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 Samplers in this chapter select sample points on the image plane at which incident radiance will be computed (recall that in the previous chapter, Cameras used Samples generated by a Sampler to construct their camera rays). Three Sampler implementations are described in this chapter, spanning a variety of approaches to the sampling problem. This chapter concludes with the Filter class. The Filter is used to determine how multiple samples near each pixel are blended together to compute the final pixel value.

Camera 256
Filter 353
Sample 299
Sampler 296

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

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.

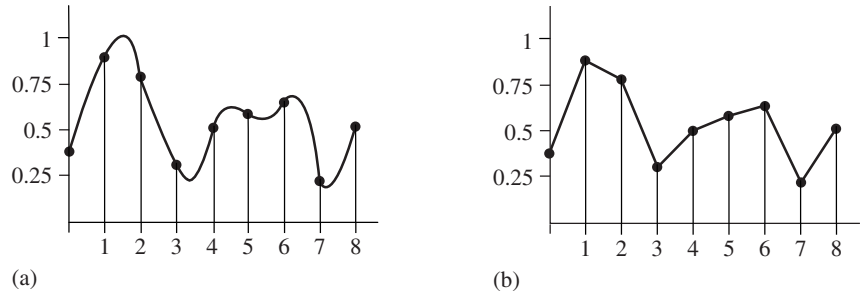


Figure 7.1: (a) By taking a set of *point samples* of $f(x)$ (indicated by black dots), we determine the value of the function at those positions. (b) The sample values can be used to *reconstruct* a function $\tilde{f}(x)$ that is an approximation to $f(x)$. The sampling theorem, introduced in Section 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.

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 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.

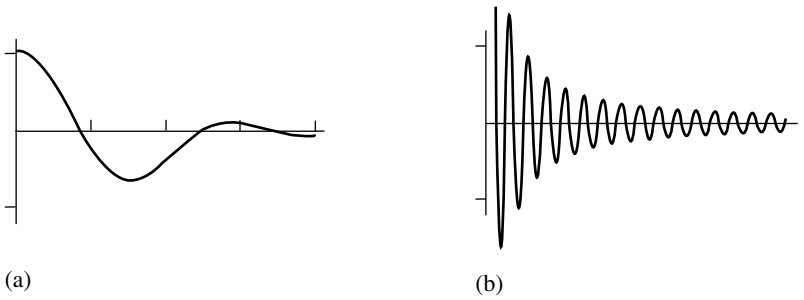


Figure 7.2: (a) Low-frequency function, and (b) high-frequency function. Roughly speaking, the higher frequency a function is, the more quickly it varies over a given region.

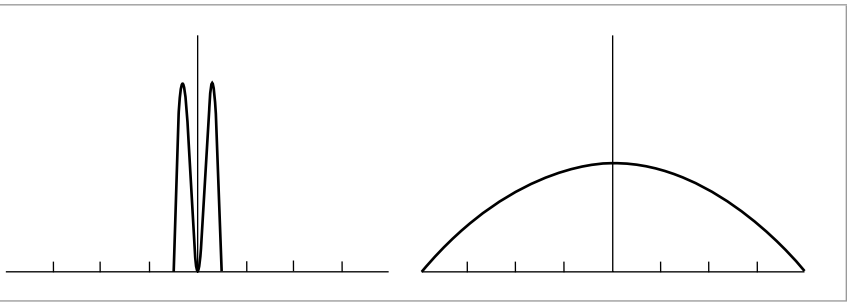


Figure 7.3: Frequency Space Representations of the Functions in Figure 7.2. The graphs show the contribution of each frequency ω to each of the functions in the spatial domain.

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.

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: $f(\omega) = \text{III}_{1/T}(\omega) = (1/T) \sum_i \delta(\omega - i/T)$

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

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

(Recall that $e^{ix} = \cos x + i \sin x$, 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*:

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

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

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

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.

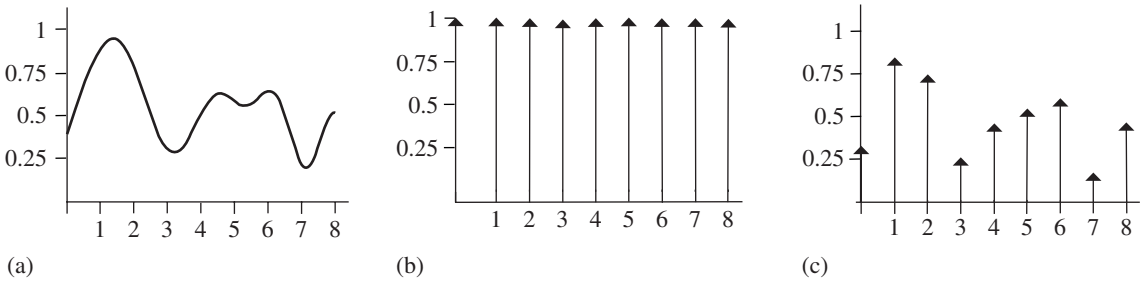


Figure 7.4: Formalizing the Sampling Process. (a) The function $f(x)$ is multiplied by (b) the shah function $\mathrm{III}_T(x)$, giving (c) an infinite sequence of scaled delta functions that represent its value at each sample point.

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 $\mathrm{III}_T(x)$ is defined as

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

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:

$$\mathrm{III}_T(x) f(x) = T \sum_{i} \delta(x - iT) f(iT).$$

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*

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

where the convolution operation $\otimes$ is defined as

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

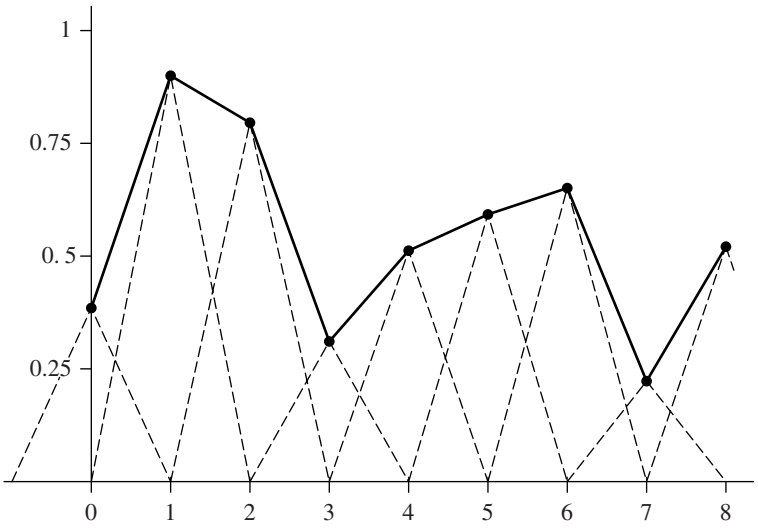


Figure 7.5: The sum of instances of the triangle reconstruction filter, shown with dashed lines, gives the reconstructed approximation to the original function, shown with a solid line.

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:

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

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 $\text{Ш}_T(x)$ 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

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

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

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

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

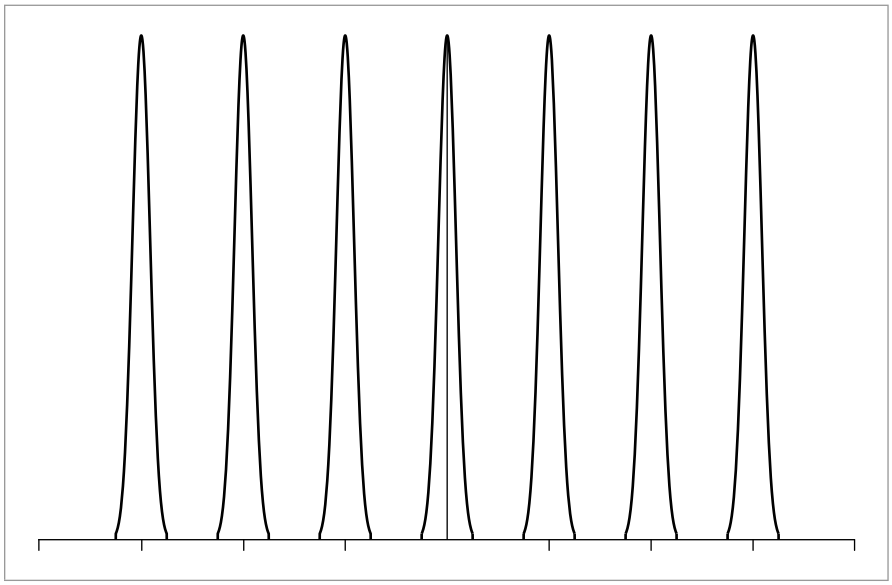


Figure 7.6: The Convolution of $F(\omega)$ and the Shah Function. The result is infinitely many copies of F.

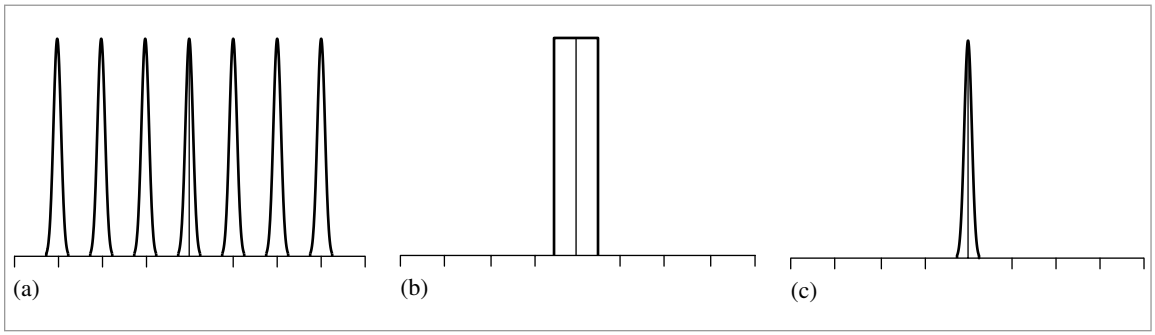


Figure 7.7: Multiplying (a) a series of copies of $F(\omega)$ by (b) the appropriate box function yields (c) the original spectrum.

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

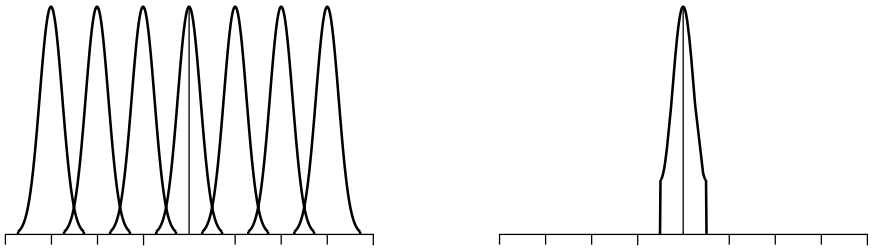


Figure 7.8: (a) When the sampling rate is too low, the copies of the function's spectrum overlap, (b) resulting in aliasing when reconstruction is performed.

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 minimize 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

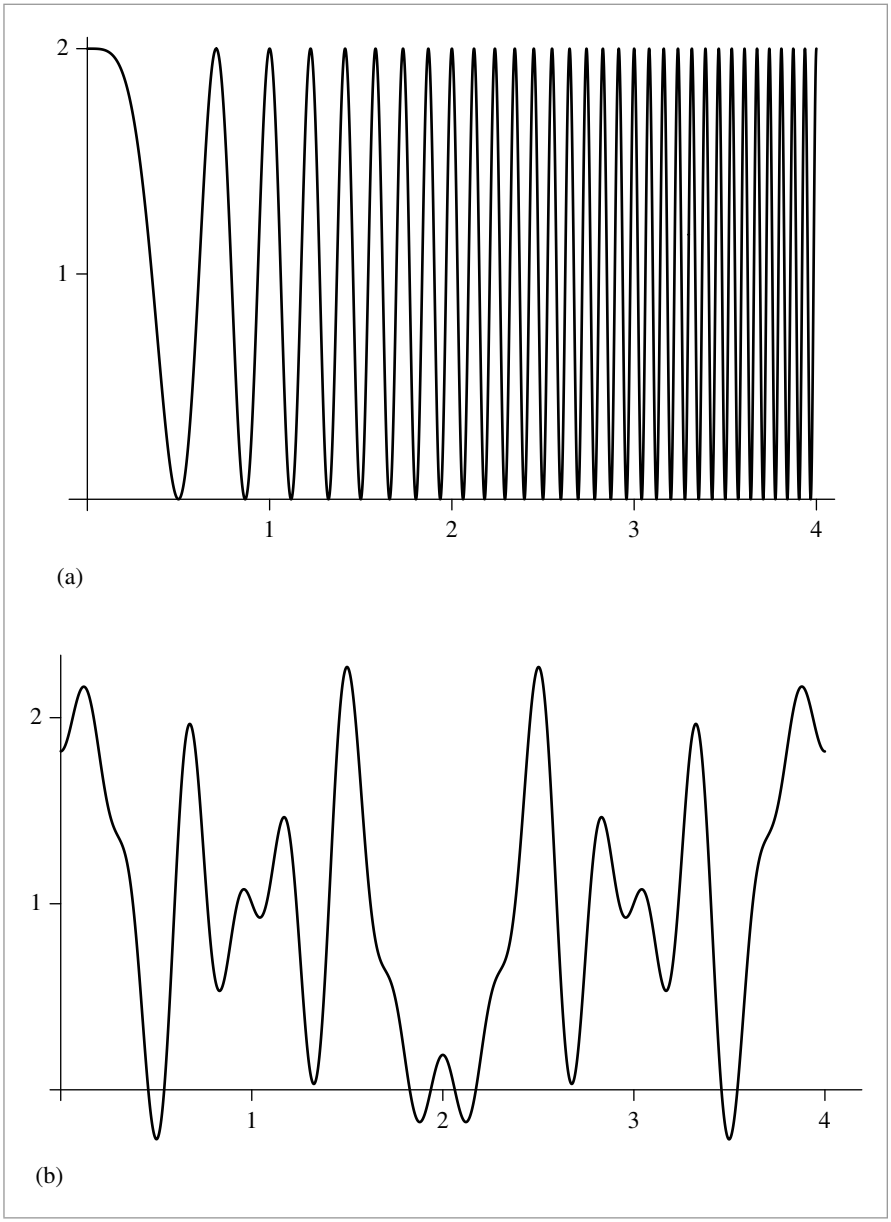


Figure 7.9: Aliasing from Point Sampling the Function $1 + \cos(4x^2)$. (a) The function. (b) The reconstructed function from sampling it with samples spaced 0.125 units apart and performing perfect reconstruction with the sinc filter. Aliasing causes the high-frequency information in the original function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

errors is broadly classified as *antialiasing*. This section reviews a number of antialiasing techniques beyond just increasing the sampling rate everywhere.

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. Unfortunately, it is hard 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 function 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 11 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.

Some areas that need supersampling will always be missed by adaptive approaches, leaving an increase in the basic sampling rate as the only recourse. Adaptive antialiasing works well at turning a very aliased image into a less aliased image, but it is usually not able to make a visually flawless image much more efficiently than increasing the sampling rate everywhere, particularly for complex scenes.

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 11. 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,

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

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 7.6. 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.

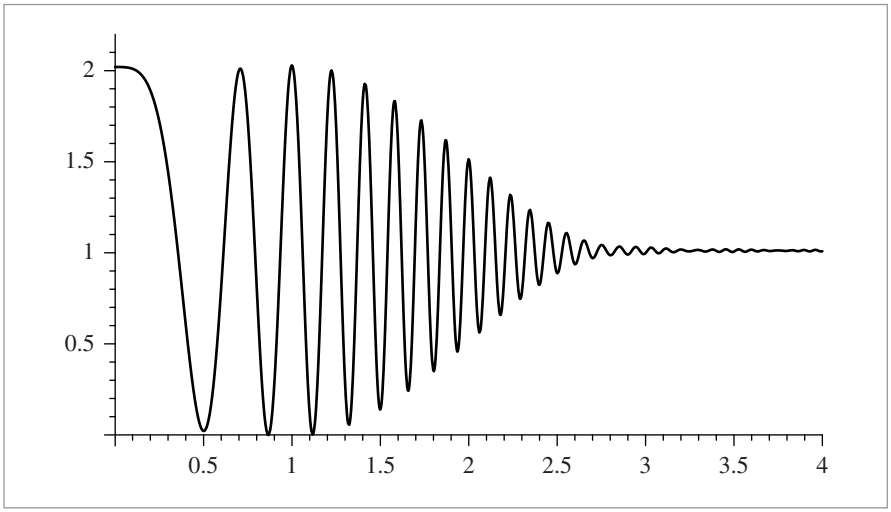


Figure 7.10: Graph of the function $1 + \cos(4x^2)$ convolved with a filter that removes frequencies beyond the Nyquist limit for a sampling rate of $T = .125$. High-frequency detail has been removed from the function, so that the new function can at least be sampled and reconstructed without aliasing.

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 16 and 17 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

Sampler 296

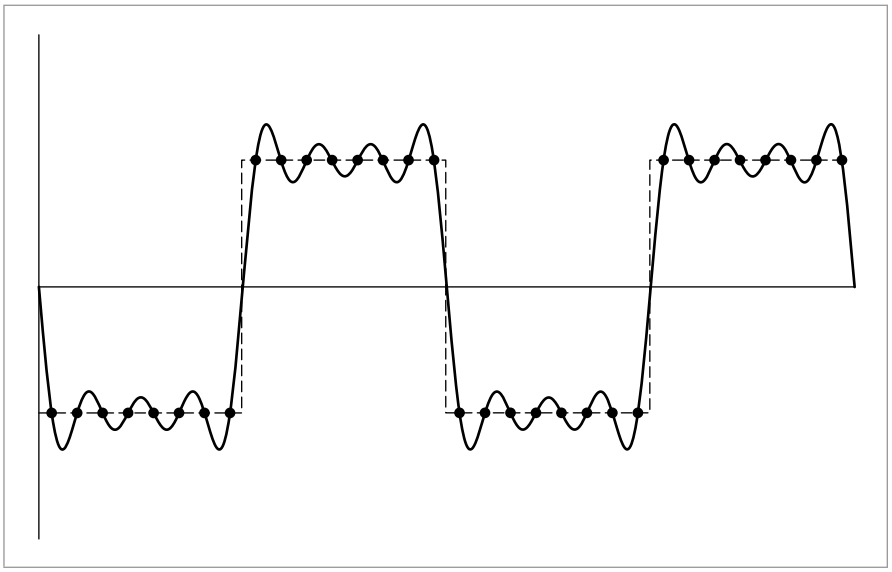


Figure 7.11: Illustration of the Gibbs Phenomenon. When a function hasn't been sampled at the Nyquist rate and the set of aliased samples is reconstructed with the sinc filter, the reconstructed function will have "ringing" artifacts, where it oscillates around the true function. Here a 1D step function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

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 11), 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.

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 $[x_0 - .5, x_1 + .5)$. Thus, any code that generates continuous sample positions for a given discrete pixel range is littered with 0.5 offsets. It is easy to forget some of these, leading to subtle errors.

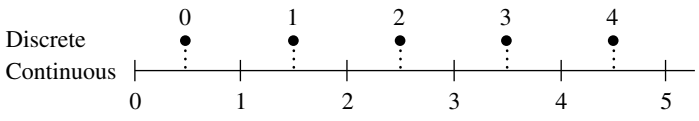
If we instead truncate continuous coordinates c to discrete coordinates d by

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

and convert from discrete to continuous by

$$c = d + .5,$$

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.



Sampler 296

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 $d + 0.5$.

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/sampling.h` and `core/sampling.cpp`. Each of the sample generation plug-ins is in its own source file in the `samplers/` directory.

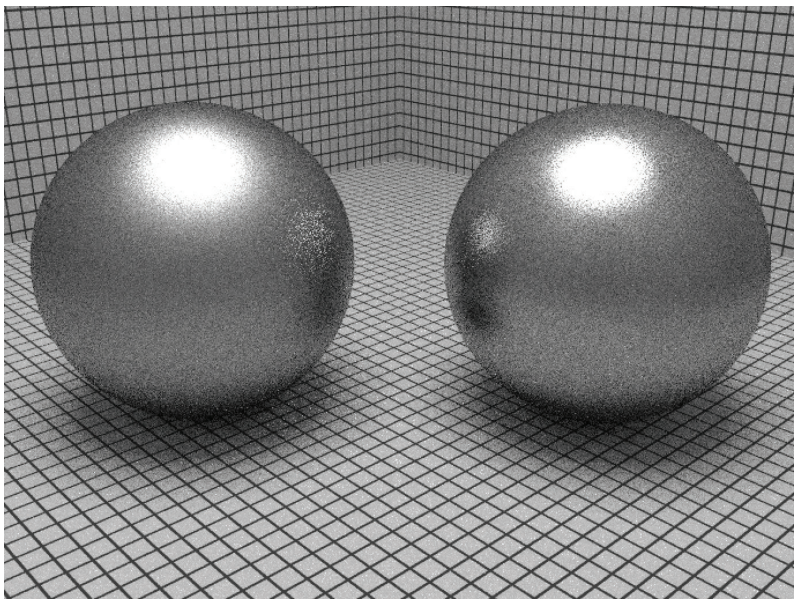
All of the sampler implementations inherit from an abstract `Sampler` class that defines their interface. The task of `Sampler`s is to generate a sequence of multidimensional sample positions. Two dimensions give the raster space image sample position, and another gives the time at which the sample should 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 Chapter 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.

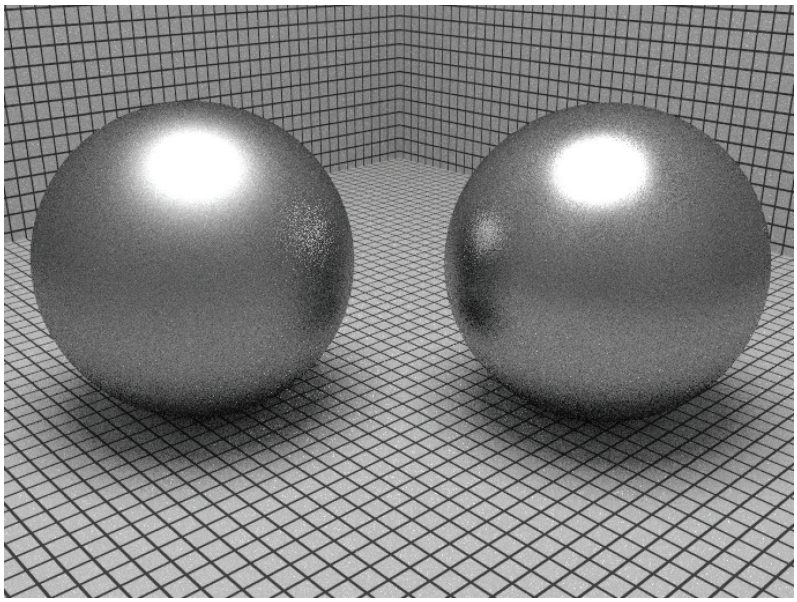
⟨*Sampling Declarations*⟩ ≡
```
class COREDLL Sampler {
public:
    ⟨Sampler Interface 298⟩
    ⟨Sampler Public Data 298⟩
};
```

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 and the number of samples the implementation expects to generate for each pixel in the final image. These values are stored in member variables for later use.

COREDLL 904
Sampler 296



(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.

⟨*Sampler Method Definitions*⟩ ≡
```
Sampler::Sampler(int xstart, int xend, int ystart, int yend, int spp) {
    xPixelStart = xstart;
    xPixelEnd = xend;
    yPixelStart = ystart;
    yPixelEnd = yend;
    samplesPerPixel = spp;
}
```

The `Sampler` implementation should generate samples for pixels with x coordinates ranging from `xPixelStart` to `xPixelEnd-1`, inclusive, and analogously for y coordinates.

⟨*Sampler Public Data*⟩ ≡ 296
```
int xPixelStart, xPixelEnd, yPixelStart, yPixelEnd;
int samplesPerPixel;
```

`Samplers` must implement the `Sampler::GetNextSample()` method, which is a pure virtual function. The `Scene::Render()` method calls this function until it returns `false`; each time it returns `true`, it should fill in the `sample` that is passed in with values that specify the next sample to be taken. All of the dimensions of the sample values it generates have values in the range [0, 1], except for `imageX` and `imageY`, which are specified with respect to the image size in raster coordinates.

⟨*Sampler Interface*⟩ ≡ 296
```
virtual bool GetNextSample(Sample *sample) = 0;
```

In order to make it easy for the main rendering loop to figure out what percentage of the scene has been rendered based on the number of samples processed, the `Sampler::TotalSamples()` method returns the total number of samples that the `Sampler` is expected to return.[4]

⟨*Sampler Interface*⟩ +≡ 296
```
int TotalSamples() const {
    return samplesPerPixel *
           (xPixelEnd - xPixelStart) *
           (yPixelEnd - yPixelStart);
}
```

Sample 299
Sampler 296
Sampler::GetNextSample() 298
Sampler::samplesPerPixel 298
Sampler::TotalSamples() 298
Sampler::xPixelEnd 298
Sampler::xPixelStart 298
Sampler::yPixelEnd 298
Sampler::yPixelStart 298
Scene::Render() 24

7.2.1 SAMPLE REPRESENTATION AND ALLOCATION

The `Sample` structure is used by `Samplers` to store a single sample. A single `Sample` is allocated in the `Scene::Render()` method. For each camera ray to be generated, the

4 The low-discrepancy and best-candidate samplers, described later in the chapter, may actually return a few more or less samples than `TotalSamples()` reports. However, since computing the actual number can't be done quickly, and since an exact number is not really required, the expected number is returned here instead.

Sample's pointer is passed to the Sampler to have its values initialized. It is then passed to the camera and integrators, which read values from it to construct the camera ray and perform lighting calculations.

Depending on the details of the light transport algorithm being used, different integrators may have different sampling needs. For example, the WhittedIntegrator doesn't do any random sampling, so it doesn't need any additional sample values, but the DirectLighting 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 will be 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 {
    ⟨Sample Public Methods 300⟩
    ⟨Camera Sample Data 299⟩
    ⟨Integrator Sample Data 301⟩
};
```

The data in the Sampler for use by the camera is fixed. We've already seen its use by the camera implementations in Chapter 6.

⟨*Camera* Sample *Data*⟩≡ 299
```
float imageX, imageY;
float lensU, lensV;
float time;
```

The Sample constructor immediately 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.

DirectLighting 723
Integrator::
RequestSamples() 723
Sample 299
Sampler 296
WhittedIntegrator 30

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 is not as good as generating a 3D sample pattern directly, it is usually acceptable in practice. If absolutely necessary, the integrator can always generate a 3D sample pattern itself.

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.

⟨*Sample Method Definitions*⟩ ≡
```
Sample::Sample(SurfaceIntegrator *surf,
               VolumeIntegrator *vol, const Scene *scene) {
    surf->RequestSamples(this, scene);
    vol->RequestSamples(this, scene);
    ⟨Allocate storage for sample pointers 301⟩
    ⟨Compute total number of sample values needed 302⟩
    ⟨Allocate storage for sample values 302⟩
}
```

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*⟩ ≡ 299
```
u_int Add1D(u_int num) {
    n1D.push_back(num);
    return n1D.size()-1;
}
```

⟨*Sample Public Methods*⟩+ ≡ 299
```
u_int Add2D(u_int 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

Integrator::
RequestSamples() 723
LDSampler 328
Sample 299
Sample::Add1D() 300
Sample::Add2D() 300
Sample::n1D 301
Sample::n2D 301
Sampler 296
Sampler::RoundSize() 301
Scene 23
SurfaceIntegrator 723
VolumeIntegrator 806

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*⟩+≡ 296
```
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 ith pattern has n1D[i] sample values. These values are stored in oneD[i][0] through oneD[i][n1D[i]-1].

⟨*Integrator* Sample *Data*⟩≡ 299
```
vector<u_int> 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

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

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

The Sample constructor first allocates memory to store the pointers. Rather than allocating memory twice, it does a single allocation that gives enough memory 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*⟩≡ 300
```
int nPtrs = n1D.size() + n2D.size();
if (!nPtrs) {
    oneD = twoD = NULL;
    return;
}
oneD = (float **)AllocAligned(nPtrs * sizeof(float *));
twoD = oneD + n1D.size();
```

AllocAligned() 842
Sample::n1D 301
Sample::n2D 301
Sample::oneD 301
Sample::twoD 301
Sampler 296

We then use the same trick to allocate memory for the actual sample values so that they are contiguous as well. First, we find the total number of float values needed:

⟨*Compute total number of sample values needed*⟩ ≡ 300

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

The constructor 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*⟩ ≡ 300

```
float *mem = (float *)AllocAligned(totSamples * sizeof(float));
for (u_int i = 0; i < n1D.size(); ++i) {
    oneD[i] = mem;
    mem += n1D[i];
}
for (u_int 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.

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.

Sample 299
Sample::n1D 301
Sample::n2D 301
Sample::oneD 301
Sample::twoD 301
StratifiedSampler 304

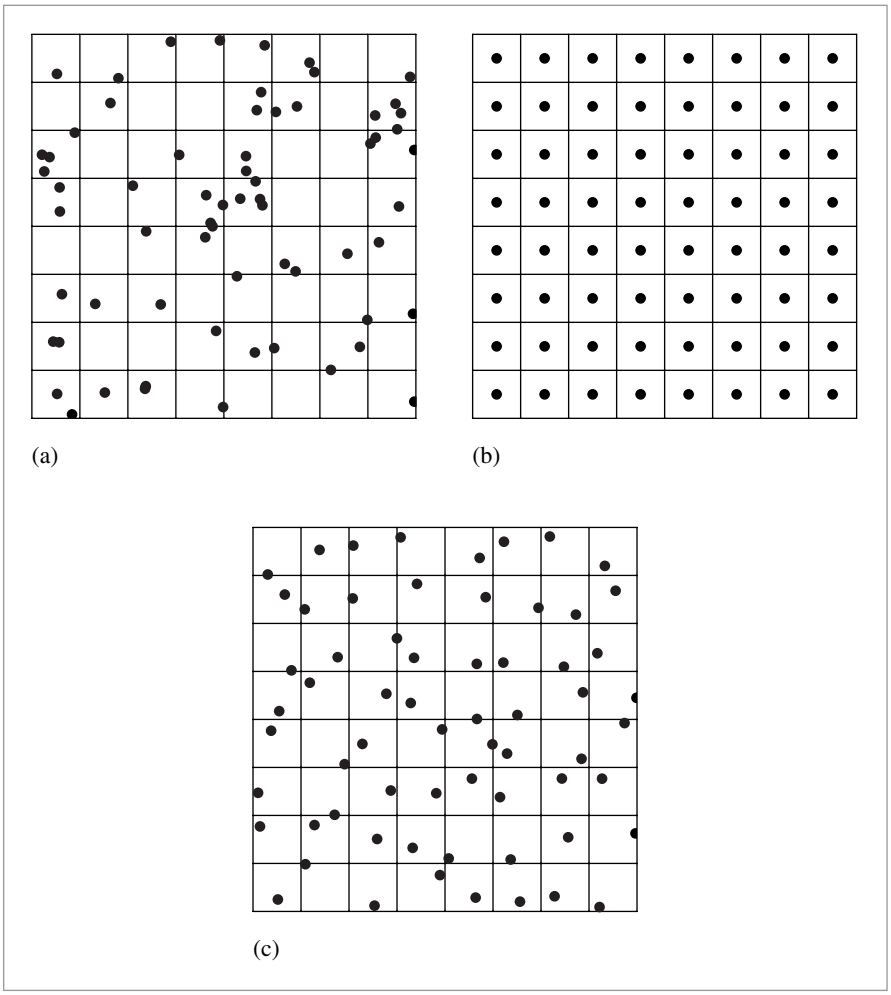


Figure 7.14: Three 2D Sampling Patterns. (a) The random pattern is an ineffective pattern, with many clumps of samples that leave large sections of the image poorly sampled. (b) A uniform stratified pattern is better distributed but can exacerbate aliasing artifacts. (c) A stratified jittered pattern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

Figure 7.14 shows a comparison of a few sampling patterns. The first is a completely random sampling pattern: we have chosen a number of samples and generated that many 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 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 Declarations*⟩ ≡
```
class StratifiedSampler : public Sampler {
public:
    ⟨StratifiedSampler Public Methods 306⟩
private:
    ⟨StratifiedSampler Private Data 304⟩
};
```

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`) and a boolean (`jitter`) that indicates whether the samples should be jittered.

⟨*StratifiedSampler Method Definitions*⟩ ≡
```
StratifiedSampler::StratifiedSampler(int xstart, int xend,
        int ystart, int yend, int xs, int ys, bool jitter)
    : Sampler(xstart, xend, ystart, yend, xs * ys) {
    jitterSamples = jitter;
    xPos = xPixelStart;
    yPos = yPixelStart;
    xPixelSamples = xs;
    yPixelSamples = ys;
    ⟨Allocate storage for a pixel's worth of stratified samples 306⟩
    ⟨Generate stratified camera samples for (xPos,yPos) 308⟩
}
```

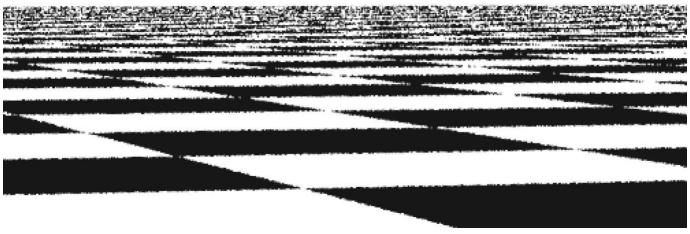
The sampler holds the coordinate of the current pixel in the `xPos` and `yPos` member variables, which are initialized to point to the pixel in the upper left of the image. Note that both the crop window and the sample filtering process can cause this corner to be at a location other than (0, 0).

⟨*StratifiedSampler Private Data*⟩ ≡ 304
```
int xPixelSamples, yPixelSamples;
bool jitterSamples;
int xPos, yPos;
```

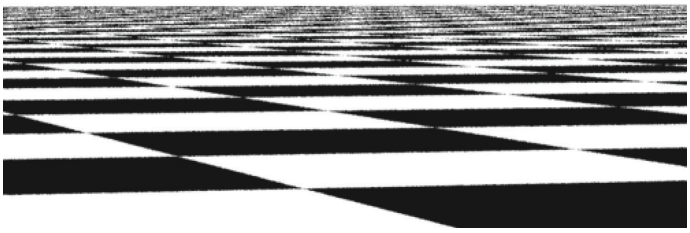
Sampler 296
Sampler::xPixelStart 298
Sampler::yPixelStart 298
StratifiedSampler 304
StratifiedSampler::jitterSamples 304
StratifiedSampler::xPixelSamples 304
StratifiedSampler::xPos 304
StratifiedSampler::yPixelSamples 304
StratifiedSampler::yPos 304

(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.

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

⟨*StratifiedSampler Public Methods*⟩ ≡ 304
```
int RoundSize(int size) const {
    return size;
}
```

The StratifiedSampler computes image, time, and lens samples for an entire pixel's worth of image samples all at once; doing so allows it to compute better-distributed patterns for the time and lens samples than if it were to compute each sample's values independently. Since the GetNextSample() method only supplies one sample at a time, here the constructor allocates enough memory to store all of the sample values for all of the samples in a single pixel.

⟨*Allocate storage for a pixel's worth of stratified samples*⟩ ≡ 304
```
imageSamples = (float *)AllocAligned(5 * xPixelSamples *
                                     yPixelSamples * sizeof(float));
lensSamples = imageSamples + 2 * xPixelSamples * yPixelSamples;
timeSamples = lensSamples + 2 * xPixelSamples * yPixelSamples;
```

⟨*StratifiedSampler Private Data*⟩ +≡ 304
```
float *imageSamples, *lensSamples, *timeSamples;
```

Naive 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. 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.

Ensuring a good distribution at the pixel level is a reasonable level of granularity: we would like the samples that are close together on the image plane to have dissimilar time and lens samples so that the sample positions are not clumped together in the

AllocAligned() 842
imageSamples 335
Integrator 722
lensSamples 335
StratifiedSampler 304
StratifiedSampler:: imageSamples 306
StratifiedSampler:: lensSamples 306
StratifiedSampler:: timeSamples 306
StratifiedSampler:: xPixelSamples 304
StratifiedSampler:: yPixelSamples 304

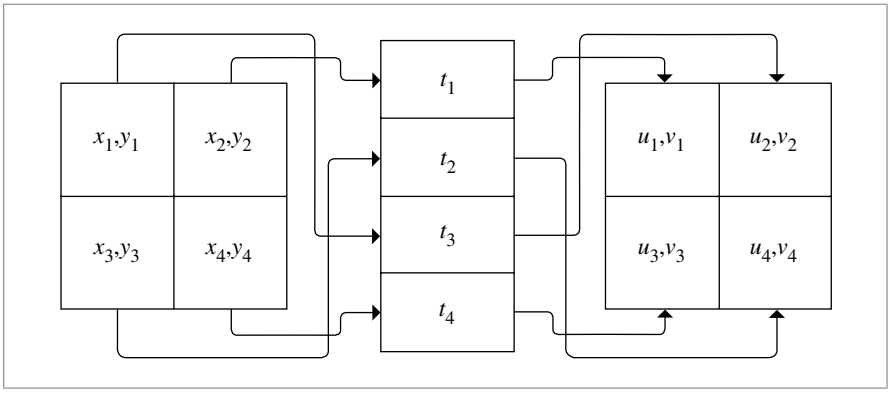


Figure 7.16: We can generate a good sample pattern that reaps the benefits of stratification without requiring that all of the sampling dimensions be stratified simultaneously. Here, we have split (x, y) image position, time t, and (u, v) lens position into independent strata with four regions each. 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.

high-dimensional sampling space. When the samples are widely distributed, the resulting images are generally better, since the samples form a more accurate representation of the image function across the domain. However, it is much less important that samples that are far away on the image plane have uncorrelated time and lens samples.[5].

BestCandidateSampler 344
StratifiedSampler 304

5 Of course, with the approach implemented here, the time and lens sample positions of the image samples for any particular output pixel are well distributed, but the time and lens sample positions of image samples for neighboring pixels are not known, and thus may be similar. The `BestCandidateSampler` can be less susceptible to this problem.

We use this technique to generate the camera samples for the current pixel in the StratifiedSampler. As GetNextSample() generates new samples, the samplePos variable tracks the current position in the image, time, and lens sample arrays.

⟨*Generate stratified camera samples for (*xPos,yPos*)*⟩ ≡ 304, 310
```
StratifiedSample2D(imageSamples, xPixelSamples, yPixelSamples,
                   jitterSamples);
StratifiedSample2D(lensSamples, xPixelSamples, yPixelSamples,
                   jitterSamples);
StratifiedSample1D(timeSamples, xPixelSamples*yPixelSamples,
                   jitterSamples);
⟨Shift stratified image samples to pixel coordinates 309⟩
⟨Decorrelate sample dimensions 309⟩
samplePos = 0;
```

⟨*StratifiedSampler Private Data*⟩+ ≡ 304
```
int samplePos;
```

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

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

⟨*Sampling Function Definitions*⟩+ ≡
```
COREDLL void StratifiedSample2D(float *samp, int nx, int ny,
                                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 ? RandomFloat() : 0.5f;
            float jy = jitter ? RandomFloat() : 0.5f;
            *samp++ = (x + jx) * dx;
            *samp++ = (y + jy) * dy;
        }
}
```

COREDLL 904
RandomFloat() 857
StratifiedSampler 304
StratifiedSampler::
  imageSamples 306
StratifiedSampler::
  jitterSamples 304
StratifiedSampler::
  lensSamples 306
StratifiedSampler::
  samplePos 308
StratifiedSampler::
  timeSamples 306
StratifiedSampler::
  xPixelSamples 304
StratifiedSampler::
  yPixelSamples 304

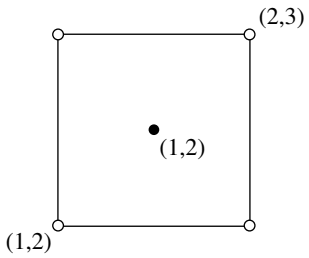


Figure 7.18: To generate stratified samples for a discrete pixel $(1, 2)$, all that needs to be done is to add $(1, 2)$ to the elements of the samples generated over $[0, 1]^2$. This gives samples with continuous pixel coordinates from $(1, 2)$ to $(2, 3)$, which end up surrounding the discrete pixel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

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, this method loops over all of the new stratified samples and adds the (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*⟩ ≡ 308

```
for (int o = 0; o < 2 * xPixelSamples * yPixelSamples; o += 2) {
    imageSamples[o]   += xPos;
    imageSamples[o+1] += yPos;
}
```

Shuffle() 310
StratifiedSample2D() 308
StratifiedSampler::imageSamples 306
StratifiedSampler::lensSamples 306
StratifiedSampler::timeSamples 306
StratifiedSampler::xPixelSamples 304
StratifiedSampler::xPos 304
StratifiedSampler::yPixelSamples 304
StratifiedSampler::yPos 304

In order to randomly associate a time and lens sample with each image sample, this method 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.

⟨*Decorrelate sample dimensions*⟩ ≡ 308

```
Shuffle(lensSamples, xPixelSamples*yPixelSamples, 2);
Shuffle(timeSamples, xPixelSamples*yPixelSamples, 1);
```

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

⟨*Sampling Function Definitions*⟩+≡
```
COREDLL void Shuffle(float *samp, int count, int dims) {
    for (int i = 0; i < count; ++i) {
        u_int other = RandomUInt() % count;
        for (int j = 0; j < dims; ++j)
            swap(samp[dims*i + j], samp[dims*other + j]);
    }
}
```

Given this infrastructure, it's now possible to implement the `GetNextSample()` method of the `StratifiedSampler`. It starts by checking whether it needs to generate a new pixel's worth of samples and whether all the samples have already been generated. It then generates new samples if needed and initializes the `Sample` pointer using the stored samples.

⟨*StratifiedSampler Method Definitions*⟩+≡
```
bool StratifiedSampler::GetNextSample(Sample *sample) {
    ⟨Compute new set of samples if needed for next pixel 310⟩
    ⟨Return next StratifiedSampler sample point 311⟩
    return true;
}
```

The `samplePos` variable keeps track of the current offset into the pixel-sized table of precomputed samples. When it reaches the end of the table, the sampler moves ahead to the next pixel.

⟨*Compute new set of samples if needed for next pixel*⟩≡ 310
```
if (samplePos == xPixelSamples * yPixelSamples) {
    ⟨Advance to next pixel for stratified sampling 310⟩
    ⟨Generate stratified camera samples for (xPos,yPos) 308⟩
}
```

To advance to the next pixel, the implementation here first tries to move one pixel over in the x direction. If doing so takes it beyond the end of the image, it resets the x position to the first pixel in the next x row of pixels and advances the y position. Because the y stratum counter `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 image, sample generation is complete and then the method returns `false`.

⟨*Advance to next pixel for stratified sampling*⟩≡ 310
```
if (++xPos == xPixelEnd) {
    xPos = xPixelStart;
    ++yPos;
}
if (yPos == yPixelEnd)
    return false;
```

COREDLL 904
RandomUInt() 857
Sample 299
Sampler::xPixelEnd 298
Sampler::xPixelStart 298
Sampler::yPixelEnd 298
StratifiedSampler 304
StratifiedSampler::
samplePos 308
StratifiedSampler::
xPixelSamples 304
StratifiedSampler::xPos 304
StratifiedSampler::
yPixelSamples 304
StratifiedSampler::yPos 304

Since the camera samples have already been computed and stored in the table at this point (either from the current call to `GetNextSample()` or a previous one), initializing the corresponding `Sample` fields is easy; the appropriate values are copied from the sample tables.

⟨*Return next* `StratifiedSampler` *sample point*⟩ ≡ 310
```
sample->imageX = imageSamples[2*samplePos];
sample->imageY = imageSamples[2*samplePos+1];
sample->lensU = lensSamples[2*samplePos];
sample->lensV = lensSamples[2*samplePos+1];
sample->time = timeSamples[samplePos];
⟨Generate stratified samples for integrators 313⟩
++samplePos;
```

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.

Sample 299
Sample::imageX 299
Sample::imageY 299
Sample::lensU 299
Sample::lensV 299
Sample::time 299
StratifiedSampler 304
StratifiedSampler::imageSamples 306
StratifiedSampler::lensSamples 306
StratifiedSampler::RoundSize() 306
StratifiedSampler::samplePos 308
StratifiedSampler::timeSamples 306

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 n 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 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.

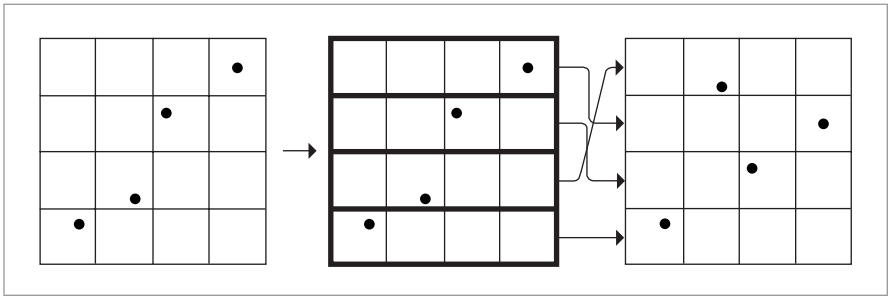


Figure 7.19: Latin hypercube sampling (sometimes called n-rooks sampling) chooses samples such that only a single sample is present in each row and each column of a grid. This can be done by generating random samples in the cells along the diagonal and then randomly permuting their coordinates. One advantage 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.

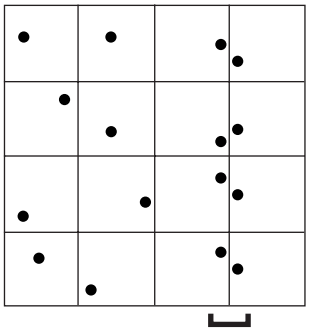


Figure 7.20: A Worst-Case Situation for Stratified Sampling. In an $n \times n$ 2D pattern, up to $2n$ of the points may project to essentially the same point on one of the axes. When "unlucky" patterns like this are generated, the quality of the results computed with them usually suffers.

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.

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*⟩ ≡ 311
```
for (u_int i = 0; i < sample->n1D.size(); ++i)
    LatinHypercube(sample->oneD[i], sample->n1D[i], 1);
for (u_int i = 0; i < sample->n2D.size(); ++i)
    LatinHypercube(sample->twoD[i], sample->n2D[i], 2);
```

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*⟩+ ≡
```
COREDLL void LatinHypercube(float *samples, int nSamples, int nDim) {
    ⟨Generate LHS samples along diagonal 313⟩
    ⟨Permute LHS samples in each dimension 314⟩
}
```

⟨*Generate LHS samples along diagonal*⟩ ≡ 313
```
float delta = 1.f / nSamples;
for (int i = 0; i < nSamples; ++i)
    for (int j = 0; j < nDim; ++j)
        samples[nDim * i + j] = (i + 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).[6]

COREDLL 904
LatinHypercube() 313
RandomFloat() 857
Sample::n1D 301
Sample::n2D 301
Sample::oneD 301
Sample::twoD 301
Shuffle() 310

6 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.

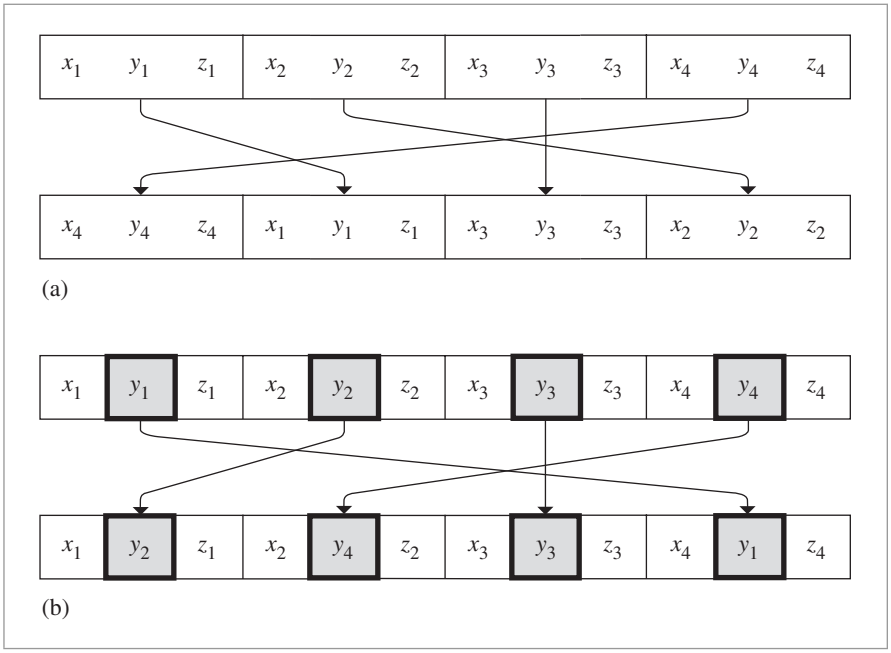


Figure 7.21: (a) The permutation done by the `Shuffle()` routine moves entire blocks of `nDims` elements around. (b) The permutation for Latin hypercube sampling permutes each dimension's samples independently. Here, the shuffling of the second dimension's samples from a four-element pattern of three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

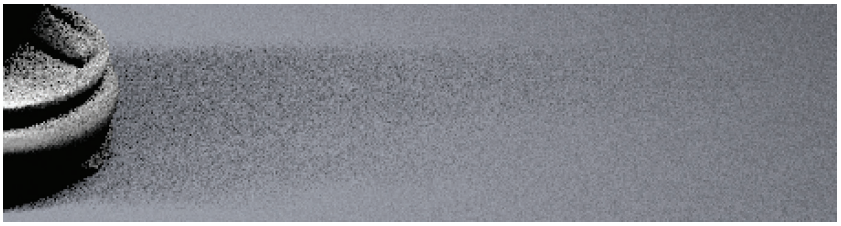
⟨*Permute LHS samples in each dimension*⟩ ≡ 313

```
for (int i = 0; i < nDim; ++i) {
    for (int j = 0; j < nSamples; ++j) {
        u_int other = RandomUInt() % nSamples;
        swap(samples[nDim * j + i],
             samples[nDim * other + i]);
    }
}
```

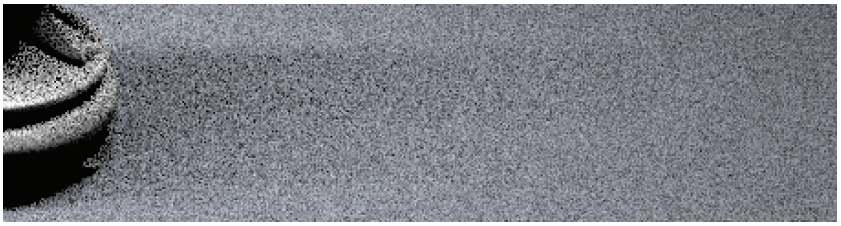
Starting with the scene in Figure 7.22, Figure 7.23 shows the improvement from good samples for the `DirectLighting` integrator. Image (a) was computed with 1 image sample per pixel, each with 16 shadow samples, and image (b) was computed with 16 image samples per pixel, each with 1 shadow sample. Because 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.

DirectLighting 723
RandomUInt() 857
Shuffle() 310
StratifiedSampler 304

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.

⋆ 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.[7] 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 of P with respect to B is[8]

`LDSampler` 328

`StratifiedSampler` 304

7 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.

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

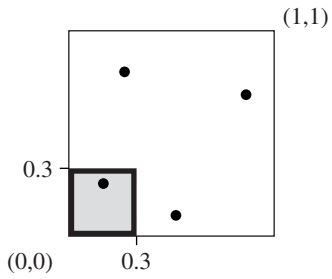


Figure 7.24: The discrepancy of a box (shaded) given a set of 2D sample points in $[0, 1]^2$. One of the four sample points is inside the box, so this set of points would estimate the box's area to be $1/4$. The true area of the box is $.3 \times .3 = .09$, so the discrepancy for this particular box 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.

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

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:

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

Then

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

The bounds for the star discrepancy of a sequence of points in one dimension has 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 pseudorandom variation.

7.4.2 CONSTRUCTING LOW-DISCREPANCY 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

$$\frac{d_i}{b^i}.$$

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.

⟨*Sampling 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 319⟩
    }
    return val;
}
```

⟨*Compute next digit of radical inverse*⟩≡ 319
```
int d_i = (n % base);
val += d_i * invBi;
n /= base;
invBi *= invBase;
```

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,

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

RadicalInverse() 319

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 inverse base b, with a different base for each dimension of the pattern. The bases used

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
⋮	⋮	⋮

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 hundred points of the 2D Hammersley sequence. Figure 7.25(b) shows the Halton sequence.

PhotonIntegrator 774

The *folded radical inverse* function can be used in place of the original radical inverse function to further reduce the discrepancy of Hammersley and Halton sequences. The folded radical inverse is defined by adding the offset i to the ith digit d_i and taking the

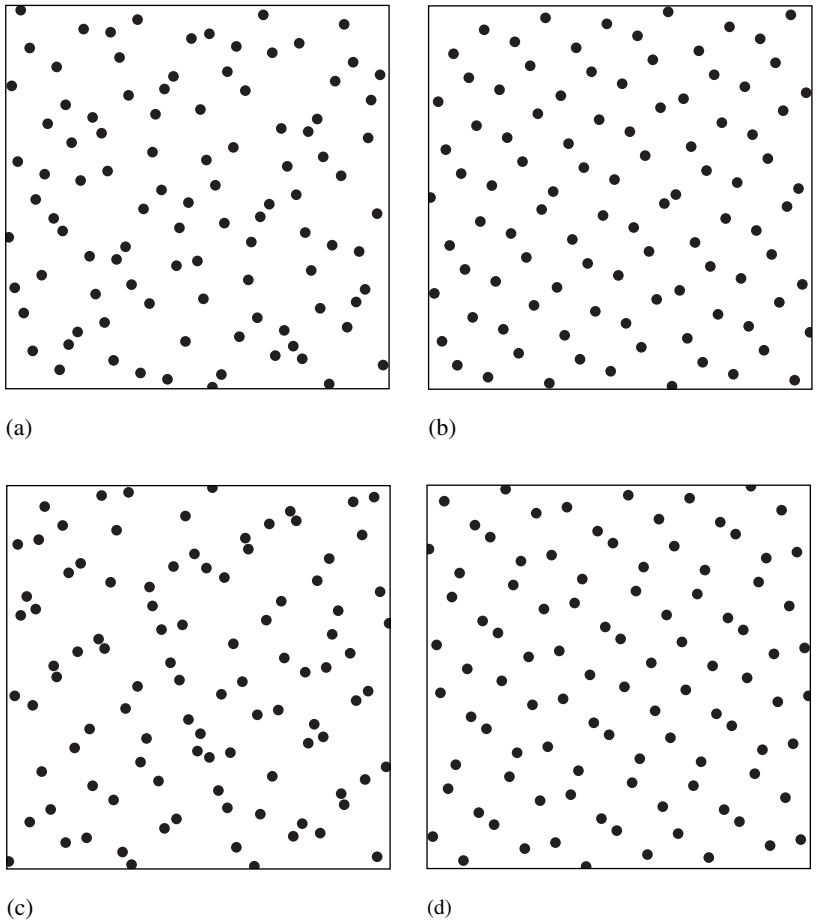


Figure 7.25: The First 100 Points of Various Low-Discrepancy Sequences in 2D. (a) Halton, (b) Hammersley, (c) Halton with folded radical inverse, (d) Hammersley with folded radical inverse.

result modulus b, then adding that result to the next digit to the right of the decimal point:

FoldedRadicalInverse() 322
RadicalInverse() 319

$$\Psi_b(n) = \sum_i \frac{(d_i + i - 1) \bmod b}{b^i}.$$

The `FoldedRadicalInverse()` function computes Ψ_b. It is similar to the original `RadicalInverse()` function, but with two modifications. First, it needs to track which digit is currently being processed, so that the appropriate offset can be added before

the modulus computation; this is done with the `modOffset` variable. Second, it needs to handle the fact that Ψ_b is actually an *infinite* sum. Even though the digits d_i are zero after a finite number of terms, the offset that is added ensures that most terms beyond this point will be nonzero. Fortunately, the finite precision of computer floating-point numbers has the effect that the implementation can conservatively stop adding digits to the folded radical inverse as soon as it detects that `invBi` is small enough that adding its contribution to `val` is certain to leave `val` unchanged.

⟨*Sampling Declarations*⟩+≡
```
inline double FoldedRadicalInverse(int n, int base) {
    double val = 0;
    double invBase = 1.f/base, invBi = invBase;
    int modOffset = 0;
    while (val + base * invBi != val) {
        ⟨Compute next digit of folded radical inverse 322⟩
    }
    return val;
}
```

⟨*Compute next digit of folded radical inverse*⟩≡ 322
```
int digit = ((n+modOffset) % base);
val += digit * invBi;
n /= base;
invBi *= invBase;
++modOffset;
```

When the folded radical inverse is used to generate the Hammersley and Halton point sets, they are known as the Hammersley-Zaremba and Halton-Zaramba point sets, after the inventor of the folded radical inverse function. Plots of the first 100 Hammersley-Zaremba and Halton-Zaremba points are shown in Figures 7.25(c) and 7.25(d). It's possible to see visually that the Hammersley sequence has lower discrepancy than the Halton sequence—there are far fewer clumps of nearby sample points. Furthermore, one can see that the folded radical inverse function reduces the discrepancy of the Hammersley sequence; its effect on the Halton sequence is less visually obvious.

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 sample positions for a pixel's worth of image samples such that each sample is well-stratified with respect not only to the other samples in the set for the pixel but also to the sample positions at the other image samples around the current pixel.

StratifiedSampler 304

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 previous stratification constraint, 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 7.26 shows all of the possibilities for dividing the domain into regions where the first 16 samples of 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

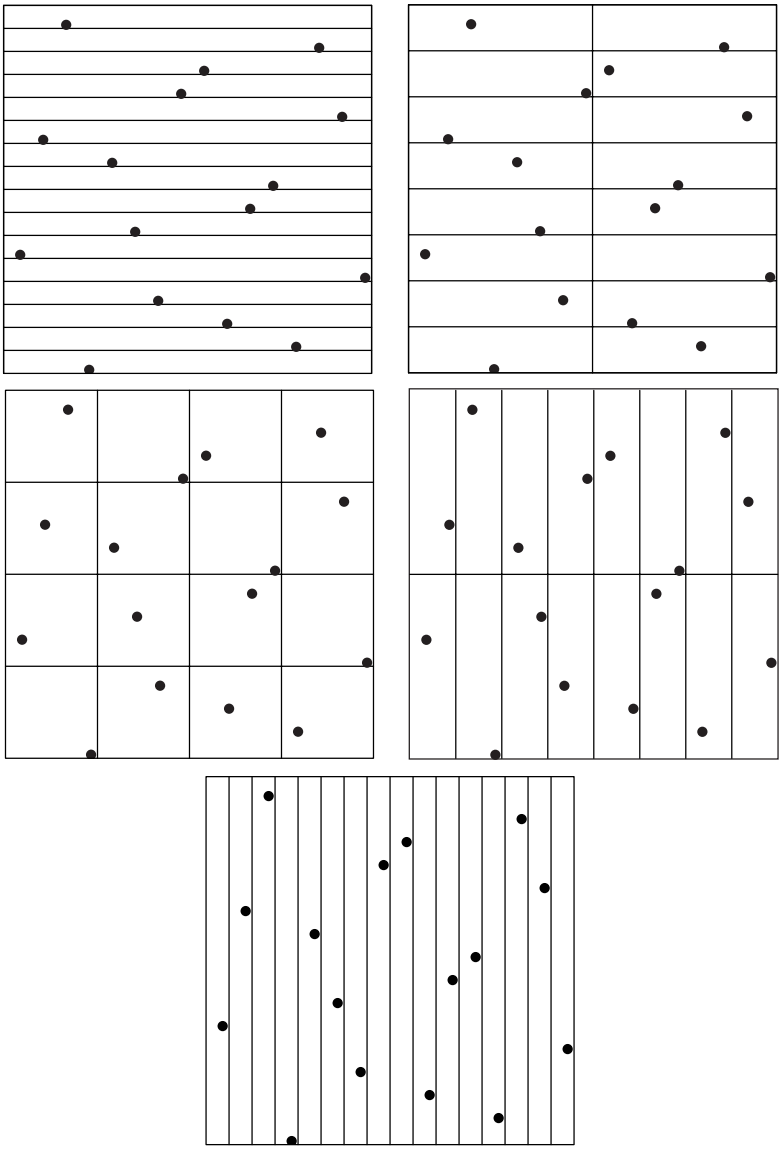


Figure 7.26: A sampling pattern that has a single sample in all of the base 2 elementary intervals. It satisfies both the 4×4 stratification and Latin hypercube constraints as well as the other stratification constraints shown.

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 7.27 shows an unscrambled (0, 2)-sequence and two randomly scrambled variations of it.

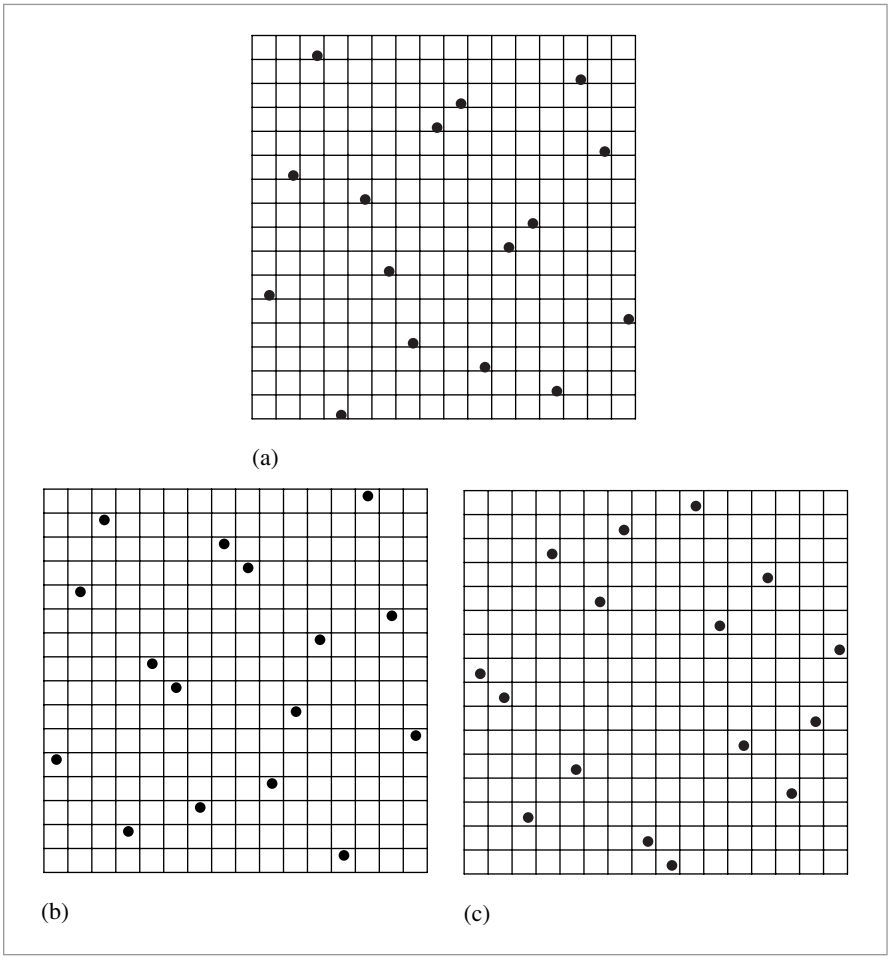


Figure 7.27: (a) A low-discrepancy (0, 2)-sequence-based sampling pattern and (b, c) two randomly scrambled instances of it. Random scrambling of low-discrepancy patterns is an effective way to eliminate the artifacts that would be present in images if we used the same sampling pattern in every pixel, while still preserving the low-discrepancy properties of the point set being used.

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 that at each level l of the recursive scrambling, the same decision will be 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 `u_int` 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(u_int n, u_int 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 `u_int` 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(u_int n, u_int scramble) {
    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);
    n ^= scramble;
    return (float)n / (float)0x100000000LL;
}
```

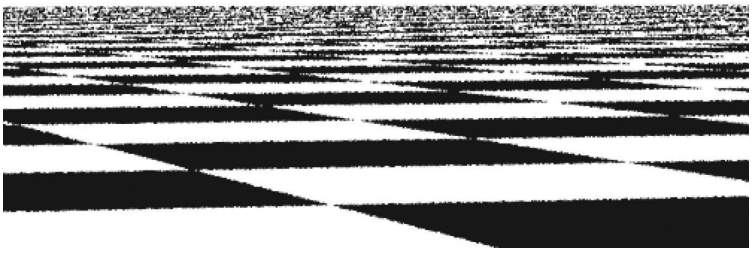
⟨*Sampling Inline Functions*⟩ +≡
```
inline float Sobol2(u_int n, u_int scramble) {
    for (u_int v = 1 << 31; n != 0; n >>= 1, v ^= v >> 1)
        if (n & 0x1) scramble ^= v;
    return (float)scramble / (float)0x100000000LL;
}
```

RadicalInverse() 319
Sobol2() 326
VanDerCorput() 326

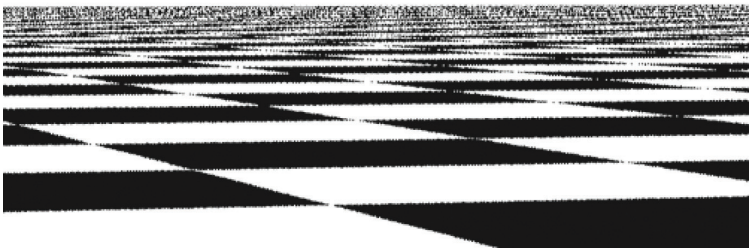
7.4.4 THE LOW-DISCREPANCY SAMPLER

We need to turn this theory into a practical `Sampler` for the renderer. One potential approach is to consider the problem as a general high-dimensional sampling problem and just use a Hammersley sequence to compute samples for all dimensions—image plane, time, lens, and integration. We have left this approach for an exercise at the end of the chapter and here will instead implement a pattern based on (0, 2)-sequences, since they give the foundation for a particularly effective approach for generating multiple samples per image sample for integration.

Another reason not to use the Hammersley point set is that it can be prone to aliasing in images. Figure 7.28 compares the results of sampling a checkerboard texture using a Hammersley-based sampler to using the stratified sampler from the previous section. Note the unpleasant pattern along edges in the foreground and toward the horizon.



(a)

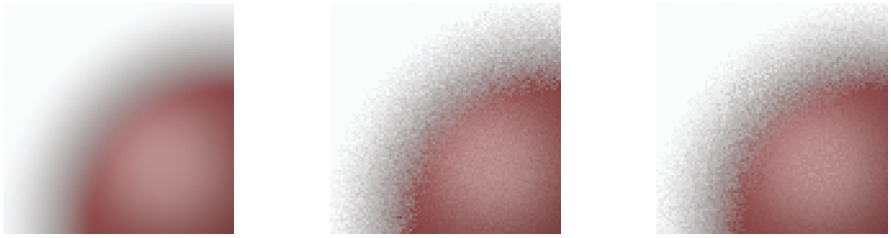


(b)

Figure 7.28: Comparison of the Stratified Sampler to a Low-Discrepancy Sampler Based on Hammersley Points on the Image Plane. (a) The jittered stratified sampler with a single sample per pixel and (b) the Hammersley sampler with a single sample per pixel. Note that although the Hammersley 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 `LDSampler` in this section performs similarly to the jittered sampler.)

`LDSampler` 328

`Sampler` 296



(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.

The `LDSampler` uses a (0, 2)-sequence to generate a number of samples for each pixel that must be a power of two. Samples for positions on the lens and for the two-dimensional integrator samples are similarly generated with scrambled (0, 2)-sequences, and one-dimensional samples for time and integrators are generated with a scrambled van der Corput sequence. Similarly to the `StratifiedSampler`, the `LDSampler` generates an entire pixel's worth of samples at once and hands them out in turn from `GetNextSample()`. 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 329⟩
private:
    ⟨LDSampler Private Data 329⟩
};
```

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. Note also that the constructor sets up the current pixel (`xPos,yPos`) and the current pixel sample `samplePos` in a way that causes the first call to `GetNextSample()` to immediately generate the samples for the true first pixel, (`xPixelStart,yPixelStart`).

`LDSampler` 328
`Sampler` 296
`StratifiedSampler` 304

⟨*LDSampler Method Definitions*⟩ ≡
```
LDSampler::LDSampler(int xstart, int xend,
                     int ystart, int yend, int ps)
    : Sampler(xstart, xend, ystart, yend, RoundUpPow2(ps)) {
    xPos = xPixelStart - 1;
    yPos = yPixelStart;
    if (!IsPowerOf2(ps)) {
        Warning("Pixel samples being rounded up to power of 2");
        pixelSamples = RoundUpPow2(ps);
    }
    else
        pixelSamples = ps;
    samplePos = pixelSamples;
    oneDSamples = twoDSamples = NULL;
    imageSamples = new float[5*pixelSamples];
    lensSamples = imageSamples + 2*pixelSamples;
    timeSamples = imageSamples + 4*pixelSamples;
}
```

⟨*LDSampler Private Data*⟩ ≡ 328
```
int xPos, yPos, pixelSamples;
int samplePos;
float *imageSamples, *lensSamples, *timeSamples;
float **oneDSamples, **twoDSamples;
```

imageSamples 335
LDSampler 328
LDSampler::imageSamples 329
LDSampler::lensSamples 329
LDSampler::oneDSamples 329
LDSampler::pixelSamples 329
LDSampler::samplePos 329
LDSampler::timeSamples 329
LDSampler::twoDSamples 329
LDSampler::xPos 329
LDSampler::yPos 329
RoundUpPow2() 855
Sampler 296
Sampler::xPixelStart 298
Sampler::yPixelStart 298
StratifiedSampler 304

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

⟨*LDSampler Public Methods*⟩ ≡ 328
```
int RoundSize(int size) const {
    return RoundUpPow2(size);
}
```

The `xPos`, `yPos`, and `samplePos` variables are just like those in the `StratifiedSampler`. The implementation here, however, precomputes both the image, lens, and time samples for the pixel as well as an entire pixel's worth of all of the samples requested by the integrator. These are stored in the `oneDSamples` and `twoDSamples` arrays, respectively.

⟨*LDSampler Method Definitions*⟩+≡
```
bool LDSampler::GetNextSample(Sample *sample) {
    if (!oneDSamples) {
        ⟨Allocate space for pixel's low-discrepancy sample tables 330⟩
    }
    if (samplePos == pixelSamples) {
        ⟨Advance to next pixel for low-discrepancy sampling 330⟩
        ⟨Generate low-discrepancy samples for pixel 331⟩
    }
    ⟨Copy low-discrepancy samples from tables 332⟩
    ++samplePos;
    return true;
}
```

It is not possible to allocate space for the integrator samples in the constructor since the number needed is not yet known when it executes. Therefore, allocation is done the first time this method is called.

⟨*Allocate space for pixel's low-discrepancy sample tables*⟩≡ 330
```
oneDSamples = new float *[sample->n1D.size()];
for (u_int i = 0; i < sample->n1D.size(); ++i)
    oneDSamples[i] = new float[sample->n1D[i] * pixelSamples];
twoDSamples = new float *[sample->n2D.size()];
for (u_int i = 0; i < sample->n2D.size(); ++i)
    twoDSamples[i] = new float[2 * sample->n2D[i] * pixelSamples];
```

The logic for advancing to the next pixel is just like in the `StratifiedSampler`.

⟨*Advance to next pixel for low-discrepancy sampling*⟩≡ 330
```
if (++xPos == xPixelEnd) {
    xPos = xPixelStart;
    ++yPos;
}
if (yPos == yPixelEnd)
    return false;
samplePos = 0;
```

There is a subtle implementation detail that must be accounted for in using these patterns in practice.[9] 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

LDSampler 328
Sample 299
StratifiedSampler 304

9 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.

one of the N light sources in the scene to sample illumination from, and 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 ith element of the one-dimensional pattern and the ith 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. The `LDShuffleScrambled1D()` and `LDShuffleScrambled2D()` functions take care of this:

⟨*Generate low-discrepancy samples for pixel*⟩ ≡ 330

```
LDShuffleScrambled2D(1, pixelSamples, imageSamples);
LDShuffleScrambled2D(1, pixelSamples, lensSamples);
LDShuffleScrambled1D(1, pixelSamples, timeSamples);
for (u_int i = 0; i < sample->n1D.size(); ++i)
    LDShuffleScrambled1D(sample->n1D[i], pixelSamples,
                         oneDSamples[i]);
for (u_int i = 0; i < sample->n2D.size(); ++i)
    LDShuffleScrambled2D(sample->n2D[i], pixelSamples,
                         twoDSamples[i]);
```

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 into a random order to eliminate correlation between the ith sample from this particular sequence and the ith sample from other sequences in this pixel.

LDSampler::imageSamples 329
LDSampler::lensSamples 329
LDSampler::oneDSamples 329
LDSampler::pixelSamples 329
LDSampler::timeSamples 329
LDSampler::twoDSamples 329
LDShuffleScrambled1D() 331
LDShuffleScrambled2D() 332
RandomUInt() 857
Sample::n1D 301
Sample::n2D 301
Shuffle() 310
VanDerCorput() 326

⟨*Sampling Inline Functions*⟩ +≡

```
inline void LDShuffleScrambled1D(int nSamples,
        int nPixel, float *samples) {
    u_int scramble = 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);
    Shuffle(samples, nPixel, nSamples);
}
```

⟨*Sampling Inline Functions*⟩+≡
```
inline void LDShuffleScrambled2D(int nSamples,
        int nPixel, float *samples) {
    u_int scramble[2] = { RandomUInt(), 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);
    Shuffle(samples, nPixel, 2 * nSamples);
}
```

Given the precomputed sample values for the pixel, initializing the `Sample` structure is a matter of extracting the appropriate values for the current sample in the pixel from the sample tables and copying them to the appropriate places.

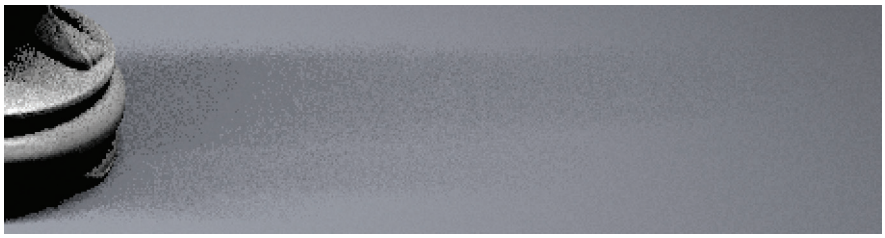
⟨*Copy low-discrepancy samples from tables*⟩≡ 330
```
sample->imageX = xPos + imageSamples[2*samplePos];
sample->imageY = yPos + imageSamples[2*samplePos+1];
sample->time = timeSamples[samplePos];
sample->lensU = lensSamples[2*samplePos];
sample->lensV = lensSamples[2*samplePos+1];
for (u_int i = 0; i < sample->n1D.size(); ++i) {
    int startSamp = sample->n1D[i] * samplePos;
    for (u_int j = 0; j < sample->n1D[i]; ++j)
        sample->oneD[i][j] = oneDSamples[i][startSamp+j];
}
for (u_int i = 0; i < sample->n2D.size(); ++i) {
    int startSamp = 2 * sample->n2D[i] * samplePos;
    for (u_int j = 0; j < 2*sample->n2D[i]; ++j)
        sample->twoD[i][j] = twoDSamples[i][startSamp+j];
}
```

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.

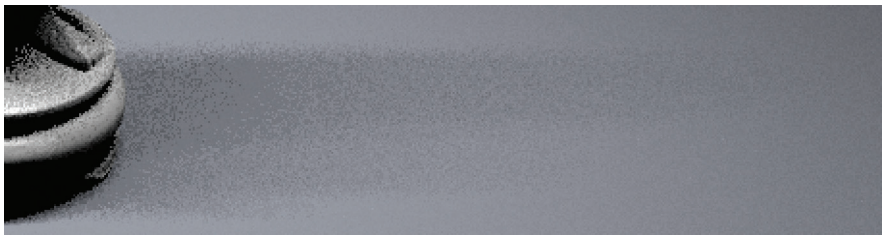
imageSamples 335
LDSampler 328
LDSampler::imageSamples 329
LDSampler::lensSamples 329
LDSampler::oneDSamples 329
LDSampler::samplePos 329
LDSampler::timeSamples 329
LDSampler::twoDSamples 329
LDSampler::xPos 329
LDSampler::yPos 329
lensSamples 335
RandomUInt() 857
Sample 299
Sample02() 326
Sample::imageX 299
Sample::imageY 299
Sample::lensU 299
Sample::lensV 299
Sample::time 299
Shuffle() 310
StratifiedSampler 304

★ 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*



(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.

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. Poisson disk patterns are usually generated by *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.

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

LDSampler 328

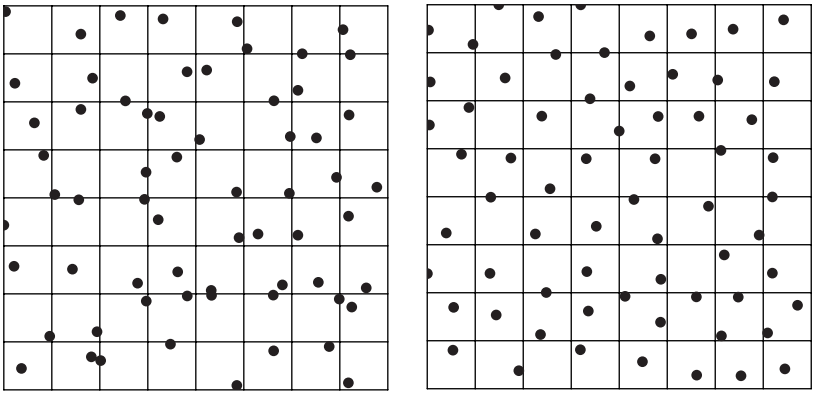


Figure 7.31: Comparison of Sampling Patterns. (a) A jittered pattern: note clumping of samples and undersampling in some areas. (b) A pattern generated with the best-candidate algorithm: it is effectively as good as the Poisson disk pattern.

number of samples than 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 will compute a good sampling pattern once in a preprocess. The pattern is stored in a table and can be efficiently used at rendering time. Rather than computing a sampling pattern large enough to sample the largest image we'd ever render, we'll compute a pattern that 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.

7.5.1 GENERATING THE BEST-CANDIDATE PATTERN

We precompute the best-candidate sampling pattern in an offline process. The code described here can be found in the file `tools/samplepat.cpp`. The results are stored in the file `samplers/sampledata.cpp`.

First, we need to define the size of the table used to store the computed sample pattern:

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

This program generates sample points in a five-dimensional space: two dimensions for the image sample location, one for the time, and two more to determine a point on the lens. Because we don't know what types of sample patterns the integrators will need a priori, the `BestCandidateSampler` uses both scrambled (0, 2)-sequences and stratified patterns for their samples.

⟨*Pattern Precomputation Local Data*⟩ ≡
```
static float imageSamples[SAMPLE_TABLE_SIZE][2];
static float timeSamples[SAMPLE_TABLE_SIZE];
static float lensSamples/[SAMPLE_TABLE_SIZE][2];
```

Sample values are computed in a multistage process. First, a well-distributed set of image sample positions is generated. Then, given the image samples, a good set of time samples is found, accounting for the positions of the time samples associated with the nearby image samples. Finally, good samples for the lens are computed, again taking into account the positions of lens samples at nearby image samples, so that close-by image samples tend to have spread-out lens samples.

⟨*Sample Pattern Precomputation*⟩ ≡
```
int main() {
    ⟨Compute image sample positions 336⟩
    ⟨Compute time samples 339⟩
    ⟨Compute lens samples 341⟩
    ⟨Write sample table to disk 342⟩
    return 0;
}
```

In order to speed up the candidate evaluation, the accepted samples are stored in a grid. This makes it possible to check only nearby samples when computing distances between samples. The grid splits up the 2D sample domain $[0, 1]^2$ into `BC_GRID_SIZE` strata in each direction and stores a list of integer sample numbers identifying the samples that overlap each cell. The `GRID()` macro maps a position in $[0, 1]$ in one of the two dimensions to the corresponding grid cell.

BC_GRID_SIZE 335
BestCandidateSampler 344
GRID() 335
SAMPLE_TABLE_SIZE 335
SQRT_SAMPLE_TABLE_SIZE 335

⟨*Global Forward Declarations*⟩ ≡
```
#define BC_GRID_SIZE 40
typedef vector<int> SampleGrid[BC_GRID_SIZE][BC_GRID_SIZE];
#define GRID(v) (int((v) * BC_GRID_SIZE))
```

To compute the image samples, the program starts by creating a sample grid, and then uses the 2D best-candidate algorithm to fill in the image samples in the grid.

⟨*Compute image sample positions*⟩ ≡ 335
```
SampleGrid pixelGrid;
BestCandidate2D(imageSamples, SAMPLE_TABLE_SIZE, &pixelGrid);
```

⟨*Sample Pattern Precomputation*⟩ +≡
```
void BestCandidate2D(float table[][2], int totalSamples,
                     SampleGrid *grid) {
    SampleGrid localGrid;
    if (!grid) grid = &localGrid;
    ⟨Generate first 2D sample arbitrarily 336⟩
    for (int currentSample = 1;
         currentSample < totalSamples;
         ++currentSample) {
        ⟨Generate next best 2D image sample 337⟩
    }
}
```

The first image sample position is chosen completely at random and recorded in the grid. For all subsequent samples, a set of candidates is generated and compared to the already accepted samples.

⟨*Generate first 2D sample arbitrarily*⟩ ≡ 336
```
table[0][0] = RandomFloat();
table[0][1] = RandomFloat();
addSampleToGrid(table, 0, grid);
```

A short utility function adds a particular point in the table of samples to a `SampleGrid`:

⟨*Pattern Precomputation Utility Functions*⟩ ≡
```
static void addSampleToGrid(float sample[][2], int sampleNum,
                            SampleGrid *grid) {
    int u = GRID(sample[sampleNum][0]);
    int v = GRID(sample[sampleNum][1]);
    (*grid)[u][v].push_back(sampleNum);
}
```

To generate the rest of the samples, a dart-throwing algorithm throws a number of candidate darts for each needed sample. The number of darts thrown is proportional to the number of samples that have been stored already; this ensures that the quality of the samples remains in a sense consistent. After throwing a dart, the program computes how close it is to all of the samples it has accepted so far. If the dart is farther away than the previous best candidate was, it is kept. At the end of the loop, the best remaining candidate is accepted.

BestCandidate2D() 336
currentSample 336
GRID() 335
imageSamples 335
RandomFloat() 857
SampleGrid 335
SAMPLE_TABLE_SIZE 335

⟨*Generate next best 2D image sample*⟩ ≡ 336
```
float maxDist2 = 0.;
int numCandidates = 500 * currentSample;
for (int currentCandidate = 0;
     currentCandidate < numCandidates;
     ++currentCandidate) {
    ⟨Generate a random candidate sample 337⟩
    ⟨Loop over neighboring grid cells and check distances 337⟩
    ⟨Keep this sample if it is the best one so far 338⟩
}
addSampleToGrid(table, currentSample, grid);
```

Candidate positions are chosen completely at random. Note that image sample locations are being computed in the range [0, 1). It is up to the `Sampler` that uses the sampling pattern to scale and translate image samples into raster space appropriately.

⟨*Generate a random candidate sample*⟩ ≡ 337
```
float candidate[2];
candidate[0] = RandomFloat();
candidate[1] = RandomFloat();
```

Given a candidate, it is necessary to compute the distances to all of the nearby samples in the grid, keeping track of the minimum distance to any of them. For efficiency, the implementation actually computes the squared distance, which gives the same result for this test and avoids expensive square root computations.

Distances are computed to only the candidates in the eight neighboring grid cells and the cell that the candidate is in. Although this means that the first few samples are not optimally distributed relative to each other, this won't matter by the time the complete set of samples has been found, as long as `BC_GRID_SIZE` is less than `SQRT_SAMPLE_TABLE_SIZE`.

⟨*Loop over neighboring grid cells and check distances*⟩ ≡ 337
```
float sampleDist2 = INFINITY;
int gu = GRID(candidate[0]), gv = GRID(candidate[1]);
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Compute (u,v) grid cell to check 338⟩
        ⟨Update minimum squared distance from cell's samples 338⟩
    }
}
```

BC_GRID_SIZE 335
currentSample 336
GRID() 335
INFINITY 856
RandomFloat() 857
Sampler 296
SQRT_SAMPLE_TABLE_SIZE 335

In determining which grid cell to check, it is necessary to handle the toroidal topology of the grid. If the cell that would be considered is out of bounds, the routine wraps around to the other end of the grid.

⟨*Compute (u,v) grid cell to check*⟩ ≡ 337, 340, 342
```
int u = gu + du, v = gv + dv;
if (u < 0)              u += BC_GRID_SIZE;
if (u >= BC_GRID_SIZE) u -= BC_GRID_SIZE;
if (v < 0)              v += BC_GRID_SIZE;
if (v >= BC_GRID_SIZE) v -= BC_GRID_SIZE;
```

Given the grid cell, the program then loops over the list of sample numbers. For each sample, it computes the squared distance to the current candidate, tracking the lowest squared distance found so far.

⟨*Update minimum squared distance from cell's samples*⟩ ≡ 337
```
for (u_int g = 0; g < (*grid)[u][v].size(); ++g) {
    int s = (*grid)[u][v][g];
    float xdist = Wrapped1DDist(candidate[0], table[s][0]);
    float ydist = Wrapped1DDist(candidate[1], table[s][1]);
    float d2 = xdist*xdist + ydist*ydist;
    sampleDist2 = min(sampleDist2, d2);
}
```

The computation for the 1D distance between two values in [0, 1] also needs to handle the wraparound issue. Consider two samples with x coordinates of .01 and .99. Direct computation will find their distance to be .98, although with wraparound, the actual distance should be .02. Whenever the initial distance is greater than 0.5, the wrapped distance will be lower. In that case, the true distance is just the distance from the higher sample to one, plus the distance from zero to the lower sample.

⟨*Pattern Precomputation Utility Functions*⟩+ ≡
```
inline float Wrapped1DDist(float a, float b) {
    float d = fabsf(a - b);
    if (d < 0.5f) return d;
    else return 1.f - max(a, b) + min(a, b);
}
```

Once the minimum squared distance has been found, its value is compared to the minimum squared distance for the previous best candidate. If the new one has a higher squared distance to its neighbors, its distance is recorded and it is tentatively put in the output `table`.

⟨*Keep this sample if it is the best one so far*⟩ ≡ 337
```
if (sampleDist2 > maxDist2) {
    maxDist2 = sampleDist2;
    table[currentSample][0] = candidate[0];
    table[currentSample][1] = candidate[1];
}
```

BC_GRID_SIZE 335
currentSample 336
Wrapped1DDist() 338

After generating all of the image samples in this manner, the sample positions for the rest of the dimensions can be found. One approach would be to generalize the Poisson disk concept to a higher-dimensional Poisson sphere. Interestingly enough, it is possible to do better than this, particularly in the five-dimensional case where a very large number of candidate samples would be needed to find good ones.

Consider the problem of choosing time values for two nearby image samples: not only should the time values not be too close together, but in fact they should be as far apart as possible. In any local 2D region of the image, we'd like the best possible coverage of the complete three-dimensional sample space. An intuition for why this is the case comes from how the sampling pattern will be used. Although we're generating a five-dimensional pattern overall, what we're interested in is optimizing its distribution across local areas of the two-dimensional image plane. Optimizing its distribution over the five-dimensional space is at best a secondary concern.

Therefore, a two-stage process is used to generate the sample positions. First, a well-distributed sampling pattern for the time and lens positions is found. Then, these samples are associated with image samples in a way that ensures that nearby image samples have sample values in the other dimensions that are well spread out.[10]

To compute the time samples, a set of one-dimensional stratified sample values over [0, 1] is generated. The `timeSamples` array is then rearranged so that the ith time sample is a good one for the ith image sample.

⟨*Compute time samples*⟩ ≡ 335
```
for (int i = 0; i < SAMPLE_TABLE_SIZE; ++i)
    timeSamples[i] = (i + RandomFloat()) / SAMPLE_TABLE_SIZE;
for (int currentSample = 1;
     currentSample < SAMPLE_TABLE_SIZE;
     ++currentSample) {
    ⟨Select best time sample for current image sample 339⟩
}
```

⟨*Select best time sample for current image sample*⟩ ≡ 339
```
int best = -1;
⟨Find best time relative to neighbors 340⟩
swap(timeSamples[best], timeSamples[currentSample]);
```

As the program looks for a good time sample for the image sample number `currentSample`, the elements of `timeSamples` from zero to `currentSample-1` have already

`currentSample` 336
`RandomFloat()` 857
`SAMPLE_TABLE_SIZE` 335
`timeSamples` 335

10 As if that isn't enough to worry about, we should also consider *correlation*. Not only should nearby image samples have distant sample values for the other dimensions, but we should also make sure that, for example, the time and lens values aren't correlated. If samples were chosen such that the time value was always similar to the lens u sample value, the sample pattern is not as good as it would be if the two were uncorrelated. We won't address this issue in our implementation here because the technique used here is not prone to introducing correlation in the first place.

been assigned to previous image samples and are unavailable. The rest of the times, from currentSample to SAMPLE_TABLE_SIZE-1, are the ones to be chosen from.

⟨*Find best time relative to neighbors*⟩ ≡ 339
```
float maxMinDelta = 0.;
for (int t = currentSample; t < SAMPLE_TABLE_SIZE; ++t) {
    ⟨Compute min delta for this time 340⟩
    ⟨Update best if this is best time so far 341⟩
}
```

Similar to the way image samples were evaluated, only the samples in the adjoining few grid cells are examined here. Of these, the one that is most different from the time samples that have already been assigned to the nearby image samples will be selected.

⟨*Compute min delta for this time*⟩ ≡ 340
```
int gu = GRID(imageSamples[currentSample][0]);
int gv = GRID(imageSamples[currentSample][1]);
float minDelta = INFINITY;
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Check offset from times of nearby samples 340⟩
    }
}
```

The implementation loops through the image samples in each of the grid cells, although it only needs to consider the ones that already have time samples associated with them. Therefore, it skips over the ones with sample numbers greater than the sample it's currently working on. For the remaining ones, it computes the offset from their time sample to the current candidate time sample, keeping track of the minimum difference.

⟨*Check offset from times of nearby samples*⟩ ≡ 340
```
⟨Compute (u,v) grid cell to check 338⟩
for (u_int g = 0; g < pixelGrid[u][v].size(); ++g) {
    int otherSample = pixelGrid[u][v][g];
    if (otherSample < currentSample) {
        float dt = Wrapped1DDist(timeSamples[otherSample], timeSamples[t]);
        minDelta = min(minDelta, dt);
    }
}
```

currentSample 336
GRID() 335
imageSamples 335
INFINITY 856
SAMPLE_TABLE_SIZE 335
timeSamples 335
Wrapped1DDist() 338

If the minimum offset from the current time sample is greater than the minimum distance of the previous best time sample, this sample is recorded as the best one so far.

⟨*Update* best *if this is best time so far*⟩ ≡ 340
```
if (minDelta > maxMinDelta) {
    maxMinDelta = minDelta;
    best = t;
}
```

Finally, the lens positions are computed. Good sampling patterns are again generated with dart throwing and then associated with image samples in the same manner that times were, by selecting lens positions that are far away from the lens positions of nearby image samples.

⟨*Compute lens samples*⟩ ≡ 335
```
BestCandidate2D(lensSamples, SAMPLE_TABLE_SIZE);
Redistribute2D(lensSamples, pixelGrid);
```

After the `BestCandidate2D()` function generates the set of 2D samples, the `Redistribute2D()` utility function takes the set of lens samples to be assigned to the image samples and reshuffles them so that each sample isn't too close to those in neighboring pixels.

⟨*Sample Pattern Precomputation*⟩ +≡
```
static void Redistribute2D(float samples[][2], SampleGrid &pixelGrid) {
    for (int currentSample = 1;
         currentSample < SAMPLE_TABLE_SIZE;
         ++currentSample) {
        ⟨Select best lens sample for current image sample 341⟩
    }
}
```

⟨*Select best lens sample for current image sample*⟩ ≡ 341
```
int best = -1;
⟨Find best 2D sample relative to neighbors 341⟩
swap(samples[best][0], samples[currentSample][0]);
swap(samples[best][1], samples[currentSample][1]);
```

As with the time samples, we would like to to choose the lens sample that has the minimum distance to the lens sample values that have already been assigned to the neighboring image samples.

BestCandidate2D() 336
currentSample 336
lensSamples 335
Redistribute2D() 341
SampleGrid 335
SAMPLE_TABLE_SIZE 335

⟨*Find best 2D sample relative to neighbors*⟩ ≡ 341
```
float maxMinDist2 = 0.f;
for (int samp = currentSample; samp < SAMPLE_TABLE_SIZE; ++samp) {
    ⟨Check distance to lens positions at nearby samples 342⟩
    ⟨Update best for 2D lens sample if it is best so far 342⟩
}
```

⟨*Check distance to lens positions at nearby samples*⟩ ≡ 341
```
int gu = GRID(imageSamples[currentSample][0]);
int gv = GRID(imageSamples[currentSample][1]);
float minDist2 = INFINITY;
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Check 2D samples in current grid cell 342⟩
    }
}
```

⟨*Check 2D samples in current grid cell*⟩ ≡ 342
```
⟨Compute (u,v) grid cell to check 338⟩
for (u_int g = 0; g < pixelGrid[u][v].size(); ++g) {
    int s2 = pixelGrid[u][v][g];
    if (s2 < currentSample) {
        float dx = Wrapped1DDist(samples[s2][0], samples[samp][0]);
        float dy = Wrapped1DDist(samples[s2][1], samples[samp][1]);
        float d2 = dx*dx + dy*dy;
        minDist2 = min(d2, minDist2);
    }
}
```

⟨*Update* `best` *for 2D lens sample if it is best so far*⟩ ≡ 341
```
if (minDist2 > maxMinDist2) {
    maxMinDist2 = minDist2;
    best = samp;
}
```

The last step is to open a file and write out C++ code that initializes the table. When the `BestCandidateSampler` is compiled, it will `#include` this file to initialize its sample table.

⟨*Write sample table to disk*⟩ ≡ 335
```
FILE *f = fopen("sampledata.cpp", "w");
if (f == NULL)
    Severe("Couldn't open sampledata.cpp for writing.");
fprintf(f, "\n/* Automatically generated %dx%d sample "
        "table (%s @ %s) */\n\n",
        SQRT_SAMPLE_TABLE_SIZE, SQRT_SAMPLE_TABLE_SIZE,
        __DATE__, __TIME__);
fprintf(f, "const float "
        "BestCandidateSampler::sampleTable[%d][5] = {\n",
        SAMPLE_TABLE_SIZE);
```

BestCandidateSampler 344
currentSample 336
GRID() 335
imageSamples 335
INFINITY 856
lensSamples 335
SAMPLE_TABLE_SIZE 335
Severe() 834
SQRT_SAMPLE_TABLE_SIZE 335
timeSamples 335
Wrapped1DDist() 338

```
for (int i = 0; i < SAMPLE_TABLE_SIZE; ++i) {
    fprintf(f, "  { ");
    fprintf(f, "%10.10ff, %10.10ff, ", imageSamples[i][0],
        imageSamples[i][1]);
    fprintf(f, "%10.10ff, ", timeSamples[i]);
    fprintf(f, "%10.10ff, %10.10ff, ", lensSamples[i][0],
        lensSamples[i][1]);
    fprintf(f, "},\n");
}
fprintf(f, "};\n");
```

7.5.2 USING THE BEST-CANDIDATE PATTERN

`BestCandidateSampler`, the `Sampler` that uses the sample table, is fairly straightforward. If an average of `pixelSamples` samples is to be taken in each pixel, a single copy of the sample table covers `SQRT_SAMPLE_TABLE_SIZE / sqrt(pixelSamples)` pixels in the x and y directions. Like the `StratifiedSampler` and `LDSampler`, this sampler scans across the image from the upper left of the crop window, going left to right and then top to bottom. Here, it generates all samples inside the sample table's extent before advancing to the next region of the image that it covers. Figure 7.32 illustrates this idea.

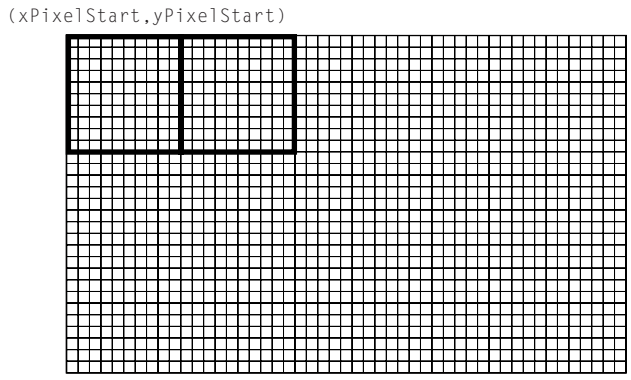


Figure 7.32: Because it is comprised of thousands of samples, the best-candidate sample table will generally cover a large number of pixels. One of the main tasks of the `BestCandidateSampler` is to tile this pattern over the image plane, mapping its sample positions to image sample positions.

BestCandidateSampler 344
LDSampler 328
Sampler 296
StratifiedSampler 304

⟨*BestCandidateSampler Declarations*⟩ ≡
```
class BestCandidateSampler : public Sampler {
public:
    ⟨BestCandidateSampler Public Methods 346⟩
private:
    ⟨BestCandidateSampler Private Data 344⟩
};
```

The sampler stores the current raster space pixel position in `xTableCorner` and `yTableCorner`, and `tableWidth` is the raster space width in pixels that the precomputed sample table spans. `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. 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 sample pattern precomputation step searches around a fixed-size region of 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 Method Definitions*⟩ ≡
```
BestCandidateSampler::BestCandidateSampler(int xstart, int xend,
                                           int ystart, int yend,
                                           int pixelSamples)
    : Sampler(xstart, xend, ystart, yend, pixelSamples) {
    tableWidth = (float)SQRT_SAMPLE_TABLE_SIZE / sqrtf(pixelSamples);
    xTableCorner = float(xPixelStart) - tableWidth;
    yTableCorner = float(yPixelStart);
    tableOffset = SAMPLE_TABLE_SIZE;
    ⟨BestCandidateSampler constructor implementation⟩
}
```

⟨*BestCandidateSampler Private Data*⟩ ≡ 344
```
int tableOffset;
float xTableCorner, yTableCorner, tableWidth;
```

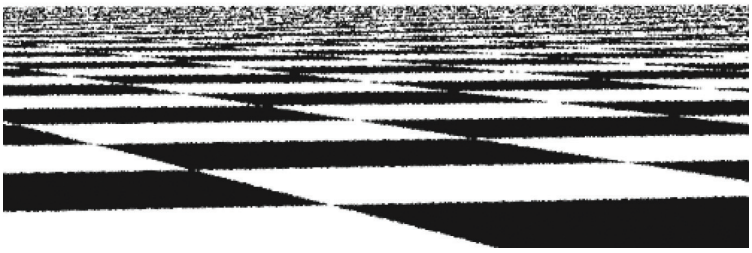
Here we incorporate the precomputed sample data stored in `samplers/sampledata.cpp`.

⟨*BestCandidateSampler Private Data*⟩ +≡ 344
```
static const float sampleTable[SAMPLE_TABLE_SIZE][5];
```

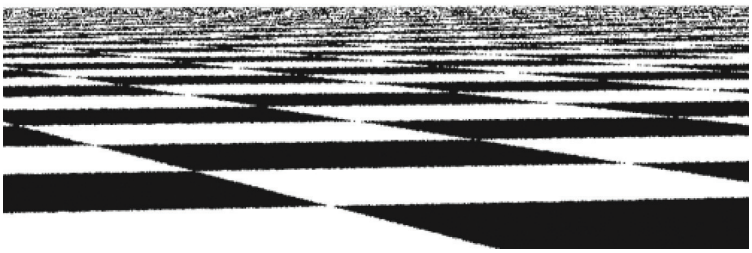
⟨*BestCandidateSampler Method Definitions*⟩ +≡
```
#include "sampledata.cpp"
```

This sampler usually generates stratified patterns for the samples for integrators (with one exception, explained shortly). In practice, these patterns work best with square-

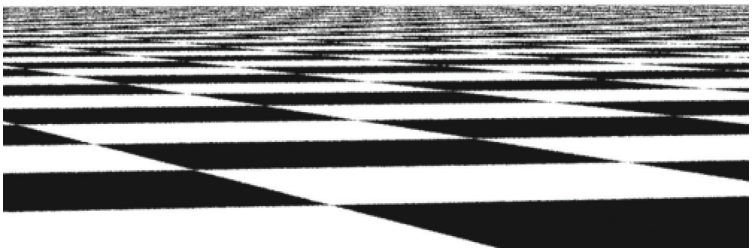
BestCandidateSampler 344
BestCandidateSampler::tableOffset 344
BestCandidateSampler::tableWidth 344
BestCandidateSampler::xTableCorner 344
BestCandidateSampler::yTableCorner 344
Sampler 296
Sampler::xPixelStart 298
Sampler::yPixelStart 298
SAMPLE_TABLE_SIZE 335
SQRT_SAMPLE_TABLE_SIZE 335



(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.

shaped strata from an equal number of samples in each direction, so the `RoundSize()` method rounds up sample size requests so that they are an integer number squared.

⟨*BestCandidateSampler Public Methods*⟩ ≡ 344
```
int RoundSize(int size) const {
    int root = Ceil2Int(sqrtf((float)size - .5f));
    return root*root;
}
```

The `BestCandidateSampler::GetNextSample()` method has a similar basic approach to the other samplers in this chapter, except that the sample pattern sometimes extends beyond the image's boundaries due to the way it is tiled. These out-of-bounds samples must be ignored, which can lead to multiple tries in order to find an acceptable sample.

⟨*BestCandidateSampler Method Definitions*⟩ +≡
```
bool BestCandidateSampler::GetNextSample(Sample *sample) {
again:
    if (tableOffset == SAMPLE_TABLE_SIZE) {
        ⟨Advance to next best-candidate sample table position 347⟩
    }
    ⟨Compute raster sample from table 349⟩
    ⟨Check sample against crop window, goto again if outside 349⟩
    ⟨Compute integrator samples for best-candidate sample 350⟩
    ++tableOffset;
    return true;
}
```

If it has reached the end of the sample table, the sampler tries to move forward by `xTableCorner`. If this leaves the raster extent of the image, it moves ahead by `yTableCorner`, and if this takes y beyond the bottom of the image, it is finished.

BestCandidateSampler 344
BestCandidateSampler::
GetNextSample() 346
BestCandidateSampler::
tableOffset 344
Ceil2Int() 856
Sample 299
SAMPLE_TABLE_SIZE 335

⟨*Advance to next best-candidate sample table position*⟩ ≡ 346
```
tableOffset = 0;
xTableCorner += tableWidth;
if (xTableCorner >= xPixelEnd) {
    xTableCorner = float(xPixelStart);
    yTableCorner += tableWidth;
    if (yTableCorner >= yPixelEnd)
        return false;
}
if (!oneDSamples) {
    ⟨Initialize sample tables and precompute strat2D values 348⟩
}
⟨Update sample shifts 348⟩
⟨Generate SAMPLE_TABLE_SIZE-sized tables for single samples 348⟩
```

Samples for integrators are handled here with a hybrid approach: If only a single sample of a particular type is needed per image sample, the sampler uses a value from a shuffled low-discrepancy sequence, extracted from an array of such samples computed for this section of the image. If multiple samples are needed, however, the sampler computes stratified samples for them.

There are three main reasons to use this approach. First, if the integrator needs a large number of samples per image sample, the storage to hold all of the sample values for the thousands of samples in the best-candidate table may be objectionable. Second, as the number of integration samples increases, the effect of not having samples that are well-distributed with respect to the neighbors is reduced, since the integrator samples at the current image sample cover the sample space well themselves. Third, the low-discrepancy sequences can cause some unusual artifacts when used with certain pixel reconstruction filters (as we will see shortly in Figure 7.36). It's still worthwhile to go through the trouble of using low-discrepancy samples for the single-sample case; this is a big help for the `PathIntegrator`, which uses many single samples like this, for example.

BestCandidateSampler::
  tableWidth 344
BestCandidateSampler::
  xTableCorner 344
BestCandidateSampler::
  yTableCorner 344
PathIntegrator 749
Sampler::xPixelEnd 298
Sampler::xPixelStart 298
Sampler::yPixelEnd 298

⟨*BestCandidateSampler Private Data*⟩ +≡ 344
```
float **oneDSamples, **twoDSamples;
int *strat2D;
```

⟨`BestCandidateSampler` *constructor implementation*⟩ ≡
```
oneDSamples = twoDSamples = NULL;
strat2D = NULL;
```

⟨*Initialize sample tables and precompute* strat2D *values*⟩ ≡ 347
```
oneDSamples = new float *[sample->n1D.size()];
for (u_int i = 0; i < sample->n1D.size(); ++i) {
    oneDSamples[i] = (sample->n1D[i] == 1) ?
        new float[SAMPLE_TABLE_SIZE] : NULL;
}
twoDSamples = new float *[sample->n2D.size()];
strat2D = new int[sample->n2D.size()];
for (u_int i = 0; i < sample->n2D.size(); ++i) {
    twoDSamples[i] = (sample->n2D[i] == 1) ?
        new float[2 * SAMPLE_TABLE_SIZE] : NULL;
    strat2D[i] = Ceil2Int(sqrtf((float)sample->n2D[i] - .5f));
}
```

The low-discrepancy samples for the single-sample case are computed with the shuffled random scrambled sampling routines defined earlier.

⟨*Generate* SAMPLE_TABLE_SIZE*-sized tables for single samples*⟩ ≡ 347
```
for (u_int i = 0; i < sample->n1D.size(); ++i)
    if (sample->n1D[i] == 1)
        LDShuffleScrambled1D(1, SAMPLE_TABLE_SIZE, oneDSamples[i]);
for (u_int i = 0; i < sample->n2D.size(); ++i)
    if (sample->n2D[i] == 1)
        LDShuffleScrambled2D(1, SAMPLE_TABLE_SIZE, twoDSamples[i]);
```

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 sampling patterns were computed with toroidal topology, 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.

⟨*Update sample shifts*⟩ ≡ 347
```
for (int i = 0; i < 3; ++i)
    sampleOffsets[i] = RandomFloat();
```

BestCandidateSampler::
oneDSamples 347
BestCandidateSampler::
sampleOffsets 349
Ceil2Int() 856
LDSampler 328
LDShuffleScrambled1D() 331
RandomFloat() 857
SAMPLE_TABLE_SIZE 335
StratifiedSampler 304

⟨*BestCandidateSampler Private Data*⟩+≡ 344
```
float sampleOffsets[3];
```

Computing the raster space sample position from the positions in the table just 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.

⟨*Compute raster sample from table*⟩≡ 346
```
#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
sample->imageX = xTableCorner + tableWidth *
                 sampleTable[tableOffset][0];
sample->imageY = yTableCorner + tableWidth *
                 sampleTable[tableOffset][1];
sample->time  = WRAP(sampleOffsets[0] +
                     sampleTable[tableOffset][2]);
sample->lensU = WRAP(sampleOffsets[1] +
                     sampleTable[tableOffset][3]);
sample->lensV = WRAP(sampleOffsets[2] +
                     sampleTable[tableOffset][4]);
```

BestCandidateSampler::
sampleOffsets 349
BestCandidateSampler::
sampleTable 344
BestCandidateSampler::
tableWidth 344
BestCandidateSampler::
xTableCorner 344
BestCandidateSampler::
yTableCorner 344
Sample::imageX 299
Sample::imageY 299
Sample::lensU 299
Sample::lensV 299
Sample::time 299
Sampler::xPixelEnd 298
Sampler::xPixelStart 298
Sampler::yPixelEnd 298
Sampler::yPixelStart 298

The sample table may spill off the edge of the image plane, so some of the generated samples may be outside the appropriate 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*⟩≡ 346
```
if (sample->imageX <  xPixelStart ||
    sample->imageX >= xPixelEnd   ||
    sample->imageY <  yPixelStart ||
    sample->imageY >= yPixelEnd) {
    ++tableOffset;
    goto again;
}
```

As explained previously, for integrator samples, the precomputed randomly scrambled low-discrepancy values are used if just one sample of this type is needed; otherwise a stratified pattern is used.

⟨*Compute integrator samples for best-candidate sample*⟩ ≡ 346

```
for (u_int i = 0; i < sample->n1D.size(); ++i) {
    if (sample->n1D[i] == 1)
        sample->oneD[i][0] = oneDSamples[i][tableOffset];
    else
        StratifiedSample1D(sample->oneD[i], sample->n1D[i]);
}
for (u_int i = 0; i < sample->n2D.size(); ++i) {
    if (sample->n2D[i] == 1) {
       sample->twoD[i][0] = twoDSamples[i][2*tableOffset];
       sample->twoD[i][1] = twoDSamples[i][2*tableOffset+1];
    }
    else
        StratifiedSample2D(sample->twoD[i], strat2D[i], strat2D[i]);
}
```

7.6 IMAGE RECONSTRUCTION

Given carefully chosen image samples, it is necessary to develop the infrastructure for converting 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.

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

BestCandidateSampler::
oneDSamples 347
BestCandidateSampler::
strat2D 347
BestCandidateSampler::
twoDSamples 347
Sample::n1D 301
Sample::n2D 301
Sample::oneD 301
Sample::twoD 301
StratifiedSample1D() 308
StratifiedSample2D() 308

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

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i) L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}, \tag{7.3}$$

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 7.35 shows a pixel at location (x, y) that has a pixel filter with extent `xWidth` in the x direction and `yWidth` in the y direction. All of the samples inside the

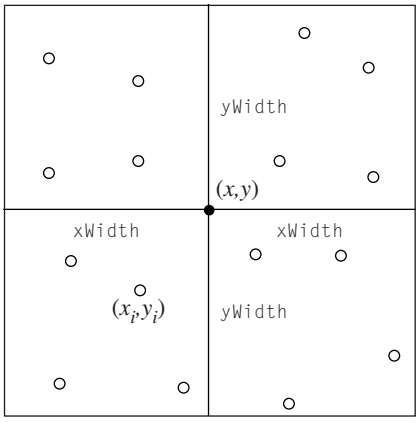
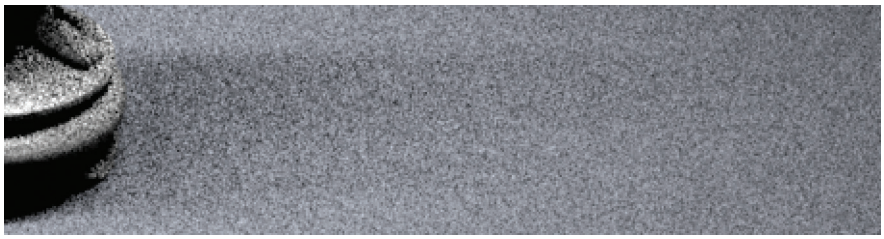


Figure 7.35: 2D Image Filtering. To compute a filtered pixel value for the pixel marked with a filled circle located at (x, y), all of the image samples inside the box around (x, y) with extent `xWidth` and `yWidth` need to be considered. Each of the image samples (x_i, y_i), denoted by open circles, 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.

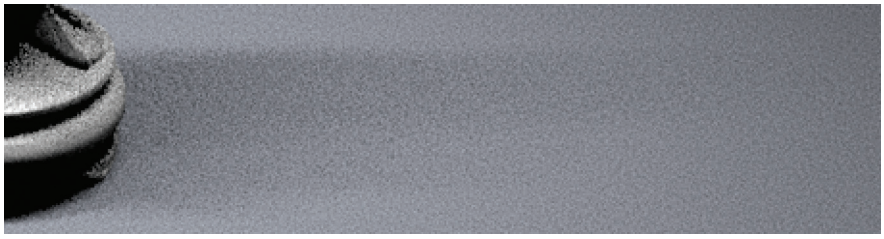
box given by the filter extent may contribute to the pixel's value, depending on the filter function's value for $f(x - x_i, y - y_i)$.

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 7.36 shows this effect in practice. Using this filter with regular stratified samples



(a)



(b)

Figure 7.36: 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.

`LDSampler` 328

`StratifiedSampler` 304

(a) (b) (c)

Figure 7.37: 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 bricks, for example.)

makes much less of a difference. Given the remarkable effectiveness of patterns generated with the `LDSampler`, this is something of a quandry: the box filter is the last filter we'd like to use, yet using it instead of another filter substantially improves the results from the `LDSampler`. Given all of these issues, `pbrt` provides a variety of different filter functions as plug-ins.

7.6.1 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 (7.3). The `Film` (described in the next chapter) stores a pointer to a `Filter` and uses it to filter the output before writing it to disk. Figure 7.37 shows comparisons of zoomed-in regions of images rendered using a variety of the filters from this section to reconstruct pixel values.

```
⟨Sampling Declarations⟩+≡
  class Filter {
  public:
      ⟨Filter Interface 354⟩
      ⟨Filter Public Data 354⟩
  };
```

Film 370
Filter 353
LDSampler 328

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 7.38).

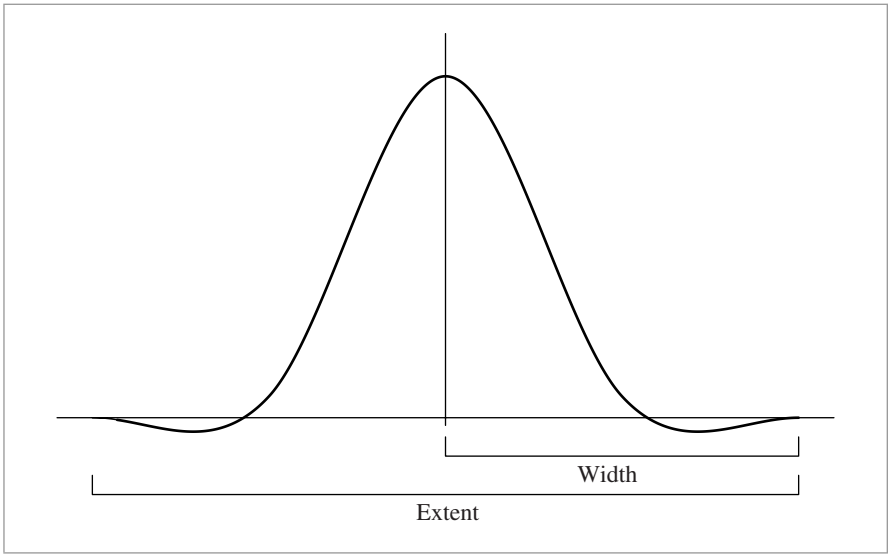


Figure 7.38: The extent of filters in `pbrt` is specified in terms of their width from the origin to its cutoff point. The support of a filter is its total nonzero extent, here equal to twice its width.

⟨*Filter Interface*⟩ ≡ 353
```
Filter(float xw, float yw)
    : xWidth(xw), yWidth(yw), invXWidth(1.f/xw), invYWidth(1.f/yw) {
}
```

⟨*Filter Public Data*⟩ ≡ 353
```
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*⟩ + ≡ 353
```
virtual float Evaluate(float x, float y) const = 0;
```

Box Filter

Filter 353

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

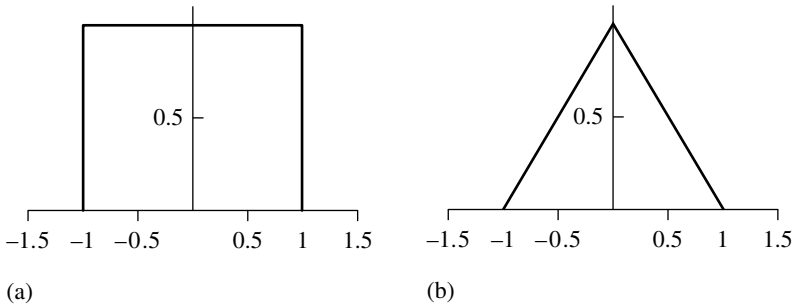


Figure 7.39: Graphs of the (a) box filter and (b) triangle filter. Although neither of these is a particularly good filter, they are both computationally efficient, easy to implement, and good baselines for evaluating other filters.

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 7.39(a) shows a graph of the box filter, and Figure 7.40 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 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.

BoxFilter 355
Filter 353

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

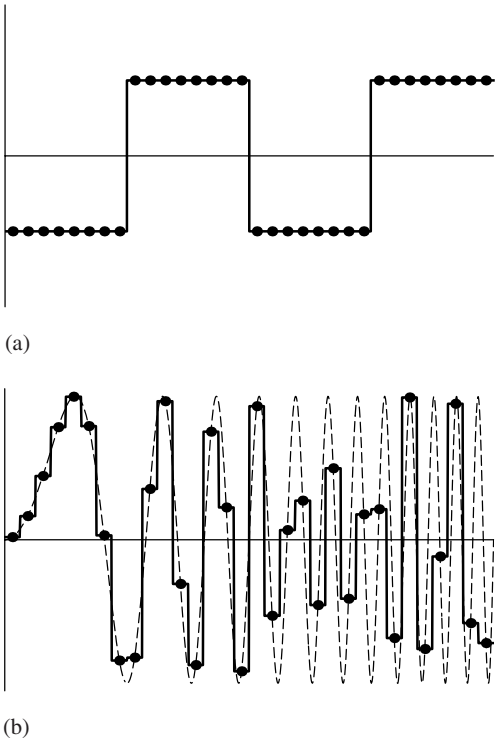


Figure 7.40: The box filter reconstructing (a) a step function and (b) a sinusoidal function with increasing frequency as x increases. This filter does well with the step function, as expected, but does an extremely poor job with the sinusoidal function.

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 7.39(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;
};
```

Filter 353
TriangleFilter 356

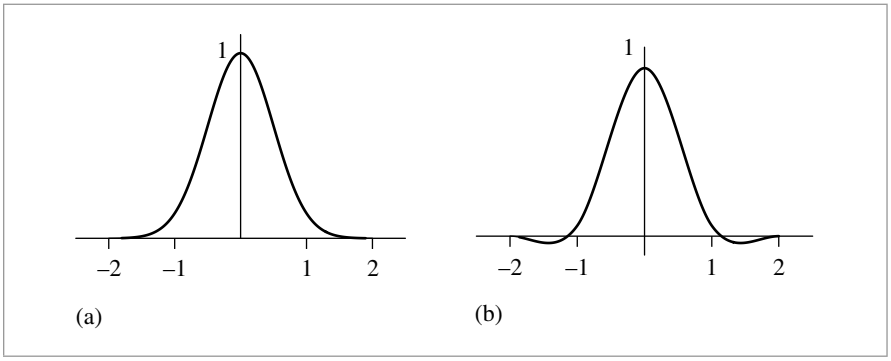


Figure 7.41: Graphs of (a) the Gaussian filter and (b) the Mitchell filter with $B = \frac{1}{3}$ and $C = \frac{1}{3}$, each with a width of two. The Gaussian gives images that tend to be a bit blurry, while the negative lobes of the Mitchell filter help to accentuate and sharpen edges in final images.

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 7.41). 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.

⟨*Gaussian Filter Declarations*⟩ ≡
```
class GaussianFilter : public Filter {
public:
    ⟨GaussianFilter Public Methods 358⟩
private:
    ⟨GaussianFilter Private Data 358⟩
    ⟨GaussianFilter Utility Functions 358⟩
};
```

Filter 353
Filter::xWidth 354
Filter::yWidth 354
TriangleFilter 356

The 1D Gaussian filter function of width w is

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

where α controls the rate of falloff of the filter. Smaller values cause a slower falloff, giving a blurrier image. For efficiency, the constructor precomputes the constant term for $e^{-\alpha w^2}$ in each direction.

⟨*GaussianFilter Public Methods*⟩ ≡ 357
```
GaussianFilter(float xw, float yw, float a)
    : Filter(xw, yw) {
    alpha = a;
    expX = expf(-alpha * xWidth * xWidth);
    expY = expf(-alpha * yWidth * yWidth);
}
```

⟨*GaussianFilter Private Data*⟩ ≡ 357
```
float alpha;
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*⟩ ≡ 357
```
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.

Filter 353
GaussianFilter 357
GaussianFilter::alpha 358
GaussianFilter::expX 358
GaussianFilter::expY 358

Note from the graph in Figure 7.41(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

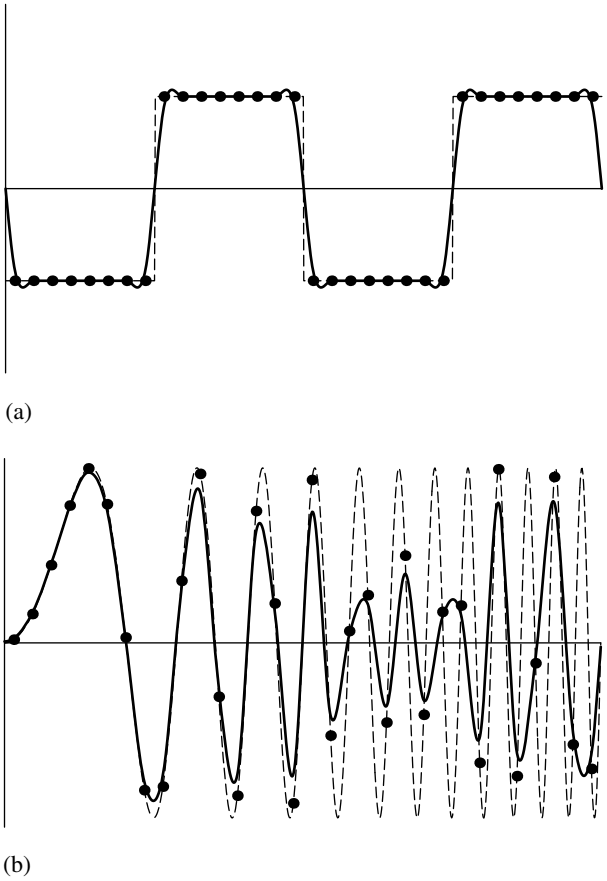


Figure 7.42: The Mitchell-Netravali Filter Used to Reconstruct the Example Functions. It does a good job with both of these functions, (a) introducing minimal ringing with the step function and (b) accurately representing the sinusoid until aliasing from undersampling starts to dominate.

can therefore become negative, they will eventually need to be clamped to a legal output range.

Figure 7.42 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 360⟩
private:
    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*⟩ ≡ 360
```
MitchellFilter(float b, float c, float xw, float yw)
    : Filter(xw, yw) { B = b; C = c; }
```

Like many 2D image filtering functions, including the earlier Gaussian filter, the Mitchell-Netravali filter is the product of one-dimensional filter functions in the x and y directions. Such filters are called *separable*. In fact, all of the provided filters in `pbrt` are separable. Nevertheless, the `Filter::Evaluate()` interface 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);
}
```

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

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

Filter 353
Filter::Evaluate() 354
Filter::invXWidth 354
Filter::invYWidth 354
MitchellFilter 360
MitchellFilter::
Mitchell1D() 361

⟨*MitchellFilter Public Methods*⟩+≡ 360

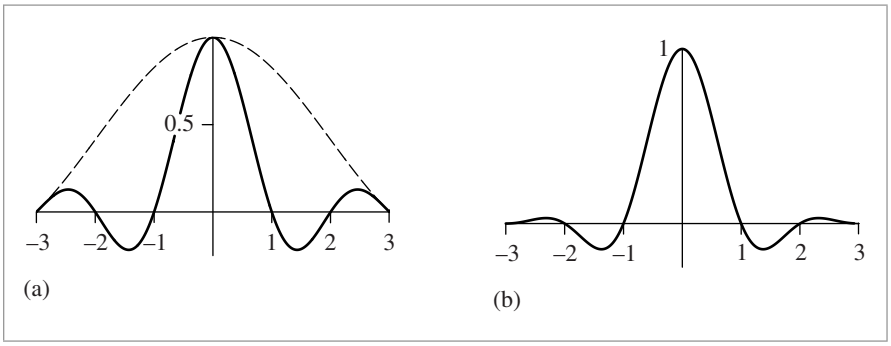
```
float Mitchell1D(float x) const {
    x = fabsf(2.f * x);
    if (x > 1.f)
        return ((-B - 6*C) * x*x*x + (6*B + 30*C) * x*x +
                (-12*B - 48*C) * x + (8*B + 24*C)) * (1.f/6.f);
    else
        return ((12 - 9*B - 6*C) * x*x*x +
                (-18 + 12*B + 6*C) * x*x +
                (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 7.43 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 7.43 also shows the filter that we will implement here, which is the product of the sinc function and the windowing function.



LanczosSincFilter 363

Figure 7.43: 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`.

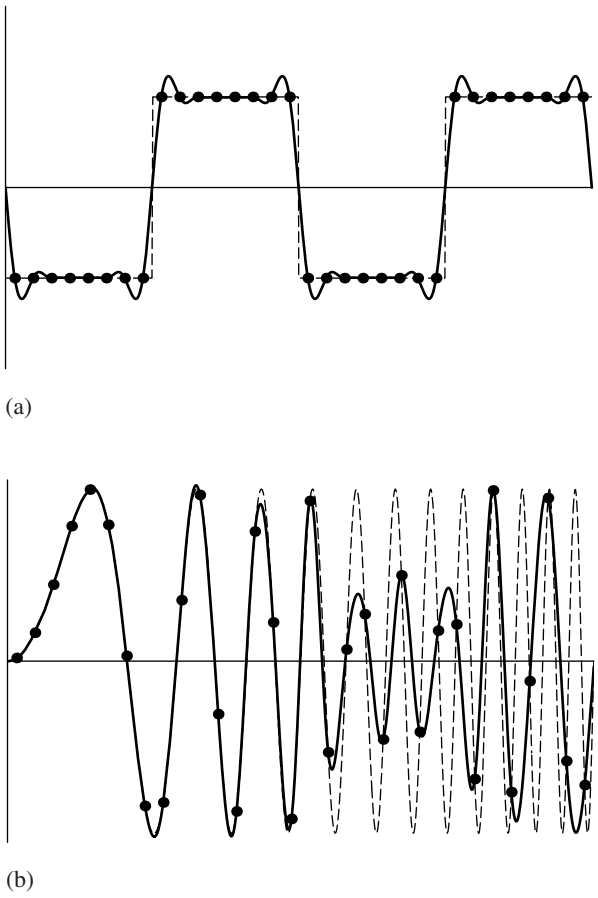


Figure 7.44: Results of Using the Windowed Sinc Filter to Reconstruct the Example Functions. Here $\tau = 3$. (a) Like the infinite sinc, it suffers from ringing with the step function, although there is much less ringing in the windowed version. (b) The filter does quite well with the sinusoid, however.

Figure 7.44 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(float xw, float yw, float t)
        : Filter(xw, yw) {
        tau = t;
    }
    float Evaluate(float x, float y) const;
    float Sinc1D(float x) const;
private:
    float tau;
};
```

Like the other filters, the sinc filter is also separable.

⟨*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.

⟨*Sinc Filter Method Definitions*⟩ + ≡
```
float LanczosSincFilter::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;
}
```

FURTHER READING

Filter 353
Filter::invXWidth 354
Filter::invYWidth 354
LanczosSincFilter 363
LanczosSincFilter::
  Sinc1D() 363
M_PI 855

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 surveys 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) has written an article that explains possible pitfalls when using floating-point coordinates for pixels and develops the conventions used here.

Mitchell has 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). Another efficient technique for generating Poisson disk patterns was developed by McCool and Fiume (1992). Hiller, Deussen, and Keller (2001) applied a technique based on relaxation that takes a random point set and improves its distribution. Cohen et al. (2003) showed how to use Wang tiles to quickly generate large point distributions that are a good approximation to a Poisson disk distribution.

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 pattern's 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 15. The seminal book on quasi-random sampling and algorithms for generating low-discrepancy patterns was written by Niederreiter (1992).

More recently, Keller and collaborators have 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 344
LDSampler 328
Sampler 296

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, although one of Mitchell's interesting findings was that low-discrepancy sampling sequences sometimes lead to visually objectionable artifacts in images that aren't present with other sampling patterns. However, Keller (2001) argues that because low-discrepancy patterns tend to converge more quickly than others, they are still preferable if one is generating high-quality imagery at a sampling rate high enough to eliminate artifacts.

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) has investigated how much better stratified sampling patterns are than random patterns in practice. In general, the smoother the function being sampled is, 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) have 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) reports 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).

EXERCISES

2 7.1 The *multijittered* sampling pattern developed by Chiu, Shirley, and Wang (1994) simultaneously satisfies Latin hypercube and stratified sampling properties. Although it isn't guaranteed to be well-distributed with respect to all elementary intervals, it is easy to implement and can generate good sampling patterns of any number of samples mn, $m \geq 1$, $n \geq 1$; it isn't limited to generating good distributions in quantities of powers of two. Use multijittered sampling patterns to improve the quality of the samples generated for the integrators by the `StratifiedSampler`.

For example, if an integrator requests six two-dimensional samples for each image sample and there are four image samples per pixel, generate a single multijittered pattern of 6×4 samples in each pixel and distribute the samples from the pattern to the image samples in a way that ensures that each image sample's set of samples is well-distributed on its own. Discuss good ways of distributing the multijittered samples to the image samples and discuss bad ways of doing this. Experiment with different approaches, and compare the results to the current implementation of the `StratifiedSampler` when computing direct lighting, for example.

2 7.2 Implement a new low-discrepancy sampler based on an n-dimensional Hammersley pattern, with an appropriate number of sample points such that the requested number of samples are taken per pixel, on average. Render images to compare its results to the samplers implemented in this chapter and discuss the relative merits of them, in theory and in practice. Include both in-focus images with high-frequency detail, in-focus images with smooth variation, out-of-focus images, and images with soft shadows or glossy reflection.

One subtlety in the implementation is handling nonsquare images. Why is scaling the Hammersley pattern in different amounts in the x and y direction to cover the image plane not a good idea? What is an effective way to handle nonsquare images with this sampler instead?

2 7.3 Surprisingly, reordering the series of samples provided by the `Sampler` can noticeably affect the performance of the system, even if doing so doesn't change the total amount of computation being done. This effect is caused by the fact that the coherence of the sampling on the image plane is related to the coherence of the rays traced through the scene, which in turn affects the memory coherence of the accesses to the scene data. Because caches on modern CPUs depend on coherent memory access patterns to give good performance, incoherent rays can reduce overall performance. If two subsequently traced rays are close to each other on the image, they are likely to access a similar subset of the scene data, thus improving performance.

`Sampler` 296
`StratifiedSampler` 304

Do a series of experiments to measure this effect. For example, you might want to write a new `Sampler` that sorts samples generated by another `Sampler` that it holds a pointer to internally. Try randomly ordering the samples and measure the resulting performance. Then, investigate techniques for sorting the samples to improve their coherence. How do the current samplers benefit differently from this sorting? Discuss reasons for this variation.

3 7.4 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.

3 7.5 Mitchell and Netravali (1988) note 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 11.1 may be fruitful for this effort.

Sampler 296