CHAPTER 19

Cinematic Rendering in UE4 with Real-Time Ray Tracing and Denoising

Edward Liu,[1] Ignacio Llamas,[1] Juan Cañada,[2] and Patrick Kelly[2]
[1]NVIDIA
[2]Epic Games

ABSTRACT

We present cinematic quality real-time rendering by integrating ray tracing in Unreal Engine 4. We improve the state-of-the-art performance in GPU ray tracing by an order of magnitude through a combination of engineering work, new ray tracing hardware, hybrid rendering techniques, and novel denoising algorithms.

19.1 INTRODUCTION

Image generation with ray tracing can produce higher-quality images than rasterization. One manifestation of this is that, in recent years, most special effects in films are done with path tracing. However, it is challenging to use ray tracing in real-time applications, e.g., games. Many steps of the ray tracing algorithm are costly, including bounding volume hierarchy (BVH) construction, BVH traversal, and ray/primitive intersection tests. Furthermore, stochastic sampling, which is commonly applied in ray tracing techniques, often requires hundreds to thousands of samples per pixel to produce a converged image. This is far outside the compute budget of modern real-time rendering techniques by several orders of magnitude. Also, until recently, real-time graphics APIs did not have ray tracing support, which made ray tracing integration in current games challenging. This changed in 2018 with the announcement of ray tracing support in DirectX 12 and Vulkan.

In this chapter we address many of these problems and demonstrate ray tracing integrated in a modern game engine: Unreal Engine 4 (UE4). Specifically:

- We adopted DirectX Raytracing (DXR) and integrated it into UE4, which allowed us to mostly reuse existing material shader code.
- We leveraged the RT Cores in the NVIDIA Turing architecture for hardware-accelerated BVH traversal and ray/triangle intersection tests.

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_19

- We invented novel reconstruction filters for high-quality stochastic rendering effects, including soft shadows, glossy reflections, diffuse global illumination, ambient occlusion, and translucency, with as few as one input sample per pixel.

A combination of hardware acceleration and software innovations enabled us to create two real-time cinematic-quality ray tracing–based demos: "Reflections" (Lucasfilm) and "Speed of Light" (Porsche).

19.2 INTEGRATING RAY TRACING IN UNREAL ENGINE 4

Integrating a ray tracing framework in a large application such as Unreal Engine is a challenging task, effectively one of the largest architectural changes since UE4 was released. Our goals while integrating ray tracing into UE4 were the following:

- *Performance:* This is a key aspect of UE4, so the ray tracing functionality should be aligned with what users expect. One of our decisions that helped performance is that the G-buffer is computed using existing rasterization-based techniques. On top of that, rays are traced to calculate specific passes, such as reflections or area light shadows.
- *Compatibility:* The output of the ray traced passes must be compatible with the existing UE4's shading and post-processing pipeline.
- *Shading consistency:* Shading models used by UE4 must be accurately implemented with ray tracing to produce consistent shading results with existing shadings in UE4. Specifically, we strictly follow the same mathematics in existing shader code to do BRDF evaluation, importance sampling, and BRDF probability distribution function evaluation for various shading models provided by UE4.
- *Minimizing disruption:* Existing UE4 users should find the integration easy to understand and extend. As such, it must follow the UE design paradigms.
- *Multiplatform support:* While initially real-time ray tracing in UE4 is entirely based on DXR, the multiplatform nature of UE4 required us to design the new system in a way that makes it possible to eventually port it to other future solutions without a major refactoring.

The integration was split in two phases. The first was a prototyping phase, where we created the "Reflections" (a Lucasfilm *Star Wars* short movie made in collaboration with ILMxLabs) and "Speed of Light" (Porsche) demos with the goal of learning the ideal way of integrating the ray tracing technology within UE in a way that could scale properly in the future. That helped us extract conclusions on

the most challenging aspects of the integration: performance, API, integration in the deferred shading pipeline, changes required in the render hardware interface (RHI), a thin layer that abstracts the user from the specifics of each hardware platform, changes in the shader API, scalability, etc.

After finding answers to most of the questions that arose during phase one, we moved to phase two. This consisted of a major rewrite of the UE4 high-level rendering system, which in addition to providing a better integration of real-time ray tracing, also brought other advantages, including a significant overall performance boost.

19.2.1 PHASE 1: EXPERIMENTAL INTEGRATION

At a high level, integrating ray tracing into a rasterization-based real-time rendering engine consists of a few steps:

- Registering the geometry that will be ray traced, such that acceleration structures can be built or updated when changing.
- Creating hit shaders such that any time a ray hits a piece of geometry, we can compute its material parameters, just like we would have done in rasterization-based rendering.
- Creating ray generation shaders that trace rays for various use cases, such as shadows or reflections.

19.2.1.1 DIRECTX RAYTRACING BACKGROUND ON ACCELERATION STRUCTURES

To comprehend the first step, it is first useful to understand the two-level acceleration structure (AS) exposed by the DirectX Raytracing API. In DXR there are two types of acceleration structures, forming a two-level hierarchy: top-level acceleration structure (TLAS) and bottom-level acceleration structure (BLAS). These are illustrated in Figure 19-1. The TLAS is built over a set of instances, each one with its own transform matrix and a pointer to a BLAS. Each BLAS is built over a set of geometric primitives, either triangles or AABBs, where AABBs are used to enclose custom geometry that will be intersected using a custom Intersection shader executed during acceleration structure traversal as AABBs are found along a ray. The TLAS is usually rebuilt each frame for dynamic scenes. Each BLAS is built at least once for each unique piece of geometry. If the geometry is static, no additional BLAS build operations are needed after the initial build. For dynamic geometry, a BLAS will need to be either updated or fully rebuilt. A full BLAS rebuild is needed when the number of input primitives changes (i.e., when triangles or AABBs need to be added or removed, for example, due to dynamic tessellation or

other procedural geometry generation methods, such as particle systems). In the case of a BVH (which is what the NVIDIA RTX implementation uses), the BLAS update entails a refit operation, where all the AABBs in the tree are updated from the leaves to the root, but the tree structure is left unchanged.

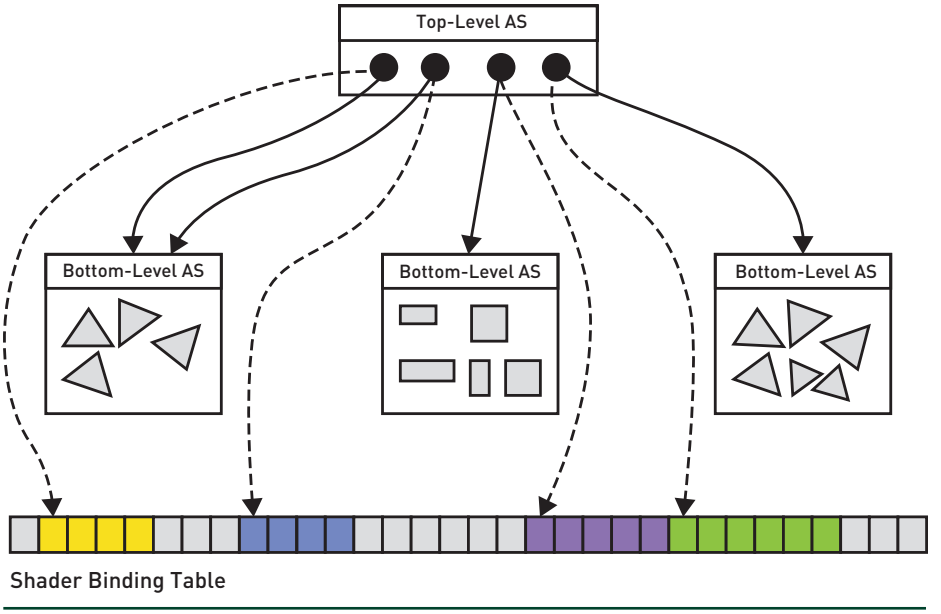


Figure 19-1. *The two-level hierarchy of the acceleration structure.*

19.2.1.2 EXPERIMENTAL EXTENSIONS TO THE UE4 RHI

In the experimental UE4 implementation, we extended the rendering hardware interface (RHI) with an abstraction inspired by the NVIDIA OptiX API, but slightly simplified. This abstraction consisted of three object types: `rtScene`, `rtObject`, and `rtGeometry`. The `rtScene` is composed of `rtObjects`, which are effectively instances, each pointing to an `rtGeometry`. An `rtScene` encapsulates a TLAS, while an `rtGeometry` encapsulates a BLAS. Both `rtGeometry` and any `rtObject` pointing to a given `rtGeometry` can be made up of multiple sections, all of which belong to the same UE4 primitive object (Static Mesh or Skeletal Mesh) and therefore share the same index and vertex buffer but may use different (material) hit shaders. An `rtGeometry` itself has no hit shaders associated with it. We set hit shaders and their parameters at the `rtObject` sections.

The engine material shader system and the RHI were also extended to support the new ray tracing shader types in DXR: Ray Generation, Closest Hit, any-hit, Intersection, and Miss. The ray tracing pipeline with these shader stages is shown

in Figure 19-2. In addition to Closest Hit and any-hit shaders, we also extended the engine to support the use of the existing vertex shader (VS) and pixel shader (PS) in their place. We leveraged a utility that extends the open-source Microsoft DirectX Compiler, providing a mechanism to generate Closest Hit and any-hit shaders from the precompiled DXIL representation of VS and PS. This utility takes as input the VS code, the Input Layout for the Input Assembly stage (including vertex and index buffer formats and strides), and the PS code. Given this input, it can generate optimal code that performs index buffer fetch, vertex attribute fetch, format conversion, and VS evaluation (for each of the three vertices in a triangle), followed by interpolation of the VS outputs using the barycentric coordinates at a hit, the result of which is provided as input to the PS. The utility is also able to generate a minimal any-hit shader to perform alpha testing. This allowed the rendering code in the engine to continue using the vertex and pixel shaders as if they were going to be used to rasterize the G-buffer, setting their shader parameters as usual.

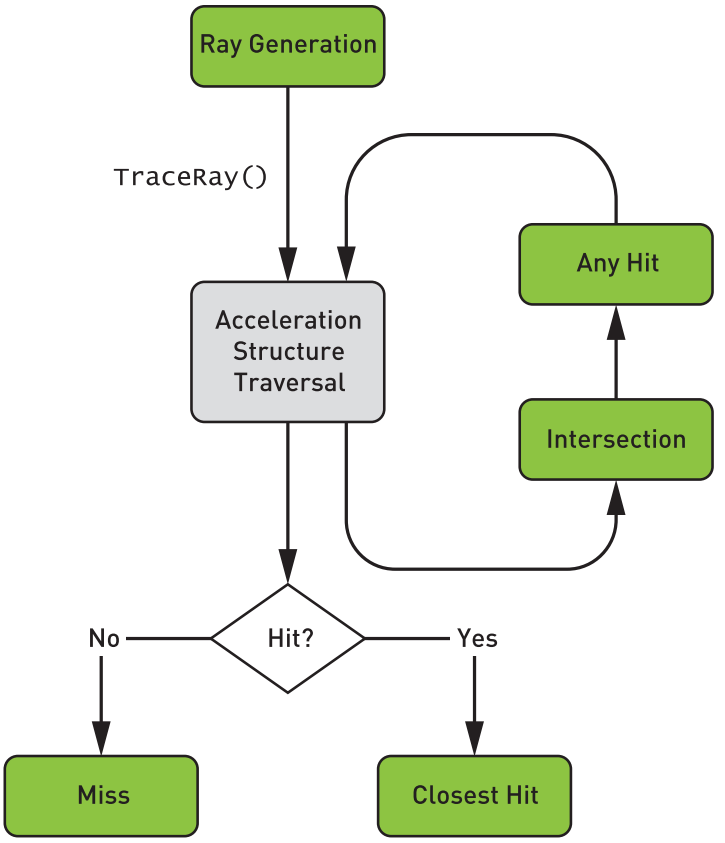


Figure 19-2. *The ray tracing pipeline.*

The implementation under this experimental RHI abstraction had two additional responsibilities: compiling ray tracing shaders (to a GPU-specific representation) and managing the Shader Binding Table, which points to the shader code and shader parameters that need to be used for each piece of geometry.

Compiling Ray Tracing Shaders Efficiently Prior to the introduction of RTX and DirectX Raytracing, the existing pipeline abstractions in graphics APIs (for rasterization and compute) required only a small number of shaders (1 to 5) to be provided to create a so-called *pipeline state object* (PSO), which encapsulates the compiled machine-specific code for the input shaders. These graphics APIs allow the creation of many of these pipeline state objects in parallel if needed, each for a different material or rendering pass. The ray tracing pipeline, however, changes this in a significant way. A *ray tracing pipeline state object* (RTPSO) must be created with all the shaders that may need to be executed in a given scene, such that when a ray hits any object, the system can execute the shader code associated with it, whatever that might be. In complex content in today's games, this can easily mean thousands of shaders. Compiling thousands of ray tracing shaders into the machine code for one RTPSO could be very time consuming if done sequentially. Fortunately, both DirectX Raytracing and all other NVIDIA ray tracing APIs enabled by RTX expose the ability to compile in parallel multiple ray tracing shaders to machine code. This parallel compilation can take advantage of the multiple cores of today's CPUs. In the experimental UE4 implementation, we used this functionality by simply scheduling separate tasks, each of which compiled a single ray tracing shader or hit group into what DXR calls a *collection*. Every frame, if no other shader compilation tasks were already executing, we checked if any new shaders were needed and not available. If any such shaders were found, we started a new batch of shader compilation tasks. Every frame we also checked if any prior set of shader compilation tasks had completed. If so, we created a new RTPSO, replacing the previous one. At any time, a single RTPSO is used for all `DispatchRays()` invocations in a frame. Any old RTPSOs replaced by a new RTPSO are scheduled for deferred deletion when no longer used by any in-flight frames. Objects for which a required shader was not yet available in the current RTPSO were removed (skipped) when building the TLAS.

The Shader Binding Table This table is a memory buffer made up of multiple records, each containing an opaque shader identifier and some shader parameters that are equivalent to what DirectX 12 calls a *root table* (for a Graphics Pipeline Draw or Compute Pipeline Dispatch). Since this first experimental implementation was designed to update shader parameters for every object in the scene at every frame, the Shader Binding Table management was simple, mimicking that of a command buffer. The Shader Binding Table was allocated as an N-buffered linear memory buffer, with size dictated by the number of objects in the scene. At every

frame we simply started writing the entire Shader Binding Table from scratch in a GPU-visible CPU-writable buffer (in an upload heap). We then enqueued GPU copies from this buffer to a GPU-local counterpart. Fence-based synchronization was used to prevent overwriting either CPU or GPU copies of the Shader Binding Table used *N* frames earlier.

19.2.1.3 REGISTERING GEOMETRY FOR A VARIETY OF ENGINE PRIMITIVES

To register geometry for acceleration structure construction, we had to make sure that the various primitives in UE4 had their RHI-level `rtGeometry` and `rtObjects` created for them. Typically, this requires identifying the right scope where `rtGeometry` and `rtObjects` should be created. For most primitives one can create an `rtGeometry` at the same scope as the vertex and index buffer geometry. This is simple to do for static triangle meshes, but for other primitives it can get more involved. For example, Particle Systems, Landscape (terrain) primitives, and Skeletal Meshes (i.e., skinned geometry, potentially using morph targets or cloth simulation) require special treatment. In UE4 we took advantage of the existing `GPUSkinCache`, a compute shader–based system that performs skinning at every frame into temporary GPU buffers that can be used as input to rasterization passes, acceleration structure updates, and hit shaders. It's also important to note that each Skeletal Mesh instance needs its own separate BLAS; therefore, in this case a separate `rtGeometry` is needed for each instance of a Skeletal Mesh, and no instancing or sharing of these is possible, as is the case for Static Meshes.

We also experimented with support for other kinds of dynamic geometry, such as Cascade Particle Systems and the Landscape (terrain) system. Particle systems can use either particle meshes, where each particle becomes an instanced pointing to a Static Mesh's `rtGeometry`, or procedurally generated triangle geometry, such as sprites or ribbons. Due to limited time, this experimental support was restricted to CPU-simulated particle systems. For the Landscape system we had some experimental support for the terrain patches, the foliage geometry, and the Spline meshes (which are meshes generated by interpolating a triangle mesh along a B-spline curve). For some of these, we relied on existing CPU code that generates the vertex geometry.

19.2.1.4 UPDATING THE RAY TRACING REPRESENTATION OF THE SCENE

At every frame, the UE4 renderer executes its Render loop body, where it performs multiple passes, such as rasterizing the G-buffer, applying direct lighting, or post-processing. We modified this loop to update the representation of the scene used

for ray tracing purposes, which at the lowest level consists of the Shader Binding Table, associated memory buffers and resource descriptors, and the acceleration structures.

From the high-level renderer point of view, the first step involves ensuring that the shader parameters for all the objects in the scene are up to date. To do so we leveraged the existing Base Pass rendering logic, typically used to rasterize a G-buffer in the deferred shading renderer. The main difference is that for ray tracing we had to perform this loop over all the objects in the scene, not only those both inside the camera frustum and potentially visible, based on occlusion culling results. The second difference is that, instead of using the VS and PS from deferred shading G-buffer rendering, the first implementation used the VS and PS from the forward shading renderer, as that seemed like a natural fit when shading hits for reflections. A third difference is that we actually had to update shader parameters for multiple *ray types*, in some cases using slightly different shaders.

19.2.1.5 ITERATING OVER ALL OBJECTS

In a large scene there may be a significant amount of CPU time spent updating shader parameters for all objects. We recognized this as a potential problem and concluded that to avoid it we should aim for what people traditionally call *retained mode* rendering. Unlike *immediate mode* rendering, where at every frame the CPU resubmits many of the same commands to draw the same objects, one by one, in retained mode rendering the work performed every frame is only that needed to update anything that has changed since the last frame in a persistent representation of the scene. Retained mode rendering is a better fit for ray tracing because, unlike rasterization, in ray tracing we need global information about the entire scene. For this reason retained mode rendering is enabled by all the GPU ray tracing APIs supported by NVIDIA RTX (OptiX, DirectX Raytracing, and Vulkan Raytracing). Most real-time rendering engines today, however, are still designed around the limitations of rasterization APIs used for the last two decades, since OpenGL. As such, renderers are written to re-execute at every frame all the shader parameter setting and drawing code, ideally for as few objects as needed to render what is visible from the camera. While this same approach worked well for the small scenes we used to demonstrate real-time ray tracing, we know it will fail to scale to huge worlds. For this reason the UE4 rendering team embarked on a project to revamp the high-level renderer to aim for a more efficient retained mode rendering approach.

19.2.1.6 CUSTOMIZING SHADERS FOR RAY TRACED RENDERING

The second difference between ray tracing and rasterization rendering was that we had to build hit shaders from VS and PS code that were slightly customized for ray tracing purposes. The initial approach was based on the code used for forward shading in UE4, except that any logic that depends on information tied to screen-space buffers was skipped. (Side note: this implied that materials making use of nodes that access screen-space buffers do not work as expected with ray tracing. Such materials should be avoided when combining rasterization and ray tracing.) While our initial implementation used hit shaders based on UE4's forward-rendering shaders, over time we refactored the shader code such that the hit shaders looked closer to the shaders used for G-buffer rendering in deferred shading. In this new mode all the dynamic lighting was performed in the ray generation shader instead of the hit shader. This reduced the size and complexity of hit shaders, avoiding the execution of nested `TraceRay()` calls in hit shaders, and allowed us to modify the lighting code for ray traced shading with much-reduced iteration times, thanks to not having to wait for the rebuilding of thousands of material pixel shaders. In addition to this change, we also optimized the VS code by ensuring we used the position computed from the ray information at a hit (`origin + t * direction`), thus avoiding memory loads and computation associated with position in the VS. Furthermore, where possible, we moved computation from the VS to the PS, such as when computing the transformed normal and tangent. Overall, this reduced the VS code to mostly data fetching and format conversion.

19.2.1.7 BATCH COMMIT OF SHADER PARAMETERS OF MULTIPLE RAY TYPES

The third difference, updating parameters for multiple ray types, meant that in some cases we had to loop over all the objects in the scene multiple times, if one of the ray types required a totally separate set of VS and PS. In some cases, though, we were able to significantly reduce the overhead of additional ray types. For example, we were able to handle the update of the two most common ray types, Material Evaluation and Any Hit Shadow, by allowing the RHI abstraction to commit shader parameters for multiple ray types at the same time, when these can use hit shaders that have compatible shader parameters. This requirement was guaranteed by the DirectX Compiler utility that transforms VS and PS pairs to hit shaders, as it ensured that the VS and PS parameter layout is the same for both the Closest Hit shader and the any-hit shader (since both were generated from the same VS and PS pair). Given this and the fact that the Any Hit Shadow ray type is simply using the same any-hit shader as the Material Evaluation ray type combined with a null Closest Hit shader, it was trivial to use the same Shader Binding Table record data for both ray types, but with different Shader Identifiers.

19.2.1.8 UPDATING INSTANCE TRANSFORMATION

During the process of filling the Shader Binding Table records, we also took care of recording their offset in the associated `rtObject`. This information needs to be provided to the TLAS build operation, as it is used by the DXR implementation to decide what hit shaders to execute and with what parameters. In addition to updating all the shader parameters, we must also update the instance transform and flags associated with every `rtObject`. This is done in a separate loop, prior to updating the shader parameters. The instance-level flags allow, among other things, control of masking and backface culling logic. Masking bits are used in UE4 to implement support for lighting channels, which allow artists to restrict specific sets of lights to interacting with only specific sets of objects. Backface culling bits are used to ensure that rasterizing and ray tracing results match visually (culling is not as likely to be a performance optimization for ray tracing as it is for rasterizing).

19.2.1.9 BUILDING ACCELERATION STRUCTURES

After updating all the ray tracing shader parameters, the `rtObject` transforms, and the culling and masking bits, the Shader Binding Table containing hit shaders is ready, and all the `rtObjects` know their corresponding Shader Binding Table records. At this point we move to the next step, which is scheduling the build or update of any bottom-level acceleration structures, as well as the rebuild of the TLAS. In the experimental implementation this step also takes care of deferred allocation of memory associated with acceleration structures. One important optimization in this phase is to ensure that any resource transition barriers that are needed after BLAS updates are deferred to be executed right before the TLAS build, instead of executing these right after each BLAS update. Deferral is important because each of these transition barriers is a synchronization step on the GPU. Having the transitions coalesced into a single point in the command buffer avoids redundant synchronization that would otherwise cause the GPU to frequently become idle. By coalescing the transitions, we perform this synchronization once, after all the BLAS updates, and allow multiple BLAS updates, potentially for many small triangle meshes, to overlap while running on the GPU.

19.2.1.10 MISS SHADERS

Our use of Miss shaders is limited. Despite exposing Miss shaders at the RHI level, we never used them from the engine side of the RHI, and we relied on the RHI implementation preinitializing a set of identical default Miss shaders (one per ray type) that simply initialized a `HitT` value in the payload to a specific negative value, as a way to indicate that a ray had not hit anything.

19.2.2 PHASE 2

After accumulating experience during the creation of two high-end ray tracing demos, we were in the position to work on a big refactoring that could make it possible to transition from code being project-specific to something that scales well for the needs of all UE4 users. One of the ultimate goals of this phase was to move the UE4 rendering system from immediate to retained mode. This led to higher efficiency, as only objects that changed at a given frame are effectively updated. Because of limitations of rasterization pipelines, UE4 was initially written following an immediate mode style. However, this would represent a serious limitation for ray tracing large scenes, since it always updates all the objects for each frame even though most of the time only a small portion changed. So, moving to a retained mode style was one of the key accomplishments of this phase.

With the ultimate goal of making it possible to integrate ray tracing in any platform in the future, we divided the requirements in different tiers, to understand what was needed for supporting each feature and how we could face limitations of any particular device without sacrificing functionality when more advance hardware was present.

19.2.2.1 TIER 1

Tier 1 describes the lowest level of functionality required to integrate basic ray tracing capabilities and is like existing ray tracing APIs such as Radeon Rays or Metal Performance Shaders. The input is a buffer that contains ray information (origin, direction), and the shader output is a buffer that contains intersection results. There is no built-in `TraceRay` intrinsic function nor are there any hit shaders available in this tier. Tier 1 is a good fit for implementing simple ray tracing effects such as opaque shadows or ambient occlusion, but going beyond these is challenging and requires complex changes in the code that introduce restrictions and make it difficult to achieve good execution efficiency.

19.2.2.2 TIER 2

Tier 2 supports ray generation shaders, which can call a `TraceRay` intrinsic function whose output is available immediately after the trace call. This level of functionality also supports dynamic shader dispatch, which is abstracted using RTPSOs and Shader Binding Tables. Tier 2 does not support recursive ray tracing, so new rays cannot be spawned from hit shaders. During phase 1 we found that this was not a big limitation in practice, and it had the positive side effect of reducing the size and complexity of hit shaders.

Tier 2 makes it possible to implement most of the goals defined in the ray tracing integration in UE4. Therefore, the design of the UE4 ray tracing pipeline has been done assuming Tier 2 capabilities.

19.2.2.3 TIER 3

Tier 3 closely follows the DXR specification. It supports all Tier 2 features and recursive calls with a predefined maximum depth. It also supports tracing rays from other shader types beyond ray generation shaders, as well as advanced features, such as customizable acceleration structure traversal. Tier 3 is the most powerful set of capabilities to date, and it enables integration of advanced ray tracing features from offline rendering, e.g., photon mapping and irradiance caching, in a modular way. The ray tracing integration in UE4 has been designed to make use of Tier 3 capabilities when the hardware supports it.

19.3 REAL-TIME RAY TRACING AND DENOISING

In addition to lessons learned from the ray tracing integration in UE4, the initial experimental phase was essential to explore the possibilities of real-time ray tracing. We started with mirror reflections and hard shadows, continued by adding denoising to approximate glossy reflections and area light shadows from a limited ray budget, and then added ambient occlusion, diffuse global illumination, and translucency.

We note that rasterization-based renderers (both offline and real-time) often split the rendering equation into multiple segments of light paths and deal with each segment separately. For example, a separate pass is performed for screen-space reflection, and another pass for direct lighting. This is less commonly used in ray tracing renderers, in particular offline path tracers, which instead render by accumulating tens, hundreds, or thousands of light paths.

Some ray tracing renderers use techniques to improve convergence or interactivity, e.g., virtual point lights (instant radiosity [6]), path space filtering [1], and a plethora of denoising algorithms. For an overview of recent denoising techniques for Monte Carlo rendering, please refer to Zwicker et al's [14] excellent state-of-the-art report.

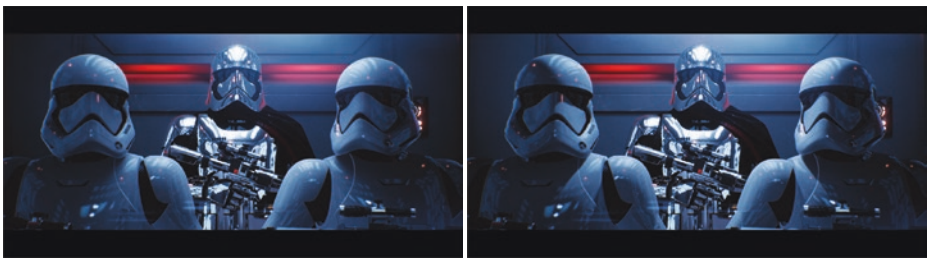
Zimmer et al. [13] split the entire ray tree into separate buffers and applied denoising filters to each buffer before compositing the final frame. In our scenario, we follow a similar approach, splitting the light paths that emerge when we try to solve the rendering equation. We apply custom filters for results from different ray types, e.g., shadow, reflection, and diffuse rays. We use a small number of rays per pixel for each effect and denoise them aggressively to make up for the insufficient

number of samples. We exploit local properties to improve denoising quality (such as the size of a light or the shape of a glossy BRDF lobe) and combine the results to generate images that approach those generated by offline renderers. We call this technique *Partitioned Light Path Filtering*.

19.3.1 RAY TRACED SHADOWS

One significant advantage of ray traced shadows over shadow maps is that ray tracing can easily simulate physically accurate penumbras even for light sources with large areas, improving the perceived realism of the rendered image. Producing high-quality soft shadows with large penumbra is one of our goals.

In the "Reflections" (Lucasfilm) demo, area lights and soft shadows are two of the most important visual components. See Figure 19-3 for one example.



(a) Ray traced shadows (b) Shadow maps

Figure 19-3. *(a) Original rendering from the "Reflections" (Lucasfilm) demo with ray traced soft shadows. Notice the soft shadows under the helmet of the two stormtroopers. (b) Without soft shadows, the lighting loses the fidelity in the area light illumination, and the image is less photorealistic.*

A similar effect is shown in the "Speed of Light" (Porsche) demo, with shadows cast by the giant area light above the Porsche 911 Speedster car. Large diffuse lights are commonly used for car exhibitions, and they produce diffuse-looking shadows with large penumbras. Accurate shadow penumbras from big area light sources are challenging with traditional rasterization-based techniques such as shadow maps. With ray tracing we can simulate this phenomenon accurately, as shown in Figure 19-4.

(a) Ray traced shadows (b) Shadow maps

Figure 19-4. *(a) Ray traced area light shadows with penumbras using one sample per pixel per light, reconstructed with our shadow denoiser. (b) Shadows rendered with shadow mapping.*

19.3.1.1 LIGHTING EVALUATION

The lighting evaluation for area light in our demo is computed using the linearly transformed cosine (LTC) approach [2], which provides a variance-free estimation of the lighting term without including visibility. To render shadows for area lights, we use ray tracing to collect a noisy estimation of the visibility term before applying advanced image reconstruction algorithms to the results. Finally we composite the denoised visibility term on top of the lighting results.

Mathematically this can be written as the following split sum approximation of the rendering Equation [4]:

$$\begin{aligned} L(\omega_o) &= \int_{S^2} L_d(\omega_i) V(\omega_i) f(\omega_o, \omega_i) |\cos\theta_i| \\ &\approx \int_{S^2} V(\omega_i) d\omega_i \int_{S^2} L_d(\omega_i) f(\omega_o, \omega_i) |\cos\theta_i| d\omega_i. \end{aligned} \tag{1}$$

Here, $L(\omega_o)$ is the radiance leaving the surface in direction ω_o; $V(\omega_i)$ is the binary visibility term in direction ω_i; the surface property f is the BRDF (bidirectional reflectance distribution function); $L_i(\omega_i)$ is the incoming light along direction ω_i; and the angle between the surface normal and the incoming light direction is θ_i, with $|\cos\theta_i|$ accounting for geometric dropoff due to this angle. For diffuse surfaces, this approximation has negligible bias and is commonly used in shadow mapping techniques. For area light shading with occlusion on glossy surfaces, one can use the ratio estimator from Heitz et al. [3] to get a more accurate result. In contrast, we directly use ray traced reflections plus denoising to handle specular area light shading with occlusion information in the "Speed of Light" (Porsche) demo. Please see Section 19.3.2.3 for more details.

19.3.1.2 SHADOW DENOISING

To get high-quality ray traced area light shadows with large penumbra, typically hundreds of visibility samples are required per pixel to get a estimate without noticeable noise. The required number of rays depends on the size of the light sources and the positions and sizes of the occluders in the scene.

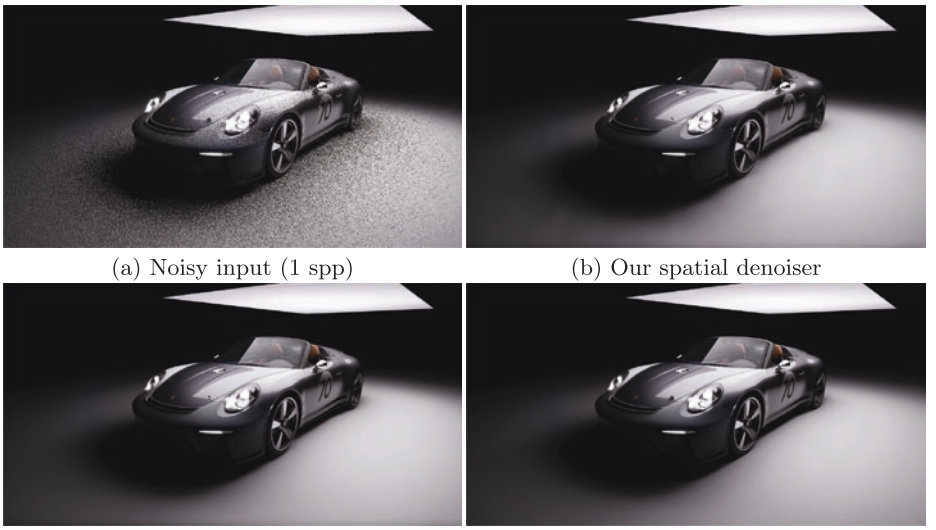
For real-time rendering we have a much tighter ray budget, and hundreds of rays are way outside our performance budget. For the "Reflections" (Lucasfilm) and "Speed of Light" (Porsche) demos, we used as few as one sample per pixel per light source. With this number of samples, the results contained a substantial amount of noise. We applied an advanced denoising filter to reconstruct a noiseless image that's close to ground truth.

We have designed a dedicated denoising algorithm for area light shadows with penumbra. The shadow denoiser has both a spatial and a temporal component. The spatial component is inspired by recent work in efficient filters based on a frequency analysis of local occlusion, e.g., the axis-aligned filtering for soft shadows [8] and the sheared filter by Yan et al. [12]. The denoiser is aware of the information about the light source, such as its size, shape, and direction and how far away it is from the receiver, as well as the hit distances for shadow rays. The denoiser uses this information to try to derive an optimal spatial filter footprint for each pixel. The footprint is anisotropic with varying directions per pixel. Figure 19-5 shows an approximated visualization of our anisotropic spatial kernel. The kernel shape stretches along the direction of the penumbra, resulting in a high-quality image after denoising. The temporal component to our denoiser increases the effective sample count per pixel to be around 8–16. The caveat is slight temporal lag if the temporal filter is enabled, but we perform temporal clamping as proposed by Salvi [10] to reduce the lag.

Figure 19-5. *Visualization (in green) of a filter kernel used in our shadow denoiser. Notice how it is anisotropic and stretches along each penumbra's direction. (From Liu [7].)*

Given that the denoiser uses information per light source, we have to denoise the shadow cast by each light source separately. Our denoising cost is linear with respect to the number of light sources in the scene. However, the quality of the denoised results is higher than that of our attempts to use a common filter for multiple lights, and we therefore opted for a filter per light for these demos.

The input image in Figure 19-6 is rendered with one shadow ray per pixel to simulate the soft illumination cast by the giant rectangular-shaped light source on top of the car. At such a sampling rate, the resulting image is extremely noisy. Our spatial denoiser removes most of the noise, but some artifacts remain. Combining a temporal and a spatial denoising component, the result is close to a ground-truth image rendered with 2048 rays per pixel.



(a) Noisy input (1 spp) (b) Our spatial denoiser

(c) Our spatiotemporal denoiser (d) Reference

Figure 19-6. *(a) Our denoiser works on noisy input rendered with a single shadow ray per pixel. (b) With only the spatial component of our denoiser, some low-frequency artifacts remain. (c) Our spatiotemporal denoiser improves the result further, and (d) it closely matches the ground truth.*

For moderately sized light sources, our spatial denoiser produces high-quality results. In the "Reflections" (Lucasfilm) demo, spatial denoising alone was sufficient to produce shadow quality results that make our artists happy. For the type of giant light sources we used in the "Speed of Light" (Porsche) demo, pure spatially denoised results did not meet our quality bar. Therefore, we also employed a temporal component to our denoiser for the "Speed of Light" (Porsche) demo, which improved reconstruction quality at the cost of slight temporal lag.

19.3.2 RAY TRACED REFLECTIONS

True reflection is another key promise of ray tracing–based rendering. Current rasterization-based techniques such as screen-space reflections (SSR) [11] often suffer from artifacts in offscreen content. Other techniques, such as pre-integrated light probes [5], do not scale well to dynamic scenes and cannot accurately simulate all the features that exist in glossy reflections, such as stretching along the surface normal directions and contact hardening. Furthermore, ray tracing is arguably the most efficient way to handle multiple-bounce reflections on arbitrarily shaped surfaces.

Figure 19-7 demonstrates the type of effect that we were able to produce with ray traced reflections in the "Reflections" (Lucasfilm) demo. Notice the multiple-bounce interreflections among portions of Phasma's armor.

Figure 19-7. *Reflections on Phasma rendered with ray tracing. Notice the accurate interreflections among portions of her armor, as well as the slightly glossy reflections reconstructed with our denoiser.*

19.3.2.1 SIMPLIFIED REFLECTION SHADING

While ray tracing makes it much easier to support dynamic reflections on arbitrary surfaces, even for offscreen content, it is expensive to compute the shading and lighting at the hit points of reflection bounces. To reduce the cost of material evaluation at reflection hit points, we provide the option to use different, artist-simplified materials for ray traced reflection shading. This material simplification has little impact on the final perceived quality, as reflected objects are often minimized on convex reflectors, and removing micro-details in the materials often is not visually noticeable yet is beneficial for performance. Figure 19-8 compares the regular complex material with rich micro-details from multiple texture maps in the primary view (left) and the simplified version used in reflection hit shading (right).

Figure 19-8. *Left: original Phasma materials with full micro-details. Right: simplified materials used for shading the reflection ray hit points.*

19.3.2.2 DENOISING FOR GLOSSY REFLECTIONS

Getting perfectly smooth specular reflections with ray tracing is nice, but in the real world most specular surfaces are not mirror-like. They usually have varying degrees of roughness and bumpiness across their surfaces. With ray tracing one would typically stochastically sample the local BRDF of the material with hundreds to thousands of samples, depending on the roughness and incoming radiance. Doing so is impractical for real-time rendering.

We implemented an adaptive multi-bounce mechanism to drive the reflected rays generation. The emission of reflection bounce rays was controlled by the roughness of the hit surface, so rays that hit geometries with higher roughness were killed earlier. On average we dedicated only two reflection rays to each pixel, for two reflection bounces, so for each visible shading point we had only one BRDF sample. The result was extremely noisy, and we again applied sophisticated denoising filters to reconstruct glossy reflections that are close to ground truth.

We have designed a denoising algorithm that works on only the reflected incoming radiance term. Glossy reflection is an integral of the product of the incoming radiance term L and the BRDF f over the hemisphere around the shading point. We separate the integral of the product into an approximate product of two integrals,

$$L(\omega_o) = \int_{S^2} L(\omega_i) f(\omega_o, \omega_i) |\cos\theta_i| \, d\omega_i \approx \int_{S^2} L(\omega_i) \, d\omega_i \int_{S^2} f(\omega_o, \omega_i) |\cos\theta_i| \, d\omega_i, \quad (2)$$

which simplifies the denoising task. We apply denoising to only the incoming radiance term $\int L(\omega_i)d\omega_i$. The BRDF integral can be separated out and pre-integrated. This is a common approximation for pre-integrated light probes [5]. In addition, the specular albedo is also included in the BRDF, so by filtering only the radiance term, we don't need to worry about overblurring the texture details.

The filter stack has both temporal and spatial components. For the spatial part, we derive an anisotropic-shaped kernel in screen space that respects the BRDF distribution at the local shading point. The kernel is estimated by projecting the BRDF lobe back to screen space, based on hit distance, surface roughness, and normals. The resulting kernel has varying kernel sizes and directions per pixel. This is shown in Figure 19-9.

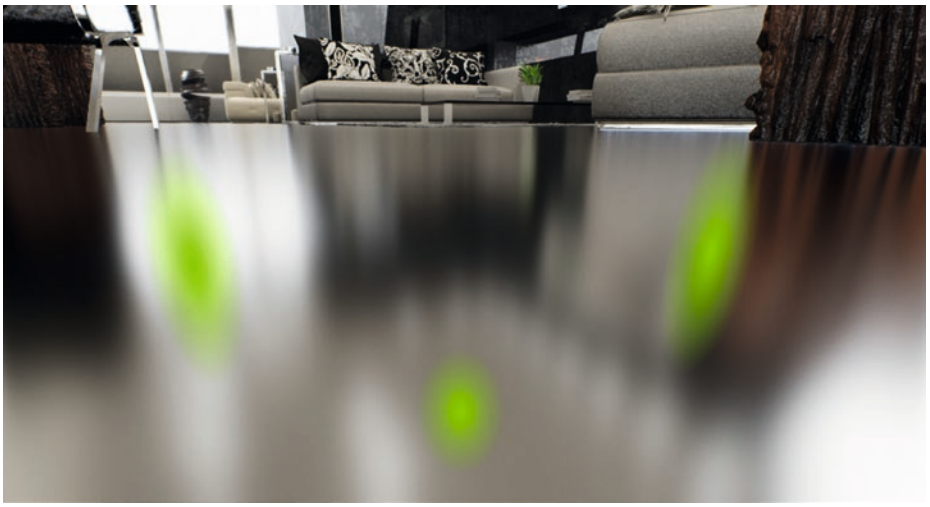


Figure 19-9. *Visualization of the BRDF-based reflection filter kernel. (From Liu [7].)*

Another noteworthy property of our BRDF-based filter kernel is that it can produce moderately rough-looking glossy surfaces by filtering from just mirror-like surfaces, as shown in Figures 19-10, 19-11, and 19-12. Our filter produces convincing results from 1 spp input, closely matching ground-truth rendering with 16384 spp. Please refer to Figures 19-11 and 12 for examples.

Figure 19-10. *The input to our reflection spatial filter, in the case where it is just a perfect mirror reflection image.*

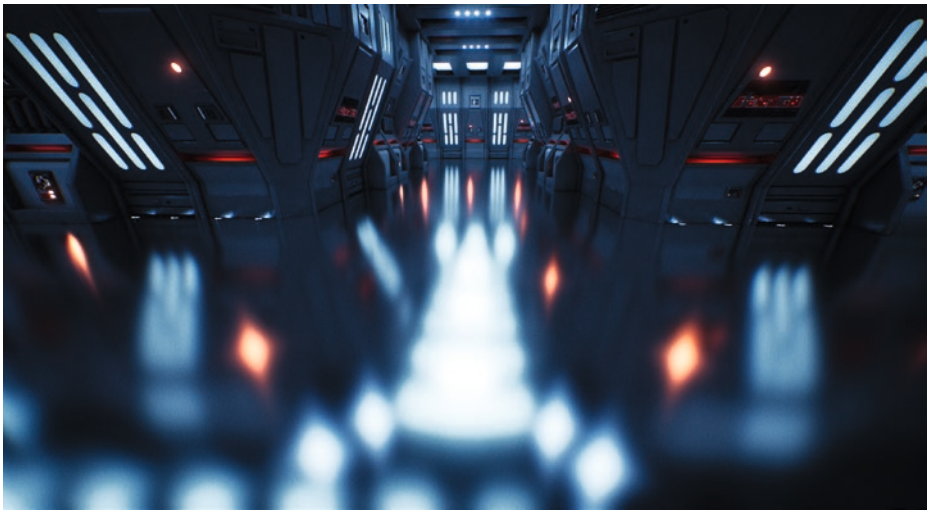


Figure 19-11. *The output of our reflection spatial filter applied on the mirror reflection image in Figure 19-10, simulating a GGX squared-roughness of 0.15. It produces all the expected features of glossy reflections, such as contact hardening and elongation along normal directions.*

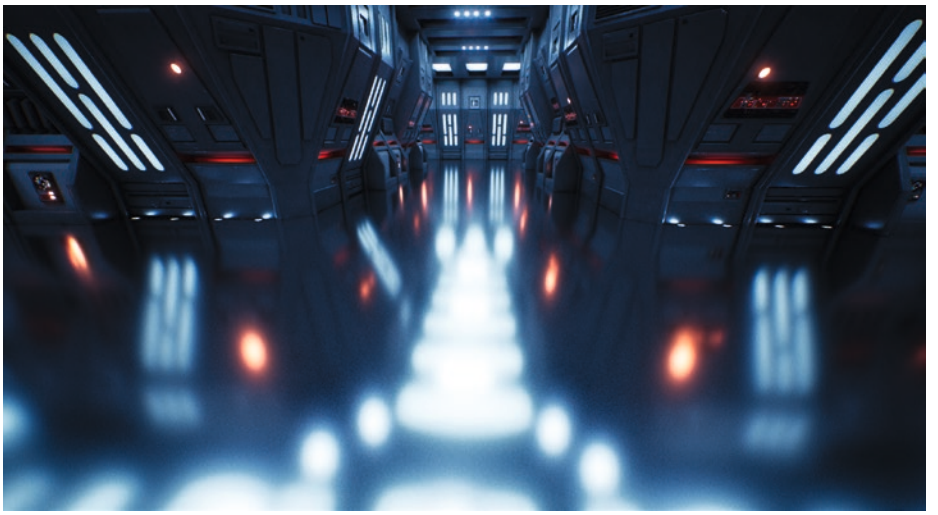


Figure 19-12. *A GGX squared-roughness of 0.15 rendered with unbiased stochastic BRDF sampling with thousands of rays per pixel.*

This spatial filter can faithfully reconstruct glossy surfaces with moderate roughness (GGX squared-roughness less than around 0.25). For higher roughness values, we apply biased stochastic BRDF sampling like Stachowiak et al. [11] and combine the temporal component with the spatial component to achieve better denoised quality.

Temporal reprojection on reflected surfaces requires motion vectors for reflected objects, which can be challenging to obtain. Previously, Stachowiak et al. [11] used the reflected virtual depth to reconstruct the motion vector caused by camera movement for reflected objects inside planar reflectors. However, that approach does not work so well for curved reflectors. In Chapter 32, Hirvonen et al. introduce a novel approach of modeling each local pixel neighborhood as a thin lens and then using thin lens equations to derive the motion vectors of reflected objects. It works well for curved reflectors, and we use this approach to calculate motion vectors in our temporal filter.

19.3.2.3 SPECULAR SHADING WITH RAY TRACED REFLECTIONS

Linearly transformed cosines (LTC) [2] is a technique that produces realistic area light shading for arbitrary roughness analytically, with the caveat that it doesn't handle occlusion. Since our reflection solution produces plausible glossy reflections with one sample per pixel, we can use it to directly evaluate the specular component of material shading of area light sources. Instead of using LTC, we simply treat area

light sources as emissive objects, shade them at the reflection hit point, and then apply our denoising filter to reconstruct the specular shading *including* occlusion information. Figure 19-13 shows a comparison of the two approaches.

(a) LTC (b) Ray traced reflections

Figure 19-13. *In the scene, the floor is a pure specular surface with a GGX squared-roughness of 0.17. (a) The lighting from the two area lights is computed with LTC. While LTC produces correct-looking highlights, the car reflections that should have occluded part of the highlight are missing, making the car feel not grounded. (b) With ray traced reflections, notice how ray tracing handles the correct occlusion from the car while also producing plausible glossy highlights from the two area lights.*

19.3.3 RAY TRACED DIFFUSE GLOBAL ILLUMINATION

In the pursuit of photorealism, in both the "Reflections" (Lucasfilm) and the "Speed of Light" (Porsche) demos, we used ray tracing to compute indirect illumination to increase the realism of the rendered image. The techniques that we used for the two demos are slightly different. For "Reflections" (Lucasfilm) we used ray tracing to fetch irradiance information from precomputed volumetric light maps to compute indirect lighting on the dynamic characters. For the "Speed of Light" (Porsche) demo, we used a more brute-force method of directly doing path tracing with two bounces of indirect diffuse rays from the G-buffer. We used next event estimation to accelerate convergence.

19.3.3.1 AMBIENT OCCLUSION

Ambient occlusion provides an approximation for global illumination that is physically inspired and artist controllable. Decoupling lighting from occlusion breaks physical correctness but gives measurable efficiency. Our technique for applying ambient occlusion is a straightforward application of the same, well-documented algorithm that has been used in film for decades. We fire several rays, in a cosine-hemispherical distribution, centered around a candidate point's shading normal. As a result, we produce a screen-space occlusion mask that globally attenuates the lighting contribution.

While Unreal Engine supports screen-space ambient occlusion (SSAO), we avoided its use in our demonstrations. SSAO suffers from noticeable shortcomings. Its dependence on the view frustum causes vignetting at the borders and does not accurately capture thin occluders that are primarily parallel to the viewing direction. Furthermore, occluders outside the view frustum do not contribute to SSAO's measure. For cinematics such as our demo, an artist would typically avoid such a scenario entirely or dampen effects from larger ray distances. With DXR, however, we can capture directional occlusion that is independent of the view frustum.

19.3.3.2 INDIRECT DIFFUSE FROM LIGHT MAPS

For "Reflections" (Lucasfilm), we desired an ambient occlusion technique that could provide effective color bleeding. While we valued the efficiency of ambient occlusion, its global darkening effect was undesirable to our artists. We implemented an indirect diffuse pass as a reference comparison. For this algorithm, in a similar fashion to traditional ambient occlusion, we cast a cosine-hemispherical distribution of rays from a candidate G-buffer sample. Rather than recording a hit-miss ratio, we recorded the BRDF-weighted result if our visibility ray hit an emitter. As expected, the number of rays needed for a meaningful result was intractable, but they provided a baseline for more approximate techniques.

Rather than resort to brute-force evaluation, we employed Unreal Engine's light mapping solution to provide an approximate indirect contribution. Specifically, we found that substituting the evaluation from our volumetric light map as emission for our ambient occlusion rays provided an indirect result that was reasonable. We also found the resulting irradiance pass to be significantly easier to denoise than the weighted visibility pass from the traditional ambient occlusion algorithm. Comparison images are presented in Figure 19-14.

Figure 19-14. *Comparison of global lighting techniques. Top: screen-space ambient occlusion. Middle: indirect diffuse from light maps. Bottom: reference one-bounce path tracing.*

19.3.3.3 REAL-TIME GLOBAL ILLUMINATION

Apart from using precomputed light maps to render indirect diffuse lighting, we also developed a path tracing solution that improved our global illumination efforts further. We used path tracing with next event estimation to render one-bounce indirect diffuse lighting, before applying the reconstruction filters detailed in Section 19.3.3.4 on the noisy irradiance, which provided much more accurate color bleeding than before.

19.3.3.4 DENOISING FOR AMBIENT OCCLUSION AND DIFFUSE GLOBAL ILLUMINATION

For both demos we used a similar denoiser, which is based on the axis-aligned filter for diffuse indirect lighting by Mehta et al. [9]. For the "Speed of Light" (Porsche) demo the denoising was more challenging. Since we were using brute-force path tracing without any precomputation, we combined the spatial filter based on Mehta et al. with a temporal filter to achieve the desired quality. For the "Reflections" (Lucasfilm) demo, since we were fetching from light map texels nearby, using temporal antialiasing combined with the spatial filter provided good enough quality.

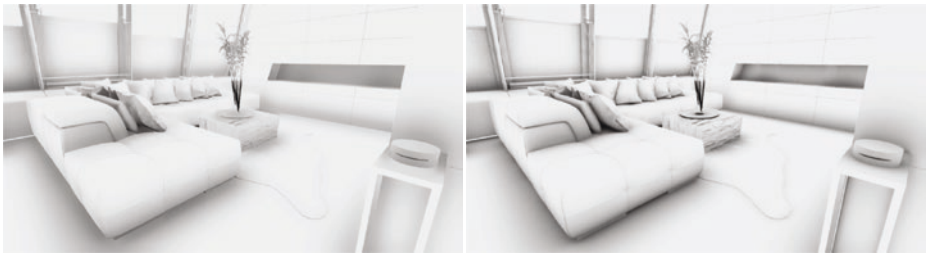
We apply our denoiser only on the indirect diffuse component of the lighting, to avoid overblurring texture details, shadows, or specular highlights, as those are filtered separately in other dedicated denoisers. For the spatial filter, we apply a world-space spatial kernel with footprint derived from hit distance as proposed by Mehta et al. Adapting the filter size with hit distance avoids over blurring details in the indirect lighting and keeps features such as indirect shadows sharper. When combined with a temporal filter, it also reduces the spatial kernel footprint based on how many reprojected samples a pixel has accumulated. For pixels with more temporally accumulated samples, we apply a smaller spatial filter footprint, hence making the results closer to ground truth.

Figures 19-15 shows comparison shots for filtering using a constant radius versus adapting the filter radius based on ray hit distance and temporal sample count. Clearly, using the adapted filter footprint provides much better fine details at the contact region.

(a) Uniform kernel (b) Adaptive kernel

Figure 19-15. *Indirect lighting filtered with (a) a uniform world-space radius and (b) an adaptive kernel. The adaptive kernel size is based on average ray hit distance and accumulated temporal sample count.*

The same idea also helps with ray traced ambient occlusion denoising. In Figure 19-16 we compare (a) denoised ray traced ambient occlusion with a constant world-space radius, with (b) denoised ambient occlusion using adaptive kernel radius guided with hit distance and temporal sample-count.



(a) Uniform kernel (b) Adaptive kernel

Figure 19-16. *Ambient occlusion filtered with (a) a uniform world-space radius and (b) an adaptive kernel. The adaptive kernel size is based on average ray hit distance and accumulated temporal sample count.*

It is clear again that using adaptive filter sizes leads to better-preserved contact details in the denoised ambient occlusion.

19.3.4 RAY TRACED TRANSLUCENCY

The "Speed of Light" (Porsche) demo presented a number of new challenges. The most obvious initial challenge to the team was that of rendering glass. Traditional methods for rendering real-time translucency conflict with deferred rendering algorithms. Often, developers are required to render translucent geometry in a separate forward pass and composite the result over the main deferred rendering. Techniques that can be applied to deferred rendering are often unsuitable for translucent geometry, creating incompatibilities that make integration of translucent and opaque geometry difficult.

Fortunately, ray tracing provides a natural framework for representing translucency. With ray tracing, translucent geometry can easily be combined with deferred rendering in a way that unifies geometry submission. It provides arbitrary translucent depth complexity as well as the capability to correctly model refraction and absorption.

19.3.4.1 RAY GENERATION

Our implementation of ray traced translucency in Unreal Engine uses a separate ray tracing pass similar to the one used for ray traced reflections. In fact, most of the shader code is shared between these two passes. There are, however, a few small nuanced differences between how the two behave. The first one is the use of early-ray termination to prevent unnecessary traversal through the scene once the throughput of the ray is close to zero; i.e., if traversed farther, its contribution is negligible. Another difference is that translucent rays are traced with a maximum ray length that prevents hitting the opaque geometry that has already being fully shaded and stored at the corresponding pixel. However, if refraction is performed, a translucent hit may result in a new ray in an arbitrary direction, and this new ray or its descendants may hit opaque geometry, which will need to be shaded. Before we perform any lighting on such opaque hits, we perform a reprojection of the opaque hit point to the screen buffer, and if valid data are found after this reprojection step, they are used instead. This simple trick allowed us to take advantage of the higher visual quality achieved when performing all the ray traced lighting and denoising on opaque geometry in the G-buffer. This can work for some limited amount of refraction, although the results can be incorrect due to specular lighting being computed with the wrong incoming direction in such cases.

Another key difference with the reflection pass is the ability for translucent rays to recursively generate reflection rays after hitting subsequent interfaces. This was not completely straightforward to implement using HLSL due to the lack of support for recursion in the language. By *recursion* we do not mean the ability to trace rays from a hit shader, but the ability for a simple HLSL function to call itself. This is simply not allowed in HLSL, but it is desirable when implementing a Whitted-style ray tracing algorithm, as we did in this case. To work around this limitation of HLSL, we instantiated the same code into two functions with different names. We effectively moved the relevant function code into a separate file and included this file twice, surrounded by preprocessor macros that set the function name each time, resulting in two different instantiations of the same function code with different names. We then made one of the two function instantiations call the other one, thus allowing us to effectively have recursion with a hard-coded limit of one level. The resulting implementation permits translucent paths, with optional

refraction, where each hit along the path could trace "recursive" reflection rays in addition to shadow rays. Reflections traced off translucent surfaces along this path could potentially bounce up to a selected number of times. However, if at any of these bounces a translucent surface was hit, we did not allow additional recursive reflection rays to be traced.

Homogeneous volumetric absorption following the Beer-Lambert law was added to our translucency pass to model thick glass and to approximate the substrate. To correctly model homogeneous bounded volumes, additional constraints were placed on the geometry. Ray traversal was modified to explicitly trace against both frontfacing and backfacing polygons to overcome issues with intersecting, non-manifold geometry. The improved visual realism was considered not worth the slight added cost for the "Speed of Light" (Porsche) demo and was not included in its final version.

19.4 CONCLUSIONS

The recent introduction of dedicated hardware for ray tracing acceleration and the addition of ray tracing support in graphics APIs encouraged us to be innovative and experiment with a new way of hybrid rendering, combining rasterization and ray tracing. We went through the engineering practice of integrating ray tracing in Unreal Engine 4, a commercial-grade game engine. We invented innovative reconstruction filters for rendering stochastic effects such as glossy reflections, soft shadows, ambient occlusion, and diffuse indirect illumination with as few as a single path per pixel, making these expensive effects more practical for use in real time. We have successfully used hybrid rendering to create two cinematic-quality demos.

ACKNOWLEDGMENTS

As a historical note, Epic Games, in collaboration with NVIDIA and ILMxLAB, gave the first public demonstration of real-time ray tracing in Unreal Engine during Epic's "State of Unreal" opening session at *Game Developers Conference* in March 2018. The demo showed *Star Wars* characters from *The Force Awakens* and *The Last Jedi* built with Unreal Engine 4. It was originally run on NVIDIA's RTX technology for Volta GPUs via Microsoft's DirectX Raytracing API.

Mohen Leo (ILMxLAB) joined Marcus Wassmer and Jerome Platteaux from Epic Games in the development and presentation of the techniques used in that demo. ILMxLAB is Lucasfilm's immersive entertainment division, known best for their work on CARNE y ARENA, *Star Wars: Secrets of the Empire*, and the upcoming *Vader Immortal: A Star Wars VR Series*. Many others who have worked or consulted

on the ray tracing content and implementation for UE4 include Guillaume Abadie, Francois Antoine, Louis Bavoil, Alexander Bogomjakov, Rob Bredow, Uriel Doyon, Maksim Eisenstein, Judah Graham, Evan Hart, Jon Hasselgren, Matthias Hollander, John Jack, Matthew Johnson, Brian Karis, Kim Libreri, Simone Lombardo, Adam Marrs, Gavin Moran, Jacob Munkberg, Yuriy O'Donnell, Min Oh, Jacopo Pantaleoni, Arne Schober, Jon Story, Peter Sumanaseni, Minjie Wu, Chris Wyman, and Michael Zhang.

Star Wars images are used courtesy of Lucasfilm.

REFERENCES

[1] Binder, N., Fricke, S., and Keller, A. Fast Path Space Filtering by Jittered Spatial Hashing. In *ACM SIGGRAPH Talks* (2018), pp. 71:1–71:2.

[2] Eric Heitz, Jonathan Dupuy, S. H., and Neubelt, D. Real-Time Polygonal-Light Shading with Linearly Transformed Cosines. *ACM Transactions on Graphics 35*, 4 (2017), 41:1–41:8.

[3] Heitz, E., Hill, S., and McGuire, M. Combining Analytic Direct Illumination and Stochastic Shadows. In *Symposium on Interactive 3D Graphics and Games* (2018), pp. 2:1–2:11.

[4] Kajiya, J. T. The Rendering Equation. *Computer Graphics (SIGGRAPH)* (1986), 143–150.

[5] Karis, B. Real Shading in Unreal Engine 4. Physically Based Shading in Theory and Practice, SIGGRAPH Courses, August 2013.

[6] Keller, A. Instant Radiosity. In *Proceedings of SIGGRAPH* (1997), pp. 49–56.

[7] Liu, E. Low Sample Count Ray Tracing with NVIDIA's Ray Tracing Denoisers. Real-Time Ray Tracing, SIGGRAPH Courses, August 2018.

[8] Mehta, S., Wang, B., and Ramamoorthi, R. Axis-Aligned Filtering for Interactive Sampled Soft Shadows. *ACM Transactions on Graphics 31*, 6 (Nov 2012), 163:1–163:10.

[9] Mehta, S. U., Wang, B., Ramamoorthi, R., and Durand, F. Axis-aligned Filtering for Interactive Physically-based Diffuse Indirect Lighting. *ACM Transactions on Graphics 32*, 4 (July 2013), 96:1–96:12.

[10] Salvi, M. An Excursion in Temporal Supersampling. From the Lab Bench: Real-Time Rendering Advances from NVIDIA Research, Game Developers Conference, 2016.

[11] Stachowiak, T. Stochastic Screen Space Reflections. Advances in Real-Time Rendering, SIGGRAPH Courses, 2018.

[12] Yan, L.-Q., Mehta, S. U., Ramamoorthi, R., and Durand, F. Fast 4D Sheared Filtering for Interactive Rendering of Distribution Effects. *ACM Transactions on Graphics 35*, 1 (2015), 7:1–7:13.

[13] Zimmer, H., Rousselle, F., Jakob, W., Wang, O., Adler, D., Jarosz, W., Sorkine-Hornung, O., and Sorkine-Hornung, A. Path-Space Motion Estimation and Decomposition for Robust Animation Filtering. *Computer Graphics Forum 34*, 4 (2015), 131–142.

[14] Zwicker, M., Jarosz, W., Lehtinen, J., Moon, B., Ramamoorthi, R., Rousselle, F., Sen, P., Soler, C., and Yoon, S.-E. Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. *Computer Graphics Forum (Proceedings of Eurographics—State of the Art Reports) 34*, 2 (May 2015), 667681.