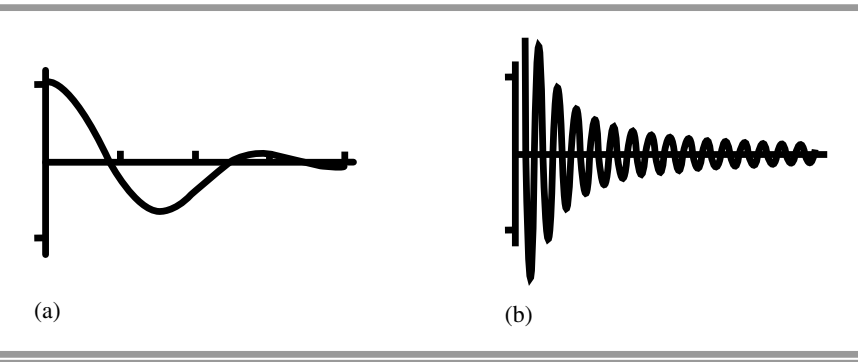


Figure 7.2: (a) Low-frequency function, and (b) high-frequency function. Roughly speaking, the higher frequency a function is, the more quickly it varies over a given region.

relevant to the sampling algorithms used in `pbrt`. The "Further Reading" section of this chapter has pointers to more detailed information about these topics.

7.1.1 THE FREQUENCY DOMAIN AND THE FOURIER TRANSFORM

One of the foundations of Fourier analysis is the Fourier transform, which represents a function in the *frequency domain*. (We will say that functions are normally expressed in the *spatial domain*.) Consider the two functions graphed in Figure 7.2. The function in Figure 7.2(a) varies relatively slowly as a function of x, while the function in Figure 7.2(b) varies much more rapidly. The slower-varying function is said to have lower frequency content. Figure 7.3 shows the frequency space representations of these two functions; the lower-frequency function's representation goes to zero more quickly than the higher-frequency function.

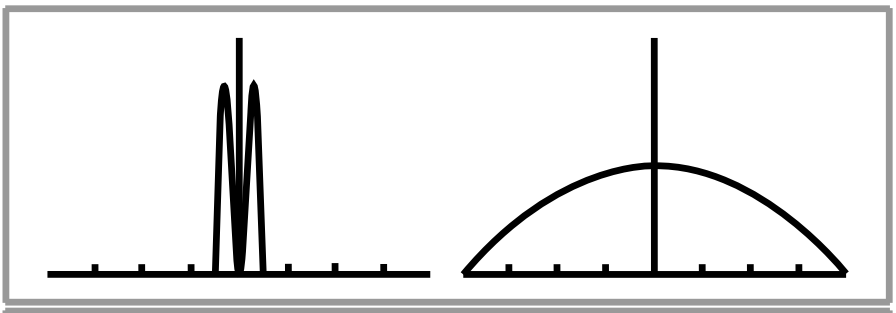


Figure 7.3: Frequency Space Representations of the Functions in Figure 7.2. The graphs show the contribution of each frequency ω to each of the functions in the spatial domain.

Most functions can be decomposed into a weighted sum of shifted sinusoids. This remarkable fact was first described by Joseph Fourier, and the Fourier transform converts a function into this representation. This frequency space representation of a function gives insight into some of its characteristics—the distribution of frequencies in the sine functions corresponds to the distribution of frequencies in the original function. Using this form, it is possible to use Fourier analysis to gain insight into the error that is introduced by the sampling and reconstruction process, and how to reduce the perceptual impact of this error.

The Fourier transform of a one-dimensional function $f(x)$ is[2]

$$F(\omega) = \int_{-\infty}^{\infty} f(x) \mathrm{e}^{-i2\pi\omega x} \mathrm{d}x. \tag{7.1}$$

(Recall that $\mathrm{e}^{ix} = \cos x + i \sin x$, where $i = \sqrt{-1}$.) For simplicity, here we will consider only *even* functions where $f(-x) = f(x)$, in which case the Fourier transform of f has no imaginary terms. The new function F is a function of *frequency*, ω.[3] We will denote the Fourier transform operator by $\mathcal{F}$, such that $\mathcal{F}\{f(x)\} = F(\omega)$. $\mathcal{F}$ is clearly a linear operator—that is, $\mathcal{F}\{af(x)\} = a\mathcal{F}\{f(x)\}$ for any scalar a, and $\mathcal{F}\{f(x) + g(x)\} = \mathcal{F}\{f(x)\} + \mathcal{F}\{g(x)\}$.

Equation (7.1) is called the *Fourier analysis* equation, or sometimes just the *Fourier transform*. We can also transform from the frequency domain back to the spatial domain using the *Fourier synthesis* equation, or the *inverse Fourier transform*:

$$f(x) = \int_{-\infty}^{\infty} F(\omega) \mathrm{e}^{i2\pi\omega x} \, \mathrm{d}\omega. \tag{7.2}$$

2 The reader should be warned that the constants in front of these integrals are not always the same in different fields. For example, some authors (including many in the physics community) prefer to multiply both integrals by $1/\sqrt{2\pi}$.

3 In this chapter, we will use the ω symbol to denote frequency. Throughout the rest of the book, ω denotes normalized direction vectors. This overloading of notation should never be confusing, given the contexts where these symbols are used. Similarly, when we refer to a function's "spectrum," we are referring to its distribution of frequencies in its frequency space representation, rather than anything related to color.

Table 7.1: Fourier Pairs. Functions in the spatial domain and their frequency space representations. Because of the symmetry properties of the Fourier transform, if the left column is instead considered to be frequency space, then the right column is the spatial equivalent of those functions as well.

Spatial Domain	Frequency Space Representation
Box: $f(x) = 1$ if $\|x\| < 1/2$, 0 otherwise	Sinc: $f(\omega) = \text{sinc}(\omega) = \sin(\pi\omega)/(\pi\omega)$
Gaussian: $f(x) = e^{-\pi x^2}$	Gaussian: $f(\omega) = e^{-\pi\omega^2}$
Constant: $f(x) = 1$	Delta: $f(\omega) = \delta(\omega)$
Sinusoid: $f(x) = \cos x$	Translated delta: $f(\omega) = \pi(\delta(1/2 - \omega) + \delta(1/2 + \omega))$
Shah: $f(x) = \text{III}_T(x) = T \sum_i \delta(x - Ti)$	Shah: $f(\omega) = \text{III}_{1/T}(\omega) = (1/T) \sum_i \delta(\omega - i/T)$

Table 7.1 shows a number of important functions and their frequency space representations. A number of these functions are based on the Dirac delta distribution, a special function that is defined such that $\int \delta(x)\mathrm{d}x = 1$, and for all $x \neq 0$, $\delta(x) = 0$. An important consequence of these properties is that

$$\int f(x)\delta(x)\,\mathrm{d}x = f(0).$$

The delta distribution cannot be expressed as a standard mathematical function, but instead is generally thought of as the limit of a unit area box function centered at the origin with width approaching zero.

7.1.2 IDEAL SAMPLING AND RECONSTRUCTION

Using frequency space analysis, we can now formally investigate the properties of sampling. Recall that the sampling process requires us to choose a set of equally spaced sample positions and compute the function's value at those positions. Formally, this corresponds to multiplying the function by a "shah," or "impulse train," function, an infinite sum of equally spaced delta functions. The shah $\text{III}_T(x)$ is defined as

$$\text{III}_T(x) = T \sum_{i=-\infty}^{\infty} \delta(x - iT),$$

where T defines the period, or *sampling rate*. This formal definition of sampling is illustrated in Figure 7.4. The multiplication yields an infinite sequence of values of the function at equally spaced points:

$$\text{III}_T(x) f(x) = T \sum_i \delta(x - iT) f(iT).$$

These sample values can be used to define a reconstructed function $\tilde{f}$ by choosing a reconstruction filter function $r(x)$ and computing the *convolution*

$$\big(\text{III}_T(x) f(x)\big) \otimes r(x),$$

where the convolution operation $\otimes$ is defined as

$$f(x) \otimes g(x) = \int_{-\infty}^{\infty} f(x')g(x - x')\,\mathrm{d}x'.$$

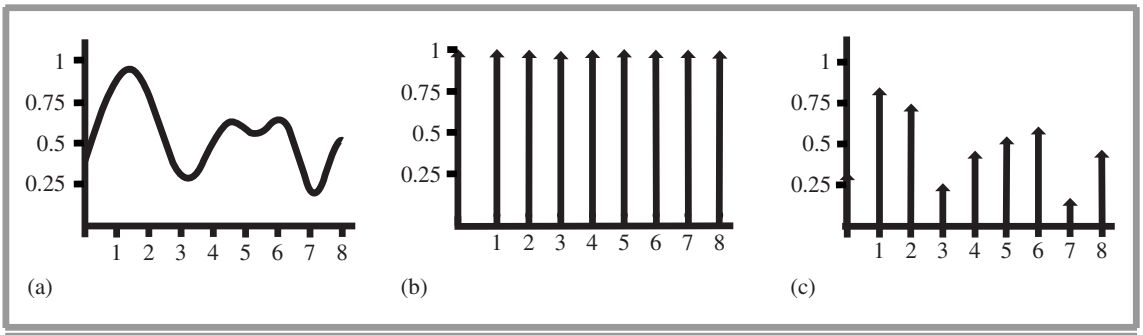


Figure 7.4: Formalizing the Sampling Process. (a) The function $f(x)$ is multiplied by (b) the shah function $\mathrm{III}_T(x)$, giving (c) an infinite sequence of scaled delta functions that represent its value at each sample point.

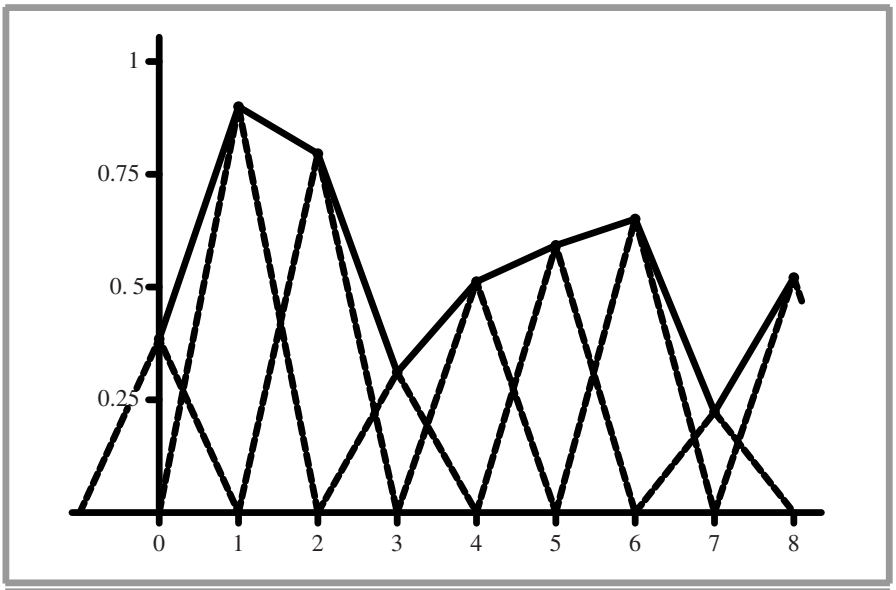


Figure 7.5: The sum of instances of the triangle reconstruction filter, shown with dashed lines, gives the reconstructed approximation to the original function, shown with a solid line.

For reconstruction, convolution gives a weighted sum of scaled instances of the reconstruction filter centered at the sample points:

$$\tilde{f}(x) = T \sum_{i=-\infty}^{\infty} f(iT)r(x - iT).$$

For example, in Figure 7.1, the triangle reconstruction filter, $f(x) = \max(0, 1 - |x|)$, was used. Figure 7.5 shows the scaled triangle functions used for that example.

We have gone through a process that may seem gratuitiously complex in order to end up at an intuitive result: the reconstructed function $\tilde{f}(x)$ can be obtained by interpolating among the samples in some manner. By setting up this background carefully, however, Fourier analysis can now be applied to the process more easily.

We can gain a deeper understanding of the sampling process by analyzing the sampled function in the frequency domain. In particular, we will be able to determine the conditions under which the original function can be exactly recovered from its values at the sample locations—a very powerful result. For the discussion here, we will assume for now that the function $f(x)$ is *band limited*—there exists some frequency ω_0 such that $f(x)$ contains no frequencies greater than ω_0. By definition, band-limited functions have frequency space representations with compact support, such that $F(\omega) = 0$ for all $|\omega| > \omega_0$. Both of the spectra in Figure 7.3 are band limited.

An important idea used in Fourier analysis is the fact that the Fourier transform of the product of two functions $\mathcal{F}\{f(x)g(x)\}$ can be shown to be the convolution of their individual Fourier transforms $F(\omega)$ and $G(\omega)$:

$$\mathcal{F}\{f(x)g(x)\} = F(\omega) \otimes G(\omega).$$

It is similarly the case that convolution in the spatial domain is equivalent to multiplication in the frequency domain:

$$\mathcal{F}\{f(x) \otimes g(x)\} = F(\omega)G(\omega). \tag{7.3}$$

These properties are derived in the standard references on Fourier analysis. Using these ideas, the original sampling step in the spatial domain, where the product of the shah function and the original function $f(x)$ is found, can be equivalently described by the convolution of $F(\omega)$ with another shah function in frequency space.

We also know the spectrum of the shah function $\text{III}_T(x)$ from Table 7.1; the Fourier transform of a shah function with period T is another shah function with period $1/T$. This reciprocal relationship between periods is important to keep in mind: it means that if the samples are farther apart in the spatial domain, they are closer together in the frequency domain.

Thus, the frequency domain representation of the sampled signal is given by the convolution of $F(\omega)$ and this new shah function. Convolving a function with a delta function just yields a copy of the function, so convolving with a shah function yields an infinite sequence of copies of the original function, with spacing equal to the period of the shah (Figure 7.6). This is the frequency space representation of the series of samples.

Now that we have this infinite set of copies of the function's spectrum, how do we reconstruct the original function? Looking at Figure 7.6, the answer is obvious: just discard all of the spectrum copies except the one centered at the origin, giving the original $F(\omega)$. In order to throw away all but the center copy of the spectrum, we multiply by a box function of the appropriate width (Figure 7.7). The box function $\Pi_T(x)$ of width T is defined as

$$\Pi_T(x) = \begin{cases} 1/(2T) & |x| < T \\ 0 & \text{otherwise.} \end{cases}$$

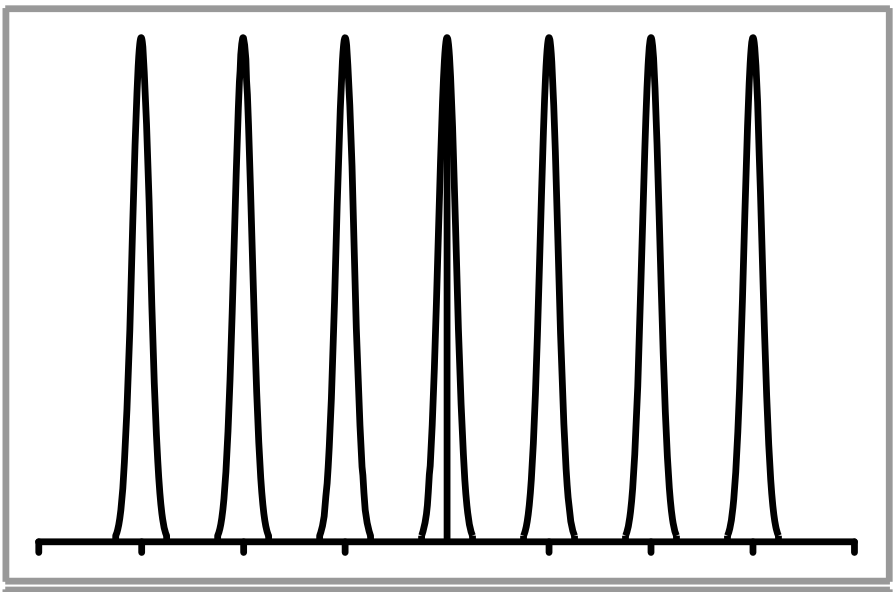


Figure 7.6: The Convolution of $F(\omega)$ and the Shah Function. The result is infinitely many copies of F.

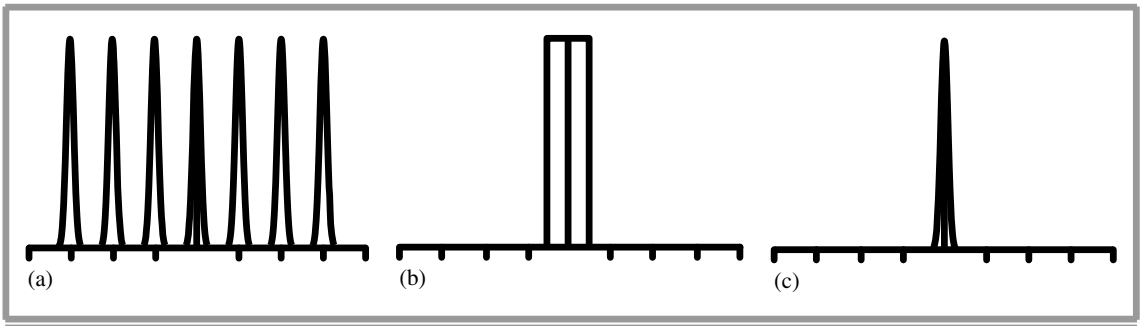


Figure 7.7: Multiplying (a) a series of copies of $F(\omega)$ by (b) the appropriate box function yields (c) the original spectrum.

This multiplication step corresponds to convolution with the reconstruction filter in the spatial domain. This is the ideal sampling and reconstruction process. To summarize:

$$\tilde{F} = \left(F(\omega) \otimes \text{III}_{1/T}(\omega)\right) \Pi_T(x).$$

This is a remarkable result: we have been able to determine the exact frequency space representation of $f(x)$, purely by sampling it at a set of regularly spaced points. Other than knowing that the function was band limited, no additional information about the composition of the function was used.

Applying the equivalent process in the spatial domain will likewise recover $f(x)$ exactly. Because the inverse Fourier transform of the box function is the sinc function, ideal reconstruction in the spatial domain is found by

$$\tilde{f} = \big(f(x)\mathrm{III}_T(x)\big) \otimes \mathrm{sinc}(x),$$

or

$$\tilde{f}(x) = \sum_{i=-\infty}^{\infty} \mathrm{sinc}(x-i)\, f(i).$$

Unfortunately, because the sinc function has infinite extent, it is necessary to use all of the sample values $f(i)$ to compute any particular value of $\tilde{f}(x)$ in the spatial domain. Filters with finite spatial extent are preferable for practical implementations even though they don't reconstruct the original function perfectly.

A commonly used alternative in graphics is to use the box function for reconstruction, effectively averaging all of the sample values within some region around x. This is a very poor choice, as can be seen by considering the box filter's behavior in the frequency domain: This technique attempts to isolate the central copy of the function's spectrum by *multiplying by a sinc*, which not only does a bad job of selecting the central copy of the function's spectrum but includes high-frequency contributions from the infinite series of other copies of it as well.

7.1.3 ALIASING

In addition to the sinc function's infinite extent, one of the most serious practical problems with the ideal sampling and reconstruction approach is the assumption that the signal is band limited. For signals that are not band limited, or signals that aren't sampled at a sufficiently high sampling rate for their frequency content, the process described earlier will reconstruct a function that is different than the original signal.

The key to successful reconstruction is the ability to exactly recover the original spectrum $F(\omega)$ by multiplying the sampled spectrum with a box function of the appropriate width. Notice that in Figure 7.6, the copies of the signal's spectrum are separated by empty space, so perfect reconstruction is possible. Consider what happens, however, if the original function was sampled with a lower sampling rate. Recall that the Fourier transform of a shah function III_T with period T is a new shah function with period $1/T$. This means that if the spacing between samples increases in the spatial domain, the sample spacing decreases in the frequency domain, pushing the copies of the spectrum $F(\omega)$ closer together. If the copies get too close together, they start to overlap.

Because the copies are added together, the resulting spectrum no longer looks like many copies of the original (Figure 7.8). When this new spectrum is multiplied by a box function, the result is a spectrum that is similar but not equal to the original $F(\omega)$: High-frequency details in the original signal leak into lower-frequency regions of the spectrum of the reconstructed signal. These new low-frequency artifacts are called *aliases* (because high frequencies are "masquerading" as low frequencies), and the resulting signal is

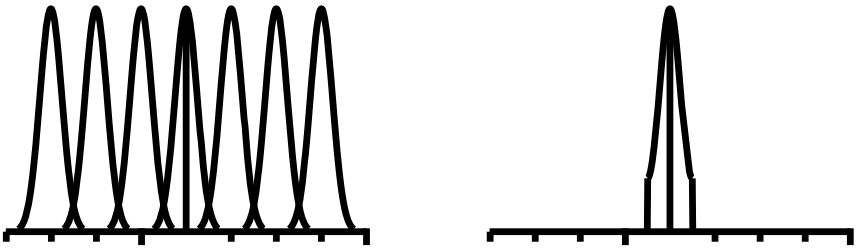


Figure 7.8: (a) When the sampling rate is too low, the copies of the function's spectrum overlap, resulting in (b) aliasing when reconstruction is performed.

said to be *aliased*. Figure 7.9 shows the effects of aliasing from undersampling and then reconstructing the one-dimensional function $f(x) = 1 + \cos(4x^2)$.

A possible solution to the problem of overlapping spectra is to simply increase the sampling rate until the copies of the spectrum are sufficiently far apart to not overlap, thereby eliminating aliasing completely. In fact, the *sampling theorem* tells us exactly what rate is required. This theorem says that as long as the frequency of uniform sample points ω_s is greater than twice the maximum frequency present in the signal ω_0, it is possible to reconstruct the original signal perfectly from the samples. This minimum sampling frequency is called the *Nyquist frequency*.

For signals that are not band limited ($\omega_0 = \infty$), it is impossible to sample at a high enough rate to perform perfect reconstruction. Non-band-limited signals have spectra with infinite support, so no matter how far apart the copies of their spectra are (i.e., how high a sampling rate we use), there will always be overlap. Unfortunately, few of the interesting functions in computer graphics are band limited. In particular, any function containing a discontinuity cannot be band limited, and therefore we cannot perfectly sample and reconstruct it. This makes sense because the function's discontinuity will always fall between two samples and the samples provide no information about the location of the discontinuity. Thus, it is necessary to apply different methods besides just increasing the sampling rate in order to counteract the error that aliasing can introduce to the renderer's results.

7.1.4 ANTIALIASING TECHNIQUES

If one is not careful about sampling and reconstruction, myriad artifacts can appear in the final image. It is sometimes useful to distinguish between artifacts due to sampling and those due to reconstruction; when we wish to be precise we will call sampling artifacts *prealiasing* and reconstruction artifacts *postaliasing*. Any attempt to fix these errors is broadly classified as *antialiasing*. This section reviews a number of antialiasing techniques beyond just increasing the sampling rate everywhere.

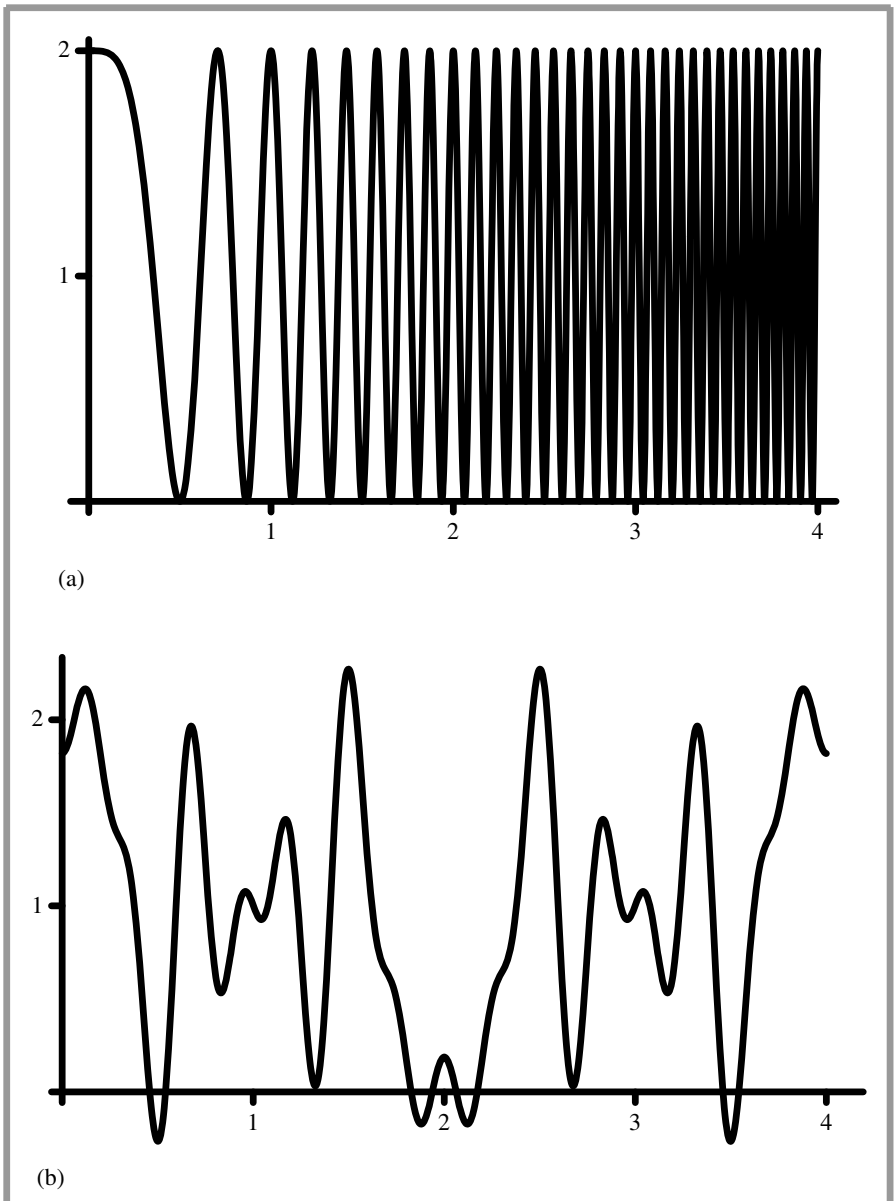


Figure 7.9: Aliasing from Point Sampling the Function $1 + \cos(4x^2)$. (a) The function. (b) The reconstructed function from sampling it with samples spaced 0.125 units apart and performing perfect reconstruction with the sinc filter. Aliasing causes the high-frequency information in the original function to be lost and to reappear as lower-frequency error.

Nonuniform Sampling

Although the image functions that we will be sampling are known to have infinite-frequency components and thus can't be perfectly reconstructed from point samples, it is possible to reduce the visual impact of aliasing by varying the spacing between samples in a nonuniform way. If ξ denotes a random number between zero and one, a nonuniform set of samples based on the impulse train is

$$\sum_{i=-\infty}^{\infty} \delta\left(x - \left(iT + \frac{1}{2} - \xi\right)\right).$$

For a fixed sampling rate that isn't sufficient to capture the function, both uniform and nonuniform sampling produce incorrect reconstructed signals. However, nonuniform sampling tends to turn the regular aliasing artifacts into noise, which is less distracting to the human visual system. In frequency space, the copies of the sampled signal end up being randomly shifted as well, so that when reconstruction is performed the result is random error rather than coherent aliasing.

Adaptive Sampling

Another approach that has been suggested to combat aliasing is *adaptive supersampling:* if we can identify the regions of the signal with frequencies higher than the Nyquist limit, we can take additional samples in those regions without needing to incur the computational expense of increasing the sampling frequency everywhere. It can be difficult to get this approach to work well in practice, because finding all of the places where supersampling is needed is difficult. Most techniques for doing so are based on examining adjacent sample values and finding places where there is a significant change in value between the two; the assumption is that the signal has high frequencies in that region.

In general, adjacent sample values cannot tell us with certainty what is really happening between them: Even if the values are the same, the functions may have huge variation between them. Alternatively, adjacent samples may have substantially different values without any aliasing actually being present. For example, the texture-filtering algorithms in Chapter 10 work hard to eliminate aliasing due to image maps and procedural textures on surfaces in the scene; we would not want an adaptive sampling routine to needlessly take extra samples in an area where texture values are changing quickly but no excessively high frequencies are actually present.

Prefiltering

Another approach to eliminating aliasing that sampling theory offers is to filter (i.e., blur) the original function so that no high frequencies remain that can't be captured accurately at the sampling rate being used. This approach is applied in the texture functions of Chapter 10. While this technique changes the character of the function being sampled by removing information from it, it is generally less objectionable than aliasing.

Recall that we would like to multiply the original function's spectrum with a box filter with width chosen so that frequencies above the Nyquist limit are removed. In the spatial domain, this corresponds to convolving the original function with a sinc filter,

$$f(x) \otimes \operatorname{sinc}(2\omega_s x).$$

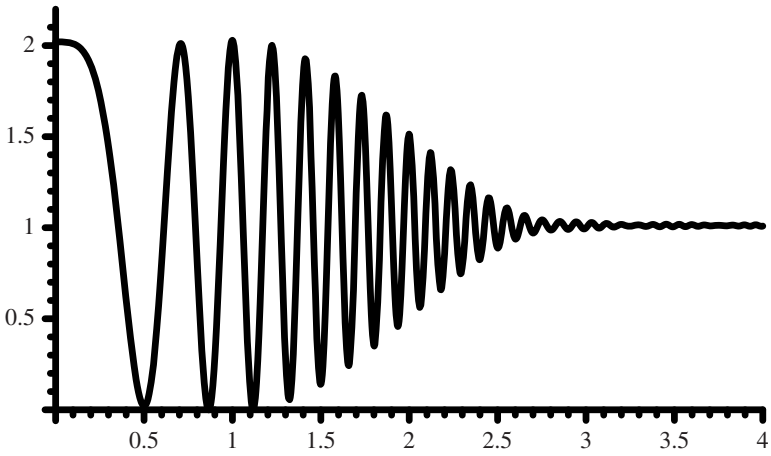


Figure 7.10: Graph of the function $1 + \cos(4x^2)$ convolved with a filter that removes frequencies beyond the Nyquist limit for a sampling rate of $T = .125$. High-frequency detail has been removed from the function, so that the new function can at least be sampled and reconstructed without aliasing.

In practice, we can use a filter with finite extent that works well. The frequency space representation of this filter can help clarify how well it approximates the behavior of the ideal sinc filter.

Figure 7.10 shows the function $1 + \cos(4x^2)$ convolved with a variant of the sinc with finite extent that will be introduced in Section 7.7. Note that the high-frequency details have been eliminated; this function can be sampled and reconstructed at the sampling rate used in Figure 7.9 without aliasing.

7.1.5 APPLICATION TO IMAGE SYNTHESIS

The application of these ideas to the two-dimensional case of sampling and reconstructing images of rendered scenes is straightforward: we have an image, which we can think of as a function of two-dimensional (x, y) image locations to radiance values L:

$$f(x, y) \rightarrow L.$$

The good news is that, with our ray tracer, we can evaluate this function at any (x, y) point that we choose. The bad news is that it's not generally possible to prefilter f to remove the high frequencies from it before sampling. Therefore, the samplers in this chapter will use both strategies of increasing the sampling rate beyond the basic pixel spacing in the final image as well as nonuniformly distributing the samples to turn aliasing into noise.

It is useful to generalize the definition of the scene function to a higher-dimensional function that also depends on the time t and (u, v) lens position at which it is sampled. Because the rays from the camera are based on these five quantities, varying any of them

gives a different ray and thus a potentially different value of f. For a particular image position, the radiance at that point will generally vary across both time (if there are moving objects in the scene) and position on the lens (if the camera has a finite-aperture lens).

Even more generally, because many of the integrators defined in Chapters 15 and 16 use statistical techniques to estimate the radiance along a given ray, they may return a different radiance value when repeatedly given the same ray. If we further extend the scene radiance function to include sample values used by the integrator (e.g., values used to choose points on area light sources for illumination computations), we have an even higher-dimensional image function

$$f(x, y, t, u, v, i_1, i_2, \ldots) \rightarrow L.$$

Sampling all of these dimensions well is an important part of generating high-quality imagery efficiently. For example, if we ensure that nearby (x, y) positions on the image tend to have dissimilar (u, v) positions on the lens, the resulting rendered images will have less error because each sample is more likely to account for information about the scene that its neighboring samples do not. The `Sampler` classes in the next few sections will address the issue of sampling all of these dimensions as well as possible.

7.1.6 SOURCES OF ALIASING IN RENDERING

Geometry is one of the most common causes of aliasing in rendered images. When projected onto the image plane, an object's boundary introduces a step function—the image function's value instantaneously jumps from one value to another. Not only do step functions have infinite frequency content as mentioned earlier, but, even worse, the perfect reconstruction filter causes artifacts when applied to aliased samples: ringing artifacts appear in the reconstructed function, an effect known as the *Gibbs phenomenon*. Figure 7.11 shows an example of this effect for a 1D function. Choosing an effective reconstruction filter in the face of aliasing requires a mix of science, artistry, and personal taste, as we will see later in this chapter.

Very small objects in the scene can also cause geometric aliasing. If the geometry is small enough that it falls between samples on the image plane, it can unpredictably disappear and reappear over multiple frames of an animation.

Another source of aliasing can come from the texture and materials on an object. *Shading aliasing* can be caused by texture maps that haven't been filtered correctly (addressing this problem is the topic of much of Chapter 10), or from small highlights on shiny surfaces. If the sampling rate is not high enough to sample these features adequately, aliasing will result. Furthermore, a sharp shadow cast by an object introduces another step function in the final image. While it is possible to identify the position of step functions from geometric edges on the image plane, detecting step functions from shadow boundaries is more difficult.

The key insight about aliasing in rendered images is that we can never remove all of its sources, so we must develop techniques to mitigate its impact on the quality of the final image.

Sampler 340

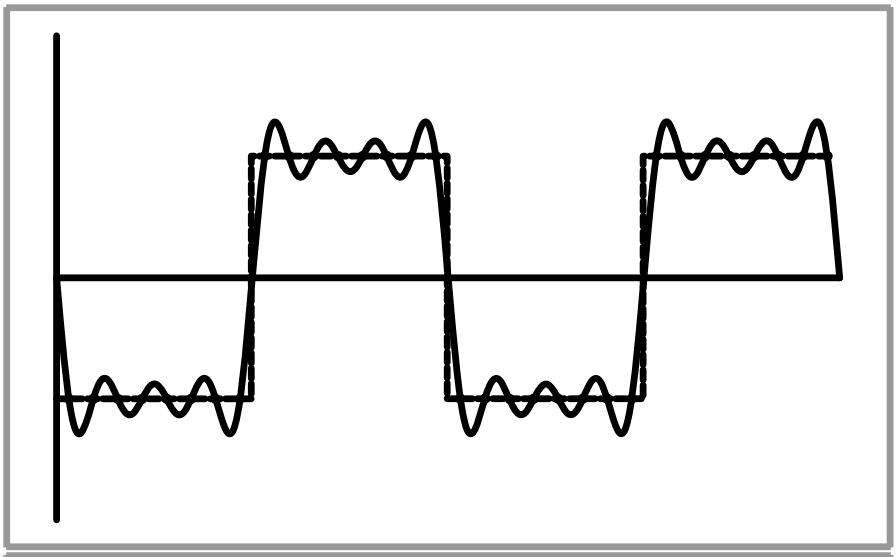


Figure 7.11: Illustration of the Gibbs Phenomenon. When a function hasn't been sampled at the Nyquist rate and the set of aliased samples is reconstructed with the sinc filter, the reconstructed function will have "ringing" artifacts, where it oscillates around the true function. Here a 1D step function (dashed line) has been sampled with a sample spacing of 0.125. When reconstructed with the sinc, the ringing appears (solid line).

7.1.7 UNDERSTANDING PIXELS

There are two ideas about pixels that are important to keep in mind throughout the remainder of this chapter. First, it is crucial to remember that the pixels that constitute an image are point samples of the image function at discrete points on the image plane; there is no "area" associated with a pixel. As Alvy Ray Smith (1995) has emphatically pointed out, thinking of pixels as small squares with finite area is an incorrect mental model that leads to a series of errors. By introducing the topics of this chapter with a signal processing approach, we have tried to lay the groundwork for a more accurate mental model.

The second issue is that the pixels in the final image are naturally defined at discrete integer (x, y) coordinates on a pixel grid, but the `Samplers` in this chapter generate image samples at continuous floating-point (x, y) positions. The natural way to map between these two domains is to round continuous coordinates to the nearest discrete coordinate; this is appealing since it maps continuous coordinates that happen to have the same value as discrete coordinates to that discrete coordinate. However, the result is that given a set of discrete coordinates spanning a range $[x_0, x_1]$, the set of continuous coordinates that covers that range is $[x_0 - 1/2, x_1 + 1/2)$. Thus, any code that generates continuous sample positions for a given discrete pixel range is littered with 1/2 offsets. It is easy to forget some of these, leading to subtle errors.

Sampler 340

If we instead truncate continuous coordinates c to discrete coordinates d by

$$d = \lfloor c \rfloor,$$

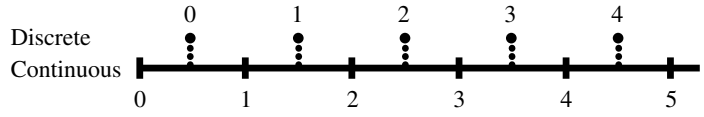


Figure 7.12: Pixels in an image can be addressed with either *discrete* or *continuous* coordinates. A discrete image four pixels wide covers the continuous pixel range $[0, 5)$. A particular discrete pixel d's coordinate in the continuous representation is $d + 1/2$.

and convert from discrete to continuous by

$$c = d + 1/2,$$

then the range of continuous coordinates for the discrete range $[x_0, x_1]$ is naturally $[x_0, x_1 + 1)$ and the resulting code is much simpler (Heckbert 1990a). This convention, which we will adopt in `pbrt`, is shown graphically in Figure 7.12.

7.2 IMAGE SAMPLING INTERFACE

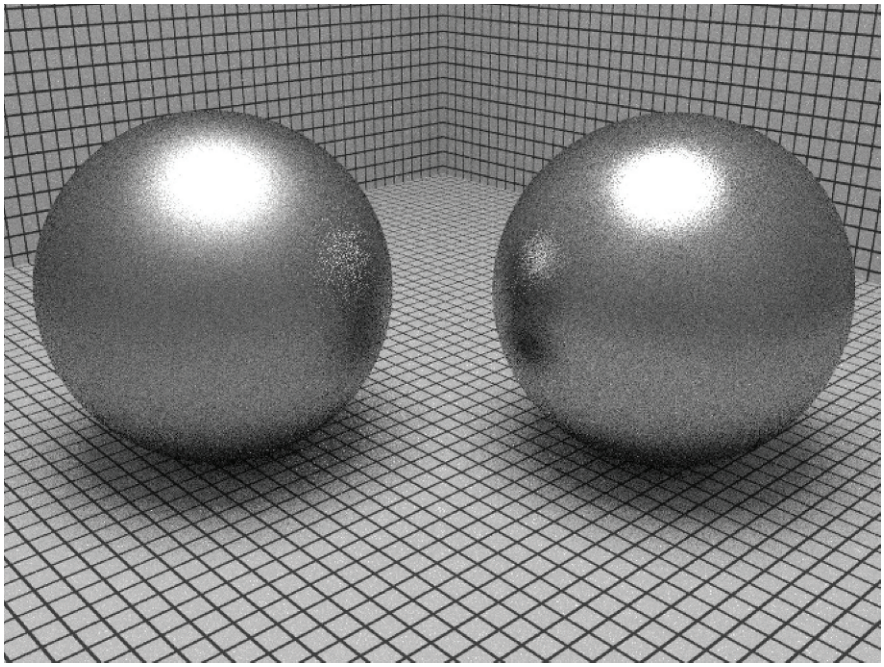
We can now describe the operation of a few classes that generate good image sampling patterns. It may be surprising to see that some of them have a significant amount of complexity behind them. In practice, creating good sample patterns can substantially improve a ray tracer's efficiency, allowing it to create a high-quality image with fewer rays than if a lower-quality pattern was used. Because the run time expense for using the best sampling patterns is approximately the same as for lower-quality patterns, and because evaluating the radiance for each image sample is expensive, doing this work pays dividends (Figure 7.13).

The core sampling declarations and functions are in the files `core/sampler.h` and `core/sampler.cpp`. Each of the sample generation implementations is in its own source file in the `samplers/` directory.

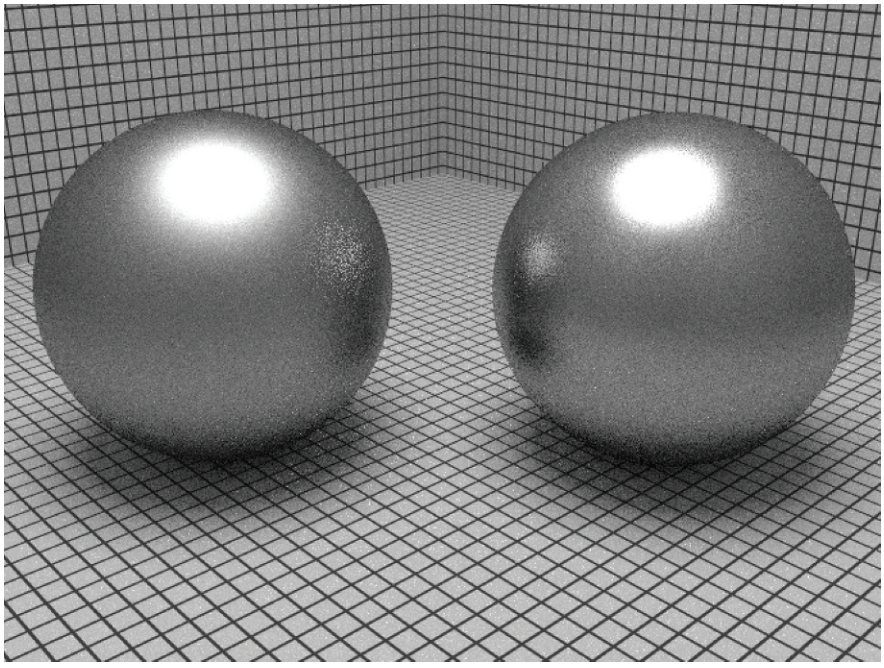
All of the sampler implementations inherit from an abstract `Sampler` class that defines their interface. The task of `Sampler`s is to generate a sequence of multidimensional sample positions. Two dimensions give the raster space image sample position, and another gives the time at which the sample should be taken; this ranges from zero to one, and is scaled by the camera to cover the time period that the shutter is open. Two more sample values give a (u, v) lens position for depth of field; these also vary from zero to one.

Just as well-placed sample points can help conquer the complexity of the 2D image function, most of the light transport algorithms in Chapters 15 and 16 use sample points for tasks like choosing positions on area light sources when estimating illumination. Choosing these points is also the job of the `Sampler`, since it is able to take the sample points chosen for adjacent image samples into account when selecting samples at new points. Doing so can improve the quality of the results of the light transport algorithms.

Sampler 340



(a)



(b)

Figure 7.13: Scene rendered with (a) a relatively ineffective sampler and (b) a carefully designed sampler, using the same number of samples for each. The improvement in image quality, ranging from the edges of the highlights to the quality of the glossy reflections, is noticeable.

⟨*Sampling Declarations*⟩ ≡
```
class Sampler {
public:
    ⟨Sampler Interface 340⟩
    ⟨Sampler Public Data 340⟩
protected:
    ⟨Sampler Protected Methods⟩
};
```

All of the Sampler implementations take a few common parameters that must be passed to the base class's constructor. These include the overall image resolution in the x and y dimensions, the number of samples the implementation expects to generate for each pixel in the final image, and the range of time over which the camera's shutter is open. These values are stored in member variables for later use.

⟨*Sampler Method Definitions*⟩ ≡
```
Sampler::Sampler(int xstart, int xend, int ystart, int yend, int spp,
                 float sopen, float sclose)
    : xPixelStart(xstart), xPixelEnd(xend), yPixelStart(ystart),
      yPixelEnd(yend), samplesPerPixel(spp), shutterOpen(sopen),
      shutterClose(sclose) { }
```

The Sampler implementation should generate samples for pixels with x coordinates ranging from xPixelStart to xPixelEnd-1, inclusive, and analogously for y coordinates. The time values for samples should similarly be in the range shutterOpen to shutterClose.

⟨*Sampler Public Data*⟩ ≡ 340
```
const int xPixelStart, xPixelEnd, yPixelStart, yPixelEnd;
const int samplesPerPixel;
const float shutterOpen, shutterClose;
```

Samplers must implement the Sampler::GetMoreSamples() method, which is a pure virtual function. The SamplerRendererTask::Run() method calls this function until it returns zero, which signifies that all samples have been generated. Otherwise, it generates one or more samples, returning the number of samples generated and filling in sample values in the array pointed to by the sample parameter. All of the dimensions of the sample values it generates should have values in the range [0, 1], except for CameraSample::imageX and CameraSample::imageY, which are specified with respect to the image size in raster coordinates.

The caller also passes a RNG for the sampler to use for any random number generation it needs to do; Samplers could equivalently store their own RNGs, though code that calls this method generally already has one available.

⟨*Sampler Interface*⟩ ≡ 340
```
virtual int GetMoreSamples(Sample *sample, RNG &rng) = 0;
```

The MaximumSampleCount() method returns the maximum number of sample values that the Sampler will ever return from its implementation of the GetMoreSamples() method. It allows the renderer to preallocate memory for the array of Samples passed into GetMoreSamples().

CameraSample::imageX 342
RNG 1003
Sample 343
Sampler 340
Sampler::
GetMoreSamples() 340
Sampler::samplesPerPixel 340
Sampler::shutterClose 340
Sampler::shutterOpen 340
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
Sampler::yPixelStart 340
SamplerRendererTask::Run() 30

⟨*Sampler Interface*⟩ +≡ 340
```
virtual int MaximumSampleCount() = 0;
```

Samplers may implement the ReportResults() method; it allows the renderer to report back to the sampler which rays were generated, what radiance values were computed, and the intersection points found for a collection of samples originally from GetMoreSamples(). The sampler may use this information for adaptive sampling algorithms, deciding to take more samples close to the ones that were returned here.

The return value indicates whether or not the sample values should be added to the image being generated. For some adaptive sampling algorithms, the sampler may want to cause an initial collection of samples (and their results) to be discarded, generating a new set to replace them completely. Because most of the Samplers in this chapter do not implement adaptive sampling, a default implementation of this method just returns true.

⟨*Sampler Method Definitions*⟩ +≡
```
bool Sampler::ReportResults(Sample *samples, const RayDifferential *rays,
        const Spectrum *Ls, const Intersection *isects, int count) {
    return true;
}
```

Recall from Section 1.3.4 that the SamplerRenderer refines the task of rendering the complete image into a number of tasks that can execute in parallel. The Sampler::GetSubSampler() method is a key part of this process. As the scene description file is parsed, a single Sampler is created; conceptually, its responsibility is to generate samples for the entire image. The GetSubSampler() method returns a new Sampler that is responsible for generating samples for a subset of the image; its num parameter ranges from 0 to count-1, where count is the total number of subsamplers being used. Implementations of this method must ensure that the aggregate set of generated samples by all of the subsamplers covers the image; how they decompose the image is not defined.

⟨*Sampler Interface*⟩ +≡ 340
```
virtual Sampler *GetSubSampler(int num, int count) = 0;
```

Most implementations of the GetSubSampler() method decompose the image into rectangular tiles and have each subsampler generate samples for a single tile. ComputeSubWindow() is a utility function that computes a pixel sampling range given a tile number num and a total number of tiles count.

⟨*Sampler Method Definitions*⟩ +≡
```
void Sampler::ComputeSubWindow(int num, int count, int *newXStart,
        int *newXEnd, int *newYStart, int *newYEnd) const {
    ⟨Determine how many tiles to use in each dimension, nx and ny 342⟩
    ⟨Compute x and y pixel sample range for sub-window 342⟩
}
```

Intersection 186
RayDifferential 69
Sample 343
Sampler 340
SamplerRenderer 25
Spectrum 263

It's desirable that the shapes of image tiles be roughly square (as opposed to, for example, being one pixel tall and many pixels wide). This leads to more coherent access to the scene data in memory and potentially to better performance. Starting with the total number of tiles in count, the number of tiles in the x dimension is successively halved while the number in y is doubled until nx is an odd number or the number of x tiles times

the y image resolution is less than or equal to the number of y tiles times the x image resolution. (When nx is odd, we can no longer halve it and have nx * ny == count.) This is a reasonable effort to make square tiles; a more complex implementation probably isn't worthwhile.

⟨*Determine how many tiles to use in each dimension,* nx *and* ny⟩ ≡ 341
```
int dx = xPixelEnd - xPixelStart, dy = yPixelEnd - yPixelStart;
int nx = count, ny = 1;
while ((nx & 0x1) == 0 && 2 * dx * ny < dy * nx) {
    nx >>= 1;
    ny <<= 1;
}
```

Given the number of tiles in each direction, the integer tile number (xo, yo) is easily computed and hence the pixel range.

⟨*Compute x and y pixel sample range for sub-window*⟩ ≡ 341
```
int xo = num % nx, yo = num / nx;
float tx0 = float(xo) / float(nx), tx1 = float(xo+1) / float(nx);
float ty0 = float(yo) / float(ny), ty1 = float(yo+1) / float(ny);
*newXStart = Floor2Int(Lerp(tx0, xPixelStart, xPixelEnd));
*newXEnd   = Floor2Int(Lerp(tx1, xPixelStart, xPixelEnd));
*newYStart = Floor2Int(Lerp(ty0, yPixelStart, yPixelEnd));
*newYEnd   = Floor2Int(Lerp(ty1, yPixelStart, yPixelEnd));
```

7.2.1 SAMPLE REPRESENTATION AND ALLOCATION

The Sample structure is used by Samplers to store a single sample. After one or more Samples is initialized by a call to the Sampler's GetMoreSamples() method, the Sampler RendererTask::Run() method passes each of the Samples to the camera and integrators, which read values from it to construct the camera ray and perform lighting calculations.

Sample inherits from the CameraSample structure; the CameraSample represents just the sample values that are needed for generating camera rays. This separation allows us to just pass a CameraSample to the Camera::GenerateRay() method, not exposing all of the details of the rest of the Sample structure to Cameras.

⟨*Sampling Declarations*⟩ +≡
```
struct CameraSample {
    float imageX, imageY;
    float lensU, lensV;
    float time;
};
```

Depending on the details of the light transport algorithm being used, different integrators may have different sampling needs beyond the camera sample values. For example, the WhittedIntegrator doesn't do any random sampling, so it doesn't need any additional sample values beyond the ones to generate the camera ray. The DirectLighting

Camera::GenerateRay() 303
Floor2Int() 1002
Lerp() 1000
Sample 343
Sampler 340
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
Sampler::yPixelStart 340
SamplerRendererTask::Run() 30
WhittedIntegrator 42

Integrator uses values from the Sampler to randomly choose a light source to sample illumination from, as well as to randomly choose positions on area light sources. Therefore, the integrators are given an opportunity to request additional sample values in various quantities. Information about these requirements is stored in the Sample object. When it is later passed to the particular Sampler implementation, it is the Sampler's responsibility to generate all of the requested types of samples.

⟨*Sampling Declarations*⟩ +≡
```
struct Sample : public CameraSample {
    ⟨Sample Public Methods 344⟩
    ⟨Sample Public Data 344⟩
private:
    ⟨Sample Private Methods⟩
};
```

The Sample constructor calls the Integrator::RequestSamples() methods of the surface and volume integrators to find out what samples they will need. The integrators can ask for multiple one-dimensional and/or two-dimensional sampling patterns, each with an arbitrary number of entries. For example, in a scene with two area light sources, where the integrator traces four shadow rays to the first source and eight to the second, the integrator would ask for two 2D sample patterns for each image sample, with four and eight samples each. A 2D pattern is required because two dimensions are needed to parameterize the surface of a light. Similarly, if the integrator wanted to randomly select a single light source out of many, it could request a 1D sample with a single value for this purpose and use its float value to randomly choose a light.

By informing the Sampler of as much of its random sampling needs as possible, the Integrator allows the Sampler to carefully construct sample points that cover the entire high-dimensional sample space well. For example, the final image is generally better when neighboring image samples tend to sample different positions on the area lights for their illumination computations, allowing more information to be discovered.

In pbrt, we don't allow integrators to request 3D or higher-dimensional sample patterns because these are generally not needed for the types of rendering algorithms implemented here. If necessary, an integrator can combine points from lower-dimensional patterns to get higher-dimensional sample points (e.g., a 1D and a 2D sample pattern of the same size can form a 3D pattern). Although this may not be as good as generating a 3D sample pattern directly, it is usually works well in practice. If absolutely necessary, the integrator can always generate a 3D sample pattern itself, though with less ability to easily ensure that nearby image samples sample different regions of the 3D space.

The integrators' implementations of the Integrator::RequestSamples() method in turn call the Sample::Add1D() and Sample::Add2D() methods, which request another sample sequence with a given number of sample values. After they are done calling these methods, the Sample constructor can continue, allocating storage for the requested sample values.

CameraSample 342
DirectLightingIntegrator 742
Integrator::
RequestSamples() 740
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sampler 340

⟨*Sample Method Definitions*⟩ ≡
```
Sample::Sample(Sampler *sampler, SurfaceIntegrator *surf,
               VolumeIntegrator *vol, const Scene *scene) {
    if (surf) surf->RequestSamples(sampler, this, scene);
    if (vol)  vol->RequestSamples(sampler, this, scene);
    AllocateSampleMemory();
}
```

The implementations of the `Sample::Add1D()` and `Sample::Add2D()` methods record the number of samples asked for in an array and return an index that the integrator can later use to access the desired sample values in the `Sample`.

⟨*Sample Public Methods*⟩ ≡ 343
```
uint32_t Add1D(uint32_t num) {
    n1D.push_back(num);
    return n1D.size()-1;
}
```

⟨*Sample Public Methods*⟩ +≡ 343
```
uint32_t Add2D(uint32_t num) {
    n2D.push_back(num);
    return n2D.size()-1;
}
```

Most `Samplers` can do a better job of generating particular quantities of these additional samples than others. For example, the `LDSampler` can generate extremely good patterns, although they must have a size that is a power of two. The `Sampler::RoundSize()` method helps communicate this information. Integrators should call this method with the desired number of samples to be taken, giving the `Sampler` an opportunity to adjust the number of samples to a more convenient one. The integrator should then use the returned value as the number of samples to request from the `Sampler`.

⟨*Sampler Interface*⟩ +≡ 340
```
virtual int RoundSize(int size) const = 0;
```

It is the `Sampler`'s responsibility to store the samples it generates for the integrators in the `Sample::oneD` and `Sample::twoD` arrays. For 1D sample patterns, it needs to generate `n1D.size()` independent patterns, where the `i`th pattern has `n1D[i]` sample values. These values are stored in `oneD[i][0]` through `oneD[i][n1D[i]-1]`.

⟨*Sample Public Data*⟩ ≡ 343
```
vector<uint32_t> n1D, n2D;
float **oneD, **twoD;
```

To access the samples, the integrator stores the sample tag returned by `Add1D()` in a member variable (for example, `sampleOffset`) and can then access the sample values in a loop like

Integrator::
RequestSamples() 740
LDSampler 373
Sample 343
Sample::Add1D() 344
Sample::Add2D() 344
Sample::
AllocateSampleMemory() 345
Sample::n1D 344
Sample::n2D 344
Sampler 340
Sampler::RoundSize() 344
Scene 22
SurfaceIntegrator 740
VolumeIntegrator 876

```
for (i = 0; i < sample->n1D[sampleOffset]; ++i) {
    float s = sample->oneD[sampleOffset][i];
    .
    .
    .
}
```

In 2D, the process is equivalent, but the `i`th sample is given by the two values `twoD[offset][2*i]` and `twoD[offset][2*i+1]`.

After all sample requests have been made, the `AllocateSampleMemory()` method allocates space for their storage.

⟨*Sample Method Definitions*⟩ +≡

```
void Sample::AllocateSampleMemory() {
    ⟨Allocate storage for sample pointers 345⟩
    ⟨Compute total number of sample values needed 345⟩
    ⟨Allocate storage for sample values 346⟩
}
```

This method first allocates memory to store the pointers. Rather than performing two allocations, it does a single allocation that gives enough memory for the pointers for both the `oneD` and `twoD` sample arrays. `twoD` is then set to point at an appropriate offset into this memory, after the last pointer for `oneD`. Splitting up a single allocation like this is useful because it ensures that `oneD` and `twoD` point to nearby locations in memory, which is likely to reduce cache misses.

⟨*Allocate storage for sample pointers*⟩ ≡ 345

```
int nPtrs = n1D.size() + n2D.size();
if (!nPtrs) {
    oneD = twoD = NULL;
    return;
}
oneD = AllocAligned<float *>(nPtrs);
twoD = oneD + n1D.size();
```

It then uses the same trick to allocate memory for the actual sample values so that they are contiguous in memory as well. First, it finds the total number of `float` values needed.

⟨*Compute total number of sample values needed*⟩ ≡ 345

```
int totSamples = 0;
for (uint32_t i = 0; i < n1D.size(); ++i)
    totSamples += n1D[i];
for (uint32_t i = 0; i < n2D.size(); ++i)
    totSamples += 2 * n2D[i];
```

AllocAligned() 1013
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344

The method can now allocate a single chunk of memory and hand it out in pieces for the various collections of samples.

⟨*Allocate storage for sample values*⟩ ≡ 345
```
float *mem = AllocAligned<float>(totSamples);
for (uint32_t i = 0; i < n1D.size(); ++i) {
    oneD[i] = mem;
    mem += n1D[i];
}
for (uint32_t i = 0; i < n2D.size(); ++i) {
    twoD[i] = mem;
    mem += 2 * n2D[i];
}
```

The `Sample` destructor, not shown here, just frees the dynamically allocated memory.

A unique instance of the `Sample` structure is created for each rendering task in Section 1.3.4; the housekeeping for doing so is handled by the `Sample::Duplicate()` method. Most of the work involves allocating memory for storing sample values. The implementation is straightforward and not included here.

⟨*Sample Public Methods*⟩ +≡ 343
```
Sample *Duplicate(int count) const;
```

7.3 STRATIFIED SAMPLING

The first sample generator that we will introduce divides the image plane into rectangular regions and generates a single sample inside each region. These regions are commonly called *strata*, and this sampler is called the `StratifiedSampler`. The key idea behind stratification is that by subdividing the sampling domain into nonoverlapping regions and taking a single sample from each one, we are less likely to miss important features of the image entirely, since the samples are guaranteed not to all be close together. Put another way, it does us no good if many samples are taken from nearby points in the sample space, since each new sample doesn't add much new information about the behavior of the image function. From a signal processing viewpoint, we are implicitly defining an overall sampling rate such that the smaller the strata are, the more of them we have, and thus the higher the sampling rate.

The stratified sampler places each sample at a random point inside each stratum by *jittering* the center point of the stratum by a random amount up to half the stratum's width and height. The nonuniformity that results from this jittering helps turn aliasing into noise, as discussed in Section 7.1. The sampler also offers a nonjittered mode, which gives uniform sampling in the strata; this mode is mostly useful for comparisons between different sampling techniques rather than for rendering final images.

Figure 7.14 shows a comparison of a few sampling patterns. The first is a completely random sampling pattern: we generated a number of samples at random without using the strata at all. The result is a terrible sampling pattern; some regions have few samples and other areas have clumps of many samples. The second is a uniform stratified pattern. In the last, the uniform pattern has been jittered, with a random offset added to each sample's location, keeping it inside its cell. This gives a better overall distribution than

AllocAligned() 1013
Sample 343
Sample::Duplicate() 346
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344
StratifiedSampler 349

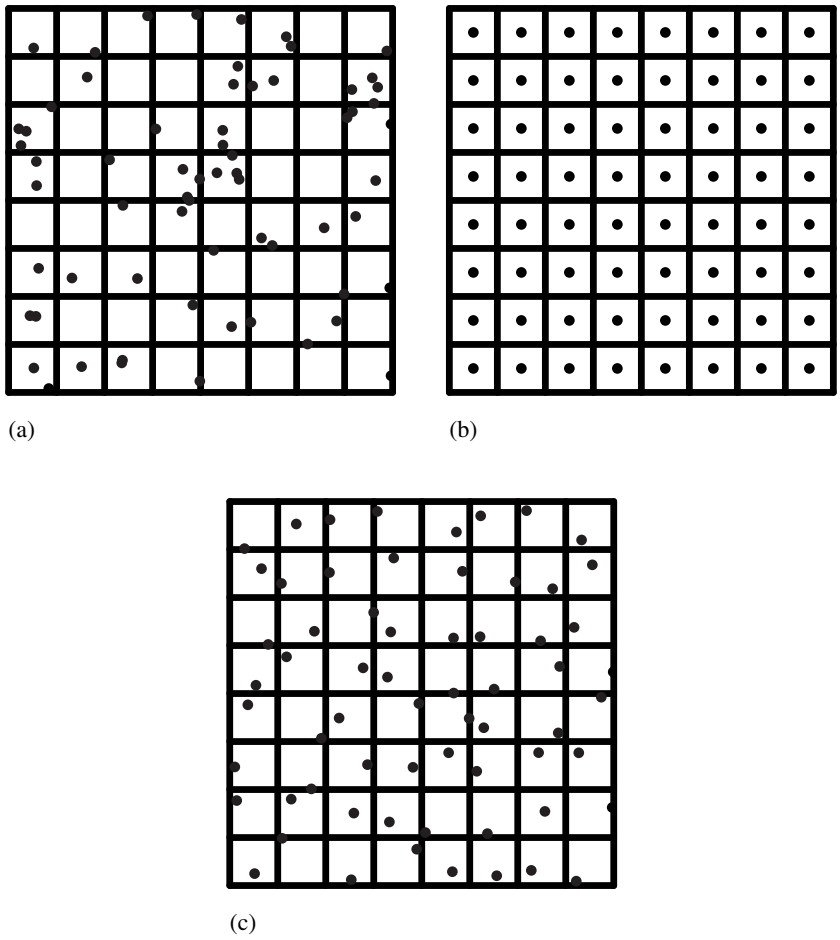


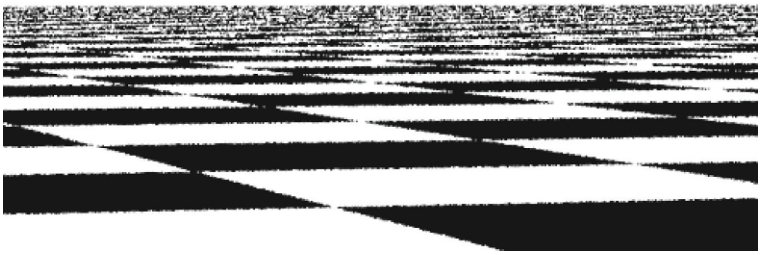
Figure 7.14: Three 2D Sampling Patterns. (a) The random pattern is an ineffective pattern, with many clumps of samples that leave large sections of the image poorly sampled. (b) A uniform stratified pattern is better distributed but can exacerbate aliasing artifacts. (c) A stratified jittered pattern turns aliasing from the uniform pattern into high-frequency noise while still maintaining the benefits of stratification.

the purely random pattern while preserving the benefits of stratification, though there are still some clumps of samples and some regions that are undersampled. We will present more sophisticated image sampling methods in the next two sections that ameliorate some of these remaining shortcomings. Figure 7.15 shows images rendered using the `StratifiedSampler` and shows how jittered sample positions turn aliasing artifacts into less objectionable noise.

StratifiedSampler 349

(a)

(b)



(c)

(d)

Figure 7.15: Comparison of Image Sampling Methods with a Checkerboard Texture. This is a difficult image to render well, since the checkerboard's frequency with respect to the pixel spacing tends toward infinity as we approach the horizon. (a) A reference image, rendered with 256 samples per pixel, showing something close to an ideal result. (b) An image rendered with one sample per pixel, with no jittering. Note the jaggy artifacts at the edges of checks in the foreground. Notice also the artifacts in the distance where the checker function goes through many cycles between samples; as expected from the signal processing theory presented earlier, that detail reappears incorrectly as lower-frequency aliasing. (c) The result of jittering the image samples, still with just one sample per pixel. The regular aliasing of the second image has been replaced by less objectionable noise artifacts. (d) The result of four jittered samples per pixel is still inferior to the reference image but is substantially better than the previous result.

⟨*StratifiedSampler Declarations*⟩ ≡
```
class StratifiedSampler : public Sampler {
public:
    ⟨StratifiedSampler Public Methods 349⟩
private:
    ⟨StratifiedSampler Private Data 349⟩
};
```

The StratifiedSampler generates samples by looping over the pixels from left to right and top to bottom, generating all of the samples for the strata in each pixel before advancing to the next pixel. The constructor takes the range of pixels to generate samples for—[xstart,ystart] to [xend-1,yend-1], inclusive—the number of strata in x and y (xs and ys), a Boolean (jitter) that indicates whether the samples should be jittered, and the range of time values where the camera shutter is open, [sopen, sclose].

The implementation of the GetMoreSamples() method below generates all xs * ys samples for a pixel at once; scratchpad memory that it uses in this process is allocated in the constructor and stored in sampleBuf.

⟨*StratifiedSampler Method Definitions*⟩ ≡
```
StratifiedSampler::StratifiedSampler(int xstart, int xend,
        int ystart, int yend, int xs, int ys, bool jitter,
        float sopen, float sclose)
    : Sampler(xstart, xend, ystart, yend, xs * ys, sopen, sclose) {
    jitterSamples = jitter;
    xPos = xPixelStart;
    yPos = yPixelStart;
    xPixelSamples = xs;
    yPixelSamples = ys;
    sampleBuf = new float[5 * xPixelSamples * yPixelSamples];
}
```

The sampler holds the coordinate of the current pixel in the xPos and yPos member variables, which are initialized to point to the pixel in the upper left of the portion of the image that the sampler is responsible for. In addition to the tiling used for parallelization, the presence of a crop window and implications of sample filtering often cause this corner to start a location other than (0, 0).

Integrator 740
Sampler 340
Sampler::xPixelStart 340
Sampler::yPixelStart 340
StratifiedSampler 349
StratifiedSampler::jitterSamples 349
StratifiedSampler::sampleBuf 349
StratifiedSampler::xPixelSamples 349
StratifiedSampler::xPos 349
StratifiedSampler::yPixelSamples 349
StratifiedSampler::yPos 349

⟨*StratifiedSampler Private Data*⟩ ≡ 349
```
int xPixelSamples, yPixelSamples;
bool jitterSamples;
int xPos, yPos;
float *sampleBuf;
```

The StratifiedSampler has no preferred sizes for the number of additional samples generated for the Integrators.

⟨*StratifiedSampler Public Methods*⟩ ≡ 349
```
int RoundSize(int size) const { return size; }
```

The GetSubSampler() method uses the ComputeSubwindow() utility routine to compute an image tile extent for the given subsampler number and then returns a new instance of a StratifiedSampler that generates samples in just that region. ComputeSubWindow() may return an empty pixel extent, where one or both of the x and y ranges are degenerate. (Consider, for example, a 2×2 image; it can't be decomposed into more than four image tiles if the tiles are no smaller than a pixel's extent.) In this case, NULL is returned. Calling code must handle this case where a finer decomposition of the image is requested than the Sampler is able to support by ignoring any returned NULL pointers.

⟨*StratifiedSampler Method Definitions*⟩ +≡
```
Sampler *StratifiedSampler::GetSubSampler(int num, int count) {
    int x0, x1, y0, y1;
    ComputeSubWindow(num, count, &x0, &x1, &y0, &y1);
    if (x0 == x1 || y0 == y1) return NULL;
    return new StratifiedSampler(x0, x1, y0, y1, xPixelSamples,
        yPixelSamples, jitterSamples, shutterOpen, shutterClose);
}
```

Direct application of stratification to high-dimensional sampling quickly leads to an intractable number of samples. For example, if we divided the five-dimensional image, lens, and time sample space into four strata in each dimension, the total number of samples per pixel would be $4^5 = 1024$. We could reduce this impact by taking fewer samples in some dimensions (or not stratifying some dimensions, effectively using a single stratum), but we would then lose the benefit of having well-stratified samples in those dimensions. This problem with stratification is known as the *curse of dimensionality*.

We can reap most of the benefits of stratification without paying the price in excessive total sampling by computing lower-dimensional stratified patterns for subsets of the domain's dimensions and then randomly associating samples from each set of dimensions. (This process is sometimes called *padding*.) Figure 7.16 shows the basic idea: we might want to take just four samples per pixel, but still have the samples be stratified over all dimensions. We independently generate four 2D stratified image samples, four 1D stratified time samples, and four 2D stratified lens samples. Then, we randomly associate a time and lens sample value with each image sample. The result is that each pixel has samples that together have good coverage of the sample space. Figure 7.17 shows the improvement in image quality from using stratified lens samples versus using unstratified random samples when rendering depth of field.

Given this approach, it's now possible to implement the GetMoreSamples() method of the StratifiedSampler. It starts by checking whether it is done generating samples for its region—because it generates samples starting from (xPixelStart, yPixelStart) and then first scans across x and then down y, it is done when yPos equals yPixelEnd. If it is done, a return value of zero indicates to the caller that the sampler has completed its work. Otherwise, the method generates a pixel's worth of samples, returning them in the samples array.

Sampler 340
Sampler::
ComputeSubWindow() 341
Sampler::shutterClose 340
Sampler::shutterOpen 340
StratifiedSampler 349
StratifiedSampler::
jitterSamples 349
StratifiedSampler::
xPixelSamples 349
StratifiedSampler::
yPixelSamples 349

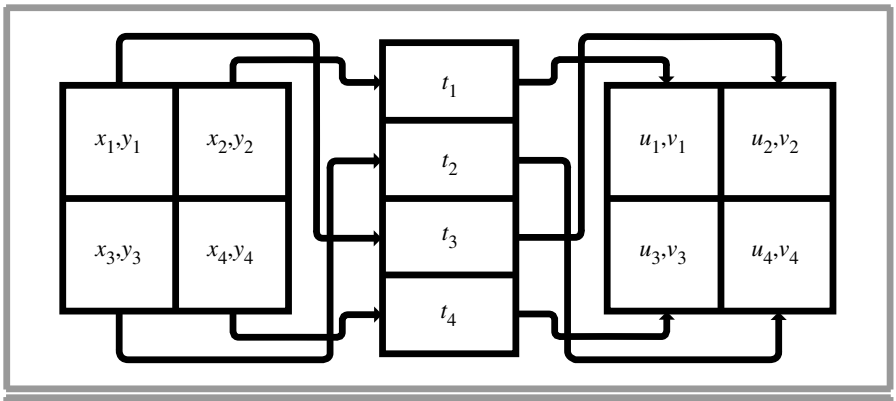


Figure 7.16: We can generate a good sample pattern that reaps the benefits of stratification without requiring that all of the sampling dimensions be stratified simultaneously. Here, we have split (x, y) image position, time t, and (u, v) lens position into independent strata with four regions each. Each is sampled independently, then a time sample and a lens sample are randomly associated with each image sample. We retain the benefits of stratification in each of the individual dimensions without having to exponentially increase the total number of samples.

(a) (b) (c)

Figure 7.17: Effect of Sampling Patterns in Rendering a Red Sphere Image with Depth of Field. (a) A high-quality reference image of the blurred edge of a sphere. (b) An image generated with random sampling in each pixel without stratification. (c) An image generated with the same number of samples, but with the `StratifiedSampler`, which stratified both the image and, more importantly for this image, the lens samples. Stratification makes a substantial improvement for this situation.

⟨*StratifiedSampler Method Definitions*⟩ +≡

```
int StratifiedSampler::GetMoreSamples(Sample *samples, RNG &rng) {
    if (yPos == yPixelEnd) return 0;
    int nSamples = xPixelSamples * yPixelSamples;
    ⟨Generate stratified camera samples for (xPos, yPos) 352⟩
    ⟨Advance to next pixel for stratified sampling 357⟩
    return nSamples;
}
```

RNG 1003
Sample 343
StratifiedSampler 349

⟨*Generate stratified camera samples for* (xPos, yPos)⟩ ≡ 351
⟨*Generate initial stratified samples into* sampleBuf *memory* 352⟩
⟨*Shift stratified image samples to pixel coordinates* 353⟩
⟨*Decorrelate sample dimensions* 353⟩
⟨*Initialize stratified* samples *with sample values* 354⟩

First, stratified sampling utility routines are used to generate two 2D and one 1D sampling patterns for the camera samples. These sample values are stored in the sampleBuf memory for now.

⟨*Generate initial stratified samples into* sampleBuf *memory*⟩ ≡ 352
```
float *bufp = sampleBuf;
float *imageSamples = bufp; bufp += 2 * nSamples;
float *lensSamples = bufp;  bufp += 2 * nSamples;
float *timeSamples = bufp;
StratifiedSample2D(imageSamples, xPixelSamples, yPixelSamples, rng,
                   jitterSamples);
StratifiedSample2D(lensSamples, xPixelSamples, yPixelSamples, rng,
                   jitterSamples);
StratifiedSample1D(timeSamples, xPixelSamples * yPixelSamples, rng,
                   jitterSamples);
```

The 1D and 2D stratified sampling routines are implemented as utility functions, since they will be useful elsewhere in pbrt. Both of them loop over the given number of strata over the [0, 1] domain and place a sample value in each one.

⟨*Sampling Function Definitions*⟩ ≡
```
void StratifiedSample1D(float *samp, int nSamples, RNG &rng,
                        bool jitter) {
    float invTot = 1.f / nSamples;
    for (int i = 0;  i < nSamples; ++i) {
        float delta = jitter ? rng.RandomFloat() : 0.5f;
        *samp++ = (i + delta) * invTot;
    }
}
```

⟨*Sampling Function Definitions*⟩ +≡
```
void StratifiedSample2D(float *samp, int nx, int ny, RNG &rng,
                        bool jitter) {
    float dx = 1.f / nx, dy = 1.f / ny;
    for (int y = 0; y < ny; ++y)
        for (int x = 0; x < nx; ++x) {
            float jx = jitter ? rng.RandomFloat() : 0.5f;
            float jy = jitter ? rng.RandomFloat() : 0.5f;
            *samp++ = (x + jx) * dx;
            *samp++ = (y + jy) * dy;
        }
}
```

RNG 1003
RNG::RandomFloat() 1003
StratifiedSample1D() 352
StratifiedSample2D() 352
StratifiedSampler::
jitterSamples 349
StratifiedSampler::
sampleBuf 349
StratifiedSampler::
xPixelSamples 349
StratifiedSampler::
yPixelSamples 349

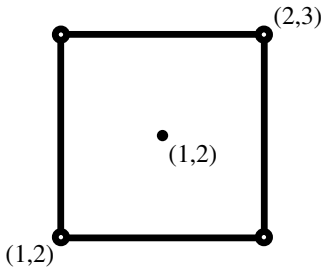


Figure 7.18: To generate stratified samples for a discrete pixel $(1, 2)$, all that needs to be done is to add $(1, 2)$ to the elements of the samples generated over $[0, 1)^2$. This gives samples with continuous pixel coordinates from $(1, 2)$ to $(2, 3)$, which end up surrounding the discrete pixel $(1, 2)$.

The `StratifiedSample2D()` utility function generates samples in the range $[0, 1)^2$, but image samples need to be expressed in terms of continuous pixel coordinates. Therefore, `GetMoreSamples()` next loops over all of the new stratified samples and adds the current (x, y) pixel number, so that the samples for the discrete pixel (x, y) range over the continuous coordinates $[x, x + 1) \times [y, y + 1)$, following the convention for continuous pixel coordinates described in Section 7.1.7. Figure 7.18 reviews the relationship between discrete and continuous coordinates as it relates to samples for a pixel.

⟨*Shift stratified image samples to pixel coordinates*⟩ ≡ 352
```
for (int o = 0; o < 2 * xPixelSamples * yPixelSamples; o += 2) {
    imageSamples[o]   += xPos;
    imageSamples[o+1] += yPos;
}
```

In order to randomly associate a time and lens sample with each image sample, this method next shuffles the order of the time and lens sample arrays. Thus, when it later initializes a `Sample` with the `i`th precomputed sample value for this pixel, it can just return the `i`th time and lens sample.

`Shuffle()` 354
`StratifiedSample2D()` 352
`StratifiedSampler::xPixelSamples` 349
`StratifiedSampler::xPos` 349
`StratifiedSampler::yPixelSamples` 349
`StratifiedSampler::yPos` 349

⟨*Decorrelate sample dimensions*⟩ ≡ 352
```
Shuffle(lensSamples, xPixelSamples*yPixelSamples, 2, rng);
Shuffle(timeSamples, xPixelSamples*yPixelSamples, 1, rng);
```

The `Shuffle()` utility function randomly permutes a sample pattern of `count` samples in `dims` dimensions.

⟨*Monte Carlo Utility Declarations*⟩ ≡
```
template <typename T>
void Shuffle(T *samp, uint32_t count, uint32_t dims, RNG &rng) {
    for (uint32_t i = 0; i < count; ++i) {
        uint32_t other = i + (rng.RandomUInt() % (count - i));
        for (uint32_t j = 0; j < dims; ++j)
            swap(samp[dims*i + j], samp[dims*other + j]);
    }
}
```

Now that the camera samples have been offset and the lens and time samples have been shuffled, the sample values can be copied out of the temporary buffer and stored in the array of `samples` passed to `GetMoreSamples()`. As this step is performed, sample values for the samples requested by the integrators are generated for each of the returned samples as well.

⟨*Initialize stratified* `samples` *with sample values*⟩ ≡ 352
```
for (int i = 0; i < nSamples; ++i) {
    samples[i].imageX = imageSamples[2*i];
    samples[i].imageY = imageSamples[2*i+1];
    samples[i].lensU = lensSamples[2*i];
    samples[i].lensV = lensSamples[2*i+1];
    samples[i].time = Lerp(timeSamples[i], shutterOpen, shutterClose);
    ⟨Generate stratified samples for integrators 356⟩
}
```

Integrators introduce a new complication since they often use multiple samples per image sample in some dimensions rather than a single sample value like the camera does for lens position and time. As we have described the topic of sampling so far, this leaves us with a quandary: if an integrator asks for a set of 64 two-dimensional sample values for each image sample, the sampler has two different goals to try to fulfill:

1. We would like each image sample's 64 integrator samples to themselves be well distributed in 2D (i.e., with an 8×8 stratified grid). Stratification here will improve the quality of the integrator's results for each individual sample.
2. We would like to ensure that the set of integrator samples for one image sample isn't too similar to the samples for its neighboring image samples. As with time and lens samples, we'd like the points to be well distributed with respect to their neighbors, so that over the region around a single pixel, there is good coverage of the entire sample space.

Rather than trying to solve both of these problems simultaneously here, the `StratifiedSampler` will only address the first one. The other samplers later in this chapter will revisit this issue with more sophisticated techniques and solve both of them simultaneously to various degrees.

A second integrator-related complication comes from the fact that they may ask for an arbitrary number of samples per image sample, so stratification may not be easily applied. (For example, how do we generate a stratified 2D pattern of seven samples?) We could just generate an $n \times 1$ or $1 \times n$ stratified pattern, but this only gives us the

CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
Lerp() 1000
RNG 1003
RNG::RandomUInt() 1003
Sampler::shutterClose 340
Sampler::shutterOpen 340
StratifiedSampler 349

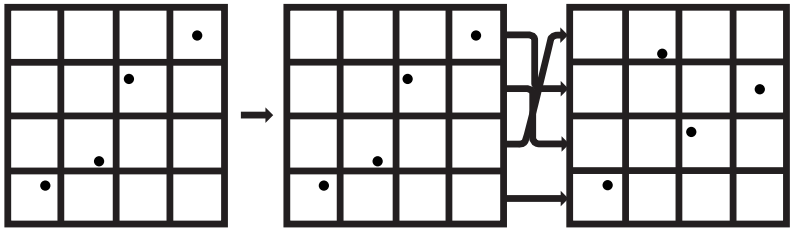


Figure 7.19: Latin hypercube sampling (sometimes called n-rooks sampling) chooses samples such that only a single sample is present in each row and each column of a grid. This can be done by generating random samples in the cells along the diagonal and then randomly permuting their coordinates. One advantage of LHS is that it can generate any number of samples with a good distribution, not just $m \times n$ samples, as with stratified patterns.

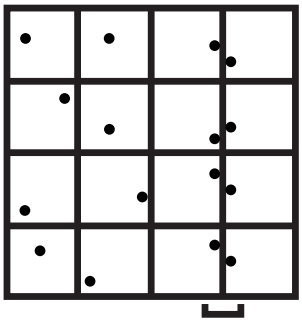


Figure 7.20: A Worst-Case Situation for Stratified Sampling. In an $n \times n$ 2D pattern, up to $2n$ of the points may project to essentially the same point on one of the axes. When "unlucky" patterns like this are generated, the quality of the results computed with them usually suffers.

benefit of stratification in one dimension, and no guarantee of a good pattern in the other dimension. The `StratifiedSampler::RoundSize()` method could round requests up to the next number that's the square of integers, but instead we will use an approach called *Latin hypercube sampling* (LHS), which can generate any number of samples in any number of dimensions with reasonably good distribution.

LHS uniformly divides each dimension's axis into n regions and generates a jittered sample in each of the n regions along the diagonal, as shown on the left in Figure 7.19. These samples are then randomly shuffled in each dimension, creating a pattern with good distribution. An advantage of LHS is that it minimizes clumping of the samples when they are projected onto any of the axes of the sampling dimensions. This is in contrast to stratified sampling, where $2n$ of the $n \times n$ samples in a 2D pattern may project to essentially the same point on each of the axes. Figure 7.20 shows this worst-case situation for a stratified sampling pattern.

StratifiedSampler::
RoundSize() 349

In spite of addressing the clumping problem, LHS isn't necessarily an improvement to stratified sampling; it's easy to construct cases where the sample positions are essentially colinear and large areas of $[0, 1]^2$ have no samples near them (e.g., when the permutation of the original samples is the identity, leaving them all where they started). In particular, as n increases, Latin hypercube patterns are less and less effective compared to stratified patterns. We will revisit this issue in the next section, where we will discuss sample patterns that are simultaneously stratified and distributed in a Latin hypercube pattern.

⟨*Generate stratified samples for integrators*⟩ ≡ 354
```
for (uint32_t j = 0; j < samples[i].n1D.size(); ++j)
    LatinHypercube(samples[i].oneD[j], samples[i].n1D[j], 1, rng);
for (uint32_t j = 0; j < samples[i].n2D.size(); ++j)
    LatinHypercube(samples[i].twoD[j], samples[i].n2D[j], 2, rng);
```

The general-purpose `LatinHypercube()` function generates an arbitrary number of LHS samples in an arbitrary dimension. The number of elements in the `samples` array should thus be `nSamples*nDim`.

⟨*Sampling Function Definitions*⟩ +≡
```
void LatinHypercube(float *samples, uint32_t nSamples, uint32_t nDim,
                    RNG &rng) {
    ⟨Generate LHS samples along diagonal 356⟩
    ⟨Permute LHS samples in each dimension 356⟩
}
```

⟨*Generate LHS samples along diagonal*⟩ ≡ 356
```
float delta = 1.f / nSamples;
for (uint32_t i = 0; i < nSamples; ++i)
    for (uint32_t j = 0; j < nDim; ++j)
        samples[nDim * i + j] = (i + (rng.RandomFloat())) * delta;
```

To do the permutation, this function loops over the samples, randomly permuting the sample points in one dimension at a time. Note that this is a different permutation than the earlier `Shuffle()` routine: that routine does one permutation, keeping all `nDim` sample points in each sample together, while here `nDim` separate permutations of a single dimension at a time are done (Figure 7.21).[4]

⟨*Permute LHS samples in each dimension*⟩ ≡ 356
```
for (uint32_t i = 0; i < nDim; ++i) {
    for (uint32_t j = 0; j < nSamples; ++j) {
        uint32_t other = j + (rng.RandomUInt() % (nSamples - j));
        swap(samples[nDim * j + i], samples[nDim * other + i]);
    }
}
```

LatinHypercube() 356
RNG 1003
RNG::RandomFloat() 1003
RNG::RandomUInt() 1003
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344
Shuffle() 354

4 While it's not necessary to permute the first dimension of the LHS pattern, the implementation here does so anyway, since making the elements of the first dimension be randomly ordered means that LHS patterns can be used in conjunction with sampling patterns from other sources without danger of correlation between their sample points.

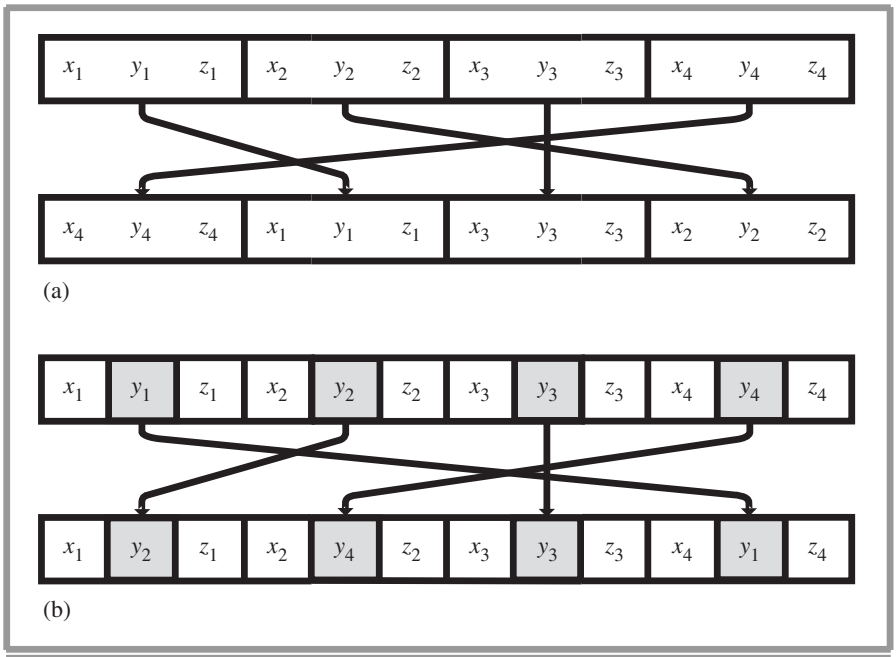


Figure 7.21: (a) The permutation done by the `Shuffle()` routine moves entire blocks of `nDims` elements around. (b) The permutation for Latin hypercube sampling permutes each dimension's samples independently. Here, the shuffling of the second dimension's samples from a four-element pattern of three dimensions is shown.

Starting with the scene in Figure 7.22, Figure 7.23 shows the improvement from good samples for the `DirectLightingIntegrator`. Image (a) was computed with 1 image sample per pixel, each with 16 shadow samples, and image (b) was computed with 16 image samples per pixel, each with 1 shadow sample. Because the `StratifiedSampler` could generate a good LHS pattern for the first case, the quality of the shadow is much better, even with the same total number of shadow samples taken.

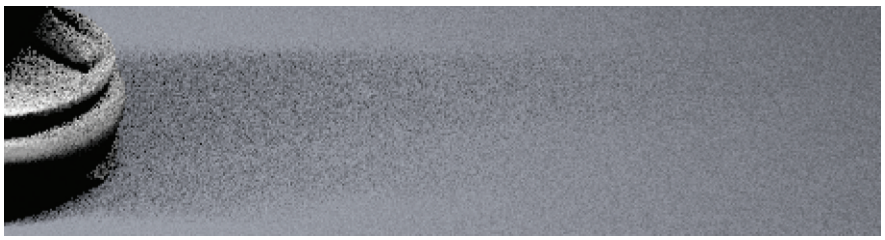
After a pixel's worth of samples have been generated, the `StratifiedSampler` advances the pixel location to the next pixel. It first tries to move one pixel over in the x direction. If doing so takes it beyond the end of the sampling range, it resets the x position to the first pixel in the next x row of pixels and advances the y position. Because the y pixel location `yPos` is advanced only when the end of a row of pixels in the x direction is reached, once the y position counter has advanced past the bottom of the sampling range, sample generation is complete.

DirectLightingIntegrator 742
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Shuffle() 354
StratifiedSampler 349
StratifiedSampler::xPos 349
StratifiedSampler::yPos 349

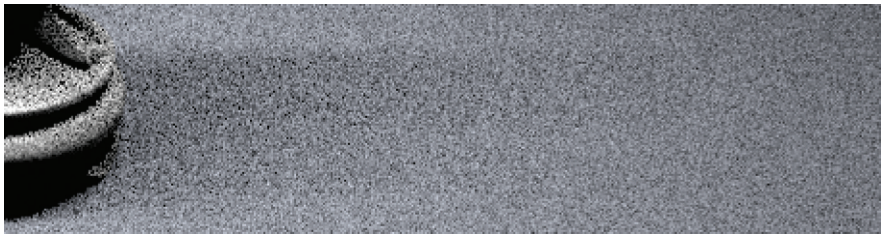
⟨*Advance to next pixel for stratified sampling*⟩ ≡ 351

```
if (++xPos == xPixelEnd) {
    xPos = xPixelStart;
    ++yPos;
}
```

Figure 7.22: Area Light Sampling Example Scene.



(a)



(b)

Figure 7.23: Sampling an Area Light with Samples from the Stratified Sampler. (a) shows the result of using 1 image sample per pixel and 16 shadow samples, and (b) shows the result of 16 image samples, each with just 1 shadow sample. The total number of shadow samples is the same in both cases, but because the version with 16 shadow samples per image sample is able to use an LHS pattern, all of the shadow samples in a pixel's area are well distributed, while in the second image the implementation here has no way to prevent them from being poorly distributed. The difference is striking.

Finally, the `MaximumSampleCount()` method can be implemented; recall that each call to `GetMoreSamples()` returns `xPixelSamples * yPixelSamples` until sampling is done.

⟨*StratifiedSampler Public Methods*⟩ +≡ 349
```
int MaximumSampleCount() { return xPixelSamples * yPixelSamples; }
```

*7.4 LOW-DISCREPANCY SAMPLING

The underlying goal of the `StratifiedSampler` is to generate a well-distributed but not uniform set of sample points, with no two sample points too close together and no excessively large regions of the sample space that have no samples. As Figure 7.14 showed, a jittered pattern does this much better than a random pattern does, although its quality can suffer when samples in adjacent strata happen to be close to the shared boundary of their two strata.

Mathematicians have developed a concept called *discrepancy* that can be used to evaluate the quality of a pattern of sample positions like these in a way that their quality can be expressed numerically. Patterns that are well distributed (in a manner to be formalized shortly) have low-discrepancy values, and thus the sample pattern generation problem can be considered to be one of finding a suitable *low-discrepancy* pattern of points.[5] A number of deterministic techniques have been developed that generate low-discrepancy point sets, even in high-dimensional spaces. This section will use a few of them as the basis for a low-discrepancy sample generator.

7.4.1 DEFINITION OF DISCREPANCY

Before defining the low-discrepancy sampling class `LDSampler`, we will first introduce a formal definition of discrepancy. The basic idea is that the "quality" of a set of points in an n-dimensional space $[0, 1]^n$ can be evaluated by looking at regions of the domain $[0, 1]^n$, counting the number of points inside each region, and comparing the volume of each region to the number of sample points inside. In general, a given fraction of the volume should have roughly the same fraction of the sample points inside of it. While it's not possible for this always to be the case, we can still try to use patterns that minimize the difference between the actual volume and the volume estimated by the points (the *discrepancy*). Figure 7.24 shows an example of the idea in two dimensions.

To compute the discrepancy of a set of points, we first pick a family of shapes B that are subsets of $[0, 1]^n$. For example, boxes with one corner at the origin are often used. This corresponds to

$$B = \{[0, v_1] \times [0, v_2] \times \cdots \times [0, v_s]\},$$

where $0 \leq v_i \leq 1$. Given a sequence of sample points $P = x_1, \ldots, x_N$, the discrepancy

LDSampler 373
StratifiedSampler 349
StratifiedSampler::
xPixelSamples 349
StratifiedSampler::
yPixelSamples 349

5 Of course, using discrepancy in this way implicitly assumes that the metric used to compute discrepancy is one that has good correlation with the quality of a pattern for image sampling, which may be a slightly different thing, particularly given the involvement of the human visual system in the process.

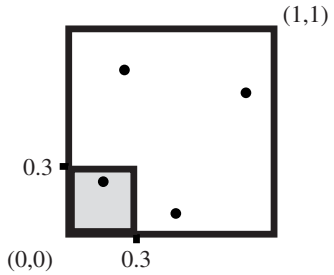


Figure 7.24: The discrepancy of a box (shaded) given a set of 2D sample points in $[0, 1]^2$. One of the four sample points is inside the box, so this set of points would estimate the box's area to be $1/4$. The true area of the box is $.3 \times .3 = .09$, so the discrepancy for this particular box is $.25 - .09 = .16$. In general, we're interested in finding the maximum discrepancy of all possible boxes (or some other shape) to compute discrepancy.

of P with respect to B is[6]

$$D_N(B, P) = \sup_{b \in B} \left| \frac{\sharp\{x_i \in b\}}{N} - \lambda(b) \right|,$$

where $\sharp\{x_i \in b\}$ is the number of points in b and $\lambda(b)$ is the volume of b.

The intuition for why this is a reasonable measure of quality is that $\sharp\{x_i \in b\}/N$ is an approximation of the volume of the box b given by the particular points P. Therefore, the discrepancy is the worst error over all possible boxes from this way of approximating volume. When the set of shapes B is the set of boxes with a corner at the origin, this is called the *star discrepancy*, $D_N^*(P)$. Another popular option for B is the set of all axis-aligned boxes, where the restriction that one corner be at the origin has been removed.

For a few particular point sets, the discrepancy can be computed analytically. For example, consider the set of points in one dimension

$$x_i = \frac{i}{N}.$$

We can see that the star discrepancy of x_i is

$$D_N^*(x_1, \ldots, x_n) = \frac{1}{N}.$$

For example, take the interval $b = [0, 1/N)$. Then $\lambda(b) = 1/N$, but $\sharp\{x_i \in b\} = 0$. This interval (and the intervals $[0, 2/N)$, etc.) is the interval where the largest differences between volume and fraction of points inside the volume are seen.

The star discrepancy of this sequence can be improved by modifying it slightly:

6 The sup operator is the continuous analog of the discrete max operator. That is, sup $f(x)$ is a constant-valued function of x that passes through the maximum value taken on by $f(x)$.

$$x_i = \frac{i - \frac{1}{2}}{N}.$$

Then

$$D_N^*(x_i) = \frac{1}{2N}.$$

The bounds for the star discrepancy of a sequence of points in one dimension have been shown to be

$$D_N^*(x_i) = \frac{1}{2N} + \max_{1 \le i \le N} \left| x_i - \frac{2i - 1}{2N} \right|.$$

Thus, the earlier modified sequence has the lowest possible discrepancy for a sequence in 1D. In general, it is much easier to analyze and compute bounds for the discrepancy of sequences in 1D than for those in higher dimensions. For less simply constructed point sequences, and for sequences in higher dimensions and for more irregular shapes than boxes, the discrepancy often must be estimated numerically by constructing a large number of shapes B, computing their discrepancy, and reporting the maximum.

The astute reader will notice that according to the low-discrepancy measure, this uniform sequence in 1D is optimal, but earlier in this chapter we claimed that irregular jittered patterns were perceptually superior to uniform patterns for image sampling in 2D since they replaced aliasing error with noise. In that framework, uniform samples are clearly not optimal. Fortunately, low-discrepancy patterns in higher dimensions are much less uniform than they are in one dimension and thus usually work reasonably well as sample patterns in practice. Nevertheless, their underlying uniformity is probably the reason why low-discrepancy patterns can be more prone to aliasing than patterns with true pseudo-random variation.

7.4.2 HAMMERSLEY AND HALTON SEQUENCES

We will now introduce a number of techniques that have been developed specifically to generate sequences of points that have low discrepancy. Remarkably, few lines of code are necessary to compute many low-discrepancy sampling patterns.

The first set of techniques that we will describe use a construction called the *radical inverse*. It is based on the fact that a positive integer value n can be expressed in a base b with a sequence of digits $d_m \ldots d_2 d_1$ uniquely determined by

$$n = \sum_{i=1}^{\infty} d_i b^{i-1}.$$

The radical inverse function Φ_b in base b converts a nonnegative integer n to a floating-point value in $[0, 1)$ by reflecting these digits about the decimal point:

$$\Phi_b(n) = 0.d_1 d_2 \ldots d_m.$$

Thus, the contribution of the digit d_i to the radical inverse is d_i / b^i.

RadicalInverse() 362

The function `RadicalInverse()` computes the radical inverse for a given number `n` in the base `base`. It first computes the value of d_1 by taking the remainder of the number n when divided by the base and adds $d_1 b^{-1}$ to the radical inverse value. It then divides n by the

base, effectively chopping off the last digit, so that the next time through the loop it can compute d_2 by finding the remainder in base `base` and adding $d_2 b^{-2}$ to the sum, and so on. This process continues until n is zero, at which point it has found the last nonzero d_i value.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
inline double RadicalInverse(int n, int base) {
    double val = 0;
    double invBase = 1. / base, invBi = invBase;
    while (n > 0) {
        ⟨Compute next digit of radical inverse 362⟩
    }
    return val;
}
```

⟨*Compute next digit of radical inverse*⟩ ≡ 362
```
int d_i = (n % base);
val += d_i * invBi;
n *= invBase;
invBi *= invBase;
```

The `RadicalInverse()` function can be implemented much more efficiently for base 2, with bit operations replacing the floating-point math above. Furthermore, for the common case where the value `n` is incremented by one for each successive call to the radical inverse function, it's possible to incrementally generate each successive point more efficiently than computing its value from scratch. Because the `RadicalInverse()` function isn't currently used in performance-critical code in `pbrt`, we leave these improvements for later work.

One of the simplest low-discrepancy sequences is the van der Corput sequence, which is a one-dimensional sequence given by the radical inverse function in base 2:

$$x_i = \Phi_2(i).$$

Table 7.2 shows the first few values of the van der Corput sequence. Notice how it recursively splits the intervals of the 1D line in half, generating a sample point at the center of each interval. The discrepancy of this sequence is

$$D_N^*(P) = O\left(\frac{\log N}{N}\right),$$

which matches the best discrepancy that has been attained for infinite sequences in d dimensions,

$$D_N^*(P) = O\left(\frac{(\log N)^d}{N}\right).$$

Two well-known low-discrepancy sequences that are defined in an arbitrary number of dimensions are the *Halton* and *Hammersley* sequences. Both use the radical inverse function as well. To generate an n-dimensional Halton sequence, we use the radical

Table 7.2: The radical inverse $\Phi_2(n)$ of the first few positive integers, computed in base 2. Notice how successive values of $\Phi_2(n)$ are not close to any of the previous values of $\Phi_2(n)$. As more and more values of the sequence are generated, samples are necessarily closer to previous samples, although with a minimum distance that is guaranteed to be reasonably good.

n	**Base 2**	$\Phi_2(n)$
1	1	$.1 = 1/2$
2	10	$.01 = 1/4$
3	11	$.11 = 3/4$
4	100	$.001 = 1/8$
5	101	$.101 = 5/8$
⋮	⋮	⋮

inverse base b, with a different base for each dimension of the pattern. The bases used must all be relatively prime to each other, so a natural choice is to use the first n prime numbers $(p_1, \ldots, p_n)$:

$$x_i = (\Phi_2(i), \Phi_3(i), \Phi_5(i), \ldots, \Phi_{p_n}(i)).$$

One of the most useful characteristics of the Halton sequence is that it can be used even if the total number of samples needed isn't known in advance; all prefixes of the sequence are well distributed, so as additional samples are added to the sequence low discrepancy will be maintained. This property will be used by the `PhotonIntegrator`, for example, which doesn't know the total number of photons that will be necessary to emit from lights in the scene ahead of time and thus uses a Halton sequence to get a well-distributed set of photons.

The discrepancy of a d-dimensional Halton sequence is

$$D_N^*(x_i) = O\left(\frac{(\log N)^d}{N}\right),$$

which is asymptotically optimal.

If the number of samples N is fixed, the Hammersley point set can be used, giving slightly lower discrepancy. Hammersley point sets are defined by

$$x_i = \left(\frac{i}{N}, \Phi_{b_1}(i), \Phi_{b_2}(i), \ldots, \Phi_{b_n}(i)\right),$$

where N is the total number of samples to be taken and as before all of the bases b_i are relatively prime. Figure 7.25(a) shows a plot of the first 100 points of the 2D Halton sequence. Figure 7.25(b) shows the Hammersley sequence.

The Halton Sampler

LDSampler 373

PhotonIntegrator 802

The `HaltonSampler` generates image, lens, and time samples using the Halton sequence. For each camera sample, it generates samples for the integrators using a Latin hypercube pattern. This sampler isn't as effective as the `LDSampler` that will be introduced shortly,

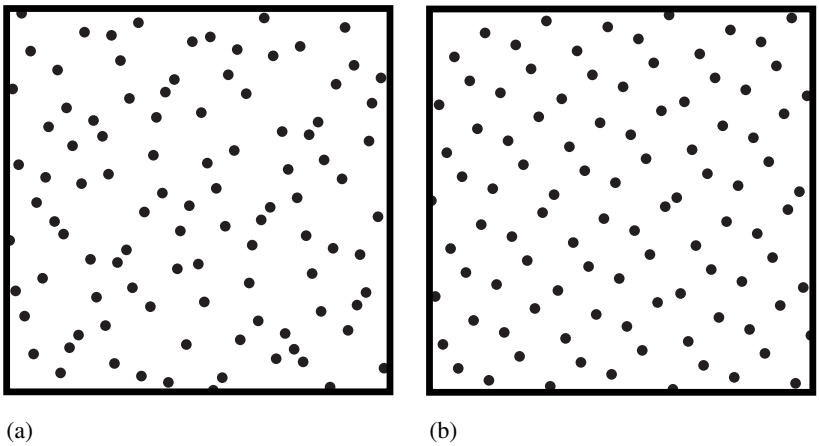


Figure 7.25: The First 100 Points of Two Low-Discrepancy Sequences in 2D. (a) Halton, (b) Hammersley.

but its implementation is straightforward and it generates reasonably good sampling patterns. However, it can be prone to aliasing in images. Figure 7.26 compares the results of sampling a checkerboard texture using a Halton-based sampler to using the stratified sampler from the previous section. Note the unpleasant pattern along edges in the foreground and toward the horizon.

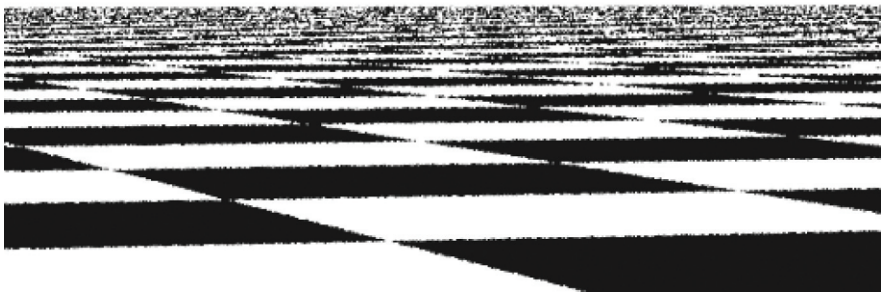
The `HaltonSampler` constructor takes the usual parameters that give the pixel extent to sample, the number of samples to take per pixel, and the shutter open range. It then computes the total number of samples it will generate, `wantedSamples`, and initializes `currentSample` to zero.

⟨*HaltonSampler Private Data*⟩ ≡
```
int wantedSamples, currentSample;
```

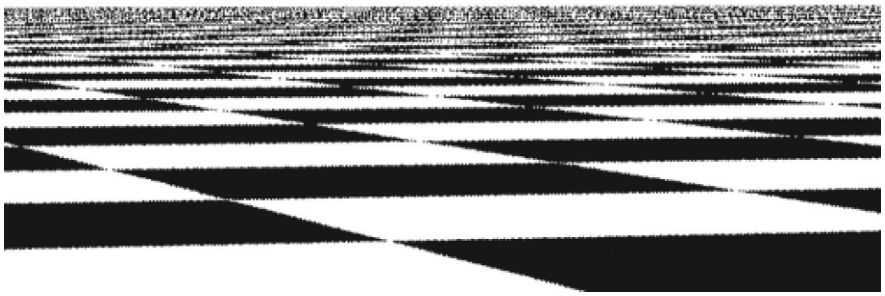
The only subtlety in the implementation is how it handles nonsquare images. The implementation here computes a total number of samples to take as if it were generating samples for a square image with resolution given by the greater of the x and y resolutions. It then ignores samples that are outside the image extent.

⟨*HaltonSampler Method Definitions*⟩ ≡
```
int HaltonSampler::GetMoreSamples(Sample *samples, RNG &rng) {
retry:
    if (currentSample >= wantedSamples) return 0;
    ⟨Generate sample with Halton sequence and reject if outside image extent 365⟩
    ⟨Generate lens, time, and integrator samples for HaltonSampler 366⟩
    return 1;
}
```

HaltonSampler::
  currentSample 364
HaltonSampler::
  wantedSamples 364
RNG 1003
Sample 343



(a)



(b)

Figure 7.26: Comparison of the Stratified Sampler to a Low-Discrepancy Sampler Based on Halton Points on the Image Plane. (a) The jittered stratified sampler with a single sample per pixel and (b) the `HaltonSampler` sampler with a single sample per pixel. Note that although the Halton pattern is able to reproduce the checker pattern farther toward the horizon than the stratified pattern, there is a regular structure to the error in the low-discrepancy pattern that is visually distracting; it doesn't turn aliasing into less objectionable noise as well as the jittered approach. (With one sample per pixel, the `HaltonSampler` performs similarly to the jittered sampler.)

The first thing `GetMoreSamples()` does is generate the `imageX` and `imageY` samples. If these are outside of the sampling extent, they are rejected and we try again with the next sample number.

⟨*Generate sample with Halton sequence and reject if outside image extent*⟩ ≡ 364

```
float u = (float)RadicalInverse(currentSample, 3);
float v = (float)RadicalInverse(currentSample, 2);
float lerpDelta = float(max(xPixelEnd - xPixelStart,
                            yPixelEnd - yPixelStart));
samples->imageX = Lerp(u, xPixelStart, xPixelStart + lerpDelta);
samples->imageY = Lerp(v, yPixelStart, yPixelStart + lerpDelta);
++currentSample;
if (samples->imageX >= xPixelEnd || samples->imageY >= yPixelEnd)
    goto retry;
```

CameraSample::imageX 342
CameraSample::imageY 342
HaltonSampler 364
HaltonSampler::
currentSample 364
RadicalInverse() 362
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
Sampler::yPixelStart 340

If the image sample is within the image extent, then samples for the remaining dimensions are generated.

⟨*Generate lens, time, and integrator samples for* HaltonSampler⟩ ≡ 364
```
samples->lensU = (float)RadicalInverse(currentSample, 5);
samples->lensV = (float)RadicalInverse(currentSample, 7);
samples->time = Lerp((float)RadicalInverse(currentSample, 11),
                     shutterOpen, shutterClose);
for (uint32_t i = 0; i < samples->n1D.size(); ++i)
    LatinHypercube(samples->oneD[i], samples->n1D[i], 1, rng);
for (uint32_t i = 0; i < samples->n2D.size(); ++i)
    LatinHypercube(samples->twoD[i], samples->n2D[i], 2, rng);
```

Randomized Halton Sequences

The Hammersley and Halton sequences have a few shortcomings. First, while their first few dimensions are irregularly distributed in desirable ways, as the base b increases, the values can exhibit surprisingly regular patterns. Second, these sequences aren't well suited to parallel rendering. If multiple threads of execution are all independently generating samples from a Halton or Hammersley sequence, they will all generate the exact same set of samples, likely computing the exact same results, which is probably not desirable. On the other hand, the synchronization overhead of having all threads draw samples from a single sequence would likely be unacceptable.

An alternative approach is to use *randomized Halton sequences*, where each instance of the sequence is independent from the others. A variety of approaches have been suggested for doing this, but a simple and effective one is to compute a permutation table for the digits and use it when computing the radical inverse:

$$\Psi_b(n) = 0.p(d_1)p(d_2)\ldots p(d_m),$$

where p is a random permutation of the digits $(0, 1, \ldots, b-1)$. Note that the same permutation is used for each digit, and the same permutation is used for generating each of the n sample points. `GeneratePermutation()` is a simple utility function to generate such a permutation for base b.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
inline void GeneratePermutation(uint32_t *buf, uint32_t b, RNG &rng) {
    for (uint32_t i = 0; i < b; ++i)
        buf[i] = i;
    Shuffle(buf, b, 1, rng);
}
```

`PermutedRadicalInverse()` is a modified version of `RadicalInverse()` that applies the given permutation to each digit.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
inline double PermutedRadicalInverse(uint32_t n, uint32_t base,
                                     const uint32_t *p) {
    double val = 0;
    double invBase = 1. / base, invBi = invBase;
```

CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
HaltonSampler::currentSample 364
LatinHypercube() 356
RadicalInverse() 362
RNG 1003
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344
Sampler::shutterClose 340
Sampler::shutterOpen 340
Shuffle() 354

```
    while (n > 0) {
        uint32_t d_i = p[n % base];
        val += d_i * invBi;
        n *= invBase;
        invBi *= invBase;
    }
    return val;
}
```

PermutedHalton is a utility class that generates a sequence of sample points in a given dimension using randomly permuted Halton values.

⟨*Monte Carlo Utility Declarations*⟩ +≡
```
class PermutedHalton {
public:
    ⟨PermutedHalton Public Methods 368⟩
private:
    ⟨PermutedHalton Private Data 368⟩
};
```

Its constructor takes the dimensionality of sample points to generate, d. It uses a precomputed array of prime numbers, primes, to determine the base b_i for each dimension and computes a permutation table for each one.

⟨*Monte Carlo Function Definitions*⟩ ≡
```
PermutedHalton::PermutedHalton(uint32_t d, RNG &rng) {
    dims = d;
    ⟨Determine bases b_i and their sum 367⟩
    ⟨Compute permutation tables for each base 368⟩
}
```

Recall that each dimension requires a permutation table of size b_i; thus, the total number of entries that must be stored for all of the permutation tables is given by the sum of the bases b_i—this value is computed here as the base for each dimension is set.

⟨*Determine bases b_i and their sum*⟩ ≡ 367
```
b = new uint32_t[dims];
uint32_t sumBases = 0;
for (uint32_t i = 0; i < dims; ++i) {
    b[i] = primes[i];
    sumBases += b[i];
}
```

PermutedHalton 367
PermutedHalton::b 368
PermutedHalton::dims 368
primes 368
RNG 1003

Memory for all of the permutation tables is allocated in a single allocation that's large enough to hold all of the entries. The first b[0] entries hold the permutation table for the first dimension, the next b[1] entries hold the table for the second dimension, and so forth.

⟨*Compute permutation tables for each base*⟩ ≡ 367
```
permute = new uint32_t[sumBases];
uint32_t *p = permute;
for (uint32_t i = 0; i < dims; ++i) {
    GeneratePermutation(p, b[i], rng);
    p += b[i];
}
```

⟨*PermutedHalton Private Data*⟩ ≡ 367
```
uint32_t dims;
uint32_t *b, *permute;
```

⟨*Sampling Local Definitions*⟩ ≡
```
static const int primes[] = {
    ⟨First 1000 prime numbers⟩
};
```

Given all this, the `PermutedHalton`'s `Sample()` method generates the `n`th sample of the sequence, storing the `dims` values in the `out` array.

⟨*PermutedHalton Public Methods*⟩ ≡ 367
```
void Sample(uint32_t n, float *out) const {
    uint32_t *p = permute;
    for (uint32_t i = 0; i < dims; ++i) {
        out[i] = PermutedRadicalInverse(n, b[i], p);
        p += b[i];
    }
}
```

7.4.3 (0,2)-SEQUENCES

To generate high-quality samples for the integrators, we can take advantage of a remarkable property of certain low-discrepancy patterns that allows us to satisfy both parts of our original goal for samplers (only one of which was satisfied with the `StratifiedSampler`): they make it possible to generate a set of integrator samples for a pixel's worth of image samples such that the samples for each pixel sample are well distributed with respect to each other, and such that the aggregate collection of integrator samples for all of the pixel samples in the pixel are collectively well distributed.

A useful low-discrepancy sequence in 2D can be constructed using the van der Corput sequence in one dimension and a sequence based on a radical inverse function due to Sobol′ in the other direction. The resulting sequence is a special type of low-discrepancy sequence known as an (0, 2)-sequence. (0, 2)-sequences are stratified in a very general way. For example, the first 16 samples in an (0, 2)-sequence satisfy the stratification constraint from Section 7.3, meaning there is just one sample in each of the boxes of extent $(\frac{1}{4}, \frac{1}{4})$. However, they also satisfy the Latin hypercube constraint, as only one of them is in each of the boxes of extent $(\frac{1}{16}, 1)$ and $(1, \frac{1}{16})$. Furthermore, there is only one sample in each of the boxes of extent $(\frac{1}{2}, \frac{1}{8})$ and $(\frac{1}{8}, \frac{1}{2})$. Figure 7.27 shows all of the possibilities for dividing the domain into regions where the first 16 samples of

GeneratePermutation() 366
PermutedHalton::b 368
PermutedHalton::dims 368
PermutedHalton::permute 368
PermutedRadicalInverse() 366
StratifiedSampler 349

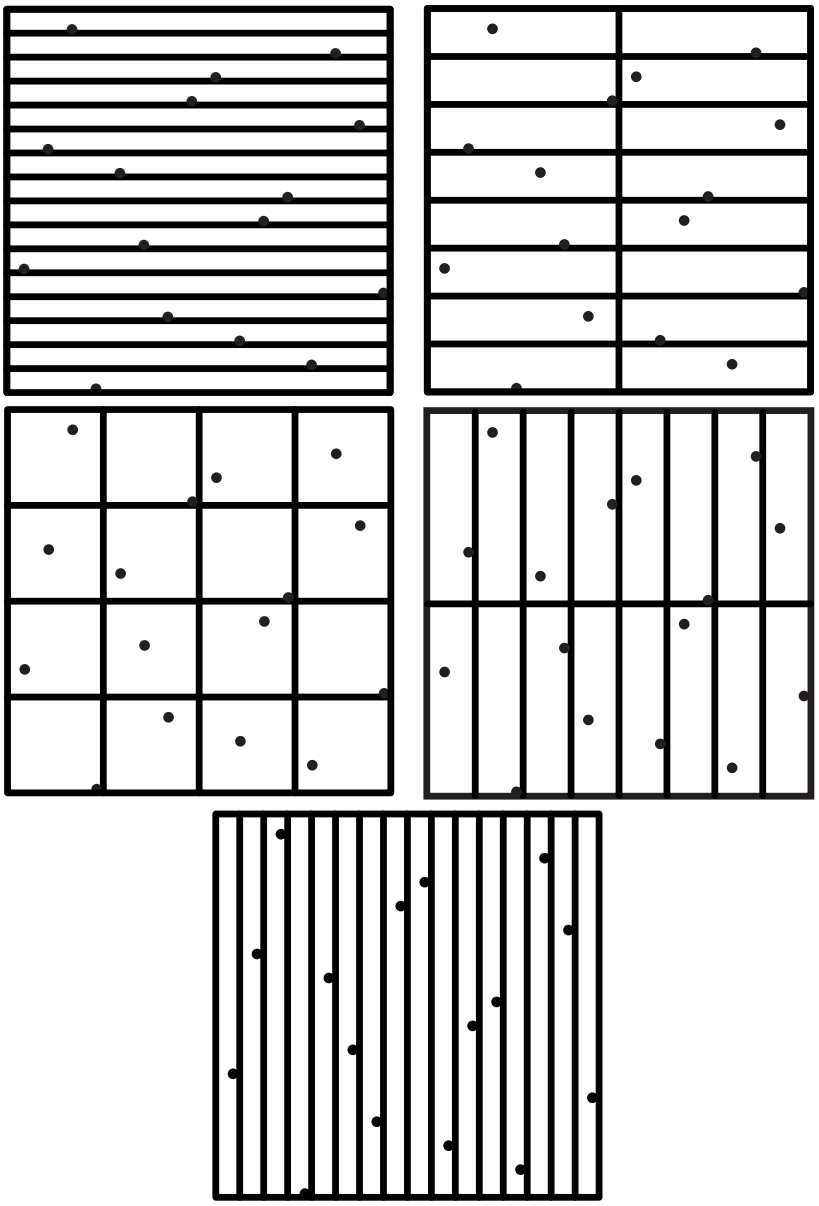


Figure 7.27: A sampling pattern that has a single sample in all of the base 2 elementary intervals. It satisfies both the 4×4 stratification and Latin hypercube constraints as well as the other stratification constraints shown.

an (0, 2)-sequence satisfy the stratification properties. Each succeeding sequence of 16 samples from this pattern also satisfies the distribution properties.

In general, any sequence of length $2^{l_1+l_2}$ (where l_i is a nonnegative integer) from an (0, 2)-sequence satisfies this general stratification constraint. The set of *elementary intervals* in two dimensions, base 2, is defined as

$$E = \left\{ \left[\frac{a_1}{2^{l_1}}, \frac{a_1+1}{2^{l_1}} \right) \times \left[\frac{a_2}{2^{l_2}}, \frac{a_2+1}{2^{l_2}} \right) \right\},$$

where the integer $a_i = 0, \ldots, 2^{l_i} - 1$. One sample from each of the first $2^{l_1+l_2}$ values in the sequence will be in each of the elementary intervals. Furthermore, the same property is true for each subsequent set of $2^{l_1+l_2}$ values.

To understand now how (0, 2)-sequences can be applied to generating 2D samples for the integrators, consider a pixel with 2×2 image samples, each with 4×4 integrator samples. The first $2 \times 2 \times 4 \times 4 = 2^6$ values of an (0, 2)-sequence are well distributed with respect to each other according to the corresponding set of elementary intervals. Furthermore, the first $4 \times 4 = 2^4$ samples are themselves well distributed according to their corresponding elementary intervals, as are the next 2^4 of them, and the subsequent ones, and so on. Therefore, we can use the first 16 (0, 2)-sequence samples for the integrator samples for the first image sample for a pixel, then the next 16 for the next image sample, and so forth. The result is an extremely well-distributed set of sample points.

There are a handful of details that must be addressed before (0, 2)-sequences can be used in practice. The first is that we need to generate multiple sets of 2D sample values for each image sample, and we would like to generate different sample values in the areas around different pixels. One approach to this problem would be to use carefully chosen nonoverlapping subsequences of the (0, 2)-sequence for each pixel. Another approach, which is used in `pbrt`, is to randomly *scramble* the (0, 2)-sequence, giving a new (0, 2)-sequence built by randomly permuting the base b digits of the values in the original sequence.

The scrambling approach we will use is due to Kollig and Keller (2002). It repeatedly partitions and shuffles the unit square $[0, 1]^2$. In each of the two dimensions, it first divides the square in half, then swaps the two halves with 50% probability. Then, it splits each of the intervals [0, 0.5) and [0.5, 1) in half and randomly exchanges each of those two halves. This process continues recursively until floating-point precision intervenes and continuing the process would no longer change the computed values. This process was carefully designed so that it preserves the low-discrepancy properties of the set of points; otherwise, the advantages of the (0, 2)-sequence would be lost from the scrambling. Figure 7.28 shows an unscrambled (0, 2)-sequence and two randomly scrambled variations of it.

Two things make this process efficient: First, because we are scrambling two sequences that are computed in base 2, the digits a_i of the sequences are all 0 or 1, and scrambling a particular digit is equivalent to exclusive-ORing it with 0 or 1. Second, the simplification is made such that at each level l of the recursive scrambling, the same decision will be

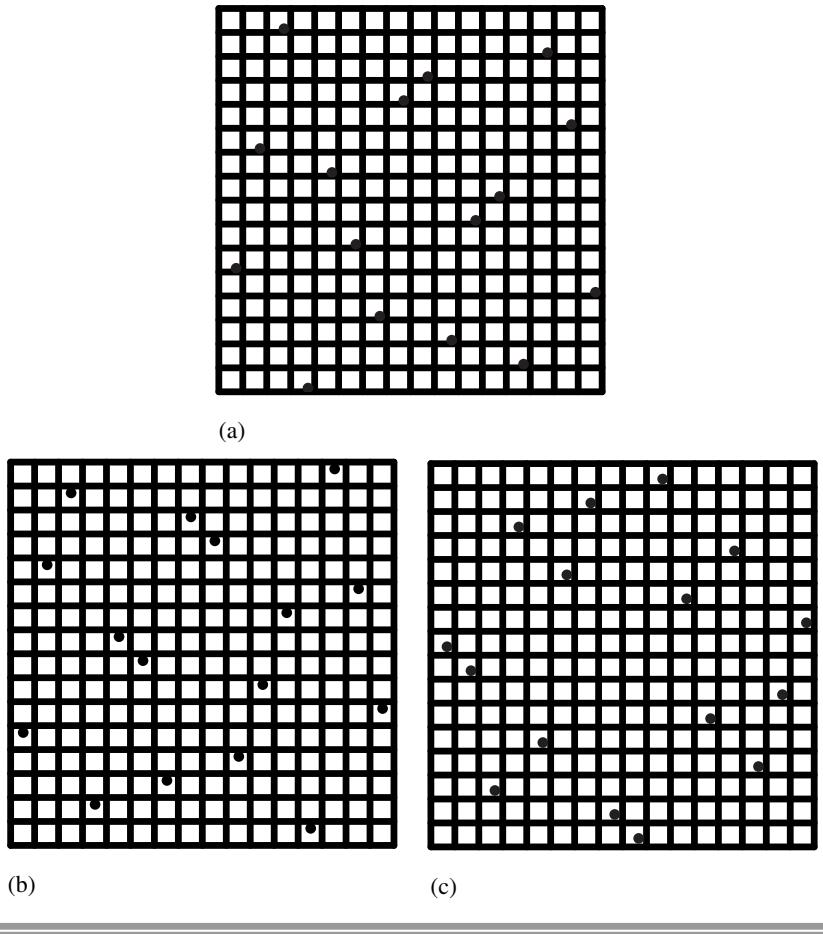


Figure 7.28: (a) A low-discrepancy (0, 2)-sequence-based sampling pattern and (b, c) two randomly scrambled instances of it. Random scrambling of low-discrepancy patterns is an effective way to eliminate the artifacts that would be present in images if we used the same sampling pattern in every pixel, while still preserving the low-discrepancy properties of the point set being used.

made as to whether to swap each of the 2^{l-1} pairs of subintervals or not. The result of these two design choices is that the scrambling can be encoded as a set of bits stored in a `uint32_t` and can be applied to the original digits via exclusive-OR operations.

The `Sample02()` function generates a sample from a scrambled (0, 2)-sequence using the given scramble values. The sequence used here is constructed from two 1D low-discrepancy sequences that together form a (0, 2)-sequence. An arbitrary pair of 1D low-discrepancy sequences will not necessarily form a (0, 2)-sequence, however.

⟨*Sampling Inline Functions*⟩ ≡
```
inline void Sample02(uint32_t n, const uint32_t scramble[2],
                     float sample[2]) {
    sample[0] = VanDerCorput(n, scramble[0]);
    sample[1] = Sobol2(n, scramble[1]);
}
```

The implementations of the van der Corput and Sobol′ low-discrepancy sequences here are also specialized for the base 2 case. Each of them takes a `uint32_t` value `scramble` that encodes a random permutation to apply and computes the `n`th value from each of the sequences as it simultaneously applies the permutation. It is worthwhile to convince yourself that the `VanDerCorput()` function computes the same values as `RadicalInverse()` in the preceding when they are called with a zero `scramble` and a `base` of two, respectively.

⟨*Sampling Inline Functions*⟩ +≡
```
inline float VanDerCorput(uint32_t n, uint32_t scramble) {
    ⟨Reverse bits of n 372⟩
    n ^= scramble;
    return ((n>>8) & 0xffffff) / float(1 << 24);
}
```

The bits of a 32-bit quantity can be efficiently reversed with a series of logical bit operations. The first line of the fragment below swaps the lower 16 bits with the upper 16 bits of the value. The next line simultaneously swaps the first 8 bits of the result with the second 8 bits and the third 8 bits with the fourth. This process continues until the last line, which swaps adjacent bits. To understand this code, it's helpful to write out the binary values of the various hexidecimal constants. For example, `0xff00ff00` is 11111111000000001111111100000000 in binary; it's easy to see that ORing with this value masks off the first and third 8-bit quantities.

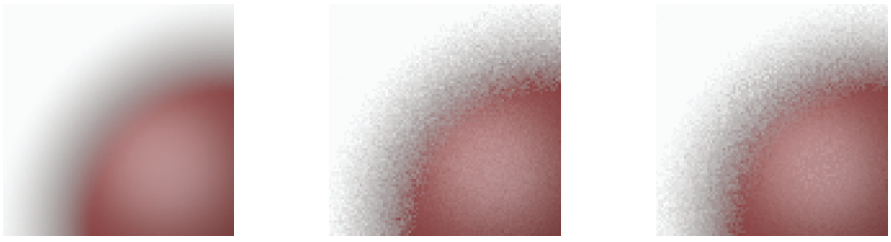
⟨*Reverse bits of* `n`⟩ ≡ 372
```
n = (n << 16) | (n >> 16);
n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
```

⟨*Sampling Inline Functions*⟩ +≡
```
inline float Sobol2(uint32_t n, uint32_t scramble) {
    for (uint32_t v = 1 << 31; n != 0; n >>= 1, v ^= v >> 1)
        if (n & 0x1) scramble ^= v;
    return ((scramble>>8) & 0xffffff) / float(1 << 24);
}
```

7.4.4 THE LOW-DISCREPANCY SAMPLER

The `LDSampler` implements a low-discrepancy pattern based on (0, 2)-sequences, since they give the foundation for an effective approach for generating multiple samples per image sample for integration. Because a (0, 2)-sequence is being used, the number of samples for each pixel must be a power of two. It generates samples for positions on the

RadicalInverse() 362
Sobol2() 372
VanDerCorput() 372



(a) (b) (c)

Figure 7.29: Comparisons of the Stratified and Low-Discrepancy Samplers for Rendering Depth of Field. (a) A reference image of the blurred edge of an out-of-focus sphere, (b) an image rendered using the `StratifiedSampler`, and (c) an image using the `LDSampler`. The `LDSampler`'s results are better than the stratified image, although the difference is far less than the difference between stratified and random sampling.

lens and for the two-dimensional integrator samples with scrambled (0, 2)-sequences, and one-dimensional samples for time and integrators are generated with scrambled van der Corput sequences. Figure 7.29 compares the result of using an (0, 2)-sequence for sampling the lens for the depth of field to using a stratified pattern.

⟨*LDSampler Declarations*⟩ ≡
```
class LDSampler : public Sampler {
public:
    ⟨LDSampler Public Methods 374⟩
private:
    ⟨LDSampler Private Data 374⟩
};
```

The constructor rounds the number of samples per pixel up to a power of two if necessary, since subsets of (0, 2)-sequences that are not a power of two in size are much less well distributed over [0, 1] than those that are. Like the `StratifiedSampler`, the `LDSampler` uses a preallocated temporary storage buffer, `sampleBuf`, during sample generation. The amount of storage that must be allocated is computed in a separate utility function; thus, other samplers (such as the `AdaptiveSampler` below) can also use its functionality.

AdaptiveSampler 386
IsPowerOf2() 1001
LDSampler 373
LDSampler::nPixelSamples 374
LDSampler::sampleBuf 374
LDSampler::xPos 374
LDSampler::yPos 374
RoundUpPow2() 1002
Sampler 340
Sampler::xPixelStart 340
Sampler::yPixelStart 340
StratifiedSampler 349
Warning() 1005

⟨*LDSampler Method Definitions*⟩ ≡
```
LDSampler::LDSampler(int xstart, int xend, int ystart, int yend, int ps,
                     float sopen, float sclose)
    : Sampler(xstart, xend, ystart, yend, RoundUpPow2(ps), sopen, sclose) {
    xPos = xPixelStart;
    yPos = yPixelStart;
    if (!IsPowerOf2(ps)) {
        Warning("Pixel samples being rounded up to power of 2");
        nPixelSamples = RoundUpPow2(ps);
    } else
        nPixelSamples = ps;
    sampleBuf = NULL;
}
```

⟨*LDSampler Private Data*⟩ ≡ 373
```
int xPos, yPos, nPixelSamples;
float *sampleBuf;
```

The implementation of the `GetSubSampler()` method is quite similar to that of the `StratifiedSampler::GetSubSampler()` method, so it isn't included here.

⟨*LDSampler Public Methods*⟩ ≡ 373
```
Sampler *GetSubSampler(int num, int count);
```

As mentioned earlier, the low-discrepancy sequences used here need power-of-two sample sizes.

⟨*LDSampler Public Methods*⟩ +≡ 373
```
int RoundSize(int size) const { return RoundUpPow2(size); }
```

Usage of `xPos` and `yPos` is just like the corresponding member variables in the `StratifiedSampler`; pixels are scanned left-to-right and top-to-bottom in the sampling region, and an entire pixel's worth of samples are generated for each call to `GetMoreSamples()`. `LDPixelSample()` fills in the `samples` array with samples for the current pixel.

⟨*LDSampler Method Definitions*⟩ +≡
```
int LDSampler::GetMoreSamples(Sample *samples, RNG &rng) {
    if (yPos == yPixelEnd) return 0;
    if (sampleBuf == NULL)
        sampleBuf = new float[LDPixelSampleFloatsNeeded(samples,
                                                        nPixelSamples)];
    LDPixelSample(xPos, yPos, shutterOpen, shutterClose,
                  nPixelSamples, samples, sampleBuf, rng);
    if (++xPos == xPixelEnd) {
        xPos = xPixelStart;
        ++yPos;
    }
    return nPixelSamples;
}
```

`LDPixelSample()` uses a temporary buffer `buf` in its work. The memory that `buf` points to must be large enough to store all of the image, lens, time, and integrator samples for a pixel. This value is computed by `LDPixelSampleFloatsNeeded()`, given a `Sample` and the pixel sampling rate. Its implementation computes the total number of `floats` needed for a single pixel sample and then returns this value multiplied by the total number of samples per pixel.

⟨*Sampling Function Definitions*⟩ +≡
```
int LDPixelSampleFloatsNeeded(const Sample *sample, int nPixelSamples) {
    int n = 5; // 2 lens + 2 pixel + time
    for (uint32_t i = 0; i < sample->n1D.size(); ++i)
        n += sample->n1D[i];
    for (uint32_t i = 0; i < sample->n2D.size(); ++i)
        n += 2 * sample->n2D[i];
```

LDPixelSample() 375
LDPixelSampleFloatsNeeded() 374
LDSampler 373
LDSampler::nPixelSamples 374
LDSampler::sampleBuf 374
LDSampler::xPos 374
LDSampler::yPos 374
RNG 1003
RoundUpPow2() 1002
Sample 343
Sample::n1D 344
Sample::n2D 344
Sampler 340
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
StratifiedSampler 349
StratifiedSampler::
GetSubSampler() 350

```
    return nPixelSamples * n;
}
```

Given the large linear temporary buffer `buf`, `LDPixelSample()` first computes pointers into the buffer for the various types of samples. After generating the samples into this buffer, it finishes by copying them into the `samples` array before returning. These extra steps are needed because the sample generation utility routines are written to generate samples into flat arrays of `float`s, rather than directly into `Sample` structures.

```
⟨Sampling Function Definitions⟩ +≡
void LDPixelSample(int xPos, int yPos, float shutterOpen,
        float shutterClose, int nPixelSamples, Sample *samples,
        float *buf, RNG &rng) {
    ⟨Prepare temporary array pointers for low-discrepancy camera samples 375⟩
    ⟨Prepare temporary array pointers for low-discrepancy integrator samples 375⟩
    ⟨Generate low-discrepancy pixel samples 376⟩
    ⟨Initialize samples with computed sample values 376⟩
}
```

For convenience, separate pointers are initialized from the temporary buffer to store the initial image, lens, and time samples.

```
⟨Prepare temporary array pointers for low-discrepancy camera samples⟩ ≡ 375
float *imageSamples = buf; buf += 2 * nPixelSamples;
float *lensSamples = buf;  buf += 2 * nPixelSamples;
float *timeSamples = buf;  buf += nPixelSamples;
```

Next, the `oneDSamples` and `twoDSamples` arrays are set up so that their entries point to memory from the temporary buffer to store their respective sample values.

```
⟨Prepare temporary array pointers for low-discrepancy integrator samples⟩ ≡ 375
uint32_t count1D = samples[0].n1D.size();
uint32_t count2D = samples[0].n2D.size();
const uint32_t *n1D = count1D > 0 ? &samples[0].n1D[0] : NULL;
const uint32_t *n2D = count2D > 0 ? &samples[0].n2D[0] : NULL;
float **oneDSamples = ALLOCA(float *, count1D);
float **twoDSamples = ALLOCA(float *, count2D);
for (uint32_t i = 0; i < count1D; ++i) {
    oneDSamples[i] = buf;
    buf += n1D[i] * nPixelSamples;
}
for (uint32_t i = 0; i < count2D; ++i) {
    twoDSamples[i] = buf;
    buf += 2 * n2D[i] * nPixelSamples;
}
```

ALLOCA() 1009
RNG 1003
Sample 343
Sample::n1D 344
Sample::n2D 344

Sample values can now be computed. The `LDShuffleScrambled*D()` utility routines, to be defined shortly, compute one- and two-dimensional scrambled low-discrepancy sampling patterns.

⟨*Generate low-discrepancy pixel samples*⟩ ≡ 375
```
LDShuffleScrambled2D(1, nPixelSamples, imageSamples, rng);
LDShuffleScrambled2D(1, nPixelSamples, lensSamples, rng);
LDShuffleScrambled1D(1, nPixelSamples, timeSamples, rng);
for (uint32_t i = 0; i < count1D; ++i)
    LDShuffleScrambled1D(n1D[i], nPixelSamples, oneDSamples[i], rng);
for (uint32_t i = 0; i < count2D; ++i)
    LDShuffleScrambled2D(n2D[i], nPixelSamples, twoDSamples[i], rng);
```

Finally, once the sample values have been computed, the `samples` array can be filled in. For image and time samples, the values have to be be mapped to the appropriate range; other sample types can be copied directly.

⟨*Initialize* `samples` *with computed sample values*⟩ ≡ 375
```
for (int i = 0; i < nPixelSamples; ++i) {
    samples[i].imageX = xPos + imageSamples[2*i];
    samples[i].imageY = yPos + imageSamples[2*i+1];
    samples[i].time = Lerp(timeSamples[i], shutterOpen, shutterClose);
    samples[i].lensU = lensSamples[2*i];
    samples[i].lensV = lensSamples[2*i+1];
    ⟨Copy integrator samples into samples[i] 376⟩
}
```

The array indexing to copy samples for the integrators is a little more tricky. The `j`th element of the `oneDSamples` array, for example, points to the start of an array of `nPixelSamples * n1D[j]` floating-point values, where the first `n1D[j]` of them are for the first pixel sample, the next `n1D[j]` are for the next pixel sample, and so forth. The code below handles copying the appropriate values into the individual samples.

⟨*Copy integrator samples into* `samples[i]`⟩ ≡ 376
```
for (uint32_t j = 0; j < count1D; ++j) {
    int startSamp = n1D[j] * i;
    for (uint32_t k = 0; k < n1D[j]; ++k)
        samples[i].oneD[j][k] = oneDSamples[j][startSamp+k];
}
for (uint32_t j = 0; j < count2D; ++j) {
    int startSamp = 2 * n2D[j] * i;
    for (uint32_t k = 0; k < 2*n2D[j]; ++k)
        samples[i].twoD[j][k] = twoDSamples[j][startSamp+k];
}
```

There is a subtle implementation detail that must be accounted for in using scrambled (0, 2)-sequences in practice.[7] Often, integrators will use samples from more than one of the sampling patterns that the sampler creates in the process of computing the values of particular integrals. For example, they might use a sample from a one-dimensional pattern to select one of the N light sources in the scene to sample illumination from, and

CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
LDSampler::nPixelSamples 374
LDSampler::xPos 374
LDSampler::yPos 374
LDShuffleScrambled1D() 377
LDShuffleScrambled2D() 377
Lerp() 1000
Sampler::shutterClose 340
Sampler::shutterOpen 340

7 Indeed, the importance of this issue wasn't fully appreciated by the authors until after going through the process of debugging some unexpected noise patterns in rendered images when this sampler was being used.

then might use a sample from a two-dimensional pattern to select a sample point on that light source, if it is an area light.

Even if these two patterns are computed with random scrambling with different random scramble values for each one, some correlation can still remain between elements of these patterns, such that the *i*th element of the one-dimensional pattern and the *i*th element of the two-dimensional pattern are related. As such, in the earlier area lighting example, the distribution of sample points on each light source would not in general cover the entire light due to this correlation, leading to unusual rendering errors.

This problem can be solved easily enough by randomly shuffling the various patterns individually after they are generated. For example, the `LDShuffleScrambled1D()` function first generates a scrambled one-dimensional low-discrepancy sampling pattern, giving a well-distributed set of samples across all of the image samples for this pixel. Then, it shuffles these samples in two ways. Consider, for example, a pixel with 8 image samples, each of which has 4 1D samples for the integrator (giving a total of 32 integrator samples). First, it shuffles samples within each of the 8 groups of 4 samples, putting each set of 4 into a random order. Next, it shuffles each of the 8 groups of 4 samples as a block, with respect to the other blocks of 4 samples.

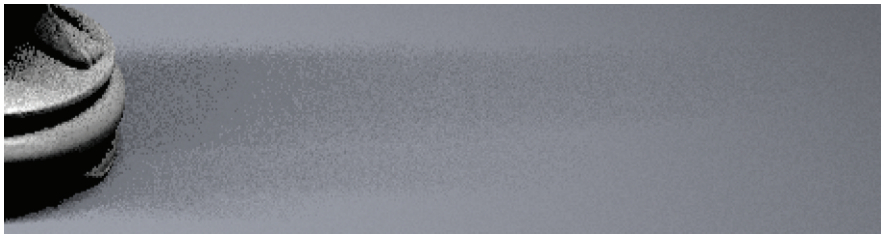
⟨*Sampling Inline Functions*⟩ +≡
```
inline void LDShuffleScrambled1D(int nSamples, int nPixel,
                                 float *samples, RNG &rng) {
    uint32_t scramble = rng.RandomUInt();
    for (int i = 0; i < nSamples * nPixel; ++i)
        samples[i] = VanDerCorput(i, scramble);
    for (int i = 0; i < nPixel; ++i)
        Shuffle(samples + i * nSamples, nSamples, 1, rng);
    Shuffle(samples, nPixel, nSamples, rng);
}
```

⟨*Sampling Inline Functions*⟩ +≡
```
inline void LDShuffleScrambled2D(int nSamples, int nPixel,
                                 float *samples, RNG &rng) {
    uint32_t scramble[2] = { rng.RandomUInt(), rng.RandomUInt() };
    for (int i = 0; i < nSamples * nPixel; ++i)
        Sample02(i, scramble, &samples[2*i]);
    for (int i = 0; i < nPixel; ++i)
        Shuffle(samples + 2 * i * nSamples, nSamples, 2, rng);
    Shuffle(samples, nPixel, 2 * nSamples, rng);
}
```

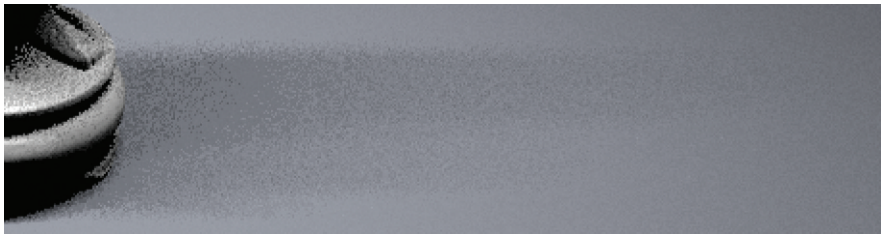
LDSampler::nPixelSamples 374
LDShuffleScrambled1D() 377
RNG 1003
RNG::RandomUInt() 1003
Sample02() 372
Shuffle() 354
VanDerCorput() 372

Given the implementation of `GetMoreSamples()` above, the maximum number of samples returned is just the total number of samples for each pixel.

⟨*LDSampler Public Methods*⟩ +≡ 373
```
int MaximumSampleCount() { return nPixelSamples; }
```



(a)



(b)

Figure 7.30: When the `LDSampler` is used for the area light sampling example, similar results are generated (a) with both 1 image sample and 16 light samples as well as (b) with 16 image samples and 1 light sample, thanks to the (0, 2)-sequence sampling pattern that ensures good distribution of samples over the pixel area in both cases. Compare these images to Figure 7.23, where the stratified pattern generates a much worse set of light samples when only 1 light sample is taken for each of the 16 image samples.

Figure 7.30 shows the result of using the (0, 2)-sequence for the area lighting example scene. Note that not only does it give a visibly better image than stratified patterns, but it also does well with one light sample per image sample, unlike the stratified sampler.

⋆ 7.5 BEST-CANDIDATE SAMPLING PATTERNS

A shortcoming of both the `StratifiedSampler` and the `LDSampler` is that they both generate good image sample patterns around a single pixel, but neither has any mechanism for ensuring that the image samples at adjacent pixels are well distributed with respect to the samples at the current pixel. For example, we would like to avoid having two adjacent pixels choose samples that are very close to their shared edge. The *Poisson disk pattern* addresses this issue of well-separated sample placement and has been shown to be an excellent image sampling pattern. The Poisson disk pattern is a group of points with no two of them closer to each other than some specified distance. Studies have shown that the rods and cones in the eye are distributed in a similar way, which suggests that this pattern might be effective for imaging. An easy way to generate Poisson disk patterns is with *dart throwing*: a program generates samples randomly, throwing away any that are closer to a previous sample than a fixed threshold distance. This can be a very expensive process, since many darts may need to be thrown before one is accepted. The "Further

`LDSampler` 373
`StratifiedSampler` 349

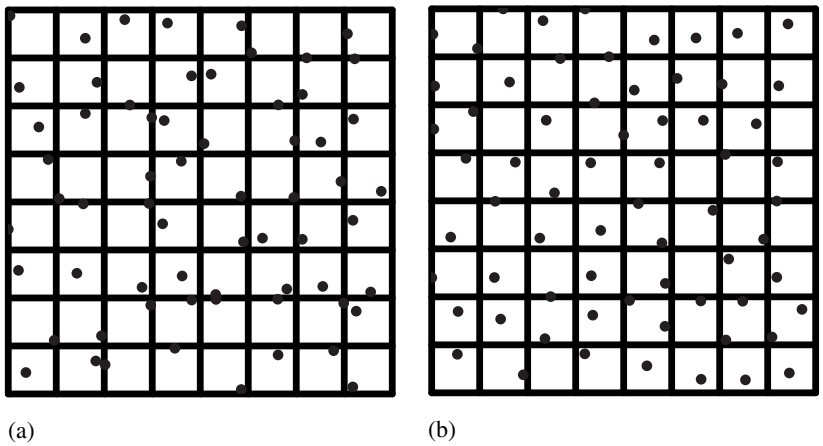


Figure 7.31: Comparison of Sampling Patterns. (a) A jittered pattern: note clumping of samples and undersampling in some areas. (b) A pattern generated with the best-candidate algorithm: it is effectively as good as the Poisson disk pattern.

Reading" section at the end of the chapter has pointers to recent work on more efficient approaches.

A related approach due to Don Mitchell is the *best-candidate* algorithm. Each time a new sample is to be computed, a large number of random candidates are generated. All of these candidates are compared to the previous samples, and the one that is farthest away from all of the previous ones is added to the pattern. Although this algorithm doesn't guarantee the Poisson disk property, it usually does quite well at finding well-separated points if enough candidates are generated. It has the additional advantage that any prefix of the final pattern is itself a well-distributed sampling pattern. Furthermore, the best-candidate algorithm makes it easier to generate a good pattern with a predetermined number of samples than is possible with the dart-throwing algorithm. A comparison of a stratified pattern to a best-candidate pattern is shown in Figure 7.31.

In this section we will present a `Sampler` implementation that uses the best-candidate algorithm to compute sampling patterns that also have good distributions of samples in the additional sampling dimensions. Because generating the sample positions is computationally intensive, we have computed a good sampling pattern once in a preprocess. The pattern encodes 5D image, time, and lens sample positions; it is stored in a table that can be efficiently used at rendering time. Rather than computing a sampling pattern large enough to sample the largest image we'd ever render, the pattern can be reused by tiling it over the image plane. This means that the pattern must have *toroidal topology*—when computing the distance between two samples, we must compute the distance between them as if the square sampling region was rolled into a torus. The code that computed the pattern can be found in the file `tools/samplepat.cpp`; the results are stored in the file `samplers/bestcandidate.out`.

The BestCandidateSampler is the Sampler that uses the best-candidate sample table. If an average of nPixelSamples samples is to be taken in each pixel, then a single copy of the sample table covers SQRT_SAMPLE_TABLE_SIZE / sqrt(nPixelSamples) pixels in the x and y directions.

⟨*BestCandidate Sampling Constants*⟩ ≡
```
#define SQRT_SAMPLE_TABLE_SIZE 64
#define SAMPLE_TABLE_SIZE (SQRT_SAMPLE_TABLE_SIZE * \
                           SQRT_SAMPLE_TABLE_SIZE)
```

We can envision the entire 2D image plane as being covered by replicated instances of this sample table; we will arbitrarily say that the upper left corner of one instance of it lies at the (0, 0) pixel location and that the placement of the other instances follows from there. Thus, if the width (and height) of the table is w, then pixels from (0, 0) to (w, w) are covered by the tile with indices (0, 0), pixels from $(w, 0)$ to $(2w, w)$ are covered by the tile (1, 0), and so forth. Specifically, the mapping from a continuous pixel location (x, y) to the 2D index (i, j) of the sample table that it lies within is given by

$$(i, j) = (\lfloor x/\text{tableWidth} \rfloor, \lfloor y/\text{tableWidth} \rfloor).$$

Figure 7.32 illustrates this idea.

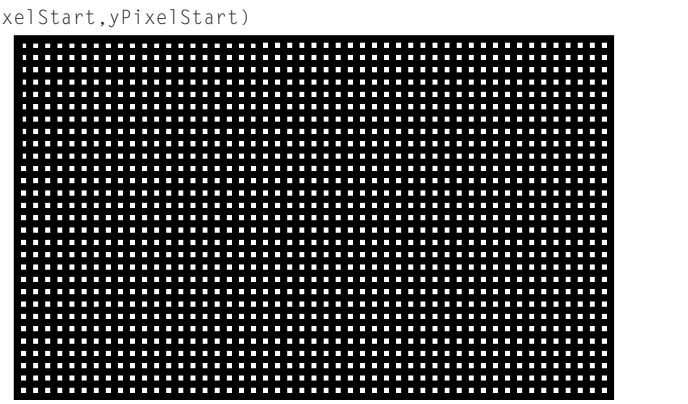


Figure 7.32: Because it is comprised of thousands of samples, the best-candidate sample table will generally cover a large number of pixels. Each instance of the table covers a width and height of some number of pixels, n, based on the size of the table and the number of samples wanted per pixel. One of the main tasks of the BestCandidateSampler is to tile this pattern over the image plane, mapping its sample positions to image sample positions.

BestCandidateSampler 381
Sampler 340
SQRT_SAMPLE_TABLE_SIZE 380

⟨*BestCandidateSampler Declarations*⟩ ≡
```
class BestCandidateSampler : public Sampler {
public:
    ⟨BestCandidateSampler Public Methods 381⟩
private:
    ⟨BestCandidateSampler Private Data 381⟩
};
```

The `tableWidth` member variable stores the raster space width in pixels that the precomputed sample table spans. The constructor then computes the range of sample table indices that must be considered to find all of the samples in the sampler's pixel region. (Because the samples within the sample table don't have any spatial organization, we need to check each one of them to see if it lies within the image tile that samples are being generated for if the tile extent at all overlaps the sampler's pixel region.) This range is stored in the member variables `xTileStart`, `xTileEnd`, `yTileStart`, and `yTileEnd`.

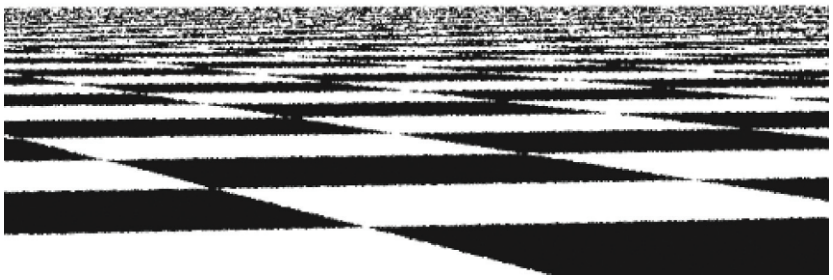
`xTile` and `yTile` hold the current integer indices of the sample table instance being used, and `tableOffset` holds the current offset into the sample table; when it is advanced to the point where it reaches the end of the table, the sampler advances to the next region of the image that the table covers by scanning `xTile` left to right and then `yTile` top to bottom.

Figure 7.33 compares the result of using the best-candidate pattern to the stratified pattern for rendering the checkerboard. Figure 7.34 shows the result of using this pattern for depth of field. For the number of samples used in that figure, the low-discrepancy sampler gives a better result, likely because the best candidate precomputation step searches around a fixed-size region of pixel samples when selecting lens samples. Depending on the actual number of pixel samples used, this region may map to much less or much more than a single pixel area.

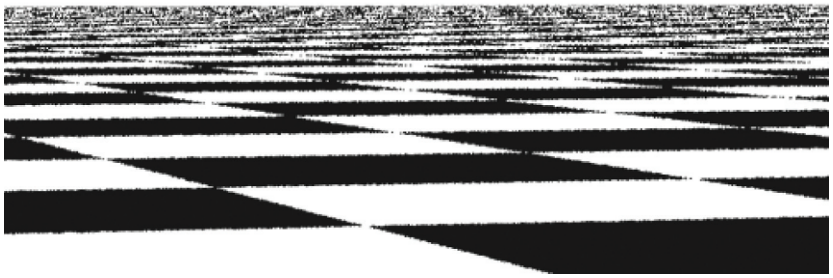
⟨*BestCandidateSampler Public Methods*⟩ ≡ 381
```
BestCandidateSampler(int xstart, int xend, int ystart, int yend,
                     int nPixelSamples, float sopen, float sclose)
    : Sampler(xstart, xend, ystart, yend, nPixelSamples, sopen, sclose) {
    tableWidth = (float)SQRT_SAMPLE_TABLE_SIZE /
                 (float)sqrtf(nPixelSamples);
    xTileStart = Floor2Int(xstart / tableWidth);
    xTileEnd = Floor2Int(xend / tableWidth);
    yTileStart = Floor2Int(ystart / tableWidth);
    yTileEnd = Floor2Int(yend / tableWidth);
    xTile = xTileStart;
    yTile = yTileStart;
    tableOffset = 0;
}
```

⟨*BestCandidateSampler Private Data*⟩ ≡ 381
```
float tableWidth;
int tableOffset;
int xTileStart, xTileEnd, yTileStart, yTileEnd;
int xTile, yTile;
```

BestCandidateSampler 381
BestCandidateSampler::
tableOffset 381
BestCandidateSampler::
tableWidth 381
BestCandidateSampler::
xTile 381
BestCandidateSampler::
xTileEnd 381
BestCandidateSampler::
xTileStart 381
BestCandidateSampler::
yTile 381
BestCandidateSampler::
yTileEnd 381
BestCandidateSampler::
yTileStart 381
Sampler 340
SQRT_SAMPLE_TABLE_SIZE 380



(a)

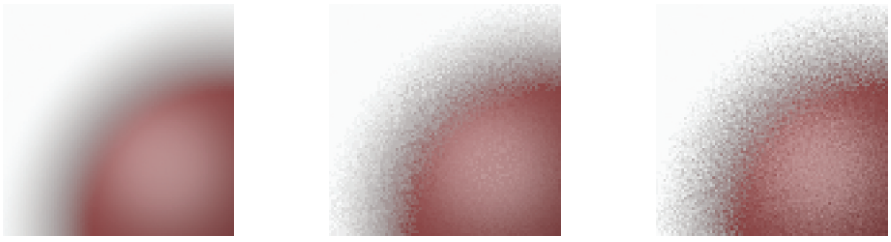


(b)

(c)

(d)

Figure 7.33: Comparison of the Stratified Sampling Pattern with the Best-Candidate Sampling Pattern. (a) The stratified pattern with a single sample per pixel. (b) The best-candidate pattern with a single sample per pixel. (c) The stratified pattern with four samples per pixel. (d) The four-sample best-candidate pattern. Although the differences are subtle, note that the edges of the checks in the foreground are less aliased when the best-candidate pattern is used, and it also does better at resolving the checks toward the horizon, particularly on the sides of the image. Furthermore, the noise from the best-candidate pattern tends to be higher frequency, and therefore more visually acceptable.



(a) (b) (c)

Figure 7.34: (a) Reference depth of field image, and images rendered with (b) the low-discrepancy and (c) best-candidate samplers. Here the low-discrepancy sampler is again the most effective.

Here we incorporate the sample data stored in `samplers/bestcandidate.out`.

⟨*BestCandidateSampler Private Data*⟩ +≡ 381
```
static const float sampleTable[SAMPLE_TABLE_SIZE][5];
```

⟨*BestCandidateSampler Method Definitions*⟩ ≡
```
#include "samplers/bestcandidate.out"
```

The implementations of the `GetSubSampler()`, `RoundSize()`, and `MaximumSampleCount()` methods are straightforward and similar to those of other previously defined samplers in this chapter.

The `BestCandidateSampler::GetMoreSamples()` method returns a single sample for each best-candidate sample point that is inside the sampler's extent.

⟨*BestCandidateSampler Method Definitions*⟩ +≡
```
int BestCandidateSampler::GetMoreSamples(Sample *sample, RNG &rng) {
again:
    if (tableOffset == SAMPLE_TABLE_SIZE) {
        ⟨Advance to next best-candidate sample table position 384⟩
    }
    ⟨Compute raster sample from table 385⟩
    ⟨Check sample against crop window, goto again if outside 385⟩
    ⟨Compute integrator samples for best-candidate sample 385⟩
    ++tableOffset;
    return 1;
}
```

BestCandidateSampler 381
BestCandidateSampler::GetMoreSamples() 383
BestCandidateSampler::tableOffset 381
RNG 1003
Sample 343
SAMPLE_TABLE_SIZE 380

If it has reached the end of the sample table, the sampler tries to move forward to the next precomputed sample tile, scanning across tiles in x and then in y.

⟨*Advance to next best-candidate sample table position*⟩ ≡ 383
```
tableOffset = 0;
if (++xTile > xTileEnd) {
    xTile = xTileStart;
    if (++yTile > yTileEnd)
        return 0;
}
⟨Update sample shifts 384⟩
```

One problem with using tiled sample patterns is that there may be subtle image artifacts aligned with the edges of the pattern on the image plane due to the same values being used repeatedly for time and lens position in each replicated sample region. Not only are the same samples used and reused (whereas the `StratifiedSampler` and `LDSampler` will at least generate different time and lens values for each image sample), but the upper-left sample in each block of samples will always have the same time and lens values, and so on.

A solution to this problem is to transform the set of sample values each time we reuse the pattern. This can be done using *Cranley–Patterson rotations*, which compute

$$X_i' = (X_i + \xi_i) \bmod 1$$

in each dimension, where X_i is the sample value and ξ_i is a random number between zero and one. Because the time and lens portions of sampling patterns were computed with toroidal topology, where the distance between two candidate samples was computed such that, for example, the distance between the points .01 and .99 was computed to be .02, rather than .98, the resulting pattern is still well distributed and seamless. The table of random offsets for time and lens position ξ_i is updated each time the sample pattern is reused.

An important detail is that the `RNG` used to compute the random offsets is seeded with a value derived from the current tile indices; this ensures that the same offsets are used when a best-candidate sample tile spans multiple image tiles used for parallelization.

⟨*Update sample shifts*⟩ ≡ 384
```
RNG tileRng(xTile + (yTile<<8));
for (int i = 0; i < 3; ++i)
    sampleOffsets[i] = tileRng.RandomFloat();
```

⟨*BestCandidateSampler Private Data*⟩ +≡ 381
```
float sampleOffsets[3];
```

Computing the raster space sample position from the positions in the table requires some simple indexing and scaling. We don't use the Cranley–Patterson shifting technique on image samples because this would cause the sampling points at the borders between repeated instances of the table to have a poor distribution. Preserving good image distribution is more important than reducing correlation. The rest of the camera dimensions do use the shifting technique; the `WRAP` macro ensures that the result stays between zero and one.

BestCandidateSampler::
sampleOffsets 384
BestCandidateSampler::
xTile 381
BestCandidateSampler::
yTile 381
LDSampler 373
RNG 1003
RNG::RandomFloat() 1003
StratifiedSampler 349

⟨*Compute raster sample from table*⟩ ≡ 383
```
#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
sample->imageX = (xTile + sampleTable[tableOffset][0]) * tableWidth;
sample->imageY = (yTile + sampleTable[tableOffset][1]) * tableWidth;
sample->time  = Lerp(WRAP(sampleOffsets[0] + sampleTable[tableOffset][2]),
                     shutterOpen, shutterClose);
sample->lensU = WRAP(sampleOffsets[1] +
                     sampleTable[tableOffset][3]);
sample->lensV = WRAP(sampleOffsets[2] +
                     sampleTable[tableOffset][4]);
```

Some of the generated samples will generally be outside the sampler's pixel sample region. The sampler detects this case by checking the sample against the region of pixels to be sampled and generating a new sample if it's out of bounds.

⟨*Check sample against crop window, goto* `again` *if outside*⟩ ≡ 383
```
if (sample->imageX < xPixelStart || sample->imageX >= xPixelEnd ||
    sample->imageY < yPixelStart || sample->imageY >= yPixelEnd) {
    ++tableOffset;
    goto again;
}
```

If the sample is inside the sampler's pixel region, samples for the integrators are generated using shuffled (0, 2)-sequences.

⟨*Compute integrator samples for best-candidate sample*⟩ ≡ 383
```
for (uint32_t i = 0; i < sample->n1D.size(); ++i)
     LDShuffleScrambled1D(sample->n1D[i], 1, sample->oneD[i], rng);
for (uint32_t i = 0; i < sample->n2D.size(); ++i)
     LDShuffleScrambled2D(sample->n2D[i], 1, sample->twoD[i], rng);
```

7.6 ADAPTIVE SAMPLING

The `AdaptiveSampler` implements a simple adaptive sampling algorithm that attempts to more-efficiently generate high-quality images by adding extra samples in parts of the image that are more complex than others. Recall the basic structure of the interaction between `Sampler`s and the main rendering loop in `SamplerRendererTask::Run()` in Section 1.3.4

1. The rendering loop calls `Sampler::GetMoreSamples()` to get a collection of samples.
2. The `SamplerRendererTask` computes the radiance values along the corresponding rays.
3. The samples, radiance values, and intersection information are passed back to the `Sampler::ReportResults()` method.
4. Based on the Boolean return value from `ReportResults()`, the `SamplerRendererTask` either discards the set of samples in preparation for a new set or passes them along to the `Film` to be incorporated into the final image.

AdaptiveSampler 386
BestCandidateSampler::sampleOffsets 384
BestCandidateSampler::sampleTable 383
BestCandidateSampler::tableWidth 381
CameraSample::imageX 342
CameraSample::imageY 342
CameraSample::lensU 342
CameraSample::lensV 342
CameraSample::time 342
Film 403
LDShuffleScrambled1D() 377
LDShuffleScrambled2D() 377
Sample::n1D 344
Sample::n2D 344
Sample::oneD 344
Sample::twoD 344
Sampler 340
Sampler::GetMoreSamples() 340
Sampler::ReportResults() 341
Sampler::shutterClose 340
Sampler::shutterOpen 340
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Sampler::yPixelEnd 340
Sampler::yPixelStart 340
SamplerRendererTask::Run() 30

This interaction allows the sampler to examine the results for a collection of samples and decide to take more samples when appropriate. The values passed to `ReportResults()` can be used in a variety of ways to make this decision.

One challenge with adaptive sampling is finding good criteria for deciding when to take more samples. The challenge is that the information the sampler has to work with is limited: as usual, all it has available to it is the values of a set of point samples, with no information about the behavior of the image function at other locations. The `AdaptiveSampler`, defined in the files `samplers/adaptive.h` and `samplers/adaptive.cpp`, supports two simple refinement criteria. The first is based on checking to see if different shapes are intersected by different samples, which indicates a likely geometric discontinuity, and the second checks for excessive contrast between the colors of different samples.

⟨*AdaptiveSampler Declarations*⟩ ≡
```
class AdaptiveSampler : public Sampler {
public:
    ⟨AdaptiveSampler Public Methods⟩
private:
    ⟨AdaptiveSampler Private Methods⟩
    ⟨AdaptiveSampler Private Data 386⟩
};
```

The `AdaptiveSampler` constructor isn't included here; its main task is to initialize the following member variables. Like most of the other samplers in this chapter, it tracks the current pixel being sampled in `(xPos, yPos)`. It generates at least `minSamples`, but possibly as many as `maxSamples` in each pixel's extent. The `sampleBuf` buffer is used for temporary storage during sample generation, and the `method` enumerant denotes which refinement criteria it will use. Finally, `supersamplePixel` indicates whether the current pixel needs extra samples; it is initialized to `false`.

⟨*AdaptiveSampler Private Data*⟩ ≡ 386
```
int xPos, yPos;
int minSamples, maxSamples;
float *sampleBuf;
enum AdaptiveTest { ADAPTIVE_COMPARE_SHAPE_ID,
                    ADAPTIVE_CONTRAST_THRESHOLD };
AdaptiveTest method;
bool supersamplePixel;
```

The `GetSubSampler()`, `RoundSize()`, and `MaxmimumSampleCount()` methods are essentially the same as those for the `LDSampler`.

In addition to recording the current pixel being sampled, the `AdaptiveSampler` also carries state that reflects whether the pixel needs additional samples in `AdaptiveSampler::supersamplePixel`. The `GetMoreSamples()` and `ReportResults()` methods interact to set it as follows:

- Initially, `supersamplePixel` is `false` and the intial set of `minSamples` is generated for the current pixel by `GetMoreSamples()`. The rendering loop evaluates those sam-

AdaptiveSampler 386
AdaptiveSampler::supersamplePixel 386
LDSampler 373
Sampler 340

ples and reports them back to ReportResults(). If further sampling is indicated, ReportResults() sets supersamplePixel to true and leaves (xPos, yPos) unchanged. Thus, the next call to GetMoreSamples() will generate a new set of maxSamples samples for the pixel. When these samples are provided to ReportResults(), supersamplePixel is reset to false and the current pixel is advanced. The implementation here only does a single stage of adaptive sampling, rather than sampling more if the results continue to vary excessively.

- If further sampling after the initial set of samples isn't needed, then ReportResults() leaves supersamplePixel as false and advances the current pixel coordinates as the other samplers do.

⟨*AdaptiveSampler Method Definitions*⟩ ≡

```
int AdaptiveSampler::GetMoreSamples(Sample *samples, RNG &rng) {
    if (!sampleBuf)
        sampleBuf = new float[LDPixelSampleFloatsNeeded(samples,
                                                        maxSamples)];
    if (supersamplePixel) {
        LDPixelSample(xPos, yPos, shutterOpen, shutterClose, maxSamples,
                      samples, sampleBuf, rng);
        return maxSamples;
    }
    else {
        if (yPos == yPixelEnd) return 0;
        LDPixelSample(xPos, yPos, shutterOpen, shutterClose, minSamples,
                      samples, sampleBuf, rng);
        return minSamples;
    }
}
```

AdaptiveSampler 386
AdaptiveSampler::
maxSamples 386
AdaptiveSampler::
minSamples 386
AdaptiveSampler::
needsSupersampling() 388
AdaptiveSampler::
sampleBuf 386
AdaptiveSampler::
supersamplePixel 386
AdaptiveSampler::xPos 386
AdaptiveSampler::yPos 386
Intersection 186
LDPixelSample() 375
LDPixelSampleFloatsNeeded() 374
RayDifferential 69
RNG 1003
Sample 343
Sampler::shutterClose 340
Sampler::shutterOpen 340
Sampler::yPixelEnd 340
Spectrum 263

Other than maintaining the appropriate state in (xPos, yPos) and supersamplePixel, the bulk of the work for ReportResults() is done by the needsSupersampling() utility method.

⟨*AdaptiveSampler Method Definitions*⟩ +≡

```
bool AdaptiveSampler::ReportResults(Sample *samples,
        const RayDifferential *rays, const Spectrum *Ls,
        const Intersection *isects, int count) {
    if (supersamplePixel) {
        supersamplePixel = false;
        ⟨Advance to next pixel for sampling for AdaptiveSampler 388⟩
        return true;
    }
    else if (needsSupersampling(samples, rays, Ls, isects, count)) {
        supersamplePixel = true;
        return false;
    }
```

```
        else {
            ⟨Advance to next pixel for sampling for AdaptiveSampler 388⟩
            return true;
        }
    }
```

⟨*Advance to next pixel for sampling for* AdaptiveSampler⟩ ≡ 387

```
    if (++xPos == xPixelEnd) {
        xPos = xPixelStart;
        ++yPos;
    }
```

As described above, the AdaptiveSampler supports two different criteria for deciding when to take more samples; the method enumerant indicates which one is used.

⟨*AdaptiveSampler Method Definitions*⟩ +≡

```
    bool AdaptiveSampler::needsSupersampling(Sample *samples,
            const RayDifferential *rays, const Spectrum *Ls,
            const Intersection *isects, int count) {
        switch (method) {
        case ADAPTIVE_COMPARE_SHAPE_ID:
            ⟨See if any shape ids differ within samples 388⟩
        case ADAPTIVE_CONTRAST_THRESHOLD:
            ⟨Compare contrast of sample differences to threshold 389⟩
        }
        return false;
    }
```

A simple test is to see if any of the shape ids for the intersected objects are different. (Recall from Section 3.1 that each Shape is assigned a unique 32-bit identifier.) If some of the rays intersect different Shapes, then it is likely that there are edges in the pixel's extent, such that further sampling may be useful.

This test is somewhat effective but fails in a number of important cases. For example, it's possible to have different shape ids but not have a geometric edge that needs additional sampling (consider two coplanar triangles with the same material that exactly share an edge). Alternatively, a complex Shape like a parametric patch can fold over itself as seen from the camera and have edges with respect to itself that could use more sampling. This criterion also doesn't capture the image function discontinuities due to hard shadow boundaries, which can be just as objectionable as geometric aliasing.

⟨*See if any shape ids differ within samples*⟩ ≡ 388

```
    for (int i = 0; i < count-1; ++i)
        if (isects[i].shapeId != isects[i+1].shapeId ||
            isects[i].primitiveId != isects[i+1].primitiveId)
            return true;
    return false;
```

A different criterion that addresses some of the above shortcomings is to compute the contrast between the colors of the radiance carried by the samples. If the contrast is exces-

AdaptiveSampler::method 386
AdaptiveSampler::xPos 386
AdaptiveSampler::yPos 386
Intersection 186
Intersection::primitiveId 186
Intersection::shapeId 186
RayDifferential 69
Sample 343
Sampler::xPixelEnd 340
Sampler::xPixelStart 340
Shape 108
Spectrum 263

sive, we assume that the pixel would benefit from additional sampling. Here, the average luminance of all of the samples is stored in `Lavg` and then the contrast is computed by finding the relative difference between each sample and the average. If this value is above a threshold, then additional samples are taken.

This test is not always successful, either. One particular problem comes from the interaction with `Textures` that filter high frequencies in their values to avoid aliasing. (The `ImageTexture` is a good example, where image maps are filtered based on the rate at which they are being sampled in order to eliminate high frequencies.) With a complex texture function, the contrast metric may indicate the need for more sampling even though the underlying image function actually doesn't suffer from aliasing.

⟨*Compare contrast of sample differences to threshold*⟩ ≡ 388

```
float Lavg = 0.f;
for (int i = 0; i < count; ++i)
    Lavg += Ls[i].y();
Lavg /= count;
const float maxContrast = 0.5f;
for (int i = 0; i < count; ++i)
    if (fabsf(Ls[i].y() - Lavg) / Lavg > maxContrast)
        return true;
return false;
```

Figure 7.35 shows the operation of these two adaptive sampling criteria with the contemporary house interior scene; each image has a white pixel for pixels that were supersampled and a black pixel otherwise.

7.7 IMAGE RECONSTRUCTION

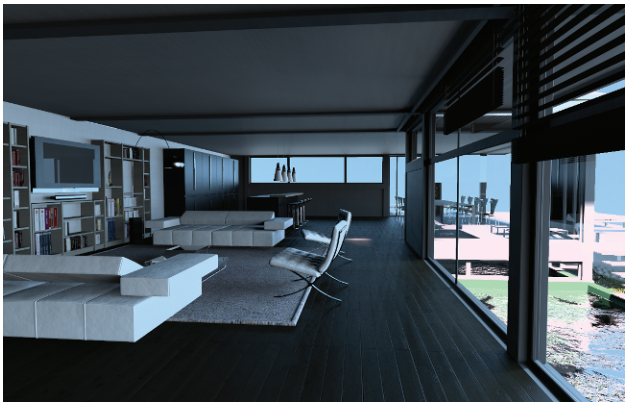
Given carefully chosen image samples, we need to convert the samples and their computed radiance values into pixel values for display or storage. According to signal processing theory, we need to do three things to compute final values for each of the pixels in the output image:

1. Reconstruct a continuous image function $\tilde{L}$ from the set of image samples.
2. Prefilter the function $\tilde{L}$ to remove any frequencies past the Nyquist limit for the pixel spacing.
3. Sample $\tilde{L}$ at the pixel locations to compute the final pixel values.

Because we know that we will be resampling the function $\tilde{L}$ at only the pixel locations, it's not necessary to construct an explicit representation of the function. Instead, we can combine the first two steps using a single filter function.

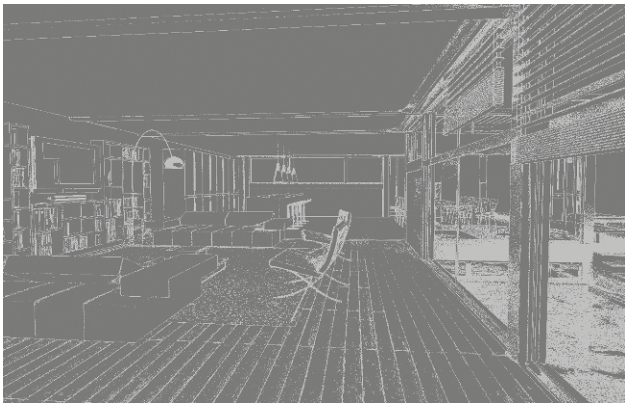
Recall that if the original function had been uniformly sampled at a frequency greater than the Nyquist frequency and reconstructed with the sinc filter, then the reconstructed function in the first step would match the original image function perfectly—quite a feat since we only have point samples. But because the image function almost always will have higher frequencies than could be accounted for by the sampling rate (due to edges, etc.), we chose to sample it nonuniformly, trading off noise for aliasing.

ImageTexture 524
Spectrum::y() 273
Texture 519



(a)

(b)



(c)

Figure 7.35: Adaptive Sampling Criteria in Action. (a) Example scene, rendered normally. (b) Pixels selected for adaptive sampling when more than one shape is intersected by the rays from the original set of samples. (c) Pixels selected for adaptive sampling when using contrast between the samples' radiance values to decide when to take more samples. *(Model courtesy of Florent Boyer.)*

The theory behind ideal reconstruction depends on the samples being uniformly spaced. While a number of attempts have been made to extend the theory to nonuniform sampling, there is not yet an accepted approach to this problem. Furthermore, because the sampling rate is known to be insufficient to capture the function, perfect reconstruction isn't possible. Recent research in the field of sampling theory has revisited the issue of reconstruction with the explicit acknowledgment that perfect reconstruction is not generally attainable in practice. With this slight shift in perspective has come powerful new techniques for reconstruction. See, for example, Unser (2000) for a survey of these developments. In particular, the goal of research in reconstruction theory has shifted from perfect reconstruction to developing reconstruction techniques that can be shown to minimize error between the reconstructed function and the original function, *regardless of whether the original was band limited*.

While the reconstruction techniques used in `pbrt` are not directly built on these new approaches, they serve to explain the experience of practitioners that applying perfect reconstruction techniques to samples taken for image synthesis generally does not result in the highest-quality images.

To reconstruct pixel values, we will consider the problem of interpolating the samples near a particular pixel. To compute a final value for a pixel $I(x, y)$, interpolation results in computing a weighted average

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i) L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}, \tag{7.4}$$

where $L(x_i, y_i)$ is the radiance value of the ith sample located at (x_i, y_i), and f is a filter function. Figure 7.36 shows a pixel at location (x, y) that has a pixel filter with extent

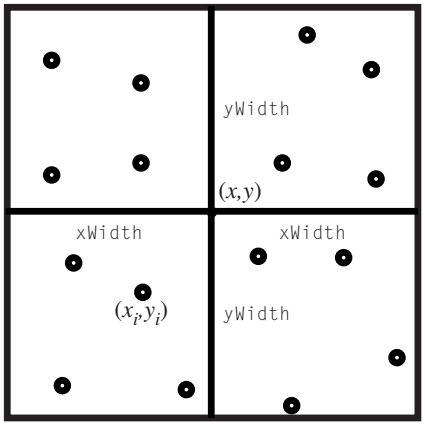
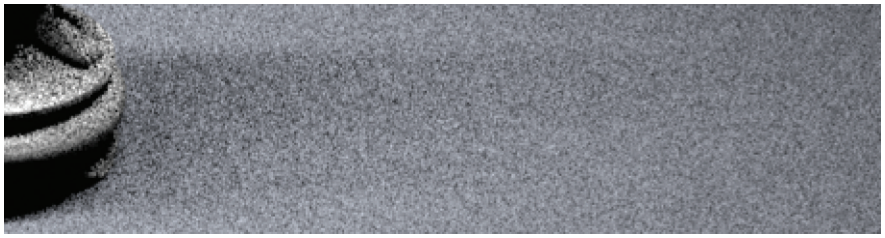
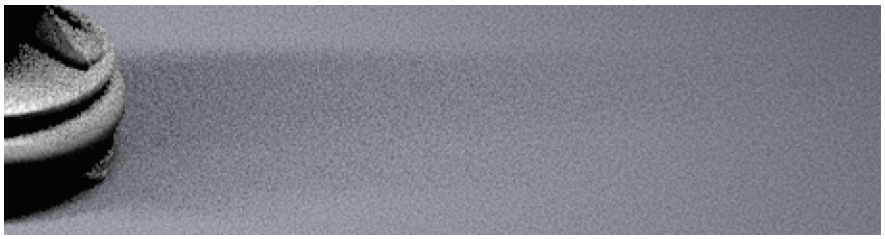


Figure 7.36: 2D Image Filtering. To compute a filtered pixel value for the pixel marked with a filled circle located at (x, y), all of the image samples inside the box around (x, y) with extent `xWidth` and `yWidth` need to be considered. Each of the image samples (x_i, y_i), denoted by open circles, is weighted by a 2D filter function, $f(x - x_i, y - y_i)$. The weighted average of all samples is the final pixel value.



(a)



(b)

Figure 7.37: The choice of pixel reconstruction filter interacts with the results from the sampling pattern in surprising ways. Here, a Mitchell filter has been used to reconstruct pixels in the soft shadows example with the `StratifiedSampler` (a) and the `LDSampler` (b). Note the significant difference and artifacts compared to the images in Figures 7.23 and 7.30. Here, we have used 16 image samples and 1 light sample per pixel.

`xWidth` in the x direction and `yWidth` in the y direction. All of the samples inside the box given by the filter extent may contribute to the pixel's value, depending on the filter function's value for $f(x - x_i, y - y_i)$.

The sinc filter is not an appropriate choice here: recall that the ideal sinc filter is prone to ringing when the underlying function has frequencies beyond the Nyquist limit (Gibbs phenomenon), meaning edges in the image have faint replicated copies of the edge in nearby pixels. Furthermore, the sinc filter has *infinite support*: it doesn't fall off to zero at a finite distance from its center, so all of the image samples would need to be filtered for each output pixel. In practice, there is no single best filter function. Choosing the best one for a particular scene takes a mixture of quantitative evaluation and qualitative judgment.

Another issue that influences the choice of image filter is that the reconstruction filter can interact with the sampling pattern in surprising ways. Recall the `LDSampler`: it generated an extremely well-distributed low-discrepancy pattern over the area of a single pixel, but samples in adjacent pixels were placed without regard for the samples in their neighbors. When used with a box filter, this sampling pattern works extremely well, but when a filter that both spans multiple pixels and isn't a constant value is used, it becomes less effective. Figure 7.37 shows this effect in practice. Using this filter with regular stratified samples makes much less of a difference. Given the remarkable effectiveness of patterns generated with the `LDSampler`, this is something of a quandary: the box filter is the last filter we'd like to use, yet using it instead of another filter substantially improves the results from

`LDSampler` 373
`StratifiedSampler` 349

the LDSampler. Given all of these issues, pbrt provides a variety of different filter function implementations.

7.7.1 FILTER FUNCTIONS

All filter implementations in pbrt are derived from an abstract Filter class, which provides the interface for the $f(x, y)$ functions used in filtering; see Equation (7.4). The Film class (described in the Section 7.8) stores a pointer to a Filter and uses it to filter the output before writing it to a file. Figure 7.38 shows comparisons of zoomed-in regions of images rendered using a variety of the filters from this section to reconstruct pixel values.

The Filter base class is defined in the files core/filter.h and core/filter.cpp.

```
⟨Filter Declarations⟩ ≡
   class Filter {
   public:
       ⟨Filter Interface 393⟩
       ⟨Filter Public Data 394⟩
   };
```

All filters define a width beyond which they have a value of zero; this width may be different in the x and y directions. The constructor takes these values and stores them along with their reciprocals, for use by the filter implementations. The filter's overall extent in each direction (its *support*) is twice the value of its corresponding width (Figure 7.39).

```
⟨Filter Interface⟩ ≡ 393
   Filter(float xw, float yw)
       : xWidth(xw), yWidth(yw), invXWidth(1.f/xw), invYWidth(1.f/yw) {
   }
```

(a) (b) (c)

Figure 7.38: The pixel reconstruction filter used to convert the image samples into pixel values can have a noticeable effect on the character of the final image. Here, we see blowups of a region of the brick wall in the Sponza atrium scene, filtered with (a) the box filter, (b) Gaussian, and (c) Mitchell–Netravali filter. Note that the Mitchell filter gives the sharpest image, while the Gaussian blurs it. The box is the least desirable, since it allows high-frequency aliasing to leak into the final image. (Note artifacts on the top edges of arches, for example.)

Film 403
Filter 393
LDSampler 373

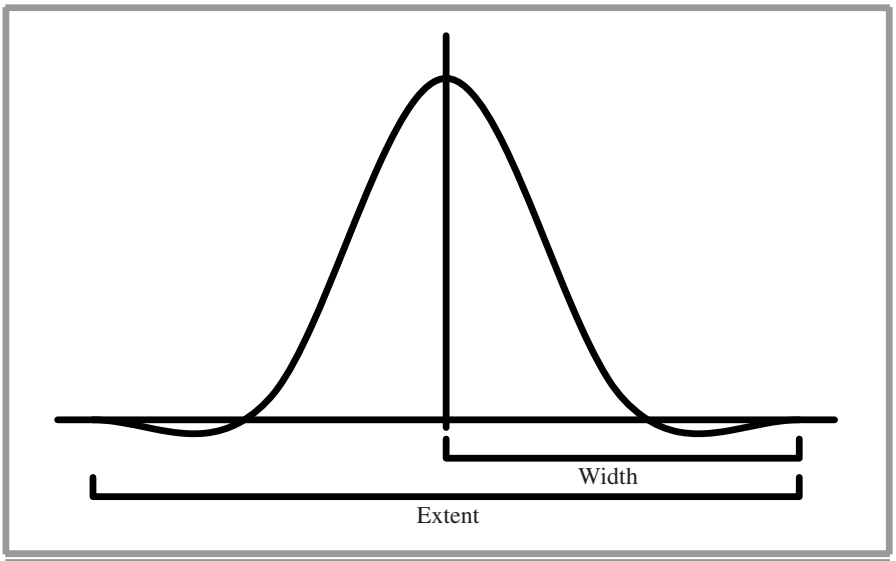


Figure 7.39: The extent of filters in `pbrt` is specified in terms of their width from the origin to its cutoff point. The support of a filter is its total non-zero extent, here equal to twice its width.

⟨*Filter Public Data*⟩ ≡ 393
```
const float xWidth, yWidth;
const float invXWidth, invYWidth;
```

The sole function that `Filter` implementations need to provide is `Evaluate()`. It takes x and y arguments, which give the position of the sample point relative to the center of the filter. The return value specifies the weight of the sample. Code elsewhere in the system will never call the filter function with points outside of the filter's extent, so filter implementations don't need to check for this case.

⟨*Filter Interface*⟩ +≡ 393
```
virtual float Evaluate(float x, float y) const = 0;
```

Box Filter

One of the most commonly used filters in graphics is the *box filter* (and, in fact, when filtering and reconstruction aren't addressed explicitly, the box filter is the *de facto* result). The box filter equally weights all samples within a square region of the image. Although computationally efficient, it's just about the worst filter possible. Recall from the discussion in Section 7.1 that the box filter allows high-frequency sample data to leak into the reconstructed values. This causes postaliasing—even if the original sample values were at a high enough frequency to avoid aliasing, errors are introduced by poor filtering.

Figure 7.40(a) shows a graph of the box filter, and Figure 7.41 shows the result of using the box filter to reconstruct two 1D functions. For the step function we used previously to illustrate the Gibbs phenomenon, the box does reasonably well. However, the results are much worse for a sinusoidal function that has increasing frequency along the x axis. Not only does the box filter do a poor job of reconstructing the function when the frequency

Filter 393

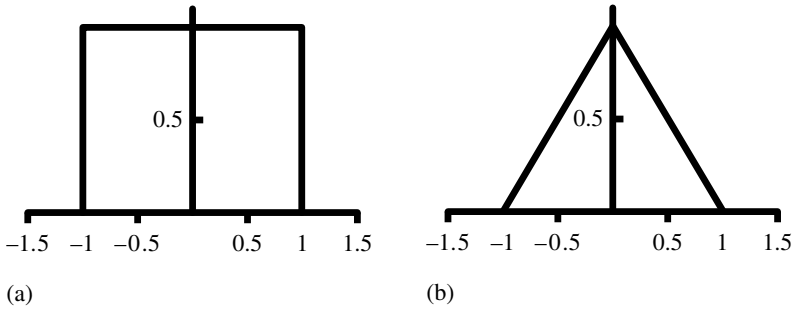


Figure 7.40: Graphs of the (a) box filter and (b) triangle filter. Although neither of these is a particularly good filter, they are both computationally efficient, easy to implement, and good baselines for evaluating other filters.

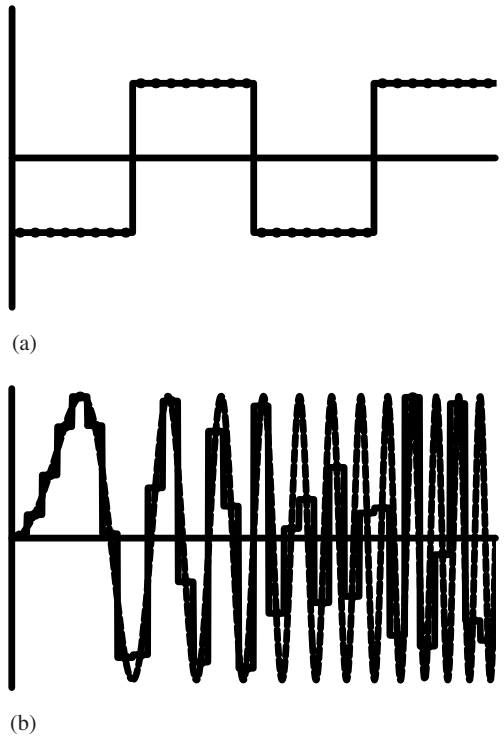


Figure 7.41: The box filter reconstructing (a) a step function and (b) a sinusoidal function with increasing frequency as x increases. This filter does well with the step function, as expected, but does an extremely poor job with the sinusoidal function.

is low, giving a discontinuous result even though the original function was smooth, but it also does an extremely poor job of reconstruction as the function's frequency approaches and passes the Nyquist limit.

⟨*Box Filter Declarations*⟩ ≡
```
class BoxFilter : public Filter {
public:
    BoxFilter(float xw, float yw) : Filter(xw, yw) { }
    float Evaluate(float x, float y) const;
};
```

Because the evaluation function won't be called with (x, y) values outside of the filter's extent, it can always return 1 for the filter function's value.

⟨*Box Filter Method Definitions*⟩ ≡
```
float BoxFilter::Evaluate(float x, float y) const {
    return 1.;
}
```

Triangle Filter

The triangle filter gives slightly better results than the box: samples at the filter center have a weight of one, and the weight falls off linearly to the square extent of the filter. See Figure 7.40(b) for a graph of the triangle filter.

⟨*Triangle Filter Declarations*⟩ ≡
```
class TriangleFilter : public Filter {
public:
    TriangleFilter(float xw, float yw) : Filter(xw, yw) { }
    float Evaluate(float x, float y) const;
};
```

Evaluating the triangle filter is simple: the implementation just computes a linear function based on the width of the filter in both the x and y directions.

⟨*Triangle Filter Method Definitions*⟩ ≡
```
float TriangleFilter::Evaluate(float x, float y) const {
    return max(0.f, xWidth - fabsf(x)) *
           max(0.f, yWidth - fabsf(y));
}
```

Gaussian Filter

Unlike the box and triangle filters, the Gaussian filter gives a reasonably good result in practice. This filter applies a Gaussian bump that is centered at the pixel and radially symmetric around it. The Gaussian's value at the end of its extent is subtracted from the filter value, in order to make the filter go to zero at its limit (Figure 7.42). The Gaussian does tend to cause slight blurring of the final image compared to some of the other filters, but this blurring can actually help mask any remaining aliasing in the image.

BoxFilter 396
Filter 393
Filter::xWidth 394
Filter::yWidth 394
TriangleFilter 396

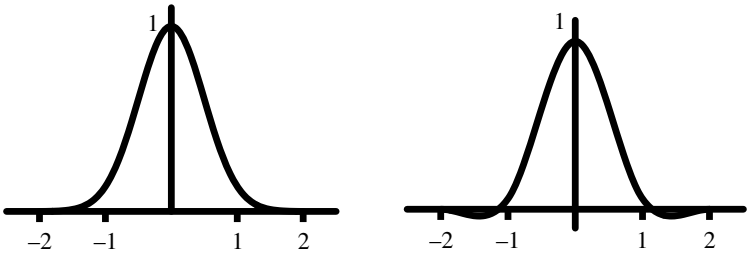


Figure 7.42: Graphs of (a) the Gaussian filter and (b) the Mitchell filter with $B = \frac{1}{3}$ and $C = \frac{1}{3}$, each with a width of two. The Gaussian gives images that tend to be a bit blurry, while the negative lobes of the Mitchell filter help to accentuate and sharpen edges in final images.

⟨*Gaussian Filter Declarations*⟩ ≡
```
class GaussianFilter : public Filter {
public:
    ⟨GaussianFilter Public Methods 397⟩
private:
    ⟨GaussianFilter Private Data 397⟩
    ⟨GaussianFilter Utility Functions 398⟩
};
```

The 1D Gaussian filter function of width w is

$$f(x) = \mathrm{e}^{-\alpha x^2} - \mathrm{e}^{-\alpha w^2},$$

where α controls the rate of falloff of the filter. Smaller values cause a slower falloff, giving a blurrier image. The second term here ensures that the Gaussian goes to zero at the end of its extent, rather than having an abrupt cliff. For efficiency, the constructor precomputes the constant term for $\mathrm{e}^{-\alpha w^2}$ in each direction.

⟨*GaussianFilter Public Methods*⟩ ≡ 397
```
GaussianFilter(float xw, float yw, float a)
    : Filter(xw, yw), alpha(a), expX(expf(-alpha * xWidth * xWidth)),
      expY(expf(-alpha * yWidth * yWidth)) { }
```

Filter 393
GaussianFilter 397
GaussianFilter::alpha 397
GaussianFilter::expX 397
GaussianFilter::expY 397

⟨*GaussianFilter Private Data*⟩ ≡ 397
```
const float alpha;
const float expX, expY;
```

Since a 2D Gaussian function is *separable* into the product of two 1D Gaussians, the implementation calls the `Gaussian()` function twice and multiplies the results.

⟨*Gaussian Filter Method Definitions*⟩ ≡
```
float GaussianFilter::Evaluate(float x, float y) const {
    return Gaussian(x, expX) * Gaussian(y, expY);
}
```

⟨*GaussianFilter Utility Functions*⟩ ≡ 397
```
float Gaussian(float d, float expv) const {
    return max(0.f, float(expf(-alpha * d * d) - expv));
}
```

Mitchell Filter

Filter design is notoriously difficult, mixing mathematical analysis and perceptual experiments. Mitchell and Netravali (1988) have developed a family of parameterized filter functions in order to be able to explore this space in a systematic manner. After analyzing test subjects' subjective responses to images filtered with a variety of parameter values, they developed a filter that tends to do a good job of trading off between *ringing* (phantom edges next to actual edges in the image) and *blurring* (excessively blurred results)—two common artifacts from poor reconstruction filters.

Note from the graph in Figure 7.42(b) that this filter function takes on negative values out by its edges; it has *negative lobes*. In practice these negative regions improve the sharpness of edges, giving crisper images (reduced blurring). If they become too large, however, ringing tends to start to enter the image. Also, because the final pixel values can therefore become negative, they will eventually need to be clamped to a legal output range.

Figure 7.43 shows this filter reconstructing the two test functions. It does extremely well with both of them: there is minimal ringing with the step function, and it does a very good job with the sinusoidal function, up until the point where the sampling rate isn't sufficient to capture the function's detail.

⟨*Mitchell Filter Declarations*⟩ ≡
```
class MitchellFilter : public Filter {
public:
    ⟨MitchellFilter Public Methods 398⟩
private:
    const float B, C;
};
```

The Mitchell filter has two parameters called B and C. Although any values can be used for these parameters, Mitchell and Netravali recommend that they lie along the line $B + 2C = 1$.

⟨*MitchellFilter Public Methods*⟩ ≡ 398
```
MitchellFilter(float b, float c, float xw, float yw)
    : Filter(xw, yw), B(b), C(c) {
}
```

The Mitchell-Netravali filter is the product of one-dimensional filter functions in the x and y directions and is therefore separable, like the Gaussian filter. (In fact, all of the provided filters in pbrt are separable.) Nevertheless, the `Filter::Evaluate()` interface

Filter 393
Filter::Evaluate() 394
GaussianFilter 397
GaussianFilter::expX 397
GaussianFilter::expY 397
MitchellFilter 398

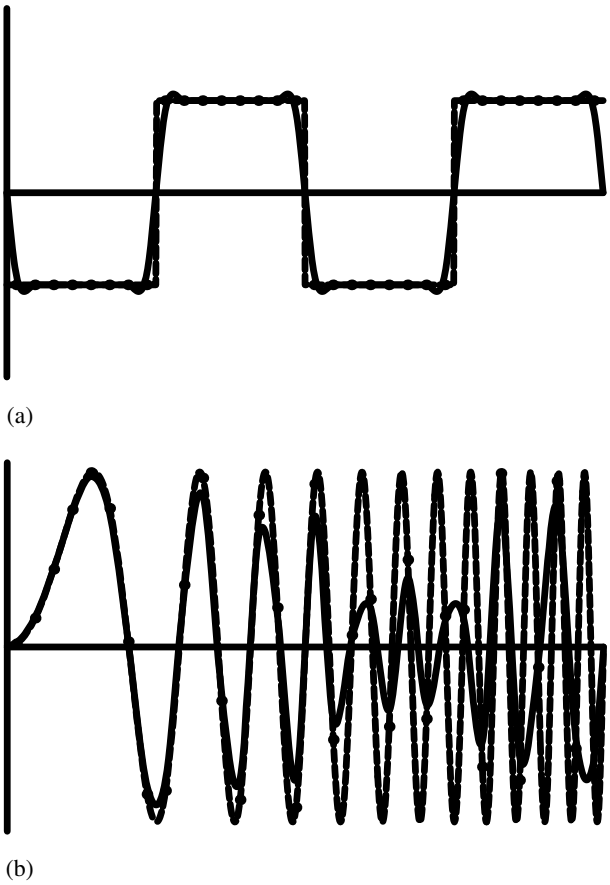


Figure 7.43: The Mitchell–Netravali Filter Used to Reconstruct the Example Functions. It does a good job with both of these functions, (a) introducing minimal ringing with the step function and (b) accurately representing the sinusoid until aliasing from undersampling starts to dominate.

does not enforce this requirement, giving more flexibility in implementing new filters in the future.

⟨*Mitchell Filter Method Definitions*⟩ ≡

```
float MitchellFilter::Evaluate(float x, float y) const {
    return Mitchell1D(x * invXWidth) * Mitchell1D(y * invYWidth);
}
```

Filter::invXWidth 394
Filter::invYWidth 394
MitchellFilter 398
MitchellFilter::Mitchell1D() 400

The 1D function used in the Mitchell filter is an even function defined over the range $[-2, 2]$. This function is made by joining a cubic polynomial defined over $[0, 1]$ with another cubic polynomial defined over $[1, 2]$. This combined polynomial is also reflected around the $x = 0$ plane to give the complete function. These polynomials are controlled by the B and C parameters and are chosen carefully to guarantee C^0 and C^1 continuity

at $x = 0$, $x = 1$, and $x = 2$. The polynomials are

$$f(x) = \frac{1}{6} \begin{cases} (12 - 9B - 6C)|x|^3 + (-18 + 12B + 6C)|x|^2 + (6 - 2B) & |x| < 1 \\ (-B - 6C)|x|^3 + (6B + 30C)|x|^2 + (-12B - 48C)|x| + (8B + 24C) & 1 \le |x| < 2 \\ 0 & \text{otherwise.} \end{cases}$$

⟨*MitchellFilter Public Methods*⟩ +≡ 398

```
float Mitchell1D(float x) const {
    x = fabsf(2.f * x);
    if (x > 1.f)
        return ((-B - 6*C) * x*x*x + (6*B + 30*C) * x*x +
                (-12*B - 48*C) * x + (8*B + 24*C)) * (1.f/6.f);
    else
        return ((12 - 9*B - 6*C) * x*x*x +
                (-18 + 12*B + 6*C) * x*x +
                (6 - 2*B)) * (1.f/6.f);
}
```

Windowed Sinc Filter

Finally, the `LanczosSincFilter` class implements a filter based on the sinc function. In practice, the sinc filter is often multiplied by another function that goes to zero after some distance. This gives a filter function with finite extent, which is necessary for an implementation with reasonable performance. An additional parameter τ controls how many cycles the sinc function passes through before it is clamped to a value of zero, Figure 7.44 shows a graph of three cycles of the sinc function, along with a graph of the windowing function we use, which was developed by Lanczos. The Lanczos window is just the central lobe of the sinc function, scaled to cover the τ cycles:

$$w(x) = \frac{\sin \pi x/\tau}{\pi x/\tau}.$$

Figure 7.44 also shows the filter that we will implement here, which is the product of the sinc function and the windowing function.

Figure 7.45 shows the windowed sinc's reconstruction results for uniform 1D samples. Thanks to the windowing, the reconstructed step function exhibits far less ringing than the reconstruction using the infinite-extent sinc function (compare to Figure 7.11). The windowed sinc filter also does extremely well at reconstructing the sinusoidal function until prealiasing begins.

⟨*Sinc Filter Declarations*⟩ ≡

```
class LanczosSincFilter : public Filter {
public:
    ⟨LanczosSincFilter Public Methods 402⟩
private:
    const float tau;
};
```

Filter 393
LanczosSincFilter 400

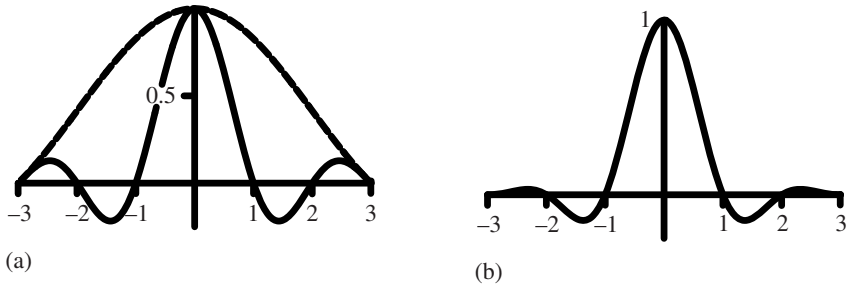


Figure 7.44: Graphs of the Sinc Filter. (a) The sinc function, truncated after three cycles (solid line) and the Lanczos windowing function (dashed line). (b) The product of these two functions, as implemented in the `LanczosSincFilter`.

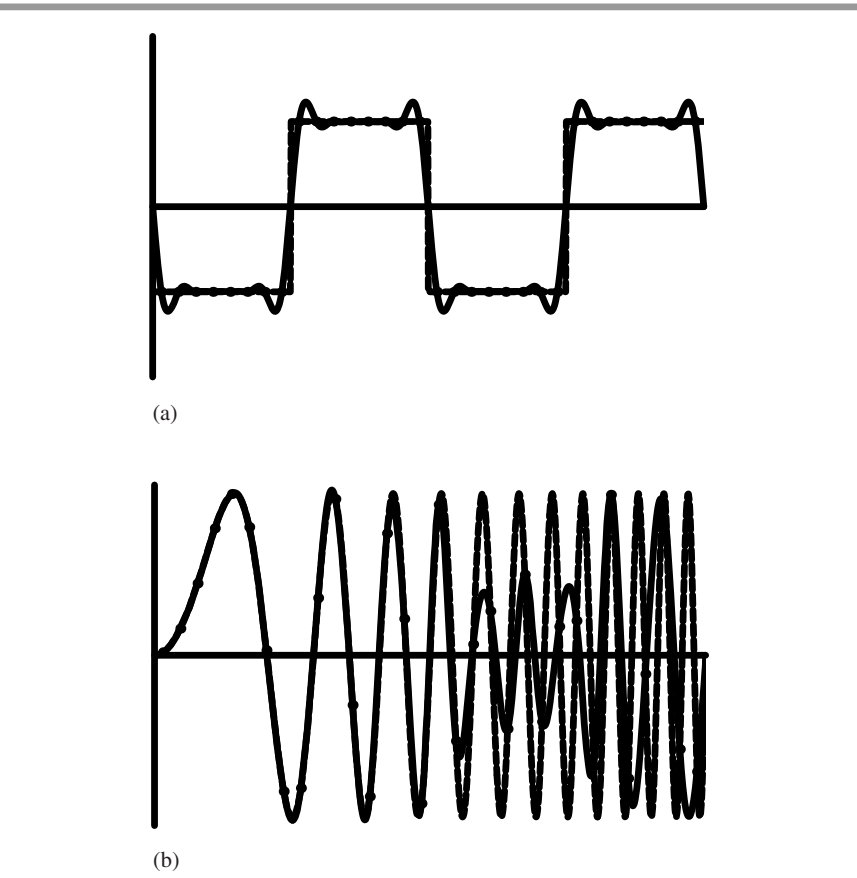


Figure 7.45: Results of Using the Windowed Sinc Filter to Reconstruct the Example Functions. Here, $\tau = 3$. (a) Like the infinite sinc, it suffers from ringing with the step function, although there is much less ringing in the windowed version. (b) The filter does quite well with the sinusoid, however.

⟨*LanczosSincFilter Public Methods*⟩ ≡ 400
```
LanczosSincFilter(float xw, float yw, float t)
    : Filter(xw, yw), tau(t) { }
```

⟨*Sinc Filter Method Definitions*⟩ ≡
```
float LanczosSincFilter::Evaluate(float x, float y) const {
    return Sinc1D(x * invXWidth) * Sinc1D(y * invYWidth);
}
```

The implementation computes the value of the sinc function and then multiplies it by the value of the Lanczos windowing function.

⟨*LanczosSincFilter Public Methods*⟩ +≡ 400
```
float Sinc1D(float x) const {
    x = fabsf(x);
    if (x < 1e-5) return 1.f;
    if (x > 1.)   return 0.f;
    x *= M_PI;
    float sinc = sinf(x * tau) / (x * tau);
    float lanczos = sinf(x) / x;
    return sinc * lanczos;
}
```

7.8 FILM AND THE IMAGING PIPELINE

The type of film or sensor in a camera has a dramatic effect on the way that incident light is eventually transformed into colors in an image. In pbrt, the Film class models the sensing device in the simulated camera. After the radiance is found for each camera ray, a Film implementation determines the sample's contribution to the nearby pixels and updates its representation of the image. When the main rendering loop exits, the Film typically writes the final image to a file on disk.

This section only provides a single Film implementation. It applies the pixel reconstruction equation to compute final pixel values and writes the image to disk with floating-point color values. For a physically based renderer, creating images in a floating-point format provides more flexibility in how the output can be used than if a typical image format with 8-bit unsigned integer values is used; floating-point formats avoid the substantial loss of information that comes from image quantization to 8-bit image formats.

In order to display such images on modern display devices, however, it is necessary to map these floating-point pixel values to discrete values for display. For example, computer monitors generally expect the color of each pixel to be described by an RGB color triple, not an arbitrary spectral power distribution. Spectra described by general basis function coefficients must therefore be converted to an RGB representation before they can be displayed. A related problem is that displays have a substantially smaller range of displayable radiance values than the range present in many real-world scenes. Therefore, the pixel values must be mapped to the displayable range in a way that causes the final displayed image to appear as close as possible to the way it would appear on an ideal

Film 403
Filter 393
Filter::invXWidth 394
Filter::invYWidth 394
LanczosSincFilter 400
LanczosSincFilter::Sinc1D() 402
M_PI 1002

display device without this limitation. These issues are addressed by research into *tone mapping*; the "Further Reading" section has more information about this topic.

7.8.1 FILM INTERFACE

The `Film` base class, defined in `core/film.h` and `core/film.cpp`, defines the abstract interface for `Film` implementations:

⟨*Film Declarations*⟩ ≡
```
class Film {
public:
    ⟨Film Interface 403⟩
    ⟨Film Public Data 403⟩
};
```

The `Film` constructor must be given the overall resolution of the image in the x and y directions; these are stored in the public member variables `Film::xResolution` and `Film::yResolution`. The `Cameras` in Chapter 6 need these values to compute some of the camera-related transformations, such as the raster-to-camera-space transformations.

⟨*Film Interface*⟩ ≡ 403
```
Film(int xres, int yres)
    : xResolution(xres), yResolution(yres) { }
```

⟨*Film Public Data*⟩ ≡ 403
```
const int xResolution, yResolution;
```

There are two methods for providing data to the film. The first is driven by `Samples` generated by the `Sampler`; for each such sample, `Film::AddSample()` is called. It takes a sample and corresponding radiance value, applies the reconstruction filter, and and updates the stored image. The expectation with this method is that the sampling density around a pixel is not related to the pixel's final value. (In other words, the final pixel value is effectively a weighted average of the nearby samples.) Under this assumption, the pixel filtering equation can be used to compute the final values.

⟨*Film Interface*⟩ +≡ 403
```
virtual void AddSample(const CameraSample &sample,
                       const Spectrum &L) = 0;
```

The second method for providing data to the film is `Splat()`. Splatting similarly updates pixel values, but rather than computing the final pixel value as a weighted average, splats are simply summed. Thus, the more splats that are around a given pixel, the brighter the pixel will be. This method is used by light transport algorithms like the one in the `MetropolisRenderer`, where more samples are generated in areas where the image is brighter. It would also be used by bidirectional light transport algorithms that followed paths starting from lights and then projected illuminated points onto the film.

⟨*Film Interface*⟩ +≡ 403
```
virtual void Splat(const CameraSample &sample, const Spectrum &L) = 0;
```

The `Film` is responsible for determining the range of integer pixel values that the `Sampler` is responsible for generating samples for. While this range would be from (0, 0) to

Camera 302
CameraSample 342
Film 403
Film::AddSample() 403
Film::xResolution 403
Film::yResolution 403
MetropolisRenderer 852
Sample 343
Sampler 340
Spectrum 263

(xResolution − 1, yResolution − 1) for a simple film implementation, in general it is necessary to sample the image plane at locations slightly beyond the edges of the final image due to the finite extent of pixel reconstruction filters. This range is returned by the `GetSampleExtent()` method.

⟨*Film Interface*⟩ +≡ 403
```
virtual void GetSampleExtent(int *xstart, int *xend,
                             int *ystart, int *yend) const = 0;
```

`GetPixelExtent()` provides the range of pixels in the actual image.

⟨*Film Interface*⟩ +≡ 403
```
virtual void GetPixelExtent(int *xstart, int *xend,
                            int *ystart, int *yend) const = 0;
```

Some `Film` implementations find it useful to be notified that a region of the pixels has recently been updated. In particular, an implementation that opened a window to show the image as it was being rendered could use this information to trigger an update of a subregion of the window after its samples had been generated. Here we provide a default empty implementation of this method so that `Film` implementations that don't need this information don't need to implement it themselves.

⟨*Film Method Definitions*⟩ ≡
```
void Film::UpdateDisplay(int x0, int y0, int x1, int y1,
                         float splatScale) {
}
```

After the main rendering loop exits, the `SamplerRenderer::Render()` method calls the `Film::WriteImage()` method, which allows the film to do any processing necessary to generate the final image and display it or store it in a file. Like `UpdateDisplay()`, it takes a scale factor for the samples provided to the `Splat()` method.

⟨*Film Interface*⟩ +≡ 403
```
virtual void WriteImage(float splatScale = 1.f) = 0;
```

7.8.2 IMAGEFILM

The `ImageFilm` is a `Film` implementation that filters image sample values with a given reconstruction filter and writes the resulting image to disk. It is implemented in the files `film/image.h` and `film/image.cpp`.

⟨*ImageFilm Declarations*⟩ ≡
```
class ImageFilm : public Film {
public:
    ⟨ImageFilm Public Methods⟩
private:
    ⟨ImageFilm Private Data 405⟩
};
```

The `ImageFilm` constructor takes a number of extra parameters beyond the overall image resolution, including a filter function, a crop window that specifies a subrectangle of the pixels to be rendered, and the filename for the output image. On some systems, the

Film 403
Film::WriteImage() 404
ImageFilm 404
SamplerRenderer::Render() 27

`ImageFilm` can be configured to open a window and show the image as it's being rendered; the `openWindow` parameter to the constructor controls this. (In the interest of brevity, we won't include the code related to this in the book here, however.)

⟨*ImageFilm Method Definitions*⟩ ≡
```
ImageFilm::ImageFilm(int xres, int yres, Filter *filt, const float crop[4],
                     const string &fn, bool openWindow)
    : Film(xres, yres) {
    filter = filt;
    memcpy(cropWindow, crop, 4 * sizeof(float));
    filename = fn;
    ⟨Compute film image extent 406⟩
    ⟨Allocate film image storage 407⟩
    ⟨Precompute filter weight table 407⟩
    ⟨Possibly open window for image display⟩
}
```

⟨*ImageFilm Private Data*⟩ ≡ 404
```
Filter *filter;
float cropWindow[4];
string filename;
```

In conjunction with the overall image resolution, the crop window gives the extent of pixels that need to be actually stored and written out. Crop windows are useful for debugging or for breaking a large image into small pieces that can be rendered on different computers and reassembled later. The crop window is specified in NDC space, with each coordinate ranging from zero to one (Figure 7.46). `ImageFilm::xPixelStart` and `ImageFilm::yPixelStart` store the pixel position of the upper-left corner of the crop window, and `ImageFilm::xPixelCount` and `ImageFilm::yPixelCount` give the total number of pixels in each direction. Their values are easily computed from the overall image resolution and the crop window, although the calculations must be done carefully such that if

Film 403
Filter 393
ImageFilm 404
ImageFilm::cropWindow 405
ImageFilm::filename 405
ImageFilm::filter 405
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406

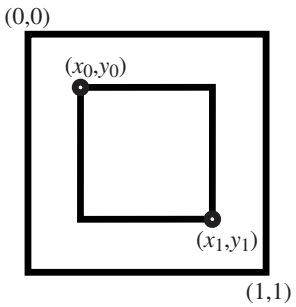


Figure 7.46: The image crop window specifies a subset of the image to be rendered. It is given in NDC space, with coordinates ranging from $(0, 0)$ to $(1, 1)$. The `ImageFilm` class only allocates space for and stores pixel values in the region inside the crop window.

an image is rendered in pieces with abutting crop windows, each final pixel will be present in only one of the subimages.

⟨*Compute film image extent*⟩ ≡ 405
```
xPixelStart = Ceil2Int(xResolution * cropWindow[0]);
xPixelCount = max(1, Ceil2Int(xResolution * cropWindow[1]) - xPixelStart);
yPixelStart = Ceil2Int(yResolution * cropWindow[2]);
yPixelCount = max(1, Ceil2Int(yResolution * cropWindow[3]) - yPixelStart);
```

⟨*ImageFilm Private Data*⟩ +≡ 404
```
int xPixelStart, yPixelStart, xPixelCount, yPixelCount;
```

Given the pixel resolution of the (possibly cropped) image, the constructor allocates an array of `Pixel` structures, one for each pixel. The running weighted sums of pixel radiance values are stored in the `Lxyz` member variable, and `weightSum` holds the sum of filter weight values for the sample contributions to the pixel. `splatXYZ` holds the (unweighted) sum of sample splats. The `pad` member is unused; its sole purpose is to ensure that the `Pixel` structure is 32 bytes large, rather than 28 as it would be otherwise. This ensures that a `Pixel` won't straddle a cache line, so that no more than one cache miss will be incurred when a `Pixel` is accessed (as long as the first `Pixel` in the array is allocated at the start of a cache line).

The `ImageFilm` uses XYZ color values to store pixel colors (Section 5.2.1). Two natural alternatives would be to use `Spectrum` values, or to store RGB color. Here, it isn't worthwhile to store complete `Spectrum` values, even when doing full spectral rendering. Because the final colors written to the output file don't include the full set of `Spectrum` samples, converting to a tristimulus value here doesn't represent a loss of information versus storing `Spectrum`s and converting to a tristimulus value on image output. Not storing complete `Spectrum` values in this case can save a substantial amount of memory if the `Spectrum` has a large number of samples.

We have chosen to use XYZ color rather than RGB to emphasize that XYZ is a display-independent representation of color, while RGB requires assuming a particular set of display response curves (Section 5.2.2). (In the end, we will, however, have to convert to RGB, since few image file formats store XYZ color.)

⟨*ImageFilm Private Data*⟩ +≡ 404
```
struct Pixel {
    Pixel() {
        for (int i = 0; i < 3; ++i) Lxyz[i] = splatXYZ[i] = 0.f;
        weightSum = 0.f;
    }
    float Lxyz[3];
    float weightSum;
    float splatXYZ[3];
    float pad;
};
BlockedArray<Pixel> *pixels;
```

BlockedArray 1017
Ceil2Int() 1002
Film::xResolution 403
Film::yResolution 403
ImageFilm::cropWindow 405
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Pixel 406
Pixel::weightSum 406
Spectrum 263

Because small rectangular blocks of pixels are typically accessed for each image sample, the `ImageFilm` uses a `BlockedArray` to store the pixels, which helps reduce the number of cache misses as samples arrive and pixel values are updated. The `BlockedArray` also ensures that the dynamically allocated memory for the array is cache aligned.

⟨*Allocate film image storage*⟩ ≡ 405
```
pixels = new BlockedArray<Pixel>(xPixelCount, yPixelCount);
```

With typical filter settings, every image sample may contribute to 16 or more pixels in the final image. Particularly for simple scenes, where relatively little time is spent on ray intersection testing and shading computations, the time spent updating the image for each sample can be significant. Therefore, the `ImageFilm` precomputes a table of filter values so that the `Film::AddSample()` method can avoid the expense of virtual function calls to the `Filter::Evaluate()` method as well as the expense of evaluating the filter and can instead use values from the table for filtering. The error introduced by not evaluating the filter at each sample's precise location isn't noticeable in practice.

The implementation here makes the reasonable assumption that the filter is defined such that $f(x, y) = f(|x|, |y|)$, so the table needs to hold values for only the positive quadrant of filter offsets. This assumption is true for all of the `Filters` currently available in `pbrt` and is true for most filters used in practice. This makes the table one-fourth the size and improves the coherence of memory accesses, leading to better cache performance.[8]

⟨*Precompute filter weight table*⟩ ≡ 405
```
#define FILTER_TABLE_SIZE 16
filterTable = new float[FILTER_TABLE_SIZE * FILTER_TABLE_SIZE];
float *ftp = filterTable;
for (int y = 0; y < FILTER_TABLE_SIZE; ++y) {
    float fy = ((float)y + .5f) *
               filter->yWidth / FILTER_TABLE_SIZE;
    for (int x = 0; x < FILTER_TABLE_SIZE; ++x) {
        float fx = ((float)x + .5f) *
                   filter->xWidth / FILTER_TABLE_SIZE;
        *ftp++ = filter->Evaluate(fx, fy);
    }
}
```

⟨*ImageFilm Private Data*⟩ +≡ 404
```
float *filterTable;
```

To understand the operation of `ImageFilm::AddSample()`, first recall the pixel filtering equation:

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i) L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}.$$

BlockedArray 1017
Film::AddSample() 403
Filter 393
Filter::Evaluate() 394
Filter::xWidth 394
Filter::yWidth 394
FILTER_TABLE_SIZE 407
ImageFilm 404
ImageFilm::AddSample() 408
ImageFilm::filter 405
ImageFilm::pixels 406
ImageFilm::xPixelCount 406
ImageFilm::yPixelCount 406
Pixel 406

8 The implementation here could further take advantage of the fact that all filters currently in `pbrt` are separable, only allocating a 1D table. However, to more easily allow different filter functions to be added, we don't assume separability here.

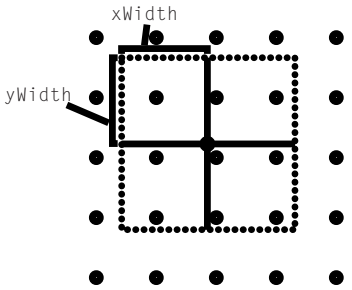


Figure 7.47: Given an image sample at some position on the image plane (solid dot), it is necessary to determine which pixel values (empty dots) are affected by the sample's contribution. This is done by taking the offsets in the x and y directions according to the pixel reconstruction filter's width (solid lines) and finding the pixels inside this region.

It computes each pixel's value $I(x, y)$ as the weighted sum of nearby samples' radiance values, using a filter function f to compute the weights. Because all of the `Filters` in `pbrt` have finite extent, this method starts by computing which pixels will be affected by the current sample. Then, turning the pixel filtering equation inside out, it updates two running sums for each pixel (x, y) that is affected by the sample. One sum accumulates the numerator of the pixel filtering equation and the other accumulates the denominator. When all of the samples have been processed, the final pixel values are computed by performing the division.

```
⟨ImageFilm Method Definitions⟩ +≡
  void ImageFilm::AddSample(const CameraSample &sample,
                            const Spectrum &L) {
      ⟨Compute sample's raster extent 408⟩
      ⟨Loop over filter support and add sample to pixel arrays 409⟩
  }
```

To find which pixels a sample potentially contributes to, `Film::AddSample()` converts the continuous sample coordinates to discrete coordinates by subtracting 0.5 from x and y. It then offsets this value by the filter width in each direction (Figure 7.47) and takes the ceiling of the minimum coordinates and the floor of the maximum, since pixels outside the bound of the extent are guaranteed to be unaffected by the sample.

```
⟨Compute sample's raster extent⟩ ≡ 408
  float dimageX = sample.imageX - 0.5f;
  float dimageY = sample.imageY - 0.5f;
  int x0 = Ceil2Int (dimageX - filter->xWidth);
  int x1 = Floor2Int(dimageX + filter->xWidth);
  int y0 = Ceil2Int (dimageY - filter->yWidth);
  int y1 = Floor2Int(dimageY + filter->yWidth);
  x0 = max(x0, xPixelStart);
  x1 = min(x1, xPixelStart + xPixelCount - 1);
```

CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
Ceil2Int() 1002
Film::AddSample() 403
Filter 393
Filter::xWidth 394
Filter::yWidth 394
Floor2Int() 1002
ImageFilm 404
ImageFilm::filter 405
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Spectrum 263

```
y0 = max(y0, yPixelStart);
y1 = min(y1, yPixelStart + yPixelCount - 1);
if ((x1-x0) < 0 || (y1-y0) < 0)
    return;
```

Given the extent of pixels that are affected by this sample—(x0,y0) to (x1,y1), inclusive—the method can now loop over all of those pixels and then filter the sample value appropriately.

Before the loop starts, it checks to see if the filter is greater than one pixel wide. If this is true, then sample values from more than one SamplerRendererTask from the SamplerRenderer may affect a particular pixel's value. (Samples near the boundary of their image tiles will affect pixels in other image tiles due to the filter's extent.) In this case, we must ensure that multiple threads entering this method at the same time correctly coordinate their updates of pixel values. On the other hand, if the filter is a pixel wide (or less), then we know that only a single SamplerRendererTask will be updating any pixel's value, and that therefore no synchronization is needed for pixel updates here.[9]

⟨*Loop over filter support and add sample to pixel arrays*⟩ ≡ 408

```
float xyz[3];
L.ToXYZ(xyz);
⟨Precompute x and y filter table offsets 410⟩
bool syncNeeded = (filter->xWidth > 0.5f || filter->yWidth > 0.5f);
for (int y = y0; y <= y1; ++y) {
    for (int x = x0; x <= x1; ++x) {
        ⟨Evaluate filter value at (x, y) pixel 410⟩
        ⟨Update pixel values with filtered sample contribution 410⟩
    }
}
```

Each discrete integer pixel (x, y) has an instance of the filter function centered around it. To compute the filter weight for a particular sample, it's necessary to find the offset from the pixel to the sample's position in discrete coordinates and evaluate the filter function. If we were evaluating the filter explicitly, the appropriate computation would be

```
filterWt = filter->Evaluate(x - dimageX, y - dimageY);
```

Instead, the implementation retrieves the appropriate filter weight from the table.

To find the filter weight for a pixel (x', y') given the sample position (x, y), this routine computes the offset $(x' - x, y' - y)$ and converts it into coordinates for the filter weights lookup table. This can be done directly by dividing each component of the sample offset by the filter width in that direction, giving a value between zero and one, and then multiplying by the table size. This process can be further optimized by noting that along each row of pixels in the x direction, the difference in y, and thus the y offset into the filter table, is constant. Analogously, for each column of pixels, the x offset is constant.

Filter::xWidth 394
Filter::yWidth 394
ImageFilm 404
ImageFilm::filter 405
Renderer 24
SamplerRenderer 25
SamplerRendererTask 29
Spectrum::ToXYZ() 272

9 Note that this represents a potentially undesirable coupling between the parallelization strategy in the SamplerRenderer and the ImageFilm here; for another Renderer that used a different parallelization scheme such that multiple tasks updated the same pixel value, this expectation would be incorrect.

Therefore, before looping over the pixels here it's possible to precompute these indices, saving repeated work in the loop.

⟨*Precompute x and y filter table offsets*⟩ ≡ 409
```
int *ifx = ALLOCA(int, x1 - x0 + 1);
for (int x = x0; x <= x1; ++x) {
    float fx = fabsf((x - dimageX) *
                     filter->invXWidth * FILTER_TABLE_SIZE);
    ifx[x-x0] = min(Floor2Int(fx), FILTER_TABLE_SIZE-1);
}
int *ify = ALLOCA(int, y1 - y0 + 1);
for (int y = y0; y <= y1; ++y) {
    float fy = fabsf((y - dimageY) *
                     filter->invYWidth * FILTER_TABLE_SIZE);
    ify[y-y0] = min(Floor2Int(fy), FILTER_TABLE_SIZE-1);
}
```

Now at each pixel, the x and y offsets into the filter table can be found for the pixel, leading to the offset into the array and thus the filter value.

⟨*Evaluate filter value at* (x, y) *pixel*⟩ ≡ 409
```
int offset = ify[y-y0]*FILTER_TABLE_SIZE + ifx[x-x0];
float filterWt = filterTable[offset];
```

For each affected pixel, we need to add the weighted XYZ color and the filter weight. If no synchronization is needed, standard addition suffices.

⟨*Update pixel values with filtered sample contribution*⟩ ≡ 409
```
Pixel &pixel = (*pixels)(x - xPixelStart, y - yPixelStart);
if (!syncNeeded) {
    pixel.Lxyz[0] += filterWt * xyz[0];
    pixel.Lxyz[1] += filterWt * xyz[1];
    pixel.Lxyz[2] += filterWt * xyz[2];
    pixel.weightSum += filterWt;
}
else {
    ⟨Safely update Lxyz and weightSum even with concurrency 411⟩
}
```

Otherwise, we have to handle the case of multiple threads potentially trying to update this pixel's values simultaneously. A number of options are possible. We could require that a mutex be acquired before updating any pixel values, but as the number of simultaneously executing threads increased, this mutex could become a point of contention: if any thread was updating any part of the image, none of the others would be able to update even a completely different part of the image.

This problem in turn could be addressed by using multiple mutexes, where each mutex was responsible for controlling access to a subset of the image (a scanline, a tile of small size, etc). In this case, we'd just need to acquire the appropriate set of mutexes, update,

ALLOCA() 1009
Filter::invXWidth 394
Filter::invYWidth 394
FILTER_TABLE_SIZE 407
Floor2Int() 1002
ImageFilm::filterTable 407
ImageFilm::pixels 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelStart 406
Pixel 406

and return. This approach may instead suffer from the overhead of multiple mutex operations, however.

Therefore, here we use atomic processor operations to safely update each of the pixel values; the processor will ensure that each update is computed correctly. Another nice property of this approach is that under the common case when no two threads are trying to modify the same pixel values, the overhead of these atomic operations is reasonable; with many hardware architectures, only as multiple threads are concurrently trying to modify the same location does the cost of atomics increase.

⟨*Safely update* `Lxyz` *and* `weightSum` *even with concurrency*⟩ ≡ 410
```
AtomicAdd(&pixel.Lxyz[0], filterWt * xyz[0]);
AtomicAdd(&pixel.Lxyz[1], filterWt * xyz[1]);
AtomicAdd(&pixel.Lxyz[2], filterWt * xyz[2]);
AtomicAdd(&pixel.weightSum, filterWt);
```

The implementation of the `Splat()` method here just has to compute which pixel the sample maps to and safely add the value. It always uses atomic operations under the expectation that, in general, it's likely that multiple threads may need to splat values to the same pixel. The implementation here effectively uses a pixel-wide box filter to filter the samples; Exercise 7.3 at the end of the chapter discusses how to address this shortcoming.

⟨*ImageFilm Method Definitions*⟩ +≡
```
void ImageFilm::Splat(const CameraSample &sample, const Spectrum &L) {
    float xyz[3];
    L.ToXYZ(xyz);
    int x = Floor2Int(sample.imageX), y = Floor2Int(sample.imageY);
    if (x < xPixelStart || x - xPixelStart >= xPixelCount ||
        y < yPixelStart || y - yPixelStart >= yPixelCount) return;
    Pixel &pixel = (*pixels)(x - xPixelStart, y - yPixelStart);
    AtomicAdd(&pixel.splatXYZ[0], xyz[0]);
    AtomicAdd(&pixel.splatXYZ[1], xyz[1]);
    AtomicAdd(&pixel.splatXYZ[2], xyz[2]);
}
```

AtomicAdd() 1038
CameraSample 342
CameraSample::imageX 342
CameraSample::imageY 342
Filter::xWidth 394
Filter::yWidth 394
Floor2Int() 1002
ImageFilm 404
ImageFilm::filter 405
ImageFilm::pixels 406
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Pixel 406
Pixel::Lxyz 406
Pixel::splatXYZ 406
Pixel::weightSum 406
Sampler 340
Spectrum 263
Spectrum::ToXYZ() 272

Because the pixel reconstruction filter spans a number of pixels, the `Sampler` must generate image samples a bit outside of the range of pixels that will actually be output. This way, even pixels at the boundary of the image will have an equal density of samples around them in all directions and won't be biased with only values from toward the interior of the image. This is also important when rendering images in pieces with crop windows, since it eliminates artifacts at the edges of the subimages.

⟨*ImageFilm Method Definitions*⟩ +≡
```
void ImageFilm::GetSampleExtent(int *xstart, int *xend,
                                int *ystart, int *yend) const {
    *xstart = Floor2Int(xPixelStart + 0.5f - filter->xWidth);
    *xend   = Floor2Int(xPixelStart + 0.5f + xPixelCount  +
                        filter->xWidth);
```

```
    *ystart = Floor2Int(yPixelStart + 0.5f - filter->yWidth);
    *yend   = Floor2Int(yPixelStart + 0.5f + yPixelCount +
                        filter->yWidth);
}
```

⟨*ImageFilm Method Definitions*⟩ +≡

```
void ImageFilm::GetPixelExtent(int *xstart, int *xend,
                               int *ystart, int *yend) const {
    *xstart = xPixelStart;
    *xend   = xPixelStart + xPixelCount;
    *ystart = yPixelStart;
    *yend   = yPixelStart + yPixelCount;
}
```

Image Output

After the main rendering loop finishes, `SamplerRenderer::Render()` calls the `Film::WriteImage()` method to store the final image in a file.

⟨*ImageFilm Method Definitions*⟩ +≡

```
void ImageFilm::WriteImage(float splatScale) {
    ⟨Convert image to RGB and compute final pixel values 412⟩
    ⟨Write RGB image 413⟩
    ⟨Release temporary image memory 413⟩
}
```

This method starts by allocating an array to store the RGB pixel values.

⟨*Convert image to RGB and compute final pixel values*⟩ ≡ 412

```
int nPix = xPixelCount * yPixelCount;
float *rgb = new float[3*nPix];
int offset = 0;
for (int y = 0; y < yPixelCount; ++y) {
    for (int x = 0; x < xPixelCount; ++x) {
        ⟨Convert pixel XYZ color to RGB 412⟩
        ⟨Normalize pixel with weight sum 413⟩
        ⟨Add splat value at pixel 413⟩
        ++offset;
    }
}
```

Given information about the response characteristics of the display device being used, the pixel values can be converted to device-dependent RGB values from the device-independent XYZ tristimulus values. This conversion is another change of spectral basis, where the new basis is determined by the spectral response curves of the red, green, and blue elements of the display device. Here, weights to convert from XYZ to the device RGB based on the HDTV standard are used. This is a good match for most modern display devices.

⟨*Convert pixel XYZ color to RGB*⟩ ≡ 412

```
XYZToRGB((*pixels)(x, y).Lxyz, &rgb[3*offset]);
```

Film::WriteImage() 404
ImageFilm 404
ImageFilm::pixels 406
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Pixel::Lxyz 406
SamplerRenderer::Render() 27
XYZToRGB() 273

As the RGB output values are being initialized, their final values from the pixel filtering equation are computed by dividing each pixel sample value by `Pixel::weightSum`. This conversion can lead to RGB values where some components are negative; these are *out-of-gamut* colors that can't be represented with the chosen display primaries. Various approaches have been suggested to deal with this issue, ranging from clamping to zero, offsetting all components to lie within the gamut, or even performing a global optimization based on all of the pixels in the image. In addition to out-of-gamut colors, reconstructed pixels may also end up with negative values due to negative lobes in the reconstruction filter function. In order to handle both of these cases, their components are clamped to zero here.

⟨*Normalize pixel with weight sum*⟩ ≡ 412
```
float weightSum = (*pixels)(x, y).weightSum;
if (weightSum != 0.f) {
    float invWt = 1.f / weightSum;
    rgb[3*offset  ] = max(0.f, rgb[3*offset  ] * invWt);
    rgb[3*offset+1] = max(0.f, rgb[3*offset+1] * invWt);
    rgb[3*offset+2] = max(0.f, rgb[3*offset+2] * invWt);
}
```

It's also necessary to add in the splatted values for this pixel to the final value.

⟨*Add splat value at pixel*⟩ ≡ 412
```
float splatRGB[3];
XYZToRGB((*pixels)(x, y).splatXYZ, splatRGB);
rgb[3*offset  ] += splatScale * splatRGB[0];
rgb[3*offset+1] += splatScale * splatRGB[1];
rgb[3*offset+2] += splatScale * splatRGB[2];
```

The `WriteImage()` function, defined in Section A.2 in Appendix A, handles the details of writing the image to a file:

⟨*Write RGB image*⟩ ≡ 412
```
::WriteImage(filename, rgb, NULL, xPixelCount, yPixelCount,
             xResolution, yResolution, xPixelStart, yPixelStart);
```

After saving the image, the working memory is freed.

⟨*Release temporary image memory*⟩ ≡ 412
```
delete[] rgb;
```

Film::xResolution 403
Film::yResolution 403
ImageFilm::filename 405
ImageFilm::pixels 406
ImageFilm::xPixelCount 406
ImageFilm::xPixelStart 406
ImageFilm::yPixelCount 406
ImageFilm::yPixelStart 406
Pixel::splatXYZ 406
Pixel::weightSum 406
WriteImage() 1004
XYZToRGB() 273

FURTHER READING

One of the best books on signal processing, sampling, reconstruction, and the Fourier transform is Bracewell's *The Fourier Transform and Its Applications* (Bracewell 2000). Glassner's *Principles of Digital Image Synthesis* (Glassner 1995) has a series of chapters on the theory and application of uniform and nonuniform sampling and reconstruction to computer graphics. For an extensive survey of the history of and techniques for interpolation of sampled data, including the sampling theorem, see Meijering (2002).

Unser (2000) also surveyed recent developments in sampling and reconstruction theory including the recent move away from focusing purely on band-limited functions.

Crow (1977) first identified aliasing as a major source of artifacts in computer-generated images. Using nonuniform sampling to turn aliasing into noise was introduced by Cook (1986) and Dippé and Wold (1985); their work was based on experiments by Yellot (1983), who investigated the distribution of photoreceptors in the eyes of monkeys. Dippé and Wold also first introduced the pixel filtering equation to graphics and developed a Poisson sample pattern with a minimum distance between samples. Lee, Redner, and Uselton (1985) developed a technique for adaptive sampling based on statistical tests that computed images to a given error tolerance.

Heckbert (1990a) wrote an article that explains possible pitfalls when using floating-point coordinates for pixels and develops the conventions used here.

Mitchell investigated sampling patterns for ray tracing extensively. His 1987 and 1991 SIGGRAPH papers on this topic have many key insights, and the best-candidate approach described in this chapter is based on the latter paper (Mitchell 1987, 1991). The general interface for `Samplers` in `pbrt` is based on an approach he has used in his implementations (Mitchell 1996a). Lagae and Dutré (2008c) surveyed the state of the art in generating Poission disk sample patterns; a number of recent approaches are substantially more efficient than the best-candidate dart-throwing method used in this chapter. See in particular the papers by Jones (2005), Dunbar and Humphreys (2006), and Wei (2008).

Shirley (1991) first introduced the use of discrepancy to evaluate the quality of sample patterns in computer graphics. This work was built upon by Mitchell (1992), Dobkin and Mitchell (1993), and Dobkin, Eppstein, and Mitchell (1996). One important observation in Dobkin et al.'s paper is that the box discrepancy measure used in this chapter and in other work that applies discrepancy to pixel sampling patterns isn't particularly appropriate for measuring a sampling pattern's accuracy at randomly oriented edges through a pixel, and that a discrepancy measure based on random edges should be used instead. This observation explains why many of the theoretically good low-discrepancy patterns do not perform as well as expected when used for image sampling.

Mitchell's first paper on discrepancy introduced the idea of using deterministic low-discrepancy sequences for sampling, removing all randomness in the interest of lower discrepancy (Mitchell 1992). Such *quasi-random* sequences are the basis of quasi–Monte Carlo methods, which will be described in Chapter 14. The seminal book on quasi-random sampling and algorithms for generating low-discrepancy patterns was written by Niederreiter (1992). Using random digit permutations with the Halton sequence was suggested by Ökten and Shah (2008).

Keller and collaborators investigated quasi-random sampling patterns for a variety of applications in graphics (Keller 1996, 1997, 2001). The (0, 2)-sequence sampling techniques used in the `LDSampler` and `BestCandidateSampler` are based on a paper by Kollig and Keller (2002). They are one instance of a general type of low-discrepancy sequence known as (t, s)-sequences and (t, m, s)-nets. These are discussed further by Niederreiter (1992).

BestCandidateSampler 381
LDSampler 373
Sampler 340

Some of Kollig and Keller's techniques are based on algorithms developed by Friedel and Keller (2000). Wong, Luk, and Heng (1997) compared the numeric error of various low-discrepancy sampling schemes. Keller (2001, 2006) argued that because low-discrepancy patterns tend to converge more quickly than others, they are the most efficient sampling approach for generating high-quality imagery.

Chiu, Shirley, and Wang (1994) suggested a *multijittered* 2D sampling technique that combined the properties of stratified and Latin hypercube approaches, although their technique doesn't ensure good distributions across all elementary intervals as (0, 2)-sequences do.

Mitchell (1996b) investigated how much better stratified sampling patterns are than random patterns in practice. In general, the smoother the function being sampled, the more effective they are. For very quickly changing functions (e.g., pixel regions overlapped by complex geometry), sophisticated stratified patterns perform no better than unstratified random patterns. Therefore, for scenes with complex variation in the high-dimensional image function, the advantages of fancy sampling schemes compared to a simple stratified pattern are likely to be minimal.

A unique approach to image sampling was developed by Bolin and Meyer (1995), who implemented a ray tracer that directly synthesized images in the frequency domain. This made it possible to implement interesting adaptive sampling approaches based on perceptual metrics related to the image's frequency content.

Cook (1986) first introduced the Gaussian filter to graphics. Mitchell and Netravali (1988) investigated a family of filters using experiments with human observers to find the most effective ones; the `MitchellFilter` in this chapter is the one they chose as the best. Kajiya and Ullner (1981) investigated image filtering methods that account for the effect of the reconstruction characteristics of Gaussian falloff from pixels in CRTs, and, more recently, Betrisey et al. (2000) described Microsoft's ClearType technology for display of text on LCDs.

There has been quite a bit of research into reconstruction filters for image resampling applications. Although this application is not the same as reconstructing nonuniform samples for image synthesis, much of this experience is applicable. Turkowski (1990b) reported that the Lanczos windowed sinc filter gives the best results of a number of filters for image resampling. Meijering et al. (1999) tested a variety of filters for image resampling by applying a series of transformations to images such that if perfect resampling had been done the final image would be the same as the original. They also found that the Lanczos window performed well (as did a few others) and that truncating the sinc without a window gave some of the worst results. Other work in this area includes papers by Möller et al. (1997) and Machiraju and Yagel (1996).

Kirk and Arvo (1991) identified a subtle problem with adaptive sampling algorithms: in short, if a set of samples is both used to decide if more samples should be taken and is also added to the image, the end result is *biased* and doesn't converge to the correct result in the limit. For this reason, the `Sampler::ReportResults()` method is able to indicate if a set of samples should be discarded and not reported to the `Film`. Mitchell (1987) observed that standard image reconstruction techniques fail in the presence of adaptive sampling:

Film 403
Sampler::ReportResults() 341

the contribution of a dense clump of samples in part of the filter's extent may incorrectly have a large effect on the final value purely due to the number of samples taken in that region. He described a multi-stage box filter that addresses this issue.

Since initial work in adaptive sampling by Lee et al. (1985), Kajiya (1986), and Purgathofer (1987), a number of sophisticated and effective adaptive sampling methods have been developed in recent years. Notable work includes Hachisuka et al. (2008a), who adaptively sampled in the 5D domain of image location, time, and lens position, rather than just in image location, and introduced a novel multidimensional filtering method, Shinya (1993) and Egan et al. (2009), who developed adaptive sampling and reconstruction methods focused on rendering motion blur, and Overbeck et al. (2009), who developed adaptive sampling algorithms based on wavelets for image reconstruction.

Lee and Redner (1990) first suggested using a median filter, where the median of a set of samples is used to find each pixel's value, as a noise reduction technique. Rushmeier and Ward (1994) suggested a nonlinear filter function that works well for reducing the visual impact of noise in images generated with Monte Carlo light transport algorithms. Their observation was that if a single sample is substantially brighter than all of the samples around it, then it is likely that the other nearby samples should also have detected the bright feature that caused the spike of brightness. More recently, Suykens and Willems (2000a) and Xu and Pattanaik (2005) have developed filtering techniques to reduce noise in images rendered using Monte Carlo algorithms.

Jensen and Christensen (1995) observed that it can be more effective to separate out the contributions to pixel values based on the type of illumination they represent; low-frequency indirect illumination can be filtered differently than high-frequency direct illumination, thus reducing noise in the final image. They developed an effective filtering technique based on this observation. An improvement to this approach was developed by Keller and collaborators with the *discontinuity buffer* (Keller 1998; Wald et al. 2002). In addition to filtering slowly varying quantities like indirect illumination separately from more quickly varying quantities like surface reflectance, the discontinuity buffer uses geometric quantities like the surface normal at nearby pixels to determine whether their corresponding values can be reasonably included at the current pixel. Kontkanen et al. (2004) built on these approaches to build a filtering approach for indirect illumination when using the irradiance caching algorithm (Section 15.5).

A number of different approaches have been developed for mapping out-of-gamut colors to the displayable range; see Rougeron and Péroche's survey article for discussion of this issue and references to various approaches (Rougeron and Péroche 1998). This topic was also covered by Hall (1989).

While most image file formats store 8 bits for each red, green, and blue channel (and possibly alpha), a number of new formats are also able to store floating-point data. Ward (1991b) suggested an efficient technique for compactly storing floating-point image data from realistic image synthesis and also developed an extension to the TIFF image file format for accurate and compact, high dynamic range color representation (Larson 1998). The OpenEXR format was developed by Kainz and Bogart at Industrial Light and Magic; one of its innovations is a 16-bit floating-point format (Kainz, Bogart, and Hess 2002). Peercy et al. (2000) were the first to use such a format for image representation.

The images that a renderer generates don't necessarily need to store color values at each pixel. For example, Séquin and Smyrl (1989) described a system that stored the tree of all rays traced at each pixel; this made it possible to modify light and material colors in the scene and generate a new image of the scene without needing to retrace any of the rays. Both Perlin (1985a) and Saito and Takahashi (1990) have shown a number of uses for images that stored 3D position, surface normal, and other geometric and material properties of the first visible object in each pixel, including fast reshading of the scene and illustration-inspired rendering styles.

Tone reproduction—algorithms for displaying high dynamic range images on low dynamic range display devices—became an active area of research starting with the work of Tumblin and Rushmeier (1993). Chiu et al. (1993) and Ward (1994a) soon followed with new approaches. The survey article of Devlin et al. (2002) summarizes most of the work in this area through 2002, giving pointers to the original papers. See also Reinhard et al.'s book on high dynamic range imaging, which includes significant coverage of this topic (Reinhard et al. 2005). For background information on properties of the human visual system, Wandell's book on vision is an excellent starting point (Wandell 1995). Ferwerda (2001) presented an overview of the human visual system for applications in graphics, and Malacara (2002) gave a concise overview of color theory and basic properties of how the human visual system processes color.

An interesting application of knowledge of the characteristics of the human visual system is perceptually driven rendering, where less work is done in areas of the image that are known to be visually unimportant and more work is done in areas that are known to be important. Both Bolin and Meyer (1998) and Ramasubramanian, Pattanaik, and Greenberg (1999) wrote renderers that performed adaptive sampling based on perceptual models. Walter, Pattanaik, and Greenberg (2002) developed a system that detected and took advantage of masking of noise artifacts due to textures to sample at lower rates in areas of the image where doing so wouldn't be noticeable.

EXERCISES

3 7.1 A substantially different software design for sample generation for integration is described in Keller's technical report (Keller 2001). What are the advantages and disadvantages of that approach compared to the one in `pbrt`? Modify `pbrt` to support sample generation along the lines of the approach described there and compare flexibility, performance, and image quality to the current implementation.

2 7.2 Although the `ImageFilm` always allocates storage for both `Pixel::Lxyz` and `Pixel::splatXYZ` values, the `SamplerRenderer` doesn't ever call `Film::Splat()` and thus the storage for splats is unused but is still loaded into the processor cache each time a `Pixel` is accessed. Modify the `ImageFilm` representation to separately allocate separate storage for the `splatXYZ` data, thus reducing the storage needed for pixels in the nonsplatting case to 4 `floats`, or 16 bytes. (You may still want to pad out the structure for `splatXYZ` values to 16 bytes, so that they never straddle cache lines.)

`Film::Splat()` 403
`ImageFilm` 404
`Pixel` 406
`Pixel::Lxyz` 406
`Pixel::splatXYZ` 406
`SamplerRenderer` 25

Although this separation won't save memory, it should marginally improve performance: twice as many `Pixels` can fit into a cache line, thus reducing cache misses. Measure the change in performance from this change. What is the difference for a scene with no geometry or lights (where the overhead of film update is a relatively larger part of overall runtime)? What about a more complex scene?

2 **7.3** As mentioned in Section 7.8.2, the `ImageFilm::Splat()` method doesn't use the current pixel filter but instead just splats the sample to the single pixel it's closest to, effectively using a box filter. In order to apply an arbitrary filter, the filter must be normalized so that it integrates to one over its domain; this constraint isn't currently required by `pbrt`. Modify the computation of `filterTable` in the `ImageFilm` constructor so that the tabularized function is normalized. (Don't forget that the table only stores one quarter of the function's extent when computing the normalization factor.) Then, modify the implementation of the `Splat()` method to use the current filter. Investigate the time and image-quality differences that result.

2 **7.4** An alternative design for parallelizing `pbrt` would be to have each rendering task have its own private memory for image storage. The tasks could could directly update this memory without using atomic instructions or a mutex. (Note that each task would only need storage in a small region of the image, just large enough to cover the pixel extent it is generating samples for as well as additional pixels at the boundary due to the filter function's extent.) As tasks finish, they could then acquire a mutex and merge their results in with the global image. Implement this approach in `pbrt` and determine if it noticeably affects system performance. You may want to use relatively simple scenes for testing, such that the portion of runtime spent in sample filtering and image update is relatively large with respect to overall runtime.

One additional advantage of this approach is that the tasks' images can potentially be merged in order, leading to more consistent results across multiple runs of the system. With `pbrt`'s current implementation, the pixel values in an image may change by small amounts with rerendering, due to tasks computing samples in different orders with respect to each other. A pixel that had a final value that came from samples from three different tasks, $v_1 + v_2 + v_3$, may sometimes be computed as $(v_1 + v_2) + v_3$ and sometimes as $(v_3 + v_1) + v_2$, for example. Due to floating-point round-off, these two values will in general be different. While these differences aren't normally a problem, they wreak havoc with automated testing scripts that might want to verify that a believed-to-be-innocuous change to the system didn't cause any differences in rendered images.

1 **7.5** Modify `pbrt` to create images where each pixel encodes the amount of time spent computing the sample values inside that pixel's filter extent. (A one-pixel-wide box filter is probably the most useful filter for this exercise.) Render images of a variety of scenes with this technique. What insight about the system's performance do the resulting images bring?

ImageFilm 404
ImageFilm::Splat() 411
Pixel 406

● 7.6 One of the advantages of the linearity assumption in radiometry is that the final image of a scene is the same as the sum of individual images that account for each light source's contribution (assuming a floating-point image file format is used that doesn't clip pixel radiance values). An implication of this property is that if a renderer creates a separate image for each light source, it is possible to write interactive lighting design tools that make it possible to quickly see the effects of scaling the contributions of individual lights in the scene without needing to rerender it from scratch. Instead, a light's individual image can be scaled and the final image regenerated by summing all of the light images again. (This technique was first applied for opera lighting design by Dorsey, Sillion, and Greenberg (1991).) Modify `pbrt` to output a separate image for each of the lights in the scene and write an interactive lighting design tool that uses them in this manner.

● 7.7 Mitchell and Netravali (1988) noted that there is a family of reconstruction filters that use both the value of a function and its derivative at the point to do substantially better reconstruction than if just the value of the function is known. Furthermore, they report that they have derived closed-form expressions for the screen space derivatives of Lambertian and Phong reflection models, although they do not include these expressions in their paper. Investigate derivative-based reconstruction and extend `pbrt` to support this technique. Because it will likely be difficult to derive expressions for the screen space derivatives for general shapes and BSDF models, investigate approximations based on finite differencing. Techniques built on the ideas behind the ray differentials of Section 10.1 may be fruitful for this effort.

● 7.8 Image-based rendering is the general name for a set of techniques that use one or more images of a scene to synthesize new images from viewpoints different than the original ones. One such approach is lightfield rendering, where a set of images from a densely spaced set of positions is used (Levoy and Hanrahan 1996; Gortler et al. 1996). Read these two papers on lightfields and modify `pbrt` to directly generate lightfields of scenes, without requiring that the renderer be run multiple times, once for each camera position. It will probably be necessary to write a specialized `Camera`, `Sampler`, and `Film` to do this. Also, write an interactive lightfield viewer that loads lightfields generated by your implementation and generates new views of the scene.

● 7.9 Rather than just storing spectral values in an image, it's often useful to store additional information about the objects in the scene that were visible at each pixel. See, for example, the SIGGRAPH papers by Perlin (1985a) and Saito and Takahashi (1990). For example, if the 3D position, surface normal, and BRDF of the object at each pixel are stored, then the scene can be efficiently rerendered after moving the light sources (Gershbein and Hanrahan 2000). Alternatively, if each sample stores information about all of the objects visible along its camera ray, rather than just the first one, new images from shifted viewpoints can be rerendered (Shade et al. (1998)). Investigate representations for deep frame

buffers and algorithms that use them; extend `pbrt` to support the creation of images like these and develop tools that operate on them.

❷ 7.10 Implement a median filter for image reconstruction: for each pixel, store the median of all of the samples within a filter extent around it. This task is complicated by the fact that filters in `ImageFilm` must be *linear*—the value of the filter function is determined solely by the position of the sample with respect to the pixel position, and the value of the sample has no impact on the value of the filter function. Because the implementation assumes that filters are linear, and because it doesn't store sample values after adding their contribution to the image, implementing the median filter will require generalizing the `ImageFilm` or developing a new `Film` implementation.

Render images using integrators like the `PathIntegrator` that have objectionable image noise with regular image filters. How successful is the median filter at reducing noise? Are there visual shortcomings to using the median filter? Can you implement this approach without needing to store all of the image sample values before computing final pixel values?

❷ 7.11 An alternative to the median filter is to discard the sample with the lowest contribution and the sample with the largest contribution in a pixel's filter region. This approach uses more of the information gathered during sampling. Implement this approach and compare the results to the median filter.

❸ 7.12 Implement the discontinuity buffer, as described by Keller and collaborators (Keller 1998; Wald et al. 2002). You will probably need to modify the `Film` interface so that geometric information about the intersection point is passed to the `Film::AddSample()` method and modify the interface to the `SurfaceIntegrators` so that they can separately return direct and indirect illumination contributions. Render images showing its effectiveness when rendering images with indirect illumination.

❸ 7.13 Implement one of the recent adaptive sampling and reconstruction techniques such as the ones described by Hachisuka et al. (2008), Egan et al. (2009), or Overbeck et al. (2009). How much more efficiently do they generate images of equal quality than just uniformly sampling at a high rate? How much more effective are they than the `AdaptiveSampler`? How do they affect running time for simple scenes where adaptive sampling isn't needed?

AdaptiveSampler 386
Film 403
Film::AddSample() 403
ImageFilm 404
PathIntegrator 766
SurfaceIntegrator 740