CHAPTER 38

CPU PERFORMANCE IN DXR

Peter Morley
NVIDIA

ABSTRACT

DirectX Raytracing (DXR) performance guides have mainly focused on accelerating ray tracing on the graphics processing unit (GPU). This chapter will focus on avoiding stalls and bottlenecks caused by DXR on the central processing unit (CPU) side.

38.1 INTRODUCTION

Most game engines must deal with streaming assets in and out of memory as the player moves throughout the built environment. Typically, the game world space is partitioned into discrete tiles or volumes, with only the elements close to the player being resident in memory, and optionally placeholders or reduced level of detail (LOD) geometry standing in for more distant items.

Implementing DXR ray tracing into a renderer adds several significant memory and compute overheads. This chapter presents techniques to optimize the management of acceleration structures and shader tables.

Methods to reduce DXR CPU overhead include the following:

- Use generic hit group shaders and use precompiled collections for a state object.
- Use incremental state object compilation.
- Reduce shader table complexity for the local root signature.
- Limit acceleration structure (AS) builds and refits.

38.2 THE RAY TRACING PIPELINE STATE OBJECT

The ray tracing pipeline state object (RTPSO) contains a network of shaders defining how various materials will be processed in a ray traced scene. The following techniques that will help reduce the amount of CPU overhead

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_38

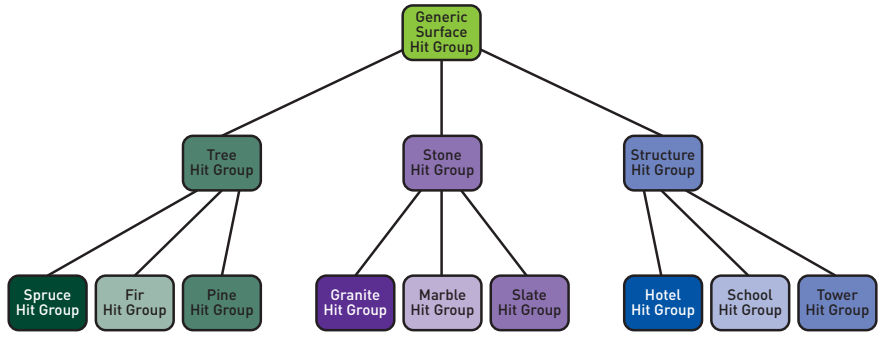

Figure 38-1. *The nine hit group shaders can be consolidated into a single generic surface hit group by unifying how common materials are shaded. Engines will separate hit groups into categories such as transparent, decals, generic, and subsurface materials, to name a few.*

associated with the RTPSO include reducing hit group shaders, as well as incremental changes to the RTPSO and state object collection multi-threaded compilation.

One way to mitigate heavy shader permutations is to break down the shader tables into generic hit group shaders that handle a collection of geometry types in order to make the number of hit groups in the RTPSO manageable. Figure 38-1 shows a state object configuration in which unique hit group shaders can be unified by functionality. Having a predefined set of hit group shaders in a state object is important because it not only reduces shader execution divergence but also reduces CPU delay from recompiling the state object every time new hit group shaders are needed.

38.2.1 INCREMENTAL STATE OBJECT MODIFICATIONS

Ray tracing pipeline state object compilation can be computationally expensive due to the driver compiling multiple shaders. `AddToStateObject` was introduced to prevent recompiling the entire RTPSO and only requires compilation of new hit group shaders as they are added. `AddToStateObject` is a very lightweight operation because the Direct3D 12 runtime does not need to validate the entire state object and the driver only needs to perform a trivial linking step after the new shaders are compiled. If complex state objects can't be avoided, then it is best practice to compile the most common hit group shaders into a state object once and incrementally add new hit group shaders with `AddToStateObject` during gameplay. The full documentation for `AddToStateObject` is available [2].

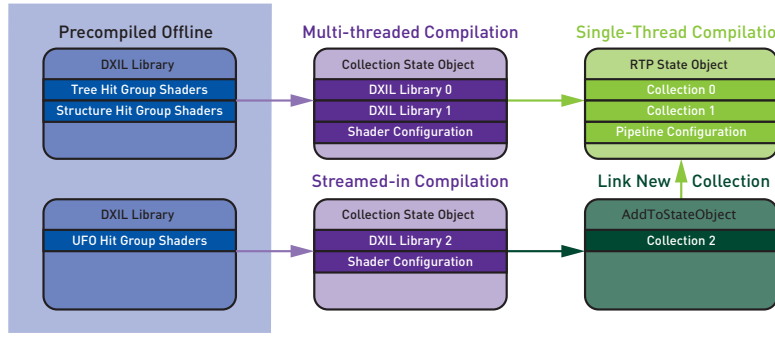

Figure 38-2. *Precompiled DXIL binaries are compiled into state object collections at startup then finally compiled into a ray tracing pipeline state object.*

38.2.2 STATE OBJECT COLLECTIONS

Another useful strategy to reduce the amount of compilation time for ray tracing pipeline state objects is to use state object collections. Collections containing ray tracing shader DirectX Intermediate Language (DXIL) libraries can be compiled separately and in parallel on the CPU. (See Figure 38-2.) RTPSOs can then reference these precompiled collections and in turn reduce the amount of time it takes to prepare a ray tracing pipeline. Using multiple CPU cores to compile collections can give a significant speedup compared to compiling the RTPSO with raw DXIL libraries on a single CPU core. RTPSO compilation should be lightweight due to multi-threaded collection compilations. New hit groups can be streamed in by asynchronously compiling new collections and using `AddToStateObject` to perform a trivial linking step, assuming the collection state objects were compiled by the driver.

38.3 THE SHADER TABLE

Generating complex shader tables is often a costly process on the CPU. A shader table contains an array of shader records that define the resource bindings of each top-level instance's local root signature in the top-level acceleration structure (TLAS).

38.3.1 BUILDING THE LOCAL ROOT SIGNATURE ON THE GPU

A local root signature (LRS) can use up to 4 KB, according to the DXR functional spec [3], in memory for descriptors and other resource data. One special advantage of the LRS is that the underlying memory can be allocated

in GPU memory for low latency during traversal shading. If the LRS uses constant buffers or root constant buffers, it can be very costly to constantly update those memory resources on the CPU side for a large amount of shader records. Instead, shader tables can be generated using a compute shader to write out shader table resource data by reading CPU system memory buffers containing the scene's resource information. Another advantage of building the shader table on the GPU and using GPU visible only memory is that this will avoid Peripheral Component Interconnect Express (PCIe) traffic reads (GPU to CPU memory) when accessing constant buffers and all the other resource descriptors.

- LRS pros:
  - Root signature memory size is 4 KB.
  - CPU overhead to update shader records.
- LRS Cons:
  - Root signature memory is configurable.

38.3.2 GLOBAL ROOT SIGNATURE

One limitation of the global root signature (GRS) is that if the resources required for state object shading exceed the memory capacity of a GRS, then resource management will require special attention. Another limitation is that a GRS is limited to 64 DWORDS (256 B) and only one can be bound. More information on limitations are defined by White and Satran [6].

For example, if root constants are needed to manage resource indexing, then the root constants would be accumulated into a single structured buffer and accessed with a shader resource view (SRV).

- GRS pros:
  - Unified root signature management.
  - No shader table management if LRS isn't used.
- GRS cons:
  - Root signature memory size is 256 B.
  - Root signature memory is required to be in system memory.
  - Requires alternative management of large root data sets.

38.3.3 GRS VERSUS LRS

A GRS is limited to 256 B and only one can be bound, while a LRS allows up to 4 KB and can be instanced. The LRS gives each instance in the TLAS access to 4 KB of resource memory rather than being limited to 256 B with a GRS. Using a LRS is a convenient way to get around some of the limitations of a GRS, but at the cost of CPU performance and more memory consumption.

38.3.4 SHARING RESOURCES WITH THE RASTERIZER

If the shader table construction can't be performed on the GPU, then techniques must be used to reduce CPU overhead of managing the resource views (constant buffer view, shader resource view, or unordered access view). This can involve extracting resources required for ray tracing from the already established rasterization engine's resources. Resource view management for the LRS typically involves heavy use of `CopyDescriptors` or `CopyDescriptorsSimple` to pull from a repository of views stored in a single large descriptor heap. Typically, these copies are required for hybrid (rasterization and ray tracing) renderers. The same resource sharing applies when the rasterizer consumes new vertex and index buffers needed to be rendered, which then must be passed along to the AS builder to include it during ray traversal.

38.3.5 BINDLESS RESOURCE ARRAYS

The preferred solution, if possible, is to avoid building the shader table altogether by dynamically indexing into bindless resources. Bindless resources are implemented using a descriptor table that contains an array of resource views. Large descriptor tables are stored in the GRS and can be accessed based on the instance and geometry index of the intersected primitive. (See also Chapter 17.)

38.4 THE ACCELERATION STRUCTURE

38.4.1 OVERVIEW

Processing millions of triangles into the ray tracing AS can become prohibitively expensive on both the GPU and the CPU. The first challenge is managing the transient geometry in the AS that enters and exits the player's rendering volume. For each frame, new bottom-level acceleration structures (BLAS) need to be built for new geometry that entered this volume, and conversely, existing acceleration structures need to be deallocated if they are

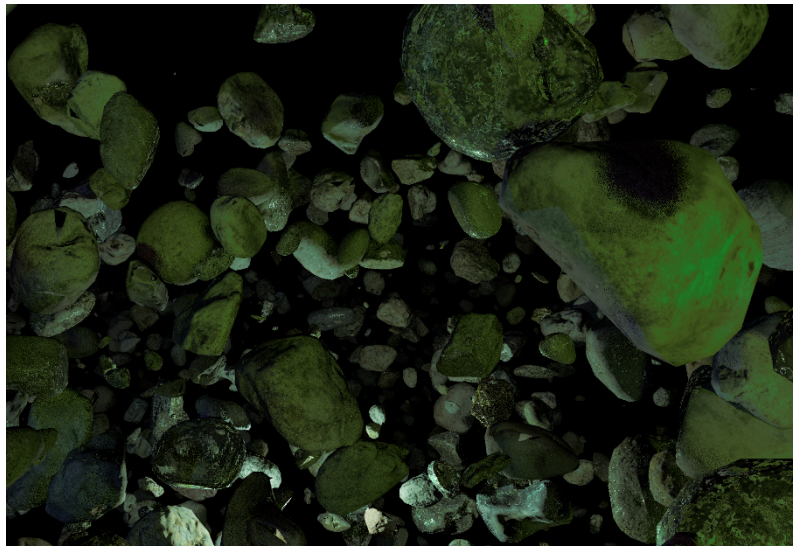

Figure 38-3. *A large collection of asteroids each stored in a unique BLAS structure. Frustum culling techniques are used to reduce the BLAS memory footprint and TLAS size.*

no longer in view. If the AS is being used for reflections or shadows, then all geometry within a certain radius of the player is required, not just the geometry in the view frustum. The movement of geometry in and out of the AS causes multiple trips to the operating system memory manager. These memory requests can introduce CPU overhead.

Figure 38-3 shows an asteroid field in which top-level asteroid instances are culled from the AS if not in the radius of inclusion. The TLAS will instance into each visible asteroid BLAS, while asteroids outside of the radius of inclusion will not be included in the TLAS and those BLAS can be deallocated.

One tool for avoiding such CPU-side memory allocation stalls involves sub-allocating the AS buffer memory. Infrequently requesting large memory pools from the operating system (OS) reduces CPU overhead because the common case would be to sub-allocate from an existing memory block. Compaction also helps in reducing the memory required for an AS by trimming the conservative memory allocation that was required for the initial build. An AS that uses compaction significantly decreases the memory footprint and, in turn, reduces CPU overhead as an added benefit. Less memory for the AS means less requests to grab sub-allocator memory blocks from the OS.

38.4.2 SHARING RESOURCES WITH THE RASTERIZER

The vertex and index buffer resources used to build the AS can hopefully be reused from the rasterization resources if the buffers are in one of the acceptable build formats. Most engines compress their vertex buffers and are forced to decompress and duplicate resources when getting ready to build the AS. The RT Cores can only interpret triangles in the form of 16- or 32-bit precision vertices. More information about RT Cores can be found in the Ampere white paper [4]. If the engine must duplicate the vertex buffers to build the AS, then be aware that deallocating those duplicated resources after building is recommended to reduce memory consumption.

38.4.3 DEFORMABLE, ANIMATED, AND STATIC AS BUILDS

There are three main categories of geometry types in the AS. These types can be described as static, animated, and deformable geometry. *Static* geometry can be defined as a triangle mesh that is uniformly transformed, such as buildings, roads, and signs. *Animated* geometry comprises a triangle mesh that has a grouping of triangles within the mesh that are transformed but adhere to the original topology, such as character animations. *Deformable* geometry is characterized by a triangle mesh in which all triangles can be transformed arbitrarily without maintaining the original topology, such as particle effects.

Static geometry has a major benefit in which it only requires a single full build. Animated geometry requires an upfront build as well but requires fast build updates every key frame, which are much less impactful on GPU performance than full builds. This requires a mesh skinning compute pass and a fast build per frame to update the AS bounding boxes for the deformed triangles. One method to reduce build processing times is to have a higher ratio of static objects compared to dynamic objects in the TLAS. Static objects only require a one-time build, whereas dynamic objects can require an update build or complete rebuild per frame. Both animated and static geometry can be compacted but require extra memory in which both the original build and the compacted build memory are resident.

Static topologies with extreme animation (running animations) or objects that change topology altogether (breakables or particles) require a full build during each of the deformable geometry's updates to maintain optimal traversal performance. AS traversal performance will degrade as triangles move farther away from their original build location and thus hurts

performance as time progresses. The trade-off is to do a full build of the deformable geometry every *N* frames and do refit builds between the *N* frames. This maintains acceptable build and traversal performance but requires tweaking for each individual engine. Putting a limit on the number of AS builds and updates per frame can also be a good way to maintain stable GPU/CPU performance with build workloads. One trade-off is that sometimes objects may appear in the AS a frame or two after they would have been traced against and show up as popping geometry. The RTX best practices blog post [5] details best practices for managing acceleration structures.

38.4.4 IMPROVING LOD PERFORMANCE

If the engine employs a LOD system, then the AS build workloads increase due to extra builds for the LODs and the management of the LODs included in the TLAS. For example, if the LOD system contains four levels of detail, then it is possible that certain assets transitioning through the LODs have the potential need to be built and compacted four separate times.

Instead, having only one LOD resident in memory prevents LOD transition popping, which is another problem set described in a stochastic LOD blog post [1]. The method described in the blog post implements smooth transitions between LODs in the acceleration structure by randomly masking out one of the LODs in the TLAS for each ray. This facilitates a gradual transition of geometry between the two LODs but at the cost of having to maintain more geometry in the AS.

One precaution with LOD selection is the potential to introduce position data bugs between rasterization and ray tracing. LOD mismatch can result in self-shadowing corruption if the acceleration structure is only using a single BLAS for each model. A technique used to mitigate self-shadowing in ray tracing is to add a bias to the ray origin. Figure 38-4 shows the self-shadowing bug when mismatching LODs for primary rays and rasterization.

The benefit to avoiding an LOD system for ray tracing prevents not only a reduction in BLAS memory but also a reduction in AS builds. The trade-off here is that geometry that is far away and highly tessellated will potentially hurt traversal performance. The TLAS is designed to prevent wasted traversal time in these highly tessellated regions, so the performance trade-off might not be as bad as expected.

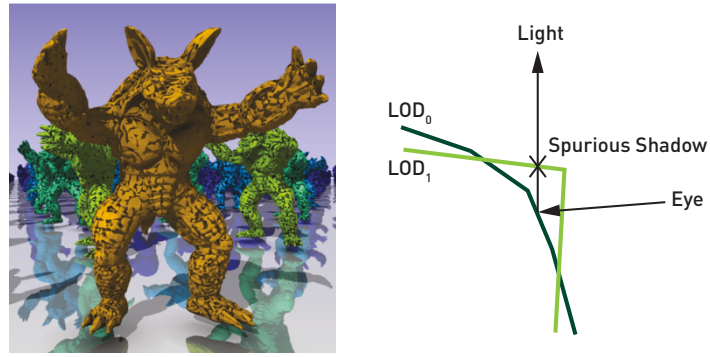

Figure 38-4. *The BLAS and rasterization vertex buffer data is different due to LOD mismatch, and the result is self-shadowing artifacts [1, Figure 3].*

38.5 CONCLUSION

The intention of this chapter is to help better understand the pros and cons associated with certain DXR design choices and how they affect CPU performance. In conclusion, the recommended approach is to simplify the RTPSO as much as possible and asynchronously compile state collections for inclusion in the RTPSO. Allocate shader tables in GPU memory and build them on the GPU to reduce CPU overhead and improve traversal shading performance. Limit the number of AS builds per frame, as to reduce memory requests to the OS and reduce the amount of AS build times on the GPU. Implementing LOD algorithms for DXR introduces a significant amount of extra memory and additional AS builds but succeeds in preventing LOD mismatch between the rasterizer and the ray tracer.

REFERENCES

[1] Lloyd, B., Klehm, O., and Stich, M. Implementing stochastic levels of detail with Microsoft DirectX Raytracing. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/implementing-stochastic-lod-with-microsoft-dxr/, June 15, 2020.

[2] Microsoft. ID3D12Device7::AddToStateObject method (d3d12.h). *Windows Developer*, https://docs.microsoft.com/en-us/windows/win32/api/d3d12/nf-d3d12-id3d12device7-addtostateobject, September 15, 2020.

[3] Microsoft. Local root signatures vs global root signatures. *DirectX Raytracing (DXR) Functional Spec*, https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html#local-root-signatures-vs-global-root-signatures, 2021.

[4] NVIDIA. Ampere GA102 GPU architecture: Second-generation RTX. White paper, https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2021.

[5] Sjoholm, J. Best practices: Using NVIDIA RTX ray tracing. *NVIDIA Developer Blog*, https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/, August 10, 2020.

[6] White, S. and Satran, M. Root signature limits. *Windows Developer*, https://docs.microsoft.com/en-us/windows/win32/direct3d12/root-signature-limits, May 31, 2018.