CHAPTER 14

THE REFERENCE PATH TRACER

Jakub Boksansky and Adam Marrs
NVIDIA

ABSTRACT

The addition of ray tracing to a real-time renderer makes it possible to create beautiful dynamic effects—such as soft shadows, reflections, refractions, indirect illumination, and even caustics—that were previously not possible. Substantial tuning and optimization are required for these effects to run in real time without artifacts, and this process is simplified when the real-time rendering engine can progressively generate a "known-correct" reference image to compare against. In this chapter, we walk through the steps to implement a simple, but sufficiently featured, path tracer to serve as such a reference.

14.1 INTRODUCTION

With the introduction of the DirectX Raytracing (DXR) and Vulkan Ray Tracing (VKR) APIs, it is now possible to integrate ray tracing functionality into real-time rendering engines based on DirectX and Vulkan. These new APIs provide the building blocks necessary for ray tracing, including the ability to (1) quickly construct spatial acceleration structures and (2) perform fast ray intersection queries against them. Critically, the new APIs provide ray tracing operations with full access to memory resources (e.g., buffers, textures) and common graphics operations (e.g., texture sampling) already used in rasterization. This creates the opportunity to reuse existing code for geometry processing, material evaluation, and post-processing and to build a hybrid renderer where ray tracing works in tandem with rasterization.

A noteworthy advantage of a hybrid ray–raster renderer is the ability to choose how the scene is sampled. Rasterization provides efficient spatially coherent ray and surface material evaluations from a single viewpoint,[1] whereas ray tracing provides convenient incoherent ray–surface evaluation from any point in any direction. The flexibility afforded by arbitrary ray casts

[1] NVIDIA's RTX 2000 series added the ability to rasterize four arbitrary viewpoints at once, but generalized hardware multi-view rasterization is not common.

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_14

Figure 14-1. *An interior scene from Evermotion's Archinteriors Vol. 48 for the Blender package [5] rendered with our reference path tracer using 125,000 paths per pixel.*

enables a hybrid real-time renderer to more naturally generate and accumulate accurate samples of a scene to construct a reference image while not being limited by the artifacts caused by discretization during rasterization (e.g., see shadow mapping).

In this chapter, we discuss how to implement a path tracer in a hybrid renderer to produce in-engine reference images for comparison with the output of new algorithms, approximations, and optimizations used in the real-time code path. We discuss tricks for handling self-intersection, importance sampling, and the evaluation of many light sources. We focus on progressively achieving "ground truth" quality and prefer simple, straightforward code over optimizations for runtime performance. Although we do not discuss optimization or the reduction of artifacts such as noise, improvements in these areas are described in other chapters of this book.

Figure 14-1 is a reference image of an interior scene from Evermotion's Archinteriors Vol. 48 for the Blender package [5]. The image is rendered progressively by our DXR-based path tracer and includes 125,000 paths per pixel with up to ten bounces per path. To help you add a reference path tracer to your renderer, the full C++ and HLSL source code of our path tracer is freely available at the book's source code website and at https://github.com/boksajak/referencePT.

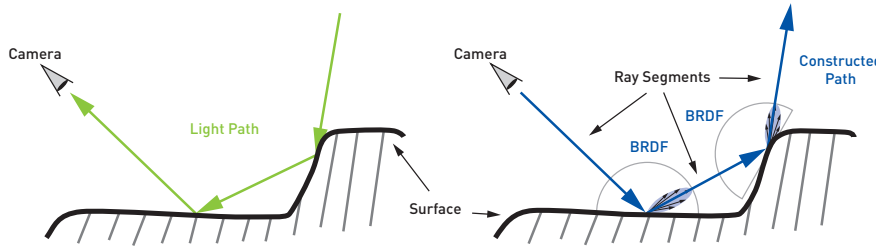

Figure 14-2. *Monte Carlo Unidirectional Path Tracing. Left: light energy bounces around the environment on its way to the camera. Right: the path tracer's model of the light transport from the left diagram. Surface material BRDFs are stochastically sampled at each point where a ray intersects a surface.*

14.2 ALGORITHM

When creating reference images, it is important to choose an algorithm that best matches the use case. A path tracer can be implemented in many ways, and that variety of algorithms is accompanied by an equivalent amount of trade-offs. An excellent overview of path tracing can be found in the book *Physically Based Rendering* [11]. Since this chapter's focus is simplicity and high image quality, we've chosen to implement a *Monte Carlo Unidirectional Path Tracer*. Let's break down what this means—starting from the end and working backward:

1. *Path Tracing* simulates how light energy moves through an environment (also known as *light transport*) by constructing "paths" between light sources and the virtual camera. A *path* is the combination of several ray segments. *Ray segments* are the connections between the camera, surfaces in the environment, and/or light sources (see Figure 14-2).

2. *Unidirectional* means paths are constructed in a single (macro) direction. Paths can start at the camera and move toward light sources or vice versa. In our implementation, paths begin exclusively at the camera. Note that this is opposite the direction that light travels in the physical world!

3. *Monte Carlo* algorithms use random sampling to approximate difficult or intractable integrals. In a path tracer, the approximated integral is the rendering equation, and tracing more paths (a) increases the accuracy of the integral approximation and (b) produces a better image. Since paths are costly to compute, we accumulate the resulting color from each path over time to construct the final image.

14.3 IMPLEMENTATION

Before implementing a reference path tracer in your engine, it is helpful to understand the basics of modern ray tracing APIs. DXR and VKR introduce new shader stages (*ray generation*, *closest-hit*, *any-hit*, and *intersection*), acceleration structure building functions, ray dispatch functions, and shader management mechanisms. Since these topics have been covered well in previous literature, we recommend Chapter 3, "Introduction to DirectX Raytracing," [19] of *Ray Tracing Gems* [6], the SIGGRAPH course of the same name [18], and Chapter 16 of this book to get up to speed. For a deeper understanding of how ray tracing works agnostic of API specifics, see Shirley's *Ray Tracing in One Weekend* series [13].

The code sample accompanying this chapter is implemented with DXR and extends the freely available IntroToDXR sample [10]. At a high level, the steps necessary to perform GPU ray tracing with the new APIs are as follows:

- At startup:
  - Initialize a DirectX device with ray tracing support.
  - Compile shaders and create the ray tracing *pipeline state object*.
  - Allocate memory for and initialize a *shader table*.
- Main loop:
  - Update *bottom-level acceleration structures* (BLAS) of geometry.
  - Update *top-level acceleration structures* (TLAS) of instances.
  - Dispatch groups of *ray generation shaders*.

With the basic execution model in place, the following sections describe the implementation of the key elements needed for a reference path tracer.

14.3.1 ACCELERATION STRUCTURE MEMORY

In-depth details regarding acceleration structure memory allocation are often omitted (rightfully so) from beginner ray tracing tutorials; however, careful acceleration structure memory management is a higher-priority topic when integrating a GPU-based path tracer into an existing rendering engine. There are two scenarios where memory is allocated for acceleration structures: (1) to store built BLAS and TLAS and (2) for use as intermediate "scratch" buffers by the API runtime during acceleration structure builds.

Memory for built acceleration structures can be managed using the same system as memory for geometry, since their lifetime is coupled with the meshes they represent. Scratch buffers, on the other hand, are temporary and may be resized or deallocated entirely once acceleration structure builds are complete. This presents an opportunity to decrease the total memory use with more adept management of the scratch buffer.

A basic scratch buffer management strategy allocates a single memory block of predefined size to use for building all acceleration structures. When the memory needed to build the current acceleration structures exceeds the predefined capacity, the memory block is increased in size. Acceleration structures typically require a small amount of memory (relative to the quantity available on current GPUs), and the maximum memory use can be predetermined in many cases with automated fly-throughs of scenes. Nevertheless, this approach's strengths *and* weaknesses stem from its simplicity. Since any part of the memory block may be in use, buffer resize and/or deallocation operations must be deferred until *all* acceleration structures are built—leading to worst-case memory requirements.

To reduce the total memory use, an alternative approach instead serializes BLAS builds (to some degree) and reuses scratch buffer memory for multiple builds. Barriers are inserted between builds to ensure that the scratch memory blocks are safe to reuse before upcoming BLAS build(s) execute. This process may occur within the scope of a single frame or across several frames. The number of BLAS builds that execute in parallel versus the maximum memory needed to build the current group of BLAS is a balancing act driven by the application's constraints and the target hardware's capabilities. Amortizing builds across multiple frames is especially useful when a scene contains a large number of meshes.

14.3.2 PRIMARY RAYS

Time to start tracing rays! We begin by implementing a ray generation shader that traces primary (camera) rays. This step is a quick way to get our first ray traced image on screen and confirm that the ray tracing pipeline is working.

To make direct comparisons with output from a rasterizer, we construct primary rays by extracting the origin and direction from the same 4×4 view and projection matrices used by the rasterizer. We compose primary ray directions using the camera's basis, the image's aspect ratio, and the camera's vertical field of view (fov). The implementation is shown in

Listing 14-1. *HLSL code to generate primary rays that match a rasterizer's output.*

```
float2 pixel = float2(DispatchRaysIndex().xy);
float2 resolution = float2(DispatchRaysDimensions().xy);

pixel = (((pixel + 0.5f) / resolution) * 2.f - 1.f);
generatePinholeCameraRay(pixel);

RayDesc generatePinholeCameraRay(float2 pixel)
{
    // Set up the ray.
    RayDesc ray;
    ray.Origin = gData.view[3].xyz;
    ray.TMin = 0.f;
    ray.TMax = FLT_MAX;

    // Extract the aspect ratio and fov from the projection matrix.
    float aspect = gData.proj[1][1] / gData.proj[0][0];
    float tanHalfFovY = 1.f / gData.proj[1][1];

    // Compute the ray direction.
    ray.Direction = normalize(
        (pixel.x * gData.view[0].xyz * tanHalfFovY * aspect) -
        (pixel.y * gData.view[1].xyz * tanHalfFovY) +
        gData.view[2].xyz);

    return ray;
}
```

Listing 14-1. Note that the aspect ratio and field of view are extracted from the projection matrix and the camera basis right (`gData.view[0].xyz`), up (`gData.view[1].xyz`), and forward (`gData.view[2].xyz`) vectors are read from the view matrix.

If the view and projection matrices are only available on the GPU as a single combined view-projection matrix, the inverse view-projection matrix (which is typically also available) can be applied to "unproject" points on screen from normalized device coordinate space. This is not recommended, however, as near and far plane settings stored in the projection matrix cause numerical precision issues when the transformation is reversed. More information on constructing primary rays in ray generation shaders can be found in Chapter 3.

To test that primary rays are working, it is useful to include visualizations of built-in DXR variables. For example, the ray hit distance to the nearest surface (`RayTCurrent()`) creates an image similar to a depth buffer. For more colorful visualizations, we output a unique color for each instance index of the

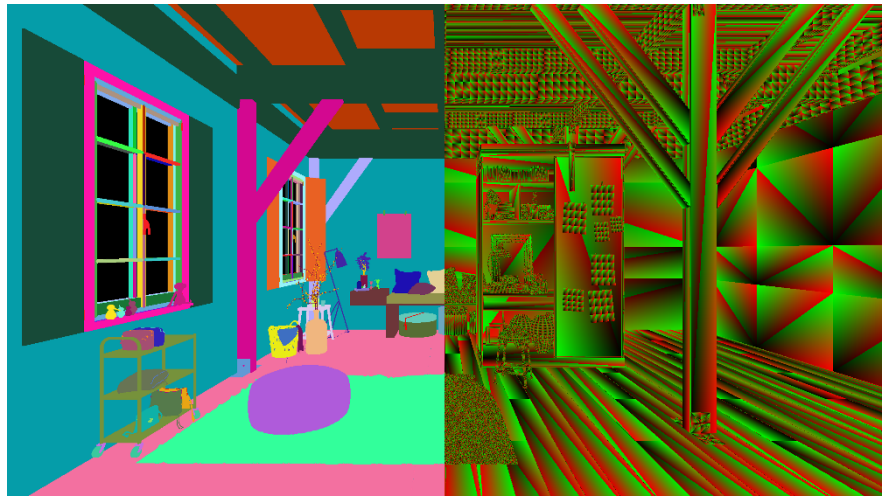

Figure 14-3. *Visualizing geometry instance indices (left) and triangle barycentrics (right) is useful when testing that primary ray tracing is working properly.*

intersected geometry (`InstanceIndex()`) and triangle barycentrics (`BuiltInTriangleIntersectionAttributes.barycentrics`). This is shown in Figure 14-3. A function for hashing an integer to a color, called `hashToColor(int)`, is available in our code sample.

It is also helpful to implement a split-view or quick-swap option to directly compare ray traced and rasterized images next to each other. Even when displaying only the albedo or hit distance, this comparison mode can reveal subtle mismatches between the ray traced and rasterized outputs. For example, geometry may be absent or flicker due to missing barriers or race conditions associated with acceleration structure builds. Another common problem is for instanced meshes to be positioned incorrectly because the DXR API expects row-major transformation matrices instead of HLSL's typical column-major packed matrices.

14.3.3 LOADING GEOMETRY AND MATERIAL PROPERTIES

Now that primary rays are traversing acceleration structures and intersecting geometry in the scene, the next step is to load the geometry and material properties of the intersected surfaces. In our code sample, we use a *bindless* approach to access resources on the GPU in both ray tracing and rasterization. This is implemented with a set of linear buffers that contain all

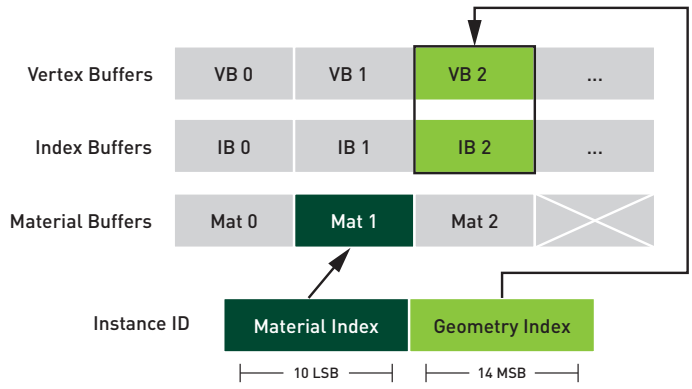

Figure 14-4. *Bindless access of geometry and material data using an encoded instance ID.*

geometry and material data used in the scene. The buffers are then marked as accessible to any invocation of any shader.

The main difference when ray tracing is that the index and vertex data of the geometry (e.g., position, texture coordinates, normals) must be explicitly loaded in the shaders and interpolated manually. To facilitate this, we encode and store the buffer indices of the geometry and material data in the 24-bit instance ID parameter of each TLAS geometry instance descriptor. Illustrated in Figure 14-4, the index into the array of buffers containing the geometry index and vertex data is packed into the 14 most significant bits of the instance ID value, and the material index is packed into the 10 least significant bits (note that this limits the number of unique geometry and material entries to 16,384 and 1,024 respectively). The encoded value is then read with `InstanceID()` in the closest (or any) hit shader, decoded, and used to load the proper material and geometry data. This encoding scheme is implemented in HLSL with two complementary functions shown in Listing 14-2.

Listing 14-2. *HLSL to encode and decode the geometry and material indices.*

```
inline uint packInstanceID(uint materialID, uint geometryID) {
    return ((geometryID & 0x3FFF) << 10) | (materialID & 0x3FF);
}

inline void unpackInstanceID(uint instanceID, out uint materialID,
                             out uint geometryID) {
    materialID = instanceID & 0x3FF;
    geometryID = (instanceID >> 10) & 0x3FFF;
}
```

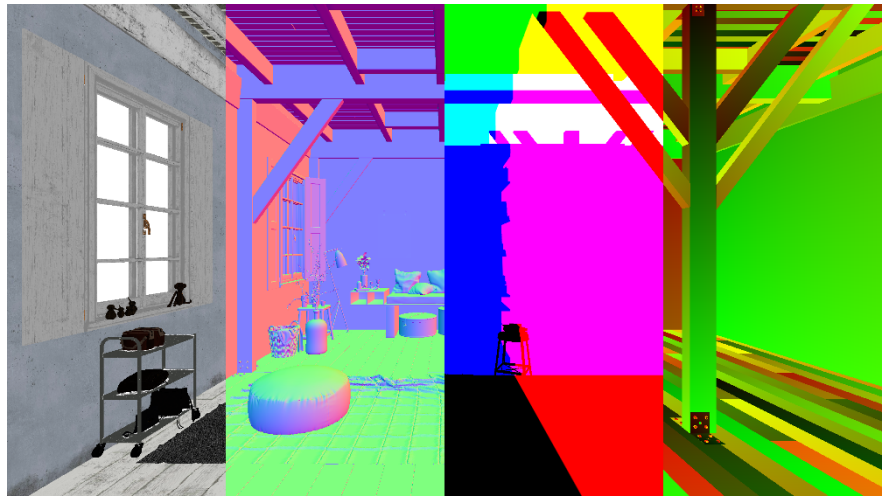

Figure 14-5. *Visualizations of various properties output by the ray generation shader. From left to right: base color, normal, world-space position, and texture coordinates.*

To confirm that the geometry and material data have been loaded and interpolated correctly, it is useful to output G-buffer–like visualizations from the ray generation shader of geometric data (e.g., world-space position, geometric normal, texture coordinates) and material properties (e.g., albedo, shading normal, roughness, etc.). These images can be directly compared with visualizations of a G-buffer generated with a rasterizer, as shown in Figure 14-5.

14.3.4 RANDOM NUMBER GENERATION

At the core of every path tracer, there is a random number generator (RNG). Random numbers are necessary to drive the sampling of materials, lights, procedurally generated textures, and much more as the path tracer simulates light transport. A high-quality RNG with a long period is an essential tool in path tracing. It ensures that each sample taken makes meaningful progress toward reducing noise and improving its approximation of the rendering equation without bias.

We rely on pseudo-random numbers, which have same statistical properties as real random numbers and are generated deterministically using one initial *seed* value. A common way to generate pseudo-random numbers on the GPU is to compute them in shader code at runtime. A fast and popular shader-based method is an xorshift function or linear congruential generator

Listing 14-3. *Initialization of the random seed for an xorshift-based RNG for the given pixel and frame number. The seed provides an initial state for the RNG and is modified each time a new random number is generated.*

```
uint jenkinsHash(uint x) {
    x += x << 10;
    x ^= x >> 6;
    x += x << 3;
    x ^= x >> 11;
    x += x << 15;
    return x;
}

uint initRNG(uint2 pixel, uint2 resolution, uint frame) {
    uint rngState = dot(pixel, uint2(1, resolution.x)) ^ jenkinsHash(frame);
    return jenkinsHash(rngState);
}
```

(LCG) seeded with a hash function [12]. First, the random seed for the RNG is established by hashing the current pixel's screen coordinates and the frame number (see Listing 14-3). This hashing ensures a good distribution of random numbers spatially across neighboring pixels and temporally across subsequent frames. We use the Jenkins's *one_at_a_time* hash [8], but other fast hash functions, such as the *Wang hash* [17], can be used as well.

Next, each new random number is generated by converting the seed into a floating-point number and hashing it again (see Listing 14-4). Notice how the `rand` function modifies the RNG's state. Since the generated number is a sequence of random bits forming an unsigned integer, we need to convert this to a floating-point number in the range [0, 1). This is achieved by using

Listing 14-4. *Generating a random number using the xorshift RNG. The `rand` function invokes the `xorshift` function to modify the RNG state in place and then converts the result to a random floating-point number.*

```
float uintToFloat(uint x) {
    return asfloat(0x3f800000 | (x >> 9)) - 1.f;
}

uint xorshift(inout uint rngState)
{
    rngState ^= rngState << 13;
    rngState ^= rngState >> 17;
    rngState ^= rngState << 5;
    return rngState;
}

float rand(inout uint rngState) {
    return uintToFloat(xorshift(rngState));
}
```

the 23 most significant bits as a mantissa of the floating-point number representation and setting the sign bits and exponent to zero. This generates numbers in the [1, 2) range, so we subtract one to shift numbers into the desired [0, 1) range.

In our code sample, we also include an implementation of the recent PCG4D generator [7]. This RNG uses an input vector of four numbers (the pixel's screen coordinates, the frame number, and the current sample number) and returns *four* random numbers. See Listing 14-5.

Listing 14-5. *Implementation of the PCG4D RNG [7]. An input vector of four numbers is transformed into four random numbers.*

```
uint4 pcg4d(uint4 v)
{
    v = v * 1664525u + 1013904223u;

    v.x += v.y * v.w;
    v.y += v.z * v.x;
    v.z += v.x * v.y;
    v.w += v.y * v.z;

    v = v ^ (v >> 16u);
    v.x += v.y * v.w;
    v.y += v.z * v.x;
    v.z += v.x * v.y;
    v.w += v.y * v.z;

    return v;
}
```

The four inputs PCG4D requires are readily available in the ray generation shader; however, they must be passed into other shader stages (e.g., closest and any-hit). For best performance, the payload that passes data between ray tracing shader stages should be kept as small as possible, so we hash the four inputs to a more compact seed value before passing it to the other shader stages. A discussion of different strategies for hashing these parameters and updating the hash for drawing subsequent samples is discussed in the recent survey by Jarzynski and Olano [7].

Both of the RNGs we've discussed generate uniform distributions (white noise) with long periods. The random numbers are generally well distributed, but there is no guarantee that similar random numbers won't appear in nearby pixels (spatially or temporally). As a result, it can be beneficial to also use *sequences* of random numbers to combat this problem. Low-discrepancy sequences, such as Halton or Sobol, are commonly used as well as blue noise. These help improve convergence speed but are typically only useful for

sampling in a few dimensions. Some sequences, such as blue noise, must be precalculated and therefore have limited length. More information on using blue noise to improve sampling can be found in Chapter 24.

14.3.5 ACCUMULATION AND ANTIALIASING

With the ability to trace rays and a dependable random number generator in hand, we now have the tools necessary to sample a scene, accumulate the results, and progressively refine an image. Shown in Listing 14-6, progressive refinement is implemented by adding the image generated by each frame to an accumulation buffer and then normalizing the accumulated colors before displaying the image shown on screen. Figure 14-6 compares a single frame's result with the normalized accumulated result from many frames.

Listing 14-6. *Implementing an accumulation buffer (ray generation shader).*

```
uint2 LaunchIndex = DispatchRaysIndex().xy;
// Trace a path for the current pixel.
// ...
// Get the accumulated color.
float3 previousColor = accumulationBuffer[LaunchIndex].rgb;

// Add the current image's results.
float3 accumulatedColor = previousColor + radiance;
accumulationBuffer[LaunchIndex] = float4(accumulatedColor, 1.f);

// Normalize the accumulated results to get the final result.
return linearToSrgb(accumulatedColor / gData.accumulatedFrames);
```

Figure 14-6. *A single path traced frame (left) and the normalized accumulated result from one million frames (right).*

The number of accumulated frames is used to ensure that each frame is weighted equally when converting the accumulation buffer's contents to the final image. We use a high-precision 32-bit floating-point format for our accumulation buffer to allow for a large number of samples to be included. Note that typical render target formats in the rasterization pipeline can depend on hardware auto-conversion from linear to sRGB color spaces when writing results. This is not currently supported for the unordered access view types commonly used for output buffers in ray tracing, so it may be necessary to manually convert the final result to sRGB. This is implemented in our code sample by the `linearToSrgb` function.

In addition to amortizing the high cost of computing paths, accumulating images generated by randomly sampling the scene also naturally antialiases the final result. Shown in Listing 14-7, antialiasing is now trivial to implement by randomly jittering ray directions (of primary rays as well as rays traced for material evaluation). For primary rays, this robust sampling not only removes jagged edges caused by geometric aliasing, but also mitigates the moiré patterns that appear on undersampled textured surfaces.

Listing 14-7. *Jittering primary ray directions for antialiasing.*

```
    float2 pixel = float2(DispatchRaysIndex().xy);
    float2 resolution = float2(DispatchRaysDimensions().xy);

    // Add a random offset to the pixel's screen coordinates.
    float2 offset = float2(rand(rngState), rand(rngState));
    pixel += lerp(-0.5.xx, 0.5.xx, offset);

    pixel = (((pixel + 0.5f) / resolution) * 2.f - 1.f);
    generatePinholeCameraRay(pixel);
```

This convenient property of accumulation makes it possible to forgo texture level of detail strategies (such as mipmapping and ray differentials) and sample *only* the highest-resolution version of any texture. Since current graphics hardware and APIs do not support automatic texture level of detail mechanisms in ray tracing, the benefits of accumulation make it possible for our reference path tracer to avoid the added code and complexity related to implementing it (at the cost of performance). Should improved performance and support for texture level of detail become necessary (e.g., to implement ray traced reflections in the real-time renderer), suitable methods are available in *Ray Tracing Gems* [2], *The Journal of Computer Graphics Techniques* [1], and Chapters 7 and 10 of this book.

```
ray = generatePrimaryRay();
throughput = 1.0;
radiance = 0.0;
for bounce ∈ {1 . . . MAX_BOUNCES} do
    Trace(ray);
    if hit surface then
        brdf, brdfPdf, ray = SampleBrdf();
        throughput *= brdf / brdfPdf;
    else
        radiance += throughput * skyColor;
        break;
return radiance;
```

Figure 14-7. *The basic path tracing loop.*

14.3.6 TRACING PATHS

Now we are ready to trace rays beyond primary rays and create full paths from the camera to surfaces and lights in the environment. To accomplish this, we extend our existing ray generation shader and implement a basic path tracing loop, illustrated in Figure 14-7. The process begins by initializing a ray to be cast from the camera into the environment (i.e., a primary ray). Next, we enter a loop and ray tracing begins. If a ray trace misses all surfaces in the scene, the sky's contribution is added to the result color and the loop is terminated. If a ray intersects geometry in the scene, the intersected surface's material properties are loaded and the associated bidirectional reflectance distribution function (BRDF) is evaluated to determine the direction of the next ray to trace along the path. The BRDF accounts for the composition of the material, so for rough surfaces, the reflected ray direction is randomized and then attenuated based on object color. Details on BRDF evaluation are described in the next section.

For a simple first test of the path tracing loop implementation, set the sky's contribution to be fully white (`float3(1, 1, 1)`). This creates lighting conditions commonly referred to as the *white furnace* because all surfaces in the scene are illuminated with white light equally from all directions. Shown in Figure 14-8, this lighting test is especially helpful when evaluating the energy conservation characteristics of BRDF implementations. After the

Figure 14-8. *The interior scene rendered using the white furnace test with white diffuse materials applied to all surfaces.*

white furnace test, try loading values from an environment map in place of a fixed sky color.

Note the two variables, radiance and throughput, declared and initialized at the beginning of the path tracing loop in Figure 14-7. *Radiance* is the final intensity of the light energy presented on screen for a given path. Radiance is initially zero and increases as light is encountered along the path being traced. *Throughput* represents the amount of energy that may be transferred along a path's ray segment after interacting with a surface. Throughput is initialized to one (the maximum) and decreases as surfaces are encountered along the path. Every time the path intersects a surface, the throughput is attenuated based on the properties of the intersected surface (dictated by the material BRDF at the point on the surface). When a path arrives at a light source (i.e., an emissive surface), the throughput is *multiplied* by the intensity of the light source and *added* to the radiance. Note how simple it is to support emissive geometry using this approach!

SURFACE MATERIALS

To render a scene with a variety of realistic materials, we implement BRDFs that describe how light interacts with surfaces. Most surfaces in the physical world reflect some subset of the incoming light (or light wavelengths) in a set

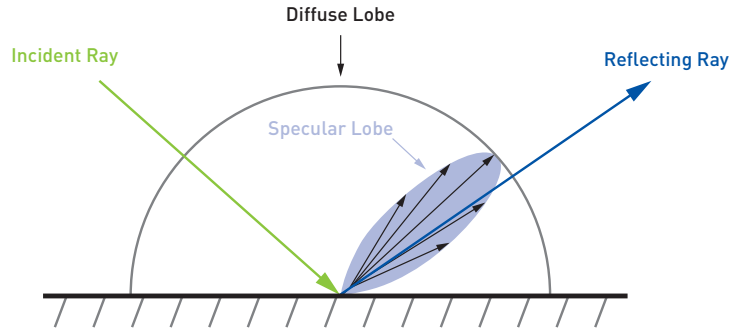

Figure 14-9. *Importance sampling of a BRDF involves selecting a direction of the reflecting ray, based on the incident ray direction and surface properties. The image shows several possible ray directions for the specular lobe (black arrows) and one selected direction highlighted in blue.*

of directions defined by the surface's micro-scale geometric composition. These parameters of variation create an incredible amount of distinct colors and appearances. To simulate this behavior and achieve a desired surface appearance, we (a) attenuate ray throughput according to a material BRDF and (b) reflect incident light within the solid angle determined by the material BRDF (often called a *lobe*). For example, metals and mirrors reflect most incoming light in the direction of perfect reflection, whereas rough matte surfaces reflect light in all directions over the hemisphere above the surface.

To properly attenuate the ray throughput, we importance-sample the material's BRDF lobes by evaluating the *brdf* and *brdfPdf* values shown in Figure 14-7. Importance sampling ensures that each new ray along the path contributes in a meaningful way to the evaluation of the surface material [9]. *brdfPdf* is a probability density function (PDF) of selecting a particular ray direction over all other possible directions, and it is specific to the importance sampling method being used. In our sample code, this process is encapsulated in the `sampleBrdf` function. This function generates a direction for the next ray to be traced along a path and computes the BRDF value corresponding to the given combination of incident and reflected ray directions. See Figure 14-9.

To ease into BRDF implementation, we recommend starting with diffuse materials. In this case, incoming light is reflected equally in all directions, which is simpler to implement. We implement this behavior by generating ray directions within a cosine weighted hemisphere (see `sampleHemisphere` in

the sample code) and using a PDF of *brdfPdf* = $\cos\omega/\pi$. This distributes rays across the entire hemisphere above the surface proportional to the cosine term found in the rendering equation. The Lambertian term *brdfWeight* = *diffuseColor*/π can be pre-divided by the PDF and multiplied by the cosine term, resulting in *brdfWeight*/*brdfPdf* being equal to the diffuse color of the surface.

To support a wide range of physically based materials, we employ a combination of specular and diffuse BRDFs commonly used in games as detailed by Boksansky [4]. Note that these implementations return BRDF values already divided by the PDF, while also accounting for the rendering equation's cosine term, so these calculations are not explicitly shown in our code sample.

SELECTING BRDF LOBES

Many material models are composed of two or more BRDF lobes, and it is common to evaluate separate lobes for the specular and diffuse components of a material. In fact, when rendering semitransparent or refractive objects, an additional *bidirectional transmittance distribution function* (BTDF) is also evaluated. This is discussed in more detail in Chapter 11.

Since the reference path tracer accumulates images, we are able to progressively sample all lobes of a material's BRDF while only tracing a single ray from each surface every frame. With this comes a new choice: *which* BRDF lobe should be sampled on each intersection? A simple solution is to randomly choose one of the lobes, each with equal probability; however, this may sample a component of the overall BRDF that ultimately doesn't contribute much to the final result. For example, highly reflective metals do not have a diffuse component, so the diffuse lobe does not need to be sampled at all.

To address this, we (a) evaluate a subset of each lobe's terms ahead of selection to use to (b) estimate the lobe's contribution and then (c) use the estimate to set the probability of selecting the lobe. Lobes with higher estimated contributions receive higher probabilities and are selected more often. This approach requires a careful balance between the number of terms evaluated before selection and the efficiency of the entire selection process. Note that some terms can only be evaluated once the reflected ray direction is known, which is expensive to generate.

```
pSpecular = getBrdfProbability();
if rand(rngState) < pSpecular) then
    brdfType = SPECULAR;
    throughput /= pSpecular;
else
    brdfType = DIFFUSE;
    throughput /= (1 - pSpecular);
```

Figure 14-10. *Importance sampling and BRDF lobe selection. We select lobes using a random number and probability $p_{specular}$. Throughput is divided by either $p_{specular}$ or $1 - p_{specular}$, depending on the selected lobe.*

In our code sample, we implement BRDF lobe selection using the importance sampling algorithm shown in Figure 14-10. The probability of choosing a specular or diffuse lobe as an estimate of their respective contributions e_{specular} and e_{diffuse} is written as

$$p_{specular} = \frac{e_{\text{specular}}}{e_{\text{specular}} + e_{\text{diffuse}}}, \tag{14.1}$$

where e_{specular} is equal to the Fresnel term F. We evaluate F using the shading normal instead of the microfacet normal (i.e., the view and reflection direction half-vector), since the true microfacet normal is known only after the reflected ray direction is computed by sampling the specular lobe. This works reasonably well for most materials, but is less efficient for rough surfaces viewed at grazing angles. Note that estimate e_{diffuse} is only based on the diffuse reflectance of the surface. Depending on the way a material model combines diffuse and specular lobes together, e_{diffuse} may also need to be weighted by $1 - F$.

This approach is simple and fast, but can generate lobe contribution estimates that are very small in some cases (e.g., rough materials with a slight specular highlight or an object that is barely semitransparent). Low-probability estimates cause certain lobes to be undersampled and can manifest as a "salt and pepper" pattern in the image. Dividing by small probability values introduces numerical precision issues and may also introduce firefly artifacts. To solve these problems, we clamp p_{specular} to the range $[0.1, 0.9]$ whenever it is not equal to zero or one. This ensures a reasonable minimal sampling frequency for each contributing BRDF lobe when the estimates are very small.

TERMINATING THE PATH TRACING LOOP

Since we are tracing paths by iteratively casting rays, we need to handle termination of the loop. Light paths naturally end when the light energy carried along the path is fully absorbed or once the path exits the scene to the sky, but reaching this state can require an extremely large number of bounces. In closed interior scenes, paths may never exit the scene. To prevent tracing paths of unbounded length, we define a maximum number of bounces and terminate the ray tracing loop once the maximum is reached, as shown in Figure 14-7. Bias is introduced when terminating paths too early, but paths that never encounter light are unnecessary and limit performance.

To balance this trade-off, we use a *Russian roulette* approach to decide if insignificant paths should be terminated early. An insignificant path is one that encounters a surface that substantially decreases ray throughput, and as a result will not significantly contribute to the final image. Shown in Listing 14-8, Russian roulette is run before tracing each non-primary ray, and it randomly selects whether to terminate the ray tracing loop or not. The probability of termination is based on the luminance of the path's throughput and is clamped to be at most 0.95 so every possible surface interaction, even those with perfect mirrors, may be terminated. To avoid introducing bias from path termination, the throughput of the non-terminated path is increased using the termination probability. We allow Russian roulette termination to start after a few initial bounces (three in our code sample), to prevent light paths from being terminated too early.

Listing 14-8. *Using a Russian roulette approach to decide if paths should terminate.*

```
    if (bounce > MIN_BOUNCES) {
        float rr_p = min(0.95f, luminance(throughput));
        if (rr_p < rand(rngState)) break;
        else throughput /= rr_p;
    }
```

Integration of Russian roulette into the path tracing loop is shown in Figure 14-11. This improvement allows us to increase the maximum number of bounces a path may have and ensures computation time is spent primarily where it is needed.

SURFACE NORMALS

Since ray segments along a path may intersect a surface on either side—not only the side facing the camera—it is best to disable backface culling when

```
Initialization...
for bounce ∈ {1 . . . MAX_BOUNCES} do
    Trace ray and evaluate direct lighting...
    russianRoulette = luminance(throughput);
    if russianRoulette < rand() then
        break;
    else
        throughput /= russianRoulette;
return radiance;
```

Figure 14-11. *The improved path tracing loop with Russian roulette path termination.*

tracing rays (which also improves performance). As a result, special care is required to ensure that correct surface normals are available on both sides of the surface. Shown in Listing 14-9, we account for this programmatically by inverting normals with the same direction as the incident ray.

Listing 14-9. *Inversion of backfacing normals.*

```
1    float3 V = -ray.Direction;
2    if (dot(geometryNormal, V) < 0.0f) geometryNormal *= -1;
3    if (dot(geometryNormal, shadingNormal) < 0.0f) shadingNormal *= -1;
```

A related issue is self-intersections that occur when a ray intersection is reported for the same surface from which a ray originates. This happens when the ray origin is erroneously placed on the opposite side of a surface due to insufficient numerical precision. A simple solution to this problem is to move the origin away from the surface along the normal by a small fixed amount ϵ ($\epsilon \approx 0.001$), but this approach is not general and almost always introduces light leaking and disconnected shadows. To prevent these problems, we use the improved approach described in Chapter 6 of *Ray Tracing Gems* [16], which is based on calculating the minimal offset necessary to prevent self-intersections (see Figure 14-12). Note that this approach works best when the ray/surface intersection position is calculated from the triangle's vertex positions and interpolated with barycentrics, as this is more precise than using the ray origin, direction, and hit distance. This method also allows us to set the `TMin` of reflecting rays to zero and prevent self-intersections by simply updating the ray origin.

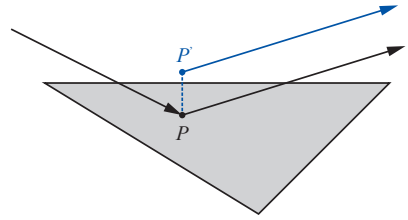

Figure 14-12. *The `offsetRay` method in our code sample calculates a new ray origin P′ originating at the ray/surface intersection point P to minimize self-intersections and light leaks caused by insufficient numerical precision.*

14.3.7 VIRTUAL LIGHTS AND SHADOW RAYS

Illumination from the sky produces beautiful results in a variety of exterior and open interior scenes, but emissive objects (i.e., lights!) are another important component when illuminating interior scenes. In this section, we cover virtual lights and shadow rays. This special treatment of light sources is sometimes referred to as *next event estimation*.

To support virtual lights, we modify the path tracing loop as shown in Figure 14-13. At every ray/surface intersection we randomly select one light (see `sampleLight` in the code sample) and cast a *shadow ray* that determines the visibility between the light and the surface. If the light is visible from the surface, we evaluate the surface's BRDF for the light, divide by the sampling PDF of the light, and add the result multiplied by the throughput to the path radiance (similar to how we handle emissive geometry). This process is illustrated in Figure 14-14.

Since point lights are just a position in space without any area, the contribution of the point light increases to *infinity* as the distance between the light and a surface approaches zero. This creates a singularity that causes invalid values (NaNs) and firefly artifacts that are important to mitigate in a path tracer. We use a method by Yuksel to avoid this obstacle when evaluating point lights [20]. Listing 14-10 shows how point light evaluation is implemented in our sample code. Our code sample supports point and directional lights, but it is straightforward to implement more light types as necessary. For debugging use, we automatically place one directional light source (the sun) and one point light source attached to the camera (a headlight) in the scene. To improve performance when evaluating lights, we

```
Initialization...
for bounce ∈ {1 . . . MAX_BOUNCES} do
    Trace(ray);
    if hit then
        lightIntensity, lightPdf = SampleLight();
        radiance += ray.throughput * EvalBrdf() * lightIntensity *
          CastShadowRay() / lightPdf;
        if bounce == MAX_BOUNCES then
            break;
        brdf, brdfPdf, ray = SampleBrdf();
        throughput *= brdf / brdfPdf;
    else
        radiance += throughput * skyColor;
        break;
    Russian roulette ray termination...
return radiance;
```

Figure 14-13. *The path tracing loop modified to support virtual lights.*

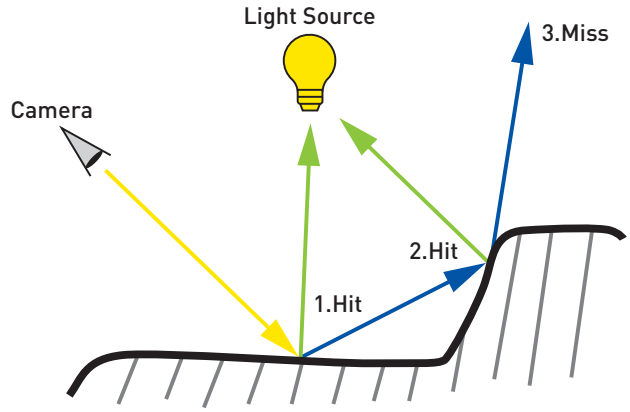

Figure 14-14. *An illustration of the path tracing loop with virtual lights. A primary ray (yellow) is generated and cast. At every surface intersection, a shadow ray (green) is cast toward selected light sources in addition to the BRDF lobe ray (blue).*

create a dedicated DXR hit group for shadow ray tracing—as these rays only need to determine visibility—and skip the closest-hit shader in the ray tracing pipeline.

Listing 14-10. *Evaluating virtual lights in a path tracer.*

```
Light light;
float lightWeight;
sampleLight(rngState, hitPosition, geometryNormal, light, lightWeight);

// Prepare data needed to evaluate the light.
float lightDistance = distance(light.position, hitPosition);
float3 L = normalize(light.position - hitPosition);

// Cast shadow ray toward the light to evaluate its visibility.
if (castShadowRay(hitPosition, geometryNormal, L, lightDistance))
{
    // Evaluate BRDF and accumulate contribution from sampled light.
    radiance += throughput * evalCombinedBRDF(shadingNormal, L, V, material)
          * (getLightIntensityAtPoint(light, lightDistance) * lightWeight);
}
```

SELECTING LIGHTS

In the majority of situations, it is not practical to evaluate and cast shadow rays from all surfaces to all virtual lights in a scene. In the previous section we depended on the `sampleLight` function to select one virtual light from the set of all lights in the scene at each ray/surface intersection. But what is the "correct" light to select and how should this work?

Shown in Listing 14-11, a simple solution is to randomly select one light from the list of lights using a uniform distribution, with a probability of selection equal to 1/*N*, where *N* is number of lights (the `lightWeight` variable in Listing 14-10 is equal to reciprocal of this PDF, analogous to the way we handle BRDF sampling). This approach is straightforward but produces noisy results because it does not consider the brightness of or distance to the selected light. On the opposite end of the spectrum, an expensive solution may evaluate the surface's BRDF for all lights to establish the probability of selecting each light, and then use importance sampling to select lights based on their actual contribution.

Listing 14-11. *Random selection of a light from all lights with a uniform distribution.*

```
uint randomLightIndex =
        min(gData.lightCount - 1, uint(rand(rngState) * gData.lightCount));
light = gData.lights[randomLightIndex];
lightSampleWeight = float(gData.lightCount);
```

A reasonable compromise between these options is a method based on resampling introduced by Talbot [15]. Instead of evaluating the full surface BRDF for all lights, we randomly select a few candidates (e.g., around eight) with a simple uniform distribution method. Next, we evaluate either the full or partial BRDF for these candidate lights and stochastically choose one light based on its contribution. With this approach, we are able to quickly discard lights on the backfacing side of the surface and select ones that are visible, brighter, and closer more often. This method is included in our code sample in the `sampleLightRIS` function. More details about light selection and resampling can be found in Chapters 22 and 23.

TRANSPARENCY

The rendering of transparent and translucent surfaces is a classic challenge in graphics, and real-time ray tracing provides us with powerful new tools to improve existing real-time techniques for transparency. In our code sample and shown in Listing 14-12, we include a simple *alpha-test* transparency mode where a surface's alpha value is compared to a specified threshold and intersections are ignored depending on the result. Any-hit shaders are invoked on every intersected surface while searching for the closest hit, which gives us the opportunity to ignore the hit if the surface's alpha is below the transparency threshold. To perform this test, the alpha value often comes from a texture and must be loaded on *all* any-hit shader invocations. This operation is expensive, so it is best to ensure that any-hit shaders are only executed for geometry with transparent materials. To do this in DXR, mark all BLAS geometry representing fully opaque surfaces with the `D3D12_RAYTRACING_GEOMETRY_FLAG_OPAQUE` flag.

Listing 14-12. *Implementation of the alpha test transparency using the any-hit shader. Note the relatively large number of operations performed on every tested surface while searching for the closest hit.*

```
MaterialData mData = materials[materialID];
float opacity = mData.opacity;

if (mData.baseColorTexIdx != -1) {
    float3 barycentrics = float3((1.0f - attrib.uv.x - attrib.uv.y), attrib.
        uv.x, attrib.uv.y);
    VertexAttributes vertex = GetVertexAttributes(geometryID, PrimitiveIndex
        (), barycentrics);
    opacity *= textures[mData.baseColorTexIdx].SampleLevel(linearSampler,
        vertex.uv, 0.0f).a;
}

if (opacity < mData.alphaCutoff) IgnoreHit();
```

The alpha-tested transparency discussed here barely scratches the surface of the complexity that transparency introduces to the rendering process. This topic deserves an entire chapter (or book!) of its own, and you can learn more about rendering transparent surfaces in Chapter 11.

14.3.8 DEFOCUS BLUR

Defocus blur, sometimes called a *depth of field* or *bokeh*, is an interesting visual effect that is difficult to implement with rasterization-based renderers, yet easy to achieve with a path tracer. To implement it, we generate primary rays in a way that emulates how light is transported through a real camera lens and then accumulate many frames over time to smooth out the results. Our code sample contains a simple implementation of this effect using a thin lens model (see the `generateThinLensCameraRay` function). More details about defocus blur can be found in Chapter 1 and *Physically Based Rendering* [11].

14.4 CONCLUSION

The path tracer described in this chapter is relatively simple, but suitable for rendering high-quality references for games and as an experimentation platform for the development of real-time effects. It has been used during the development of an unreleased AAA game and has proved to be a helpful tool. The code sample accompanying this chapter is freely available, contains a functional path tracer, and can be extended and integrated into game engines without any restrictions.

There are many ways this path tracer can be improved, including support for refraction, volumetrics, and subsurface scattering as well as denoising techniques and performance optimizations to produce noise-free results while running at real-time frame rates. The accompanying chapters in this book, its previous volume [6], and *Physically Based Rendering* [11] are excellent sources of inspiration for improvements as well. We recommend the blog posts "Effectively Integrating RTX Ray Tracing into a Real-Time Rendering Engine" [14] and "Optimizing VK/VKR and DX12/DXR Applications Using Nsight Graphics: GPU Trace Advanced Mode Metrics" [3] for guidance on the integration of real-time ray tracing into existing engines and on optimizing performance.

REFERENCES

[1] Akenine-Möller, T., Crassin, C., Boksansky, J., Belcour, L., Panteleev, A., and Wright, O. Improved Shader and Texture Level of Detail Using Ray Cones. *Journal of Computer Graphics Techniques*, 10(1):1–24, 2021. http://jcgt.org/published/0010/01/01/.

[2] Akenine-Möller, T., Nilsson, J., Andersson, M., Barré-Brisebois, C., Toth, R., and Karras, T. Texture Level of Detail Strategies for Real-Time Ray Tracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 20, pages 321–345. Apress, 2019.

[3] Bavoil, L. Optimizing VK/VKR and DX12/DXR Applications Using Nsight Graphics: GPU Trace Advanced Mode Metrics. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/optimizing-vk-vkr-and-dx12-dxr-applications-using-nsight-graphics-gpu-trace-advanced-mode-metrics, March 30, 2020. Accessed March 17, 2021.

[4] Boksansky, J. Crash Course in BRDF Implementation. https://boksajak.github.io/blog/BRDF, February 28, 2021. Accessed March 17, 2021.

[5] Evermotion. 3D Archinteriors Vol. 48 for Blender. https://www.turbosquid.com/3d-models/3d-archinteriors-vol-48-blender-1308896, 2021. Accessed March 17, 2021.

[6] E. Haines and T. Akenine-Möller, editors. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 2019.

[7] Jarzynski, M. and Olano, M. Hash Functions for GPU Rendering. *Journal of Computer Graphics Techniques*, 9(3):20–38, 2020. http://jcgt.org/published/0009/03/02/.

[8] Jenkins, B. Hash Functions. *Dr. Dobb's*, https://www.drdobbs.com/database/algorithm-alley/184410284, September 1, 1997. Accessed March 17, 2021.

[9] Kahn, H. and Marshall, A. Methods of Reducing Sample Size in Monte Carlo Computations. *Journal of the Operations Research Society of America*, 1(5):263–278, 1953. DOI: 10.1287/opre.1.5.263.

[10] Marrs, A. Introduction to DirectX Raytracing. https://github.com/acmarrs/IntroToDXR, 2018. Accessed March 17, 2021.

[11] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, third edition, 2016.

[12] Reed, N. Quick and Easy GPU Random Numbers in D3D11. http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/, January 12, 2013. Accessed March 17, 2021.

[13] Shirley, P. Ray Tracing in One Weekend. https://raytracing.github.io/books/RayTracingInOneWeekend.html, 2018. Accessed March 17, 2021.

[14] Sjoholm, J. Effectively Integrating RTX Ray Tracing into a Real-Time Rendering Engine. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/effectively-integrating-rtx-ray-tracing-real-time-rendering-engine, October 29, 2018.

[15] Talbot, J., Cline, D., and Egbert, P. Importance Resampling for Global Illumination. In *Eurographics Symposium on Rendering*, pages 139–146, 2005. DOI: 10.2312/EGWR/EGSR05/139-146.

[16] Wächter, C. and Binder, N. A Fast and Robust Method for Avoiding Self-Intersection. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 6, pages 77–85. Apress, 2019.

[17] Wang, T. Integer Hash Function. http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm, 1997. Accessed March 17, 2021.

[18] Wyman, C., Hargreaves, S., Shirley, P., and Barré-Brisebois, C. Introduction to DirectX Raytracing. *ACM SIGGRAPH 2018 Courses*, http://intro-to-dxr.cwyman.org, 2018.

[19] Wyman, C. and Marrs, A. Introduction to DirectX Raytracing. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 3, pages 21–47. Apress, 2019.

[20] Yuksel, C. Point Light Attenuation Without Singularity. In *ACM SIGGRAPH 2020 Talks*, 18:1–18:2, 2020. DOI: 10.1145/3388767.3407364.