CHAPTER 24

USING BLUE NOISE FOR RAY TRACED SOFT SHADOWS

Alan Wolfe
NVIDIA

ABSTRACT

Ray tracing is nearly synonymous with noise. Importance sampling and low-discrepancy sequences can help reduce noise by converging more quickly, and denoisers can remove noise after the fact, but sometimes we still cannot afford the number of rays needed to get the results we want. If using more rays is not an option, the next best thing is to make the noise harder to see and easier to remove. This chapter talks about using blue noise toward those goals in the context of ray traced soft shadows, but the concepts presented translate to nearly all ray tracing techniques.

24.1 INTRODUCTION

In these early days of hardware-accelerated ray tracing, our total ray budgets are as low as they are ever going to be. Even as budgets increase, the rays are going to be eaten up by newer techniques that push rendering even further. Low sample count ray tracing is going to be a topical discussion for a long time.

Though many techniques exist to make the most out of every ray—such as importance sampling and low-discrepancy sequences—blue noise is unique in that the goal is not to reduce noise, but to make it harder to see and easier to remove.

Blue noise is a cousin to low-discrepancy sequences because they both aim to spread sample points uniformly in the sampling domain. This is in contrast to white noise, which has clumps and voids (Figure 24-1). Clumped samples give redundant data, and voids are missing data. Blue noise avoid clumps and voids by being roughly uniform in space.

Being blue noise or low discrepancy is not mutually exclusive, but this chapter is going to focus on blue noise. For a more thorough understanding of

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_24

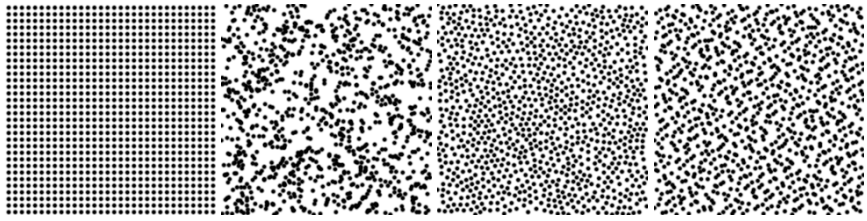

Figure 24-1. *Arrays of 1024 samples. Left to right: regular grid, white noise, blue noise, and the Halton sequence. Clumps of samples are redundant; gaps are missing information. Patterns can cause aliasing.*

modern sampling techniques, low-discrepancy sequences should be considered as well.

You may think that a regular grid would give good coverage over the sampling domain, but the diagonal distances between points on a grid are about 40% longer than non-diagonal distances, which makes it anisotropic, giving different results based on the orientation of what is being sampled. If you were to address that problem, you would get a hexagonal lattice, like a honeycomb. Using that for sampling, you could still have aliasing problems, though, and convergence may not be much better than a regular grid, depending on what you are sampling.

Blue noise converges at about the same rate as white noise, but has lower starting error. See Figure 24-2 for convergence rates. The power of blue noise is not in removing error, but making the error harder to see and easier to

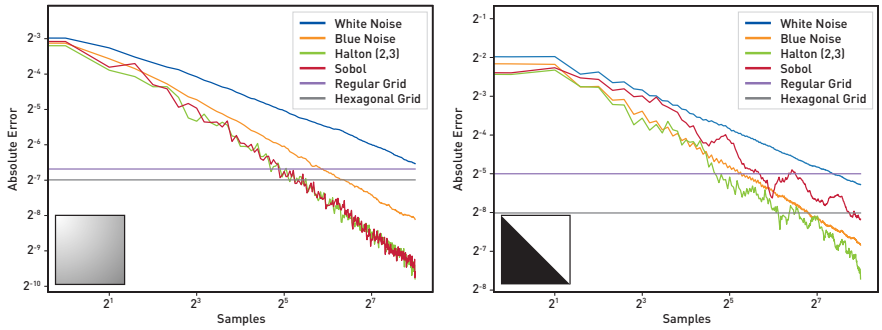

Figure 24-2. *Integrating two functions with 256 samples, with absolute error averaged over 1000 runs. Left: a crop of a Gaussian function. Right: a diagonal step function. Regular and hexagonal grids are not progressive, so only the final error is graphed.*

remove, compared to other sequences. It does this by having low aliasing and having randomization (noise) only in high frequencies.

24.2 OVERVIEW

We are going to go into more detail about blue noise samples in Section 24.3 and blue noise masks in Section 24.4. Then, we are going to look at blue noise from a digital signal processing perspective in Section 24.6.

After that, we are going to explore how to use both blue noise sampling and blue noise masks for ray traced soft shadows in Section 24.7. As a comparison, we will show an alternate noise pattern called interleaved gradient noise (IGN) presented by Jimenez [21], which is specifically designed for use with temporal antialiasing (TAA) [23] in Section 24.8, and then look at all of these things using the ꟻLIP perceptual error metric in Section 24.9.

24.3 BLUE NOISE SAMPLES

Two-dimensional blue noise samples are points in a square that are randomized but fill the square well, and are roughly uniform (often toroidally), as we see in Figure 24-3.

Samples can be blue noise distributed in other domains, too. On a sphere, blue noise would be points that covered the entire surface of the sphere and were randomized, but also were roughly uniform. These points on a sphere can also be viewed as blue noise distributed unit vectors. The similarity metric between vectors would be the negative dot product between those vectors instead of the distance between points on the surface of the sphere. Both perspectives would give equivalent samples, but show how blue noise conceptually can go beyond 2D points in a square.

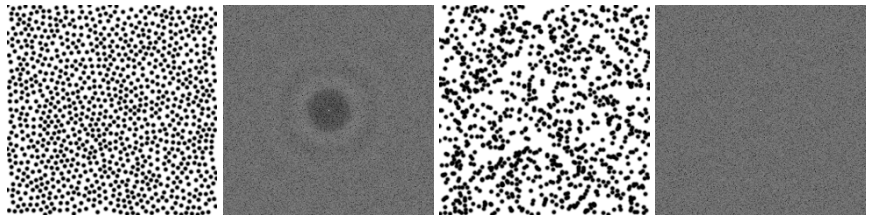

Figure 24-3. *For 1024 dots, blue noise samples (left) and discrete Fourier transform (DFT; middle left) showing attenuated low frequencies, and white noise samples (middle right) and DFT (right) showing all frequency content.*

```
samples = [];
for sampleIndex ∈ [0, NumSamples) do
    bestScore = -∞;
    bestCandidate = {};
    for candidateIndex ∈ [0, sampleIndex + 1) do
        score = ∞;
        candidate = GenerateRandomCandidate();
        for testSample ∈ samples do
            testScore = ToroidalDistance(candidate, testSample);
            score = min(score, testScore);
        if score > bestScore then
            bestScore = score;
            bestCandidate = candidate;
    samples.add(bestCandidate);
```

Figure 24-4. *Mitchell's best-candidate algorithm. `ToroidalDistance()` is the similarity metric being used.*

Generally speaking, blue noise samples can be thought of as blue noise in the form $\mathbf{v} = f(N)$, where N is an integer index into the samples and $\mathbf{v}$ could be a scalar, a point, a vector, or anything else in any sampling domain for which you could define a similarity metric.

Mitchell's best-candidate algorithm [15] can generate high-quality blue noise samples. The algorithm only requires that you are able to generate uniform random samples and have a similarity metric for pairs of samples. See the pseudocode in Figure 24-4. A more modern algorithm can be found in de Goes et al. [5]

Generating blue noise sample points is often computationally expensive. For this reason, blue noise sample points are usually generated in advance, and used as constants at runtime. Because blue noise is about squeezing the most quality out of visuals that you can, precalculating the samples lets you spend more time making better samples in advance, and then having an inexpensive runtime cost. A quicker dart throwing algorithm is compared to Mitchell's best-candidate blue in Figure 24-5, showing that the quicker algorithm does not give equivalent results.

Mitchell's best-candidate algorithm generates progressive blue noise, which means that any length of samples starting at index 0 are blue noise.

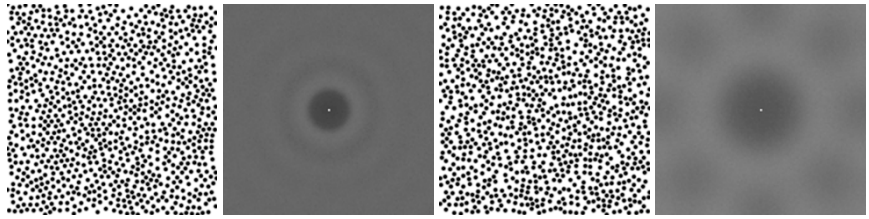

Figure 24-5. *For 1024 samples and 100 averaged DFTs, best-candidate samples (left) and dart throwing samples (right). Dart throwing samples look comparable to best-candidate samples (left half), but frequency analysis (right half) shows that they are significantly different.*

The more modern algorithms usually make non-progressive samples, which mean needing to use all samples before they are blue noise. You can make non-progressive samples progressive after they are generated, though, and the void and cluster algorithm [22] in the Section 24.5 does this as part of its work.

Progressive samples are useful when you do not know how many samples you want to take at runtime—e.g., if you want to take more samples where there is more variance. With progressive blue noise samples, you can generate some number of samples, then choose how many you want to use at runtime. You do not need to generate blue noise samples for every amount you may possibly want to sample, only the maximum amount.

An improved type of blue noise is found in Reinert et al. [19] called projective blue noise. If you generate 2D blue noise and look at the *x*- or *y*-values by themselves, they will look like white noise. If you were using your blue noise to sample a shadow cast by a vertical wall, or nearly vertical wall, the *y*-values of your samples would end up not mattering much to the result of your sampling, so that only the *x*-values mattered. If your *x*-axis looks like white noise, you are not going to do good sampling. Projective blue noise addresses this by making it so that blue noise samples are also blue on all axis-aligned subspace projections.

A nice property of blue noise is that it tends to keep its desirable properties better than other sequences when undergoing transformations like importance sampling [13]. That can be interesting when you have hybrid sampling types, such as the Sobol sequence that has blue noise projections [16].

There has been other progress on combining the integration speed of low-discrepancy sequences, while keeping the pleasing error pattern from blue noise in "Low-Discrepancy Blue Noise Sampling" [1], "A Low-Discrepancy Sampler That Distributes Monte Carlo Errors as a Blue Noise in Screen Space" [10], "Screen-Space Blue-Noise Diffusion of Monte Carlo Sampling Error via Hierarchical Ordering of Pixels" [2], "Orthogonal Array Sampling for Monte Carlo Rendering" [12], and "Progressive Multi-jittered Sample Sequences" [4].

24.4 BLUE NOISE MASKS

The other type of blue noise is blue noise masks, which are commonly referred to as blue noise textures. Blue noise textures are images where neighboring pixels are very different from each other. You can see a blue noise mask in Figure 24-6.

The most common type of blue noise mask is a 2D image with a single value stored at each pixel, but that is not the only way it can be done. You could have a 3D volume texture that stores a unit vector at each pixel, for instance, or any other sort of data on a grid you can imagine.

More generally, blue noise masks can be thought of as blue noise in the form $\mathbf{v} = f(\mathbf{u})$, where $\mathbf{u}$ is some discrete value of any dimension, and $\mathbf{v}$ could be a scalar, a point, a vector, or similar.

Generating blue noise of this form can be done using the void and cluster algorithm [22], which is detailed in Section 24.5. You can also download a zip file of textures made with the void and cluster method from the Internet [17]. How blue noise textures are usually used is that the textures are generated in advance, shipped as regular texture assets, and read by shaders at runtime. This is because, just like blue noise samples, blue noise textures are

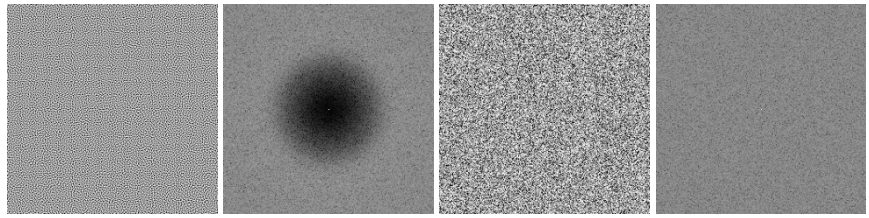

Figure 24-6. *Blue noise mask (left) and DFT (middle left) showing attenuated low frequencies. White noise mask (middle right) and DFT (right) showing all frequency content.*

computationally expensive to generate and can be made higher-quality when precomputed, giving high quality with low overhead at runtime.

There are other algorithms for generating blue noise masks. In Georgiev and Fajardo [7], a texture is initialized to white noise and pixel values are swapped if it reduces the energy function of the texture. Simulated annealing is used to help reach a better local optimum. This algorithm allows for vector-valued blue noise masks, unlike the void and cluster method, which is limited to scalar values.

Another way to make a blue noise mask is to start with white noise and high pass filter it. If you filter white noise, the histogram becomes uneven, though, and if you fix the histogram, it damages the frequency content. A decent way to get around this problem is to do several iterations of filtering the noise and fixing the histogram.

Using these alternate techniques, you can end up with blue noise that looks correct both as a texture and in frequency space, but they both lack an important detail that the void and cluster algorithm has.

Void and cluster blue noise masks have the useful property that thresholding the texture at 10% leaves 10% of the pixels remaining, and those pixels will be in a blue noise sample pattern. That is true for any threshold level. This can be useful if you want to use a blue noise texture for something like stochastic transparency because it will make the pixels that survive the alpha test be in a blue noise pattern.

To use a blue noise mask, the blue noise texture is tiled on the screen and used as the source of per-pixel random numbers. Doing this, the random numbers used by each pixel are going to be very different from neighboring pixels, producing very different results from neighbors, which is blue noise. Blue noise tiles well due to not having any low-frequency structure for your eye to pick up.

An example use of blue noise masks is when quantizing data to lower bit depths. If you add noise before quantization, it turns banding artifacts into noise, which looks more correct. Using blue noise instead of white noise means that the noise is harder to notice, and easier to remove. This can be useful when bringing high dynamic range (HDR) values from rendering into smaller bit depths for display. This can also be useful when quantizing data to G-buffer fields including color data like albedo, as well as non-color data such as normals.

Kopf et al. [14] has a technique that allows you to zoom in or out of blue noise, keeping the noise consistent hierarchically.

Some other details of blue noise masks and their usage are looked at in "The Rendering of INSIDE: High Fidelity, Low Complexity" [8], including triangular distributed blue noise that makes error patterns be more independent of signal, animating blue noise and dealing with blue noise dithering at the low and high ends of values to avoid a problem at the clamping zones.

24.5 VOID AND CLUSTER ALGORITHM

The void and cluster algorithm generates blue noise masks with the additional property that thresholding them causes the surviving pixels to be distributed as blue noise sample points. The algorithm is made up of the following steps:

1. Initial binary pattern.
2. Phase I: Make pattern progressive.
3. Phase II: First half of pixels.
4. Phase III: Second half of pixels.
5. Finalize texture.

The algorithm requires storage for each pixel to remember whether a pixel has been turned on or not, and an integer ordering value for that pixel. All pixels are initialized to being turned off and having an invalid ordering value.

The concept of voids and clusters are used to find where to add or remove points during the algorithm. Voids (gaps) are the low-energy areas in the image, and clusters are the high-energy areas.

Every pixel $\mathbf{p} = (p_x, p_y)$ that is turned on gives energy to every point $\mathbf{q}$ in the energy field using

$$E(\mathbf{p}, \mathbf{q}) = \exp\left(-\frac{\|\mathbf{p} - \mathbf{q}\|^2}{2\sigma^2}\right), \tag{24.1}$$

where $\mathbf{p}$ and $\mathbf{q}$ are the integer coordinates and distances are computed on wrapped boundaries, i.e., *toroidal* wrapping. The σ is a tunable parameter that controls energy falloff over distance, and thus frequency content.
Ulichney [22] recommends $\sigma = 1.5$. That equation may look familiar—it is just a Gaussian blur!

The algorithm can be sped up beyond a naive implementation by calculating the energy field using multithreading—calculating the energy for all pixels is an $O(N^4)$ operation where N is the side length of the output texture. It can be further sped up by actually using a texture to store the energy field as a quick lookup table, where updates are done to it iteratively as pixels are turned on or turned off. Lastly, though technically all pixels should be able to gain energy from all other pixels, in practice there is a radius at which the amount of energy gained is so negligible that it can be ignored. This means that based on the σ used in the Gaussian, you can calculate a pixel radius that contains the desired proportion of the Gaussian's energy. This lets you consider fewer pixels when calculating energy in the energy field. See the following equation for calculating the size (diameter) of the kernel for a given σ, where t is the threshold and a t value of 0.005 means all but 0.5% of the energy is accounted for:

$$\left\lfloor 1 + 2 * \sqrt{-2\sigma^2 \ln t} \right\rfloor . \tag{24.2}$$

24.5.1 INITIAL BINARY PATTERN

The first step is to generate an initial *binary* pattern where not more than half of the pixels are turned on. This can be done using white noise or nearly any other pattern. These pixels need to be transformed into blue noise points before continuing. That is done by repeatedly turning off the largest-valued cluster pixel and turning on the lowest-valued void pixel. This process is repeated until the same pixel is found for both operations. At that point, the algorithm has converged and the initial binary pattern is blue noise distributed.

The reason not more than half of the pixels should not be turned on in this step is because there is different logic required when processing the second half of the pixels compared to the first half. The logic for the first half of the pixels works better for sparser points, whereas the logic for the second half works better for denser points.

24.5.2 PHASE I: MAKE PATTERN PROGRESSIVE

The initial binary pattern is now blue noise distributed, but has no ordering and must be made into a progressive blue noise sequence. This is done by repeatedly removing the highest-energy pixel, i.e., the largest *cluster*, and giving that pixel an ordering of how many pixels are on after it is turned off.

When this is done, the pixels in the initial binary pattern are turned back on. At this point, the initial binary pattern pixels have an ordering that makes them an ordered (progressive) blue noise sampling sequence and we are ready for phase II.

24.5.3 PHASE II: FIRST HALF OF PIXELS

Next, pixels are turned on, one at a time, until half of the pixels are turned on. This is done by finding the lowest-energy pixel, i.e., the smallest *void*, and turning that pixel on. The ordering given to that pixel is the number of pixels that were on before it was turned on.

24.5.4 PHASE III: SECOND HALF OF PIXELS

At this point, the states of all the pixels are reversed. Pixels that are on are turned off, and vice versa. This phase repeatedly finds the largest-valued cluster and turns it off, giving it the ordering of the number of pixels that were off before it was turned off. When there are no more pixels turned off, this phase is finished and all pixels have an ordering.

24.5.5 FINALIZE TEXTURE

After all pixels are ordered, that ordering is turned into pixel values in the output image. If the output image has a resolution of 256×256 pixels, then their ordering will go from 0 to 65,535. If the output is an 8-bit image, the values will need to be remapped from 0 to 255. This will create non-unique values in the output texture, but there will be an equal number of each pixel value—256 of each.

You now have a high-quality blue noise mask texture ready for use!

24.6 BLUE NOISE FILTERING

When stopping before convergence in rendering, there is going to be error left behind in the image. If using a randomized sampling pattern like white noise or blue noise, the error will be left as a randomized noise pattern, and usually you want to remove it as much as possible.

The most common way to remove noise is to blur the image. In digital signal processing terms, a blur is a low pass filter, meaning it lets low frequencies through the filter, while reducing or removing high-frequency details.

Blurring is most commonly done by convolving an image against a blur kernel. This is mathematically the same as if we took the image and the blur

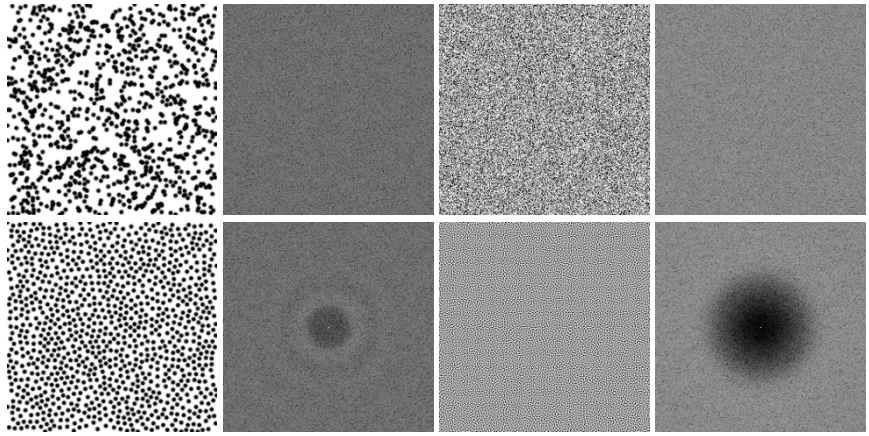

Figure 24-7. *Noise and DFT for noise samples and masks. Top: white noise. Bottom: blue noise.*

kernel into frequency space, multiplied them together, and brought the result out of frequency space again.

In Figure 24-7, you can see how white noise is present in all frequencies, while blue noise is only present in higher frequencies. In Figure 24-8 you can see how various blur kernels look in frequency space. When blurring noise, these are the things getting multiplied together in frequency space.

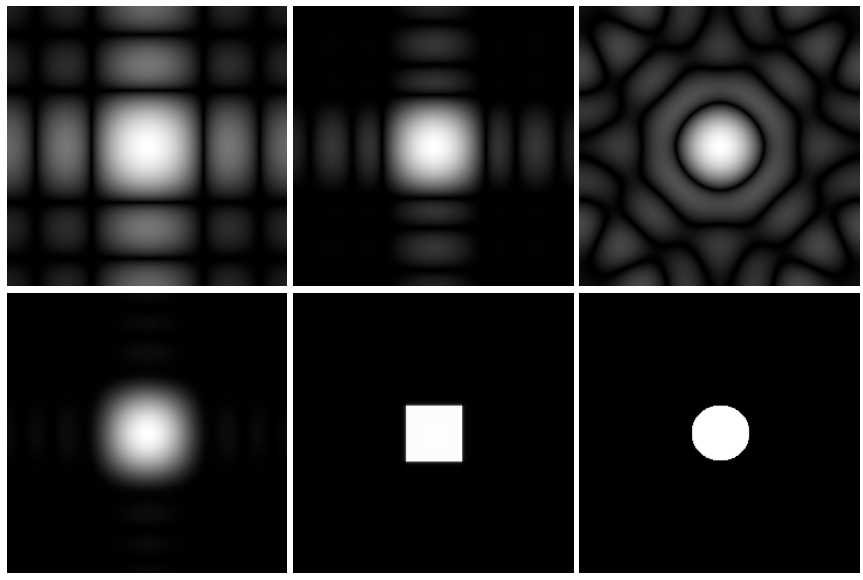

Figure 24-8. *DFT of various blur or low pass filter kernels. Clockwise from upper left: box, a trous, disk, circular sinc, sinc, and Gaussian.*

You can see how multiplying the white noise by any of the kernels is going to leave something that looks like white noise but is in the shape of the filter used, which is going to leave noise of those frequencies behind in the image. The only way you would be able to get rid of white noise completely is by removing all frequencies, which would leave nothing of your render.

Alternatively, you can see how if you were to multiply blue noise in frequency space by the Gaussian kernel or a sinc kernel, the result will be noiseless because the place where the blur kernel is white, the blue noise is black. There is no overlap, so the blue noise will have gone away.

Noisy renders are a mix of data and noise, so when you do a low pass filter to remove the blue noise from the result, it will also remove or reduce any data that is also in the frequency ranges that the blue noise occupies, but this is a big improvement over white noise, which is present in all frequencies.

From a digital signal processing perspective, the ideal isotropic low pass filter in 2D is the circular sinc filter, but from the blue noise filtering perspective, the ideal kernel will look like the missing part of the blue noise frequencies, so would be bright in the center where the low frequencies are and then fade out circularly to the edge of where the blue noise has full amplitude. Looking at these kernels, it seems that a Gaussian blur is the best choice.

If using the other blurs, you may keep frequencies that you did not intend to keep, or you may remove frequencies that you did not need to remove from the render. When doing a Gaussian blur, you want to choose σ such that the filter in frequency space is roughly the same size as the frequency hole in your blue noise. If it is too small, it will leave some blue noise residue, which will look like blobs, and if it is too large, it will remove more of the details than is needed. You can also choose to leave some blue noise residue behind if that preserves sharper details that you care about.

24.7 BLUE NOISE FOR SOFT SHADOWS

Now it is finally time to put all this together into a rendering technique!

24.7.1 LIGHTS AND SHADOWS

When ray tracing shadows, you use ray tracing to answer the question of whether a specific point on a surface can see a specific point on a light. If it can, you add the lighting contribution to the surface.

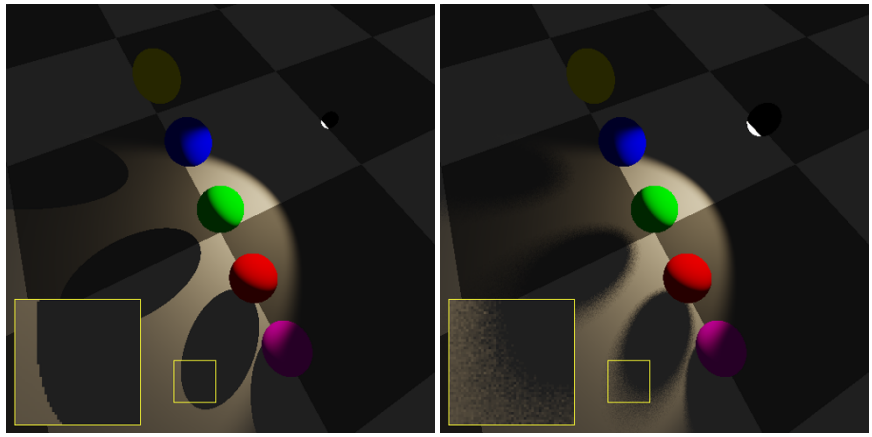

Figure 24-9. *Left: discrete light shadows are noiseless but hard at one ray per pixel. Right: area lights are soft but noisy in the penumbra at 16 white noise samples per pixel.*

Classically, the most common type of light in real-time graphics has been discrete lights, which are infinitely small and come in the usual flavors of positional lights, directional lights, and spotlights. When checking to see if a point on a surface can see a light, there is only one single ray direction toward which to shoot a ray and get a Boolean answer of yes or no. For this reason, when you ray trace shadows for discrete lights, they are completely noiseless at one sample per pixel, but they have hard shadow edges as you can see in Figure 24-9, left.

In more recent times, real-time graphics has moved away from discrete lights toward area lights, which have area and volume. These lights also come in the usual flavors of positional lights, directional lights, and spotlights, but also have others such as capsule lights, line lights, mesh lights, and even texture-based lights, to give variety and added realism to lighting conditions. Because these lights have volume, handling shadows is not as straightforward. The answer is no longer a Boolean visibility value but is more complex because you need to integrate every visible part of the light with the surface's BRDF. Not only will the surface react to the light differently at different angles, the light may shine different colors and intensities in different locations on the light, or in different directions from the light.

The correct way to handle this is discussed by Heitz et al. [11], but we are going to simplify the problem and use ray tracing to tell us the percentage of the light that can be seen by a point on a surface, then use that as a shadow

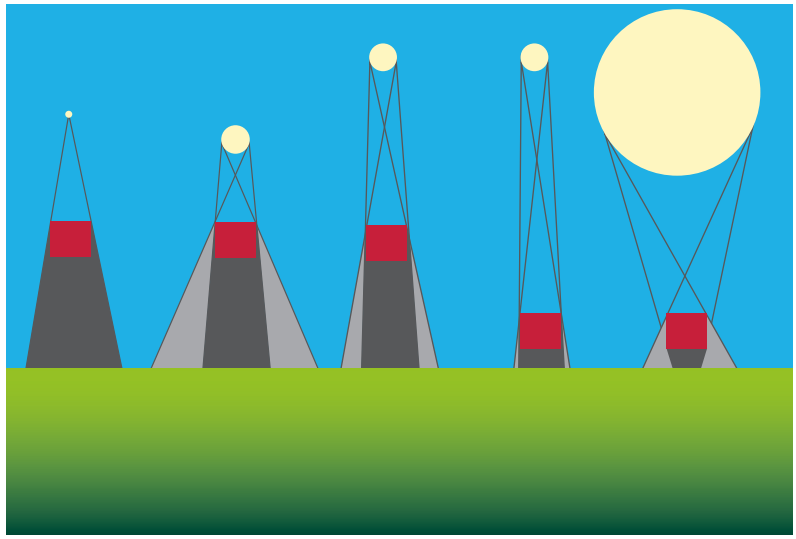

Figure 24-10. *Geometry of shadows. Dark gray is full shadow (umbra), and light gray is partial shadow (penumbra).*

multiplier for the light's contribution to shading. To do this, we will just shoot rays from the surface that we are shading to random points on the light source and calculate the percentage of rays that hit the light. We are also going to limit our examples to spherical positional lights, spherical spotlights, and circular directional lights, but the techniques apply to other light shapes and light types.

When we shoot multiple rays toward different places on a light source, we start to get noise at the edges of the shadow. We get noise in the penumbra, which is Latin for "almost a shadow" (Figure 24-9, right). The penumbra is the only place where there is noise because all other places are either completely in light or completely in shadow (umbra) and all rays will agree on visibility of the light. The size of the penumbra is based on the size and distance of the light source, as well as the size and distance of the shadow caster. Figure 24-10 shows the geometric relationships.

24.7.2 SPHERICAL DIRECTIONAL LIGHTS

The sun is a good example of a spherical directional light—a glowing ball that is so far away that no matter where you move, it will always be at the same point, angle, and direction in the sky.

We see the ball as a circle, though, so to do ray traced shadows, we are going to shoot rays at points on that circle and see what percentage were blocked for our shadowing term.

Doing this ends up being easy. We need to define the direction to the light and the solid angle radius of the circle in the sky, then we can use this GLSL code to get a direction in which to shoot a ray for a single visibility test:

```
// rect.x and rect.y are between 0 and 1.
vec2 MapRectToCircle(in vec2 rect)
{
  float radius = sqrt(rect.x);
  float angle = rect.y * 2.0 * 3.14159265359;
  return vec2(
    radius * cos(angle),
    radius * sin(angle)
  );
}

// rect.x and rect.y are between 0 and 1. direction is normalized direction
     to light. radius could be ~0.1.
vec3 SphericalDirectionalLightRayDirection(in vec2 rect, in vec3 direction,
     in float radius)
{
  vec2 point = MapRectToCircle(rect) * radius;
  vec3 tangent = normalize(cross(direction, vec3(0.0, 1.0, 0.0)));
  vec3 bitangent = normalize(cross(tangent, direction));
  return normalize(direction + point.x * tangent + point.y * bitangent);
}
```

To shoot eight shadow rays for a pixel, you would generate eight random vec2s, put them through `SphericalDirectionalLightRayDirection()` to get eight ray directions for that light, and multiply the percentage of hits by the lighting from this light. That gives you ray traced soft shadows from area lights.

As we know from previous sections though, not all random numbers are created equal!

The first thing we are going to do is use blue noise samples in a square, instead of white noise. If we use the same samples per pixel, we get the strange patterns you see in Figure 24-11.

The typical way to address all pixels using the same sampling pattern is to use Cranley–Patterson rotation: you use a per-pixel random number to modify the samples in some way to make them different. Instead of using regular random numbers, we can instead read a screen-space tiled blue noise mask texture as our source of per-pixel random numbers. We need two blue noise values, so we will read at (x, y) and will also read at an offset

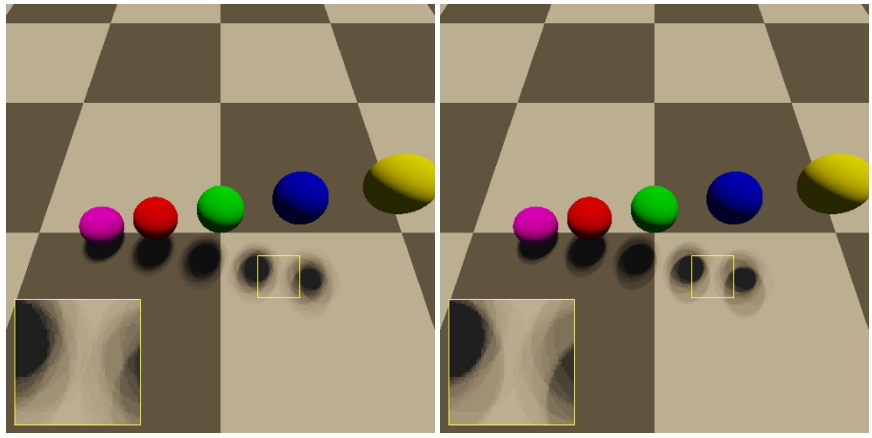

Figure 24-11. *The same 16 samples for each pixel. Left: blue noise. Right: white noise.*

$(x + 13, y + 41)$ to get two values between 0 and 1. The two values read from the texture are an (x, y) offset that we will add to each 2D blue noise sample, and we use `fract` to keep them between 0 and 1. When we do that, we get what we see in the leftmost column of Figure 24-12.

The offset of (13, 41) is somewhat arbitrary, but it was chosen to be not very close on the blue noise texture to where we read previously. Blue noise textures have correlation between pixels over small distances, so if you want uncorrelated values, you don't want to read too closely to the previous read location.

What we have looks good for still images, but in real-time graphics we also have the time dimension. For algorithms that temporally integrate rendering, such as temporal antialiasing (TAA) or deep learning super sampling (DLSS), using the same blue noise every frame is not giving them any new information, which leaves image quality unrealized. Even without temporal integration algorithms, better temporal sampling may look better to our eyes, or be implicitly integrated by the display. Individual screenshots will not look as nice if they aren't filtered over time, however.

The most straightforward way to animate these soft shadows would be to have something like eight different blue noise textures instead of one and to use the texture `frameNumber % 8` for a specific frame. This would animate the blue noise ray traced soft shadows, and every single frame would be good quality, but the problem is that looking at an individual pixel, it would become white noise over time. This is because each blue noise texture was made

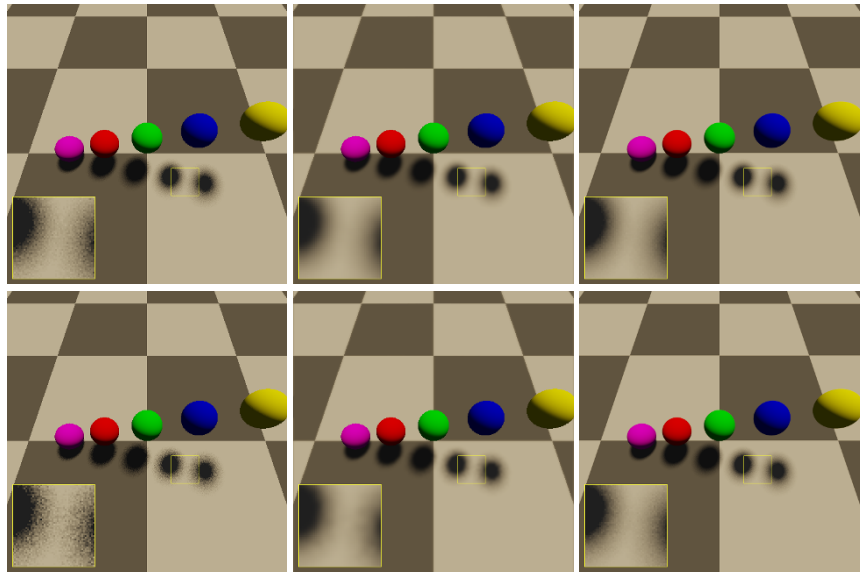

Figure 24-12. *Example of spherical directional light. Top: blue noise. Bottom: white noise. Left to right: raw, depth-aware Gaussian blur, and temporal antialiasing. Notice how the Gaussian blurred penumbra of the white noise looks much lumpier than the blue noise. This is because blue noise does not have frequencies low enough to survive the low pass filter, but white noise does.*

independently of the others. We know that white noise sampling is among the worst you can do usually, so we should be able to do better.

Another way would be to offset the texture reads by some amount every frame, essentially having a global texel offset for the pixels to add to their read location into the blue noise. This is what was done in INSIDE [8] using a Halton sequence to get the offset. Blue noise has correlation over small distances, but not over large distances, so doing this with a low discrepancy sequence results in white noise over time. This gives similar results to flipping through multiple textures, but has the benefit of only being a single texture, and you can have a flip cycle up to as long as the number of pixels in your texture.

Another way to animate blue noise is to read a blue noise texture, add the frame number multiplied by the golden ratio to it, then use modulus to keep it between 0 and 1. What this does is make each pixel use the golden ratio additive recurrence low-discrepancy sequence on the time axis, which is a high-quality 1D sampling sequence, making the time axis well-sampled. Unfortunately, this comes at the cost of modifying frequency content, so it damages the quality over space a little bit. Overall, it is a net win, though, and

this is the method that we are going to use. You can see this animated noise under TAA compared to white noise in the right most column of Figure 24-12.

```
float AnimateBlueNoise(in float blueNoise, in int frameIndex)
{
  return fract(blueNoise + float(frameIndex % 32) * 0.61803399);
}
```

You might think that the correct way to animate a 2D blue noise mask would be to make a 3D volumetric blue noise texture, but surprisingly, it is not, and 2D slices of 3D blue noise are not good blue noise at all [18]. You might also think that you could scroll a blue noise texture over time because neighboring pixels are blue noise in relation to each other, so that should make samples blue over time. This ends up not working well because 1D slices of 2D blue noise are also not good 1D blue noise, so you do not get good sampling over time; also, there will be a correlation between your samples along the scrolling direction.

Besides temporal integration and good noise patterns over time, you can also look at denoising a single frame in isolation. In the middle column of Figure 24-12, we use a depth-aware Gaussian blur on the lighting contributions before multiplying by albedo to get the final render.

At this point, we have spherical directional lights that have decent noise characteristics over both space and time.

24.7.3 SPHERICAL POSITIONAL LIGHTS

Spherical positional lights are glowing balls in the world. As you move around, the direction to them changes, and so does their brightness through distance attenuation. You can see one rendered in Figure 24-13.

Starting with the spherical directional light shader code, we only need to make two changes:

1. Light direction is no longer constant but is whatever the direction from the surface to the center of the light is.

2. The radius of the light circle is no longer constant either but changes as you get closer or farther from the light. Though being the correct thing to do, this also handles distance attenuation implicitly.

Here is the spherical directional light code modified to calculate the direction and radius as the first two lines, instead of taking them as parameters to the function. The rest works the same as before.

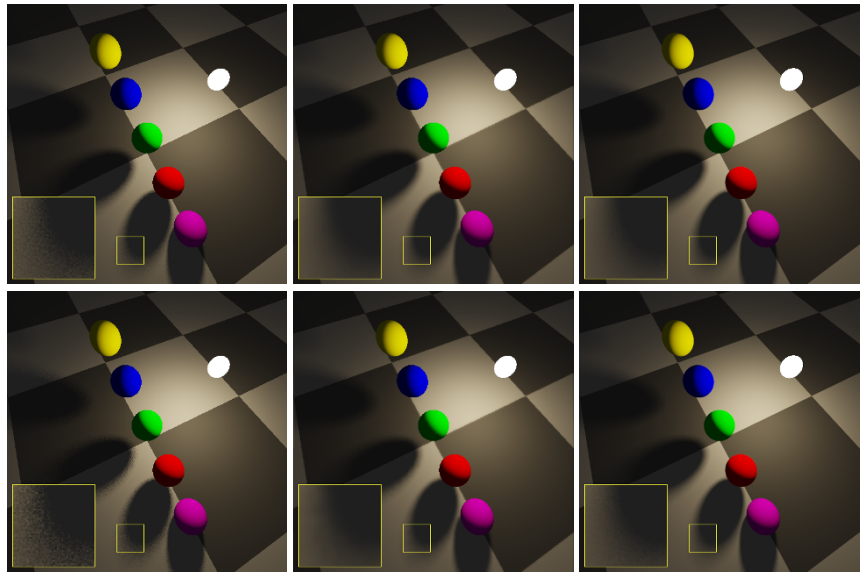

Figure 24-13. *Example of spherical positional light. Top: blue noise. Bottom: white noise. Left to right: raw, depth-aware Gaussian blur, and TAA.*

```
// rect.x and rect.y are between 0 and 1. surfacePos and lightPos are in
    world space.  worldRadius is in world units and could be ~5.
vec3 SphericalPositionalLightRayDirection(in vec2 rect, in vec3 surfacePos,
    in vec3 lightPos, in float worldRadius)
{
  vec3 direction = normalize(lightPos - surfacePos);
  float radius = worldRadius / length(lightPos - surfacePos);

  vec2 point = MapRectToCircle(rect) * radius;
  vec3 tangent = normalize(cross(direction, vec3(0.0, 1.0, 0.0)));
  vec3 bitangent = normalize(cross(tangent, direction));
  return normalize(direction + point.x * tangent + point.y * bitangent);
}
```

24.7.4 SPHERICAL SPOTLIGHTS

Last comes spherical spotlights, which are just like spherical positional lights except that they do not emit light from all angles. You can see one rendered in Figure 24-14.

To account for this, we are going to modify the spherical positional light code to take a spotlight direction as a parameter to control where the light is shining, and a cosine inner and cosine outer parameter to control how focused it is in that direction.

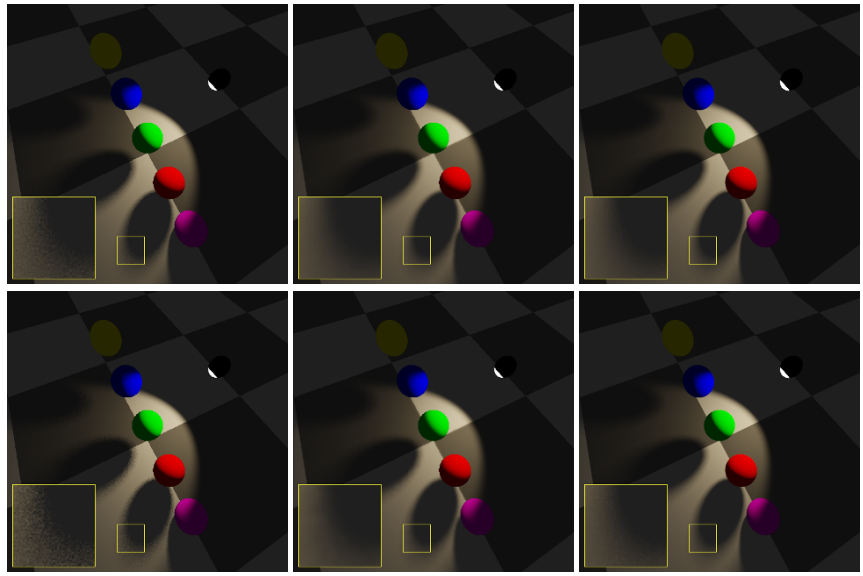

Figure 24-14. *Example of spherical spotlight. Top: blue noise. Bottom: white noise. Left to right: raw, depth-aware Gaussian blur, and TAA.*

We are going to dot product the direction from the light to the surface, by the direction in which the light shines. If it is less than the cosine inner value, it is fully bright. If it is greater than the cosine outer value, it is fully dark. If it is in between, we will use `smoothstep` (a cubic spline) to interpolate between fully bright and fully dark. This brightness value will be returned to the caller, which can be multiplied by the shadow sample result (0 or 1) before being averaged into the final result.

```
// rect.x and rect.y are between 0 and 1. surfacePos and lightPos are in
      world space.  worldRadius is in world units and could be ~5. angleAtten
       will be between 0 and 1.
vec3 SphericalSpotLightRayDirection(in vec2 rect, in vec3 surfacePos, in
      vec3 lightPos, in float worldRadius, in vec3 shineDir, in float
      cosThetaInner, in float cosThetaOuter, out float angleAtten)
{
  vec3 direction = normalize(lightPos - surfacePos);
  float radius = worldRadius / length(lightPos - surfacePos);

  angleAtten = dot(direction, -shineDir);
  angleAtten = smoothstep(cosThetaOuter, cosThetaInner, angleAtten);

  vec2 point = MapRectToCircle(rect) * radius;
  vec3 tangent = normalize(cross(direction, vec3(0.0, 1.0, 0.0)));
  vec3 bitangent = normalize(cross(tangent, direction));
  return normalize(direction + point.x * tangent + point.y * bitangent);
}
```

24.7.5 REDUCING RAY COUNT

One way to reduce the number of rays used for shadow rays is to shoot a few rays to start and, if they all agree on visibility, return the result as the answer. This makes an educated guess about the surface being either fully lit or fully shadowed. The more rays used, the more often this is correct, but the more expensive it is. If those few rays do not all agree, you know that you are in the penumbra and so should shoot more rays for better results.

You can also use a hybrid approach where you have a shadow map, but you only use it to tell if you are in the penumbra or not. You would do this by reading a shadow map value and, if you get a full 0 or 1 back as a result, use that without shooting any rays; otherwise, you would shoot your rays to get the penumbra shadowed value. Regular shadow maps are made from discrete light sources though, so for best results you probably would want to use something like percentage-closer soft shadows (PCSS) [6].

You can reduce the ray count all the way to one sample per pixel (spp) and still get somewhat decent results, as shown in Figure 24-15. You can take it below 1 spp by doing a ray trace for shadows smaller than full resolution, then filtering or interpolating the results for each individual pixel.

24.7.6 REDUCING NOISE

Noise is more apparent where there is higher contrast between the shadowed and unshadowed areas. In real life, shadows are almost never completely black, but instead have indirect global illumination bounce lighting of some amount in them. In this way, global illumination, ambient lighting, or similar can help reduce the appearance of noise without reducing the noise, but just reducing contrast.

Sample count also affects the contrast of your noise. If you only take one sample per pixel, you can only have 0 or 1 as a result, which means that the pixel is fully bright or fully in shadow, so your noise will all be either fully bright or fully in shadow. If you take two samples per pixel, you now have three values: fully in shadow, half in shadow, and fully in light. The noise in your shadows will then also have those three values and will have less contrast. The more samples you add, the more shades you have, with the shade count being the number of rays plus one.

Though we've been talking about shooting a ray toward a uniform random point on a light source, there are ways to sample area light sources that result

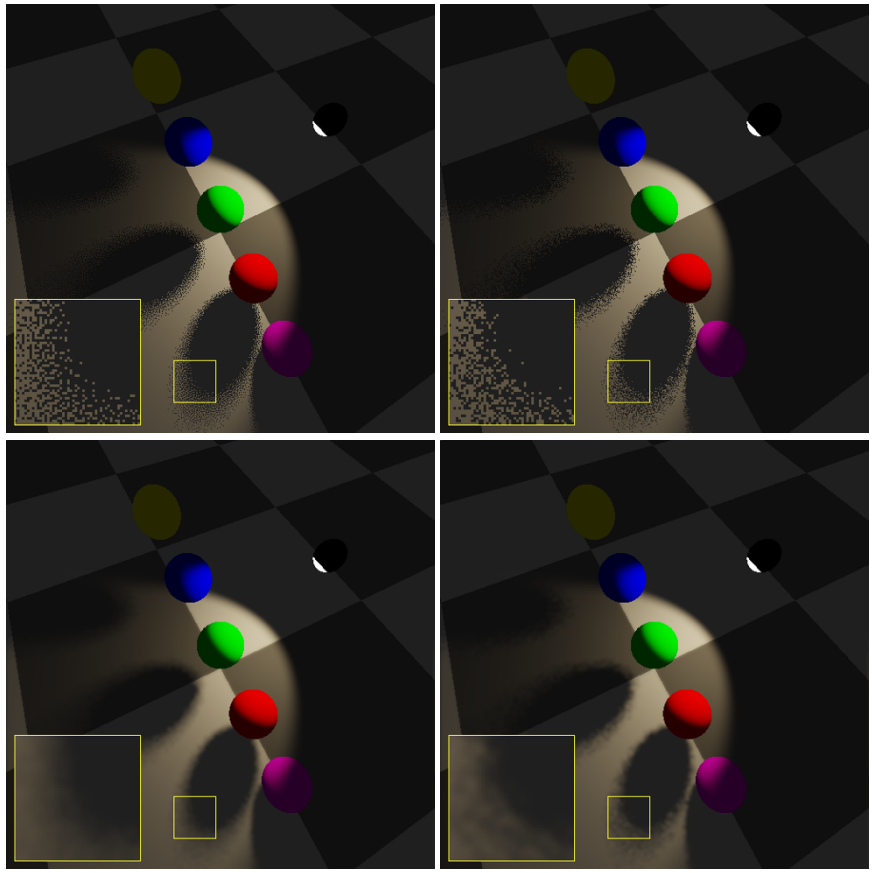

Figure 24-15. *One spp shadows. Top: raw. Bottom: depth-aware Gaussian blurred with $\sigma = 2$. Left: blue noise. Right: white noise.*

in less noise, to which you can then apply blue noise. You can read about some by Hart et al. [9] and Schutte [20].

When denoising ray traced shadows, it is best to keep your albedo, diffuse lighting, and specular lighting in separate buffers so that you can denoise your diffuse and specular lighting separately, before multiplying by albedo to get the final render. If you denoise the final render directly with a simpler denoising technique such as a depth-aware Gaussian blur, you will also end up blurring details in your albedo that come from textures, despite them not being noisy in the first place.

This chapter has focused on combining blue noise samples with blue noise masks, but you can use either of these things independently. For instance,

you might find that you get better convergence rates by using another sampling sequence per pixel, but still using a blue noise mask to vary the sampling sequence per pixel, making the resulting error pattern be blue noise in screen space.

Noise appears only in penumbras, and smaller light sources make smaller penumbras, so you can try shrinking your lights if you have too much noise. Glancing angles of lighting also make for longer penumbras, and larger areas of noise, like the sun being low in the sky, making long shadows. If you can shrink your penumbra, you will shrink the area where noise can be.

24.8 COMPARISON WITH INTERLEAVED GRADIENT NOISE

Interleaved gradient noise was presented by Jimenez [21] and is tailored toward temporal antialiasing. IGN is fast to generate from an x and y pixel coordinate, so there is no precomputed texture or a texture read. In this way, IGN is a competitor to blue noise in that it is used for the same things, is fast, and gives situationally desirable properties.

```
float IGN(int x, int y)  // x and y are in pixels.
{
  return fract(52.9829189f * fract(0.06711056f*float(x)
                                   + 0.00583715f*float(y)));
}
```

In most modern TAA implementations, a pixel will sample its 3×3 neighborhood to get a minimum and maximum color cube, possibly in another color space such as YCoCg. The previous frame's pixel color is then constrained to this color cube before interpolating toward the current frame's pixel color, possibly using a reversible tone mapping operation [23]. This constraint is used to effectively keep or reject history based on how different the history is from the local neighborhood of this frame.

If you wanted to do something like stochastic transparency for an object that has 1/9th transparency, during the G-buffer fill, you could use a per-pixel random value (white noise) and discard the pixel write if that random value was greater than the transparency value. This makes 1/9 of the pixels survive.

For the neighborhood color constraint, what you would hope for is that out of every 3×3 group of pixels under stochastic alpha, exactly one of them should survive the alpha test to most accurately represent the color cube. The problem is that white noise has clumps and voids, so there will be many 3×3 groups of pixels that have no surviving pixels, and others that have more than

one. When there are no pixels, the history is erroneously rejected and the semitransparent pixel does not contribute to the final pixel color. When there is more than one pixel, that color is over-represented, and it will look more opaque than it is.

If you use blue noise for this instead of white noise, the pixels that survive the alpha test will be roughly uniform; So, often one pixel will survive in every 3×3 block of pixels, but this is not guaranteed. Sometimes you will have more or less.

Interleaved gradient noise does guarantee this, however. This is because every 3×3 block of pixels in IGN has all values approximately 0/9, 1/9, 2/9, . . . 8/9. This is a generalized Sudoku that is impossible to solve because there are too many constraints. IGN gets around this by having small numerical drift, which also makes IGN values fully continuous, unlike blue noise, which often comes from a U8 texture and so only has 256 different possible values.

The same situation seen in stochastic transparency comes up when you ray-trace shadows (or doing other techniques). Using IGN as a per-pixel random value means that in every 3×3 group of pixels, you'll have a histogram of results that more closely matches the actual possible histogram of results, compared to white noise. That leads to more accurate history acceptance and rejection.

So, if you are using TAA with neighborhood sampling for history rejection, you may want to try using IGN instead of a blue noise mask, as a per-pixel random number. Figure 24-16 shows a comparison.

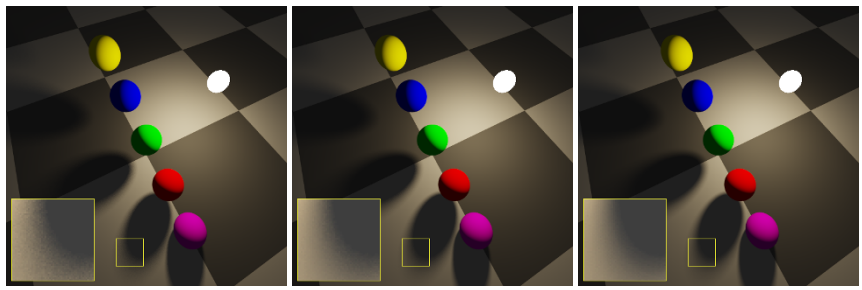

Figure 24-16. *Sixteen spp shadows under TAA. Inset images have +1 fstop exposure to make differences more visible. Left to right: white noise, blue noise samples with blue noise mask, and blue noise samples with IGN mask.*

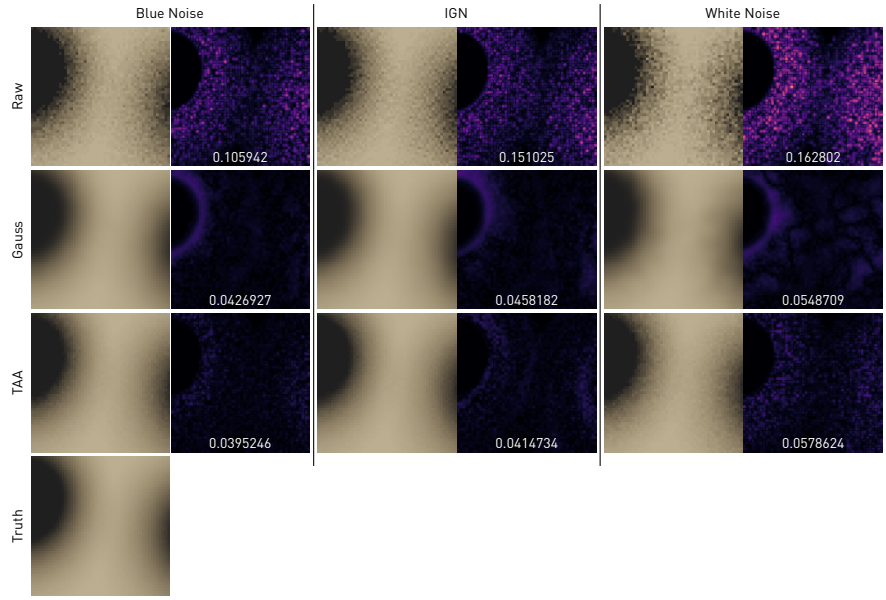

Figure 24-17. *Directional light scene ꟻLIP means (inset number) and heat maps (right images). Truth is ten thousand white noise samples.*

24.9 PERCEPTUAL ERROR EVALUATION

As we've seen, different arrangements of error can affect how an image looks even if it has the same amount of error. Because of this, perceptual error metrics are an active area of research.

Results of comparisons using the ꟻLIP perceptual error metric [3] are shown in Figure 24-17. The heat map of perceptual error is shown, as well as the mean. Lower mean values are better.

ꟻLIP tells us that blue noise is best in all situations, IGN is a little bit worse, and white noise is significantly worse. It's a little surprising that IGN didn't get a better score than blue noise under TAA since it seems to have less noisy pixels, but the scores are very close. Similarly, TAA in general seems to get a better score than Gaussian blurred results, despite looking a little noisier.

See also Chapter 19 about the ꟻLIP metric.

24.10 CONCLUSION

Offline rendering has a wealth of knowledge to give real-time rendering in the way of importance sampling and low-discrepancy sequences, but real-time rendering has much lower rendering computation budgets.

Because of this, we need to think more about what to do with four samples per pixel, one sample per pixel, or fewer than one sample per pixel, which is not as big of a topic in offline rendering.

Sampling well over both space and time are important, as are considerations to the frequency content of the noise for filtering and perceptual qualities. Blue noise can be a great choice, but it is not the only tool for the job as we saw with IGN's effectiveness under TAA. If you ever find yourself using white noise, though, chances are that you are leaving money on the table.

I believe that in the future this topic will continue to bloom, allowing us to squeeze every last drop of image quality out of our rendering capabilities, with the same number of rays.

REFERENCES

[1] Ahmed, A. G. M., Perrier, H., Coeurjolly, D., Ostromoukhov, V., Guo, J., Yan, D.-M., Huang, H., and Deussen, O. Low-discrepancy blue noise sampling. *ACM Transactions on Graphics*, 35(6):247:1–247:13, Nov. 2016. DOI: 10.1145/2980179.2980218.

[2] Ahmed, A. G. M. and Wonka, P. Screen-space blue-noise diffusion of Monte Carlo sampling error via hierarchical ordering of pixels. *ACM Transactions on Graphics*, 39(6):244:1–244:15, Nov. 2020. DOI: 10.1145/3414685.3417881.

[3] Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., and Fairchild, M. D. ꟻLIP: A difference evaluator for alternating images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2):15:1–15:23, 2020. DOI: 10.1145/3406183.

[4] Christensen, P., Kensler, A., and Kilpatrick, C. Progressive multi-jittered sample sequences. *Computer Graphics Forum*, 37(4):21–33, 2018. DOI: 10.1111/cgf.13472.

[5] De Goes, F., Breeden, K., Ostromoukhov, V., and Desbrun, M. Blue noise through optimal transport. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 31:171:1–171:11, 6, 2012. DOI: 10.1145/2366145.2366190.

[6] Fernando, R. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, 35–es, 2005. DOI: 10.1145/1187112.1187153.

[7] Georgiev, I. and Fajardo, M. Blue-noise dithered sampling. In *ACM SIGGRAPH 2016 Talks*, 35:1, 2016. DOI: 10.1145/2897839.2927430.

[8] Gjoel, M. and Svendsen, M. The rendering of INSIDE: High fidelity, low complexity. Game Developer's Conference, https://www.gdcvault.com/play/1023002/Low-Complexity-High-Fidelity-INSIDE, 2016.

[9] Hart, D., Pharr, M., Müller, T., Lopes, W., McGuire, M., and Shirley, P. Practical product sampling by fitting and composing warps. Eurographics Symposium on Rendering (EGSR'20), http://casual-effects.com/research/Hart2020Sampling/index.html, 2020.

[10] Heitz, E., Belcour, L., Ostromoukhov, V., Coeurjolly, D., and Iehl, J.-C. A low-discrepancy sampler that distributes Monte Carlo errors as a blue noise in screen space. In *ACM SIGGRAPH 2019 Talks*, 68:1–68:2, 2019. DOI: 10.1145/3306307.3328191.

[11] Heitz, E., Hill, S., and McGuire, M. Combining analytic direct illumination and stochastic shadows. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2:1–2:11, 2018. DOI: 10.1145/3190834.3190852.

[12] Jarosz, W., Enayet, A., Kensler, A., Kilpatrick, C., and Christensen, P. Orthogonal array sampling for Monte Carlo rendering. *Computer Graphics Forum*, 38(4):135–147, 2019. DOI: 10.1111/cgf.13777.

[13] Keller, A., Georgiev, I., Ahmed, A., Christensen, P., and Pharr, M. My favorite samples. In *ACM SIGGRAPH 2019 Courses*, 15:1–15:271, 2019. DOI: 10.1145/3305366.3329901.

[14] Kopf, J., Cohen-Or, D., Deussen, O., and Lischinski, D. Recursive Wang tiles for real-time blue noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):509–518, 2006. DOI: 10.1145/1179352.1141916.

[15] Mitchell, D. P. Spectrally optimal sampling for distribution ray tracing. *SIGGRAPH Computer Graphics*, 25(4):157–164, July 1991. DOI: 10.1145/127719.122736.

[16] Perrier, H., Coeurjolly, D., Xie, F., Pharr, M., Hanrahan, P., and Ostromoukhov, V. Sequences with low-discrepancy blue-noise 2-D projections. *Computer Graphics Forum (Proceedings of Eurographics)*, 37(2):339–353, 2018. https://hal.archives-ouvertes.fr/hal-01717945.

[17] Peters, C. Free blue noise textures. http://momentsingraphics.de/BlueNoise.html, 2016. Accessed January 15, 2021.

[18] Peters, C. The problem with 3D blue noise. http://momentsingraphics.de/3DBlueNoise.html, 2017. Accessed February 8, 2021.

[19] Reinert, B., Ritschel, T., Seidel, H.-P., and Georgiev, I. Projective blue-noise sampling. *Computer Graphics Forum*, 35(1):285–295, 2015. DOI: 10.1111/cgf.12725.

[20] Schutte, J. Sampling the solid angle of area light sources. https://schuttejoe.github.io/post/arealightsampling/, 2018. Accessed January 20, 2021.

[21] Tatarchuk, N., Karis, B., Drobot, M., Schulz, N., Charles, J., and Mader, T. Advances in real-time rendering in games, Part I. In *ACM SIGGRAPH 2014 Courses*, 10:1, 2014. DOI: 10.1145/2614028.2615455.

[22] Ulichney, R. A. Void-and-cluster method for dither array generation. In J. P. Allebach and B. E. Rogowitz, editors, *Human Vision, Visual Processing, and Digital Display IV*, volume 1913 of *Proceedings*, pages 332–343. SPIE, 1993. DOI: 10.1117/12.152707.

[23] Yang, L., Liu, S., and Salvi, M. A survey of temporal antialiasing techniques. *Computer Graphics Forum*, 39(2):607–621, 2020. DOI: https://doi.org/10.1111/cgf.14018.