CHAPTER 48

RAY TRACING IN FORTNITE

Patrick Kelly,[1] Yuriy O'Donnell,[1] Kenzo ter Elst,[1] Juan Cañada,[1] and Evan Hart[2]
[1]*Epic Games*
[2]*NVIDIA*

ABSTRACT

In this chapter we describe implementation details of some of the Unreal Engine 4 ray tracing effects that shipped in *Fortnite* Season 15. In particular, we dive deeply into ray traced reflections and global illumination. This includes goals, practical considerations, challenges, and optimization techniques.

48.1 INTRODUCTION

DirectX Raytracing (DXR) in Unreal Engine 4 (UE4) was showcased for the first time at Game Developers Conference 2018 in *Reflections*, a Lucasfilm *Star Wars* short movie made in collaboration with ILMxLAB and NVIDIA. A year after that, the first implementation of ray tracing in UE4 was released in

Figure 48-1. *Fortnite rendered with ray tracing effects.*

© NVIDIA 2021
A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_48

UE 4.22. Initially, the focus of the engineering team was set on feature completion and stability rather than on performance for real-time graphics applications. For this reason, during the first year and a half, ray tracing adoption was significantly higher in enterprise applications for architecture, automotive, or film rather than in games. A detailed description of how ray tracing was implemented in UE4 can be found in the chapter "Cinematic Rendering in UE4 with Real-Time Ray Tracing and Denoising" in the first volume of *Ray Tracing Gems* [10].

The feasibility of ray tracing for games increased quickly due to the vast improvements happening both in hardware and software since 2018. UE4 licensees started to include some ray tracing effects in games and other applications that demand high frame rates. Early in 2020, Unreal Engine ray tracing was transitioning from beta stage to production, and the engineering team decided it was the right time to battle-test it under challenging conditions. *Fortnite* was perfect for this task, not only due to its massive scale but also because it possessed many other characteristics that make it difficult to implement in-game ray tracing. Namely, the content creation pipeline is very well defined and changing it to embrace ray tracing was not an option. Moreover, content updates happen very often so it is not possible to adjust parameters to make specific versions look good, but any modification should be relatively permanent and behave correctly in future updates.

This chapter explains how the ray tracing team at Epic Games implemented ray tracing techniques in *Fortnite* Season 15. We describe the challenges, the trade-offs, and the solutions found.

48.2 GOALS

The main goal of the project was to ship ray tracing in *Fortnite*, both to improve the game's visuals and to battle-proof the UE4 ray tracing technology. From the technical perspective, the initial objective was to run the game on a system with an 8-core CPU (i7-7000 series or equivalent) and an NVIDIA 2080 Ti graphics card with the following requirements:

- Frame rate: 60 FPS.
- Resolution: 1080p.
- Ray tracing effects: shadows, ambient occlusion, and reflections.

As described in the following sections, improvements that happened during the project helped to achieve more ambitious goals. Novel developments in

ray traced reflections, global illumination, and denoising, plus the integration of NVIDIA's Deep Learning Super Sampling (DLSS), made it possible to target higher resolutions and more sophisticated lighting effects such as ray traced global illumination (RTGI).

From the art and content creation point of view, the team was not aiming for a dramatic change in the look, but the goal was to achieve specific improvements on key lighting effects that could make the visuals more pleasant by removing some artifacts introduced by screen-space effects. *Fortnite* is a non-photorealistic game, and the intention was to avoid an experience where ray tracing and rasterization looked too different.

48.3 CHALLENGES

From the performance side, initial tests showed both the CPU and GPU were far away from the initial performance goals when ray tracing was enabled. When running on the target hardware, CPU time was around 24 ms per frame on average, with some peaks of more than 30 ms. GPU performance was also far from the goal. While some scenarios were fast enough, others with more complex lighting were in the 30–40 ms per frame range. Some pathological cases with many dynamic geometries (such as trees) were extremely slow—on the order of 100 ms per frame.

Besides performance, there were other areas that presented interesting challenges. A remarkable one was the content creation pipeline. *Fortnite* manages an extraordinarily large amount of assets that are updated at very high frequency. Changing assets to look better in ray tracing was not possible because the overload on the content team would be unacceptable. For example, to improve the performance of ray traced reflections, the team considered adding a flag to set if an object was casting reflection rays or not. However, after further evaluation it was clear that such a solution would not scale. Any improvement had to work reasonably well *automatically* for all the existing and future content.

The look and feel of the game also presented a challenge. *Fortnite* does not have a photorealistic visual style. Most of the surfaces are highly diffuse. Reflections do not play an important role, except when water is present. Many talented artists and engineers have worked hard for years adjusting content and creating technology to make the game look good and to avoid artifacts that screen-space rasterized techniques produce. Making ray tracing shine

under these circumstances—without being able to change content—was a difficult task.

Another challenge worth mentioning was that the rendering API (known in Unreal Engine as the *Render Hardware Interface* (RHI)) used with ray tracing was not the default one. The default RHI in *Fortnite* is DirectX 11, but DXR runs on DirectX 12. This means that when enabling ray tracing, the game was exercising code paths that have been tested significantly less than the default ones. As expected, this revealed stability and performance issues that were not ray tracing specific but were critical to fix to make the game shippable. Improving the DirectX 12 RHI has been one of the most positive side effects of this initiative.

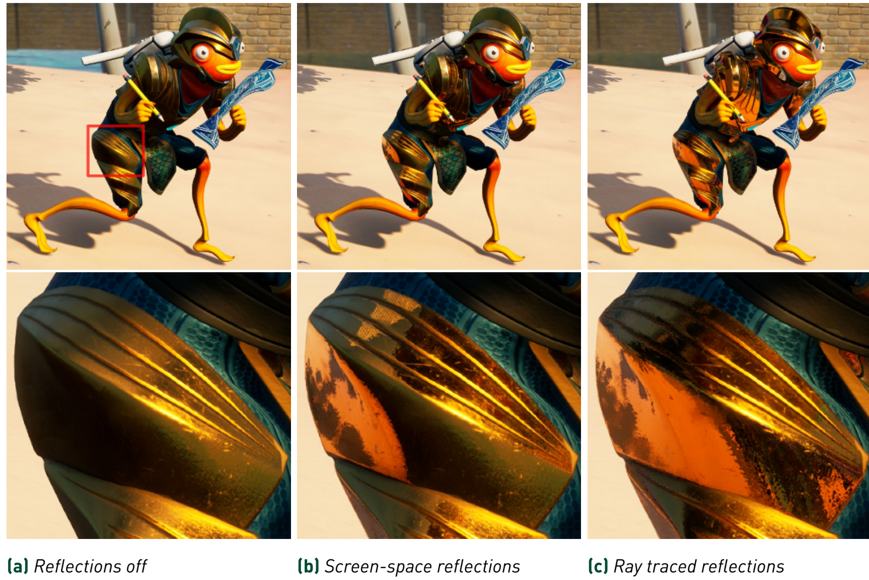
Lastly, another challenge came from a self-imposed limitation. All the technology developed for this project had to be included in the UE4 code base without any modification. Neither *Fortnite*, nor any other project developed at Epic Games, is allowed to customize the engine code. Though this can represent a problem in some cases, the long-term benefit of maintaining only one game engine overcomes any difficulties.

48.4 TECHNOLOGIES

This section describes the most important technologies that were improved or entirely developed during the project.

48.4.1 REFLECTIONS

Ray traced reflections have been a part of Unreal Engine for some time and were primarily used for cinematic rendering [10]; however, the *Fortnite* Season 15 release was the first time this effect was used in a game at Epic. Though our previous use cases required real-time performance, the goals were quite different from a game. A typical cinematic demo ran at 24 hertz at 1080p resolution and could require a high-end GPU. The *Fortnite* ray tracing target was *at least* 60 hertz at 1080p (4K with DLSS) on an NVIDIA 2080 Ti. Some optimizations and sacrifices had to be made to reach this goal. We made an experimental ray traced reflection implementation targeted specifically at games. It shared the main ideas from our original reflection shader, but shed most of the high-end rendering features such as multi-bounce reflections, translucent materials in reflections, physically based clear coat, and more. Figure 48-2 shows a comparison of the new ray traced reflections mode with screen-space reflections and with no reflections.



(a) *Reflections off* **(b)** *Screen-space reflections* **(c)** *Ray traced reflections*

Figure 48-2. *A comparison of reflection rendering modes.*

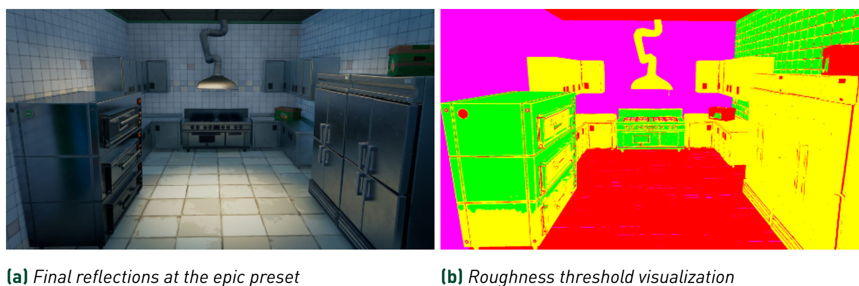
ALGORITHM OVERVIEW

The Unreal Engine reflection pipeline uses a sorted-deferred material evaluation scheme (Figure 48-3). First, reflection rays are generated based on G-buffer data and then traced to find the closest surfaces and their associated material IDs. The hit points are then sorted by material ID / shader. Finally, sorted hit points are used to dispatch another ray tracing pass that performs material evaluation and lighting using the full ray tracing pipeline state object (RTPSO).

The goal of this sorted pipeline is to improve material shader execution coherence (SIMD efficiency). Because reflection rays are randomized, pixels that are close in screen space will often generate rays that hit surfaces that are far apart—increasing the probability that they use different materials. If

Trace Rays → **Sort Hits by Material** → **Evaluate Materials** → **Lighting**

Figure 48-3. *The reflection pipeline using sorted-deferred material evaluation.*



(a) *Final reflections at the epic preset* **(b)** *Roughness threshold visualization*

Figure 48-4. *A visualization of different roughness threshold levels and corresponding GPU performance. Green: medium reflection quality preset, roughness ≤ 0.35, 0.7 ms. Yellow: high preset, roughness ≤ 0.55, 1.28 ms. Red: epic preset, roughness ≤ 0.75, 1.72 ms. Magenta: culled surfaces with roughness > 0.75, 2.09 ms. Timings given for NVIDIA RTX 3090 at* 1920×1080 *resolution.*

hit points with different materials end up in the same GPU wave,[1] the performance will drop roughly proportionally to the number of unique materials. Though theoretically a high-level ray tracing API such as DirectX Raytracing allows for automatic sorting to avoid this performance issue, in practice none of the drivers or hardware available at the time implemented this optimization. Implementing explicit sorting at the application level significantly improved performance [1], as described later.

RAY GENERATION

Reflection rays are generated using GGX distribution sampling based on G-buffer data [7]. To save GPU time, a roughness threshold is used to decide if a simple reflection environment map lookup can be used instead of tracing rays. The threshold value is mapped to the reflection quality parameter in *Fortnite* graphics options. We chose values 0.35, 0.55, and 0.75 for *medium*, *high*, and *epic* quality presets, respectively. All surfaces with roughness over 0.75 are culled, as performance cost is too high compared to the visual improvement. A special *low* preset also exists, disabling ray traced reflections on everything except water. Figure 48-4 shows a visualization of these quality presets and their GPU performance.

Because *Fortnite* content was not designed with ray traced reflection technology in mind, most of the assets were mastered for screen-space

[1] HLSL terminology is used in this chapter. *Wave* refers to a set of GPU threads executed simultaneously in a SIMD fashion. Similar concepts include Subgroup (Vulkan), Warp (NVIDIA), Wavefront (AMD).

Listing 48-1. *HLSL source code of the reflection roughness remapping function.*

```
1 float ApplySmoothBias(float Roughness, float SmoothBias)
2 {
3     // SmoothStep-like function for Roughness values
4     // lower than SmoothBias, original Roughness otherwise.
5     float X = saturate(Roughness / SmoothBias);
6     return Roughness * X * X * (3.0 - 2.0 * X);
7 }
```

reflections, which use pure mirror-like rays and therefore look quite sharp. A simple roughness threshold is used to cull screen-space reflections from most surfaces that are meant to appear rough/diffuse. Unfortunately, this means that physically based ray traced reflections appeared quite dull most of the time. Tweaking all materials manually was not an option due to the sheer amount of content in the game. Shown in Listing 48-1, an automatic solution was implemented that biases surface roughness during GGX sampling, making surfaces slightly shinier.

Shown in Figure 48-5, this remapping function pushes roughness values below some threshold closer to zero, but leaves higher values intact. This particular function was designed to preserve material roughness map

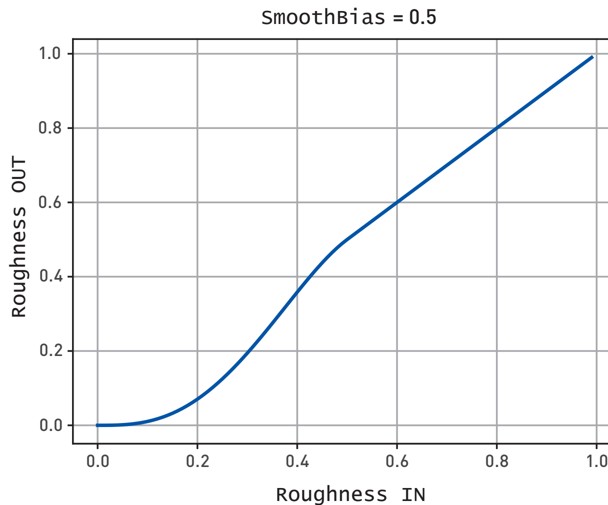


Figure 48-5. *A graph of the roughness remapping function for* `ApplySmoothBias(Roughness, 0.5)`*. This makes surfaces that are already smooth even smoother, while leaving rougher ones unchanged.*

(a) *Screen space* **(b)** *SmoothBias* = 0.0 **(c)** *SmoothBias* = 0.5 **(d)** *SmoothBias* = 1.0

Figure 48-6. *The visual effect of varying the smoothness bias.*

contributions without clipping, while remaining smooth over the full roughness range. A bias value of 0.5 was used in *Fortnite*, as it is a good compromise between the desired look and physical accuracy. As a small bonus, GPU performance was slightly improved in some scenes because mirror-like reflection rays are naturally more coherent (making them faster to trace). Figure 48-6 compares the visual results produced by varying the smoothness bias value.

MATERIAL ID GATHERING

Unreal Engine uses a specialized lightweight pipeline state object for the initial reflection ray tracing. It consists of a single ray generation shader, a trivial miss shader, and a common tiny closest-hit shader for all geometry in the scene. Shown in Listing 48-2, this shader aims to find the closest intersections without incurring any shading overhead.

Listing 48-2. *The material ID gathering closest-hit shader.*

```
1  struct FDeferredMaterialPayload
2  {
3      float  HitT;       // Ray hit depth or -1 on miss
4      uint   SortKey;    // Material ID
5      uint   PixelCoord; // X in low 16 bits, Y in high 16 bits
6  };
7
8  [shader("closesthit")]
9  DeferredMaterialCHS(
10     FDeferredMaterialPayload Payload,
11     FDefaultAttributes Attributes)
12 {
13     Payload.SortKey = GetHitGroupUserData(); // Material ID
14     Payload.HitT    = RayTCurrent();
15 }
```

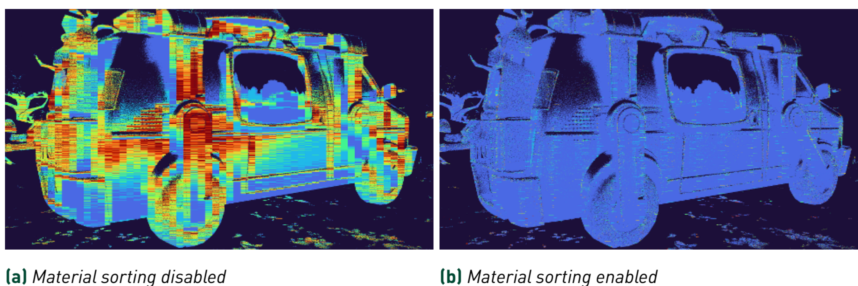
Results of the material ID gather pass are written to a deferred material payload buffer in 64×64 tile order for subsequent sorting.

RAY SORTING

Reflection ray hit points are sorted by material ID using a compute shader. Sorting is performed in blocks of 64×64 pixel screen-space tiles (4,096 total pixels). Rather than performing a full sort, rays are binned into buckets per tile. The number of buckets is the number of total pixels in a tile divided by an expected thread group size, e.g., 4,096 pixels / 32 threads = 128 buckets. There may be many more different materials in the full scene (there are approximately 500 materials in an average *Fortnite* RTPSO), but it is unlikely that a given tile will contain all of them. If there are more than 128 different materials in a tile, it is not possible to sort them into perfectly coherent groups anyway. Increasing the number of bins does not improve much beyond reducing the chance of collisions in the material ID $\rightarrow$ bucket ID mapping. In practice, we did not see any efficiency improvement from increasing the number of buckets.

The binning is done as a single compute shader pass with one thread group per tile, using `groupshared` memory to store intermediate results. A straightforward binning algorithm is used here: (1) load elements from the deferred material buffer, (2) count the number of elements per sorting bucket using atomics, (3) build a prefix sum over the counts to compute the sorted index, and (4) write the elements back into the same deferred material buffer for material evaluation. Note that the original ray dispatch index must be preserved throughout the full pipeline and is read by hit shaders from the ray payload structure (the `DispatchRaysIndex()` intrinsic may not be used outside of the original (unsorted) ray generation shader).

As demonstrated by Figures 48-7 and 48-8, the sorting scheme reduces shader execution divergence quite effectively. Most waves in a typical *Fortnite* frame contain a single material shader when sorting is used. Although this is effective, note that the GPU performance impact is very scene dependent. For cases where most rays naturally hit the same material or the sky, there may be no performance improvement or even a small slowdown due to the sorting overhead. However, for complex scenes with many high-roughness surfaces, the speedup may be as high as $3\times$. The average performance improvement in *Fortnite* is around $1.6\times$. Sorting is used for reflections on everything except water. Reflection rays from water are highly coherent and typically hit the sky, so there is little benefit from sorting.



(a) *Material sorting disabled* **(b)** *Material sorting enabled*

Figure 48-7. *A visualization of the SIMD execution efficiency improvement from ray sorting. Dark blue areas belong to waves that did not require material shaders at all (rays that hit the sky or were culled by the roughness threshold). Brighter colors show waves that contained between 1 (light blue) and 8+ (dark red) different shaders.*

Sorting Disabled	**4.17 ms**
Tracing	0.62 ms
Shading	3.55 ms
Sorting Enabled	**2.12 ms**
Tracing	0.62 ms
Sorting	0.15 ms
Shading	1.35 ms

(a) *Raw reflection output* **(b)** *GPU time measurements*

Figure 48-8. *Sorting performance comparison on the NVIDIA RTX 3090 at* 1920×1080 *resolution.*

MATERIAL EVALUATION

The material evaluation step loads ray parameters from a buffer that was written during the initial ray generation phase, but shortens rays to cover only a small segment around the triangle that was previously hit. Full material shaders are then invoked using `TraceRay`. We have found that despite the extra cost of tracing a second ray, this approach is significantly faster than naive `TraceRay` in a monolithic reflection pipeline.

Unreal Engine uses platform-specific APIs to launch closest-hit shaders directly, without incurring the traversal cost where possible. A viable alternative path on PC could be to use DXR callable shaders for all closest-hit shading, while using any-hit shaders for alpha mask evaluation. This comes with its own set of trade-offs, such as the need for all ray generation shaders

to explicitly pass hit parameters to callable shaders via the payload. Ultimately, the shortened ray approach was a good compromise between performance and simplicity at the time.

Avoiding any-hit shader processing as much as possible is an important performance optimization when ray tracing. Unfortunately, typical game scenes do contain alpha-masked materials that must look correct in reflections. We found that the vast majority of reflection rays in practice tend to hit either entirely opaque materials or opaque parts of alpha-masked materials, such as the solid part of a tree leaf mask texture. This makes it possible to evaluate opacity in the closest-hit shader and write the opacity status into the ray payload. All initial ray tracing can then use `RAY_FLAG_FORCE_OPAQUE`, and the ray generation shader gains the ability to decide if a "full-fat" ray needs to be traced without the `FORCE_OPAQUE` flag. This is shown in Listing 48-3.

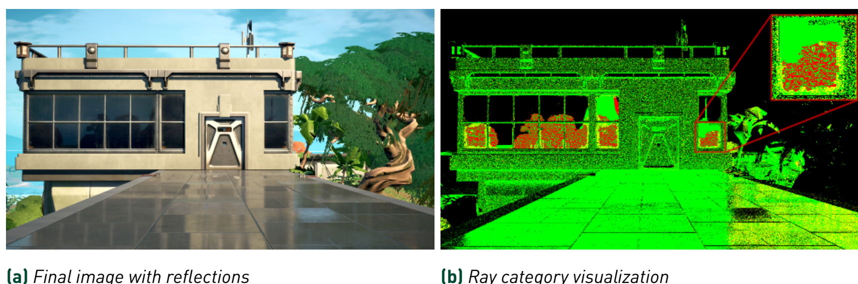
Listing 48-3. *Pseudocode for alpha-masked material ray culling in the ray generation shader.*

```
1 TraceRay(TLAS, RAY_FLAG_FORCE_OPAQUE, ..., Payload);
2 if (GBuffer.Roughness <= Threshold && Payload.IsTransparent())
3 {
4     TraceRay(TLAS, RAY_FLAG_NONE, ..., Payload);
5 }
```

Fortnite uses an aggressive any-hit roughness threshold of 0.1, meaning that alpha-masked materials, such as vegetation, are rendered fully opaque in reflections on rough surfaces. Only near-perfect mirrors show the proper alpha cutouts, as shown in Figure 48-9. Though this is a quality concession, it works fairly well for *Fortnite* in practice. Performance improvement from this optimization depends on the scene, but we measured approximately 1.2× speedup on average in *Fortnite*, with some scenes approaching 2×. We found that the benefit of `FORCE_OPAQUE` *easily* pays for the cost of retracing some rays in our typical frame.

LIGHTING

Direct lighting evaluation in Unreal Engine's ray traced effects is mostly split between ray generation and miss shaders. Only emissive and indirect lighting comes from closest-hit shaders, as it may involve reading from textures or light maps. The ray generation shader contains a light loop with grid-based culling and light shape sampling. Shadows for lights in reflections are always calculated using ray tracing (instead of shadow mapping). Light irradiance is



(a) *Final image with reflections* **(b)** *Ray category visualization*

Figure 48-9. *A visualization of any-hit material evaluation. Green areas show reflection rays that hit opaque geometries or opaque parts of alpha-masked materials (as reported by the closest-hit shader). Yellow areas show where non-opaque ray tracing was skipped due to the roughness threshold. Red areas show where non-opaque rays were traced. Performance on NVIDIA RTX 3090 at* 1920×1080 *resolution was 1.8 ms with opaque ray optimization and 2.4 ms without.*

computed in a miss shader. If a light does not use shadows, it still goes through the common `TraceRay` path with a forced miss by setting `TMin = TMax` and `InstanceInclusionMask = 0`. This is similar to launching a callable shader, but avoids an extra transition in and out of the ray generation shader. Using miss shaders in this way slims down the ray generation shader code and results in better occupancy, leading to a performance increase on all of our target platforms.

This design also improves SIMD efficiency during lighting, as rays within one wave may hit different materials. Execution diverges during material evaluation, but re-converges for lighting. Material sorting does not fully solve the divergence problem as it's not possible to always perfectly fill waves, leaving some waves only partially utilized.

Keeping all lighting calculations in the ray generation shaders allows for smaller closest-hit shaders. This is beneficial in many ways, from iteration speed to code modularity and game patch sizes. Unreal Engine uses a single common set of material hit shaders for all ray tracing effects and a single main material ray payload structure, shown in Listing 48-4. There is a trade-off akin to forward versus deferred shading in raster graphics pipelines, where the G-buffer provides an opaque interface between different rendering stages and allows decoupled/hot-swappable algorithms. However, this comes at the cost of a large G-buffer memory footprint or a larger ray payload structure.

Listing 48-4. *The standard UE4 ray tracing material ray payload.*

```
1  struct FPackedMaterialClosestHitPayload
2  {
3      float HitT                                          // 4  bytes
4      uint PackedRayCone;                                 // 4  bytes
5      float MipBias;                                      // 4  bytes
6      uint RadianceAndNormal[3];                          // 12 bytes
7      uint BaseColorAndOpacity[2];                        // 8  bytes
8      uint MetallicAndSpecularAndRoughness;               // 4  bytes
9      uint IorAndShadingModelIDAndBlendingModeAndFlags; // 4  bytes
10     uint PackedIndirectIrradiance[2];                   // 8  bytes
11     uint PackedCustomData;                              // 4  bytes
12     uint WorldTangentAndAnisotropy[2];                  // 8  bytes
13     uint PackedPixelCoord;                              // 4  bytes
14 }; // 64 bytes total
```

LIGHT SOURCE CULLING

In addition to optimizing the material evaluation costs, we needed to balance the cost of illuminating reflected surfaces. *Fortnite*'s expansive world creates scenarios where a large number of lights may potentially impact a surface. As surfaces often come close to being affected by up to 256 lights (the maximum supported in reflections), we required a strategy to select only the lights that meaningfully affect a surface. Our chosen approach uses a world-aligned, camera-centered 3D grid to perform light culling.

Fortnite's large world requires a trade-off with cell sizes: large cells hurt the efficiency of culling, but small cells are not practical due to the required coverage area. A compromise was made with cells increasing in size exponentially based on the distance to the camera. To allow the cells to fit together in a grid, scaling is applied independently for each axis. This produces modest 8 meter^3 ($2 \times 2 \times 2$) cells near the camera, while still having discrete cells 100 meters from the camera. A further tweak, which simplifies the mathematics, keeps the first two layers of cells the same size. Figure 48-10 shows the layout of the grid in two dimensions, and Listing 48-5 shows the HLSL shader code used to compute a cell address for an arbitrary position in world space.

Shown in Figure 48-11, the final representation of the light culling data is implemented as a three-level structure on the GPU. The grid is the topmost structure, with 128 bits stored per grid cell that encodes either a list of up to 11 indices of 10 bits each or a count and offset into an auxiliary buffer. This auxiliary buffer holds indices for cells that contain too many lights for the

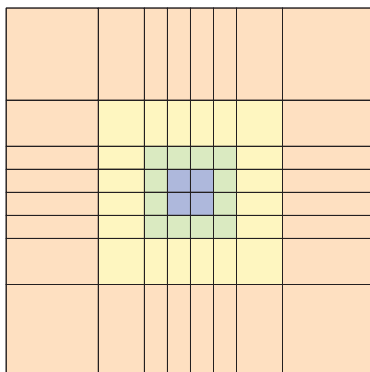


Figure 48-10. *A 2D slice of the light grid showing the four closest rings of cells.*

Listing 48-5. *HLSL code for light grid cell addressing.*

```
1  int3 ComputeCell(float3 WorldPosition)
2  {
3      float3 Position = WorldPos - View.WorldViewOrigin;
4      Position /= CellScale;
5
6      // Use symmetry about the viewer.
7      float3 Region = sign(Position);
8      Position = abs(Position);
9
10     // Logarithmic steps with the closest cells being 2x2x2 scale units
11     Position = max(Position, 2.0f);
12     Position = min(log2(Position) - 1.0f, (CellCount/2 - 1));
13
14     Position = floor(Position);
15     Position += 0.5f;              // Move the edge to the center.
16     Position *= Region;            // Map it back to quadrants.
17
18     // Remap [-CellCount/2, CellCount/2] to [0, CellCount].
19     Position += (CellCount / 2.0f);
20
21     // Clamp to within the volume.
22     Position = min(Position, (CellCount - 0.5f));
23     Position = max(Position, 0.0f);
24
25     return int3(Position);
26 }
```

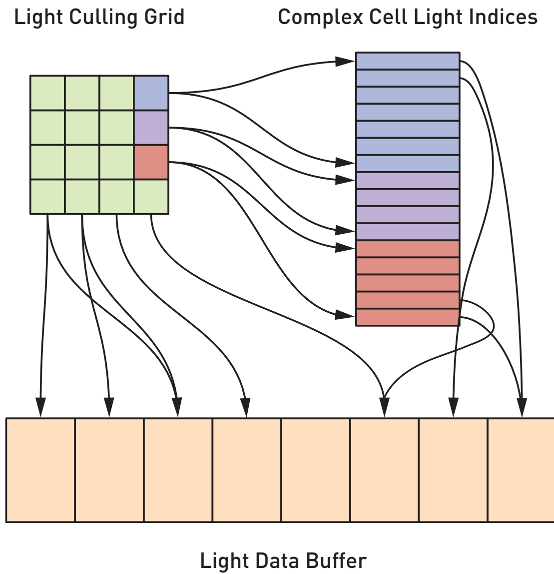


Figure 48-11. *Grid cells either address the light buffer directly or offset into an array of light indices for complex cells.*

compact format. The lowest level is a structured buffer of light data parameters to which the indices refer. Listing 48-6 shows how indices are retrieved for a grid cell. Both the grid structure and the auxiliary buffer are generated each frame by a compute shader culling lights against the grid.

48.4.2 GLOBAL ILLUMINATION

Global illumination was not an initial requirement for ray tracing in *Fortnite*. Originally released as an experimental algorithm in Unreal Engine 4.22, brute-force global illumination was not practical for uses that demanded real-time performance. Instead, the brute-force algorithm was only viable for interactive and cinematic frame rates. The original algorithm was reformulated into the "final gather" algorithm in Unreal Engine 4.24. Ongoing development, strict lighting constraints, and substantial qualitative improvements from denoising and upscaling led to adoption of the final gather approach as a potential real-time global illumination solution for *Fortnite*. In this section, we outline the two algorithms and discuss the technological improvements and optimizations that led to achieving real-time performance. Figure 48-12 shows the visual result achieved in-game.

Listing 48-6. *HLSL code to retrieve a light index for a grid cell.*

```
1 int GetLightIndex(int3 Cell, int LightNum)
2 {
3     int LightIndex = -1; // Initialized to invalid
4     const uint4 LightCellData = LightCullingVolume[Cell];
5
6     // Whether the light data is inlined in the cell
7     const bool bPacked = (LightCellData.x & (1 << 31)) > 0;
8
9     const uint LightCount = bPacked ? (LightCellData.w >> 20) & 0x3ff
10                                    : LightCellData.x;
11
12    if (bPacked)
13    {
14        // Packed lights store 3 lights per 32-bit quantity.
15        uint Shift = (LightNum % 3) * 10;
16        uint PackedLightIndices = LightCellData[LightNum / 3];
17        uint UnpackedLightIndex = (PackedLightIndices >> Shift) & 0x3ff;
18
19        if (LightNum < LightCount)
20        {
21            LightIndex = UnpackedLightIndex;
22        }
23    }
24    else
25    {
26        // Non-packed lights use an external buffer
27        // with the offset in the cell data.
28        if (LightNum < LightCount)
29        {
30            LightIndex = LightIndices[LightCellData.y + LightNum];
31        }
32    }
33    return LightIndex;
34 }
```

BRUTE FORCE

The experimental brute-force algorithm uses Monte Carlo integration to solve the diffuse interreflection component of the rendering equation [8]. As with other ray tracing passes, the global illumination pass begins at the G-buffer, where diffuse rays are generated in accordance to the world-space normal of the rasterized depth-buffer's position. In this manner, the brute-force algorithm behaves very similarly to that of an ambient occlusion algorithm. However, instead of casting visibility rays to tabulate sky occlusion, the global illumination algorithm casts more expensive rays, evaluating secondary surface material information by invoking the closest-hit shader.

Figure 48-12. *Fortnite's Risky Reels, before and after applying ray traced global illumination. Note the bounce lighting from the grass to the grill.*

Along with expensive evaluation of the closest-hit shader, the algorithm must also apply direct lighting to secondary surfaces. Instead of applying the traditional light loop, the global illumination algorithm uses next event estimation. *Next event estimation* (NEE) is a stochastic process that chooses a candidate light with some probability. The first process, known as light selection, decides which light to sample, according to some selection probability. The second process, similar to traditional light sampling, samples an outgoing direction to the light source. The NEE process constructs a shadow ray to test the visibility of the shading point in relation to the selected light. If the visibility ray successfully connects to the light source, diffuse lighting evaluation is recorded.

NEE generates a smaller per-ray cost than the traditional lighting loop, as it only evaluates a subset of the total number of candidate lights. Though the NEE is typically considered to select one light source per invocation, another secondary stochastic process may be invoked to draw multiple NEE samples. Drawing multiple samples has the effect of lowering per-ray variance while also amortizing the cost of casting an expensive material evaluation ray. We find that drawing two NEE samples per material evaluation ray works well in practice. See Listing 48-7 for an abbreviated code example.

Listing 48-7. *Next event estimation.*

```
1 float3 CalcNextEventEstimation(
2     float3 ShadingPoint,
3     inout FPayload Payload,
4     inout FRandomSampleGenerator RNG,
5     uint SampleCount
6 )
7 {
8     float3 ExitantRadiance = 0;
9     for (uint NeeSample = 0; NeeSample < SampleCount; ++NeeSample)
10     {
11         uint LightIndex;
12         float SelectionPdf;
13         SelectLight(RNG, LightIndex, SelectionPdf);
14 
15         float3 Direction;
16         float Distance;
17         float SamplePdf;
18         SampleLight(LightIndex, RNG, Direction, Distance, SamplePdf);
19 
20         RayDesc Ray = CreateRay(ShadingPoint, Direction, Distance);
21         bool bIsHit = TraceVisibilityRay(TLAS, Ray);
22         if ( !bIsHit )
23         {
24             float3 Radiance = CalcDiffuseLighting(LightIndex, Ray, Payload);
25             float3 Pdf = SelectionPdf * SamplePdf;
26             ExitantRadiance += Radiance / Pdf;
27         }
28     }
29     ExitantRadiance /= SampleCount;
30 
31     return ExitantRadiance;
32 }
```

Depending on the maximum number of bounces allowed by the artist, the brute-force algorithm may fire another diffuse ray from the secondary surface and repeat the process to extend the path chain. Subsequent bounces are susceptible to early termination and are governed by a Russian roulette process. Our cinematics department prefers to use two bounces of global illumination as their default configuration.

Global illumination computed in this manner matches closely with our path tracing integrator. However, also like our path tracing integrator, this process requires a high number of samples to converge. A strong denoising kernel helps to generate a smooth final result, but we find that between 16 and 64 samples per pixel are still necessary for sufficient quality, depending on the lighting conditions and overall environment. This is problematic, because casting multiple material evaluation rays per frame quickly pushes the

Figure 48-13. *The brute-force global illumination technique in our development test environment, rendered with two samples per pixel at `ScreenResolution = 50`. Note the yellow color bleeding from the nearby wall onto the bush.*

algorithm beyond interactive rates and is only applicable for cinematic burnouts. Figure 48-13 shows the application of the brute-force global illumination to our *Fortnite* development level.

FINAL GATHER

In an effort to keep the nice properties of the brute-force integrator, the algorithm was amended in September 2019 to render diffuse interreflection for our Archviz Interior Rendering sample [4] at a cinematic frame rate of 24 Hz.

The key insight to accelerate the brute-force algorithm involves transforming the costly material evaluation rays into relatively inexpensive visibility rays. To achieve this, we time-slice the material evaluation rays over consecutive frames. We invoke the brute-force integrator, but only at one sample per pixel, and use previous frame simulation data to accumulate as though we actually fired the desired number of samples per pixel. Accumulating previous frame data requires sample reprojection and may cause severe ghosting artifacts, especially when the previous frames' simulation data is no longer valid. To help reconcile discrepancies associated with accumulating previous frame data, we choose to employ primary path reconnection [6] to

reuse previously traced paths. Previous path data is cached in an intermediary structure that we call *gather points*. Each gather point encodes the position of the secondary surface, along with the recorded irradiance and path creation probability density function (PDF). The world position of the pixel is also cached and is used to test against reprojection criteria for reuse in subsequent frames. The gather point buffer is interpreted as a circular buffer, where the buffer length is governed by the number of samples per pixel of the algorithm. In a manner similar to Bekaert et al. [2], we fire secondary visibility rays to test for successful path reconnection. Doing so correctly requires carrying both the exitant radiance and the probability density of the gather point creation from the active shading point in the previous simulation. Like next event estimation, a successful path reconnection event records the diffuse lighting at the gather point.

A wave of diffuse rays is dispatched as an individual pass each frame. This pass has similar execution flow to the brute-force algorithm, but records lighting data to a secondary *gather point buffer*. The gather point buffer records the secondary surface position and diffuse exitant radiance from stochastic light evaluation, as well as the probability density of generating the gather point from the simulation. The original creation point is also supplied so that we can reject gather points that do not meet sufficient criteria for reuse in the current frame. From this data, we can cast path reconnection events to these points and incorporate the cached lighting evaluation. The gather points pass is accelerated using the same sorted-deferred material evaluation pipeline used with ray traced reflections. (See Listing 48-8.)

Once the gather points are created, the final gather pass is executed. The final gather pass loops through all gather points associated with the current pixel and reprojects them to the active frame. Gather points that successfully reproject are candidates for path reconnection. A visibility ray is fired to potentially connect the world position of the shading point to the world position of the gather point. If a path reconnection attempt is successful, the shading point records diffuse lighting from the gather point. We found that we still needed approximately 16 path reconnection events to have good qualitative results. Figure 48-14 and Table 48-1 present visual and runtime comparisons, respectively, of the two global illumination methods.

Listing 48-8. *Individual gather samples are cached in a structured buffer of gather points where the dimensions are governed by the pass resolution and the samples per pixel. To conserve space, the irradiance field of a gather point encapsulates both the irradiance and creation PDF of the gather sample.*

```
1  struct FGatherSample
2  {
3      float3 CreationPoint;
4      float3 Position;
5      float3 Irradiance;
6      float Pdf;
7  };
8
9  struct FGatherPoint
10 {
11     float3 CreationPoint;
12     float3 Position;
13     uint2 Irradiance;
14 };
15
16 uint2 PackIrradiance(FGatherSample GatherSample)
17 {
18     float3 Irradiance = ClampToHalfFloatRange(GatherSample.Irradiance);
19     float Pdf = GatherSample.Pdf;
20
21     uint2 Packed = (uint2)0;
22     Packed.x = f32tof16(Irradiance.x) | (f32tof16(Irradiance.y) << 16);
23     Packed.y = f32tof16(Irradiance.z) | (f32tof16(Pdf) << 16);
24     return Packed;
25 }
26
27 FGatherPoint CreateGatherPoint(FGatherSample GatherSample)
28 {
29     FGatherPoint GatherPoint;
30     GatherPoint.CreationPoint = GatherSample.CreationPoint;
31     GatherPoint.Position = GatherSample.Position;
32     GatherPoint.Irradiance = PackIrradiance(GatherSample);
33     return GatherPoint;
34 }
```

The final gather algorithm is limited to one bounce of diffuse interreflection. The technique could be extended to multiple bounces, by allowing stochastic gather point creation at a given event, but for simplicity we opted for one bounce. Because artificially limiting the bounce count has the impact of both lower runtime cost and lower variance, we felt this was an appropriate compromise given the general expense of the technique. Unfortunately, reprojection and path reconnection failures can result in slower convergence compared to the brute-force method. This is possible when experiencing significant camera or object motion.

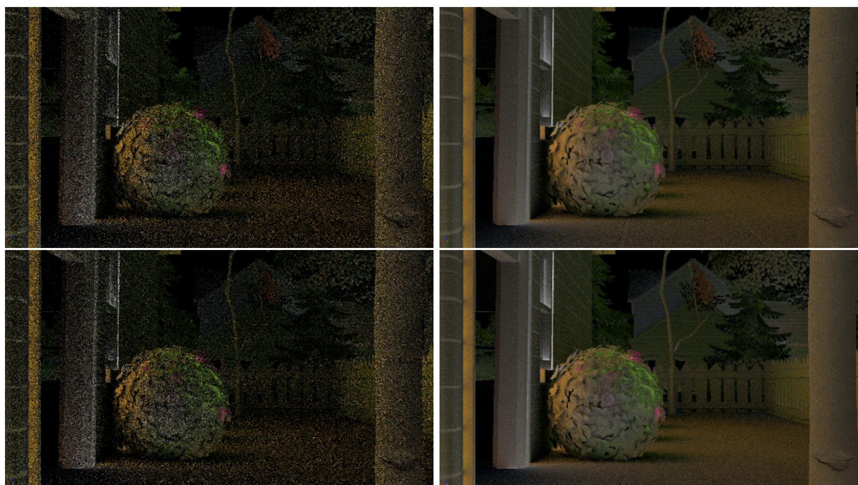


Figure 48-14. *Results from the two global illumination algorithms. Top: the brute-force method. Bottom: the final gather method. Each result is rendered at `ScreenPercentage = 100` for clarity and shown at one sample per pixel (left) and 16 samples per pixel (right). The images have been brightened for visualization purposes.*

SPP	Brute Force (ms)	Final Gather (ms)
1	19.78	11.63
2	46.95	13.39
4	121.33	13.49
8	259.48	15.86
16	556.31	20.15

Table 48-1. *GPU times for the global illumination passes presented in Figure 48-14 at `ScreenPercentage = 50` using an NVIDIA RTX 2080 Ti.*

FEASIBILITY FOR FORTNITE

Ray traced global illumination development was suspended shortly after our UE 4.24 release. With Unreal Engine 5 technology entering full development, it no longer made sense to extend an algorithm that would ultimately compete against newer initiatives.

Technologies such as NVIDIA Deep Learning Super Sampling (DLSS) [11, 3, 9] started new discussions, however. Early experiments with our test *Fortnite* level revealed that, with DLSS in performance mode, we could run our entire suite of ray tracing shaders at roughly 50 frames per second! Ray traced

shadow, ambient occlusion, reflection, sky light, and global illumination algorithms ran close to our intended budget for release. This proved to be very exciting initially, but the production map was still much more complex. For global illumination, in particular, the primitive stochastic light selection method could not cope with *Fortnite*'s large active light count. Even with necessary improvements to light selection, significant sample noise (mostly with interior lighting) made the final gather algorithm a difficult prospect for adoption.

As happens in production, other feature targets changed over the course of development, too. One of the most notable decisions was the omission of ray traced shadows for all lights except the sun (directional light). We concluded that if ray traced shadows were only included for the sun, we could apply similar exclusionary rules for global illumination as well. Of course, this also restricted bounce lighting to exterior environments, but if we limited global illumination to the sun, we could eliminate our algorithmic inefficiency with regard to light selection. Other potential sampling issues, such as illumination from large area lights, also went away. By adjusting the next event estimation samples to one, we avoided the cost of firing a second shadow ray and gained additional savings. With the feature set significantly culled, employing ray traced global illumination actually seemed feasible.

IMPROVEMENTS

Despite culling a large swath of algorithmic requirements, deploying the final gather algorithm for *Fortnite* was still challenging, due to remaining performance and sampling issues.

It became clear that we needed to operate at a coarser resolution to remain in budget. We expected that operation at half resolution or smaller would be required for the project. Unfortunately, we found that operating at smaller resolutions typically required more reconnection events to help mitigate noise. Doing so was not compatible with our temporal strategy; however, the likelihood of successful reconnection events diminished as our temporal lag increased. Overall dependence on temporal history proved difficult for a fast-moving game like *Fortnite*.

We started experimenting with extended strategies for gather point reprojection and path reconnection. Our simple, world-based reprojection of gather point data was improved with time-varying camera projection to improve stability for fast-moving camera motion. Though the first

implementation of the final gather algorithm exploits temporal path reprojection of gather points, we expected that both spatial and temporal reuse would be necessary to increase our effective samples per pixel. Previous experiments in this area had failed, and we discovered, as was also previously presented by Bekaert et al. [2], that using a regular neighborhood reconstruction kernel exhibited strongly objectionable structured noise, despite the observed error reduction in the final result.

Further experimentation in this area was halted as we also had to address our general denoising problem. Our previous global illumination denoiser operates well at full resolution, but quickly degrades when rendering at lower resolutions. Thankfully, NVIDIA presented us with an implementation of spatiotemporal variance-guided filtering (SVGF) [13]. SVGF proved to be a game-changer for the global illumination algorithm. We found that SVGF tolerated downsampled, noisy images with better results than our integrated denoiser. SVGF readily accepted the structured noise present with the newer spatial path reconnection strategy and generated very pleasing results while increasing the overall effective samples per pixel. Unfortunately, our implementation of SVGF exhibited bleeding artifacts with high-albedo surfaces when attempting to upscale to the required resolution (Figure 48-15). Though the occurrence was not frequent, it was prevalent in the Sweaty Sands

Figure 48-15. *Fortnite's Misty Meadows, before and after applying ray traced global illumination. Note the red color bleeding from the rooftops onto the buildings.*

(a) *No spatial gather point reconnection* **(b)** *Spatial gather point reconnection*

Figure 48-16. *Fortnite's Sweaty Sands, before and after applying ray traced global illumination. Sharing neighborhood samples creates strong structured artifacts, but SVGF is still able to reconstruct a smooth result while also dampening temporal artifacts.*

point of interest (see Figure 48-16) and needed to be addressed. Facing tough deadlines, we opted to implement an upsampling prepass so that G-buffer associations would not confuse the filter. We intend to address this exorbitant cost in the future, should we employ SVGF on another project.

We present a final breakdown of iterative improvements to the final gather algorithm in Table 48-2 and a final cost breakdown per pass in Table 48-3. Limiting global illumination to the directional light shows a significant cost savings when avoiding light selection. DLSS allows the method to work at a fractional scale, applying another significant speedup. A moderate speed gain is achieved by limiting lighting to one next event estimation sample. Costs are reintroduced to maintain temporal stability, when applying our spatial reconnection strategy with SVGF. SVGF offers pleasing results at both half and quarter resolution dimensions, which drives our *high* and *low* quality settings.

48.4.3 CPU OPTIMIZATIONS

GPU BUFFER MANAGEMENT

While profiling the CPU cost when ray tracing is enabled, we quickly noticed that our D3D12 data buffer management code needed to be improved. A lot of

Improvement	Time (ms)
Final gather baseline	10.25
Directional light only	6.14
1 NEE sample	5.59
DLSS mode: Performance	2.52
Spatial reconnection	2.78
SVGF denoiser	4.42
Screen percentage 25%	3.22

Table 48-2. *Incremental improvements applied to Figure 48-15. The last two entries correspond to the current global illumination user settings, high and low, respectively. GPU times reported at 1080p resolution on an NVIDIA RTX 2080 Ti*

Pass	Low Quality (ms)	High Quality (ms)
Gather point tracing	0.18	0.23
Gather point sorting	0.02	0.03
Gather point shading	0.49	0.92
Final gather tracing	0.49	1.18
SVGF	2.03	2.03

Table 48-3. *Per-pass execution costs associated with rendering Figure 48-15. SVGF costs are constant due to our need to run an upsampling prepass to the required DLSS resolution. GPU times reported at 1080p resolution on an NVIDIA RTX 2080 Ti.*

time in *Fortnite* was spent creating and destroying acceleration structure data buffers during gameplay due to mesh data streaming.

An easy optimization was to sub-allocate all these resources from dedicated heaps instead of using individual committed or placed resources. This is possible because acceleration structure buffers must be kept in the D3D12 `RAYTRACING_ACCELERATION_STRUCTURE` state, while scratch buffers are kept in the `UNORDERED_ACCESS` state. This means that there are no state transitions and no per-buffer state tracking required. This adjustment also saves a lot of memory due to smaller alignment overhead (placed resources in D3D12 require a 64K alignment, but most buffers are a lot smaller).

We had to make sure that committed resources are never created during gameplay, as this can introduce huge CPU spikes (sometimes 100+ ms, in our measurements). The buffer pooling scheme in UE4's D3D12 backend was adjusted to support the large allocations needed for top- and bottom-level acceleration structure data (the maximum allocation size was increased).

Readback buffers, used to get the information for compaction, are pooled and sub-allocated as placed resources from a dedicated heap.

Another problem is that almost all static bottom-level acceleration structure (BLAS) buffers were temporarily created at full size and then compacted. Compaction requires readback of the final BLAS size and a copy into the new compacted acceleration structure buffer. This can cause a lot of fragmentation and memory waste. We did not implement pool defragmentation for *Fortnite* due to time constraints, but this was done later for Unreal Engine 5.

DYNAMIC RAY TRACING GEOMETRIES

Another substantial CPU bottleneck we faced was in collecting and updating all the dynamic meshes in the scene before BLAS data is updated. A compute shader is run for each mesh to generate the dynamic vertex data for that frame. This temporary vertex data is then used to update/refit the BLAS. Potentially hundreds of dispatches and build operations can be kicked off in a single frame. Each dispatch might use a different compute shader and output vertex buffer, causing a lot of CPU overhead when generating the command lists. This performance overhead comes from switching the shaders/state and from binding different shader parameters.

We first optimized this process by sorting all dynamic geometry update requests by shader, but it was not quite enough. Additional overhead comes from switching the buffers for each mesh and performing the internal resource state transitions. Most dynamic meshes, such as characters or deformable objects, need to be updated every frame so the updated vertex data does not have to be persistently stored. This allowed us to use transient per-frame buffers with simple linear sub-allocation within them per mesh, minimizing the cost for state tracking. Each compute shader dispatch writes to a different part of the buffer; therefore, unordered access view (UAV) barriers are not necessary between dispatches. Finally, we parallelized the generation of the dynamic geometry update command list with the BLAS update command list to hide the surprisingly high `BuildRaytracingAccelerationStructure` CPU costs.

BUILDING THE SHADER BINDING TABLES

The largest CPU cost (by far) is building the ray tracing shader binding table (SBT) for each scene and ray tracing pipeline state object. The Unreal Engine

does not use persistent shader resource descriptor tables, so all resource bindings have to be manually collected and copied into a single shared descriptor heap every frame.

To reduce the cost of copying all the descriptors for all the meshes, we introduced several levels of caching. The biggest win came from deduplicating SBT entries with the same shader and resources, such as the same material being applied to different meshes or multiple instances of the same mesh. Unreal Engine groups shader resource bindings into high-level tables ("Uniform Buffers"), and a typical shader references three or four of them (view, vertex factory, material, etc.). Each uniform buffer in turn may contain references to textures, buffers, samplers, etc. We can cache SBT records by simply looking at the high-level uniform buffers without inspecting their contents.

A lower-level cache was added to deduplicate the actual resource descriptor data in descriptor heaps. This was mostly needed to deduplicate the sampler descriptors, as a single D3D12 sampler heap can only have 2048 entries, and only one sampler heap can be bound at a time. We found that the cost of the D3D12 `CopyDescriptors` call is roughly the same as (or larger than) the cost of descriptor hashing and hash table lookup/insert.

Finally, we parallelized the SBT record building, but found that improvements from adding worker threads diminished very quickly. As we still wanted to use descriptor deduplication, each worker thread needed to use its own local descriptor cache to avoid synchronization overheads. Global descriptor heap space is then allocated by each worker in chunks, using atomics. This reduces the cache efficiency and increases the total number of used descriptor heap slots. We found that four or five worker threads is the sweet spot for parallel SBT generation.

GEOMETRY CULLING

A simple but effective technique used to speed up both CPU and GPU render times was to skip instances entirely if they were not relevant to the current frame. When ray tracing, every object in the scene can affect what is seen by the camera. However, in real life the contribution of many objects can be negligible. Initially, we experimented with culling geometries behind the camera that were placed further than a certain distance threshold. However, this solution was discarded because it was producing popping artifacts when objects with large coverage where rejected in a frame and accepted in the

next one. The solution was to change the culling criteria to also take into account the projected area of the bounding sphere for the instance, and to discard it only if it was small enough. This simple change removed the popping while greatly improving speed. On average, we observed gains of 2–3 ms per frame after making this change.

DLSS

The integration of NVIDIA's DLSS technology was instrumental in improving performance. DLSS made it possible to enable ray traced global illumination and run the game with ray tracing enabled at higher resolutions. The engine changes made to integrate DLSS are available now in the public UE4 codebase [5] and the DLSS plugin for UE4 is now available on the Unreal Engine Marketplace [12].

48.5 FORTNITE CINEMATICS

Though it took some time until ray tracing was used in the game, the *Fortnite* team adopted ray tracing from the very early days for other purposes such as cinematic trailers and marketing content. Moving to a ray tracing pipeline allowed artists to reduce iteration times and increase the overall quality achieved. It also helped lighting artists with a background in offline rendering to speed up their initial learning of Unreal Engine because they were already familiar with similar rendering tools. An interesting aspect of this work is that, though the use cases are different and the quality requirements are higher than in the game, these cinematographic pieces are made using the same technology and assets used in the game.

48.6 CONCLUSION

Ray Tracing was shipped in *Fortnite* Season 15 (September 2020) and achieved more ambitious goals than what we initially set out to accomplish. Though the project was challenging on many levels, it ended up being a success. Not only does the game look better when ray tracing is enabled, but all the improvements are now part of the UE4 public codebase [5].

This initiative is not yet finished. There are many open problems in real-time ray tracing that need more work, including scenarios with a high number of dynamic geometries that must be updated every frame, complex light transport that requires multi-bounce scattering, and scenes with large amounts of dynamic lights. These problems are difficult and will require

multi-year efforts from the computer graphics community. The engineering team at Epic Games will continue improving the technologies developed for this project, as well as additional novel methods that we have on our road map, with the goal of making ray tracing a feasible solution for any kind of game or real-time graphics application.

ACKNOWLEDGMENTS

We would like to thank the rendering team at Epic Games for their help and feedback: Yujiang Wang, Guillaume Abadie, Chris Bunner, Michal Valiant, Marcus Wassmer, Kim Libreri, and all the great engineers that have contributed to make ray tracing in Unreal Engine.

Many thanks also to the *Fortnite* art ninjas: Jordan Walker, Mike Mulholland, Andrew Harris, Juan Collado, Kirsten Todd, Paul Mader, Alex Gonzalez, Luke Tannenbaum, Maddie Duque, Tim Elek, and many others.

Also, this endeavor would not have been possible without the amazing support the production and QA teams gave us: Scott James, Paul Oakley, Shak Khavarian, Kirstin Riddle, Katie McGovern, Petra Pintar, Ari Patrick, Brandon Grable, Stan Hormell, Will Fissler, Zachary Wilson, etc.

The authors would also like to thank the book editors for their very helpful suggestions and insights.

REFERENCES

[1] Aalto, T. Bringing ray tracing into Remedy's Northlight engine. https://www.youtube.com/watch?v=ERlcRbRoJF0, Nov. 22, 2018. Accessed May 05, 2021.

[2] Bekaert, P., Sbert, M., and Halton, J. Accelerating path tracing by re-using paths. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 125–134, 2002.

[3] Edelsten, A., Jukarainen, P., and Patney, A. Truly next-gen: Adding deep learning to games and graphics. Presentation at Game Developers Conference, https://www.gdcvault.com/browse/gdc-19/play/1026184, 2019.

[4] Epic Games. New Archviz interior rendering sample project now available! https://www.unrealengine.com/en-US/blog/new-archviz-interior-rendering-sample-project-now-available, Dec. 19, 2019.

[5] Epic Games. Unreal Engine source code repository. https://www.unrealengine.com/en-US/ue4-on-github, Jan. 1, 2021.

[6] Fascione, L., Hanika, J., Heckenberg, D., Kulla, C., Droske, M., and Schwarzhaupt, J. Path tracing in production: Part 1: Modern path tracing. In *ACM SIGGRAPH 2019 Courses*, 19:1–19:113, 2019. DOI: 10.1145/3305366.3328079.

[7] Heitz, E. Sampling the GGX distribution of visible normals. *Journal of Computer Graphics Techniques*, 7(4):1–13, 2018. http://jcgt.org/published/0007/04/01/.

[8] Kajiya, J. T. The rendering equation. 20(4), 1986.

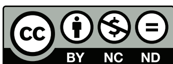
[9] Liu, E. DLSS 2.0: Image reconstruction for real-time rendering with deep learning. Presentation at Game Developers Conference, https://www.gdcvault.com/play/1026697/DLSS-Image-Reconstruction-for-Real, 2020.

[10] Liu, E., Llamas, I., Cañada, J., and Kelly, P. Cinematic rendering in UE4 with real-time ray tracing and denoising. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, chapter 19, pages 289–319. Apress, 2019.

[11] NVIDIA. Deep Learning Super Sampling (DLSS). https://www.nvidia.com/en-us/geforce/technologies/dlss/, 2019. Accessed December 19, 2019.

[12] NVIDIA. DLSS plugin for Unreal Engine 4. *Unreal Engine Marketplace*, https://www.unrealengine.com/marketplace/en-US/product/nvidia-dlss, Feb. 10, 2021.

[13] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, 2:1–2:12, 2017. DOI: 10.1145/3105762.3105770.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.