OUT-OF-CORE GPU PATH TRACING ON LARGE INSTANCED SCENES VIA GEOMETRY STREAMING

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Jeremy Berchtold

June 2022

COMMITTEE MEMBERSHIP

TITLE: Out-of-Core GPU Path Tracing on Large Instanced Scenes via Geometry Streaming

AUTHOR: Jeremy Berchtold

DATE SUBMITTED: June 2022

COMMITTEE CHAIR: Zoë Wood, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Maria Pantoja, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Chris Lupo, Ph.D.
Professor of Computer Science

ABSTRACT

Out-of-Core GPU Path Tracing on Large Instanced Scenes via Geometry Streaming

Jeremy Berchtold

We present a technique for out-of-core GPU path tracing of arbitrarily large scenes that is compatible with hardware-accelerated ray-tracing. Our technique improves upon previous works by subdividing the scene spatially into streamable chunks that are loaded using a priority system that maximizes ray throughput and minimizes GPU memory usage. This allows for arbitrarily large scaling of scene complexity. Our system required under 19 minutes to render a solid color version of Disney's Moana Island scene (39.3 million instances, 261.1 million unique quads, and 82.4 billion instanced quads [30]) at a resolution of 1024x429 and 1024spp on an RTX 5000 (24GB memory total, 22GB used, 13GB geometry cache, with the remainder for temporary buffers and storage). As a scalability test, our system rendered 26 Moana Island scenes without multi-level instancing (1.02 billion instances, 2.14 trillion instanced quads, ∼230GB if all resident) in under 1h:28m. Compared to state-of-the-art hardware-accelerated renders of the Moana Island scene, our system can render larger scenes on a single GPU. Our system is faster than the previous out-of-core approach [12] and is able to render larger scenes than previous in-core approaches given the same memory constraints [32, 27]

ACKNOWLEDGMENTS

Thanks to:

- My mom, for always supporting and encouraging my learning
- Zoë Wood, for great advising and mentoring both on technical and logistical problems
- David Hart and Mark Leone, for their incredibly helpful advice both on high-level ideation as well as detailed optimizations

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1 Introduction 1

2 Background 4

2.1 Rendering 4

2.2 GPU Compute 6

2.2.1 Optimizing GPU Performance 6

2.2.2 RTX and OptiX 7

2.2.3 Streaming 7

3 Related Work 9

4 Algorithm 13

4.1 Scene Partitioning 13

4.2 Raytracing Pipeline and Streaming 15

4.3 Asset Loading 18

4.4 Asset Cache 19

4.5 Re-tracing of Overlapping Chunks 20

5 Supporting Implementation Details 23

5.1 System Overview 23

5.2 OptiX Kernel 23

5.3 CUDA Kernels 24

5.4 Memory Management 25

6 Results 27
6.1 Moana Island Scene 28
6.2 26x Moana Island Scene 28
6.3 Asset Cache Effectiveness 29
6.4 Straggling Path Cut-off 31
7 Conclusions & Future Work 35
REFERENCES 38

LIST OF TABLES

Table Page

5.1 OptiX raytracing entrypoint functions [15] 24

6.1 Render time versus non-converged pixels remaining in the rendered image 34

LIST OF FIGURES

Figure		Page
2.1	A rendered scene. Left: 3D geometry data. Right: Final image. Scene credit: Alex Treviño [25]	4
2.2	Scene rendered with an earlier version of our system using texture streaming. Scene credit: NVIDIA [19]	8
4.1	An example of a toy scene, recursively divided until the instance count reaches a given threshold	14
4.2	Empty chunks are discarded	14
4.3	Overview of ray tracing pipeline	15
4.4	Ray buffer operations	17
4.5	Re-tracing method: Ray hit	21
4.6	Re-tracing method: Ray miss	22
6.1	Moana Island scene. 39.3 million instances. 1024spp, 1024x429. 18m:33s	28
6.2	26x Moana Island test scene. No multi-level instancing. 1.02 billion instances. ∼230GB memory usage if all resident at once. 1024spp, 1024x429. 1h:27m:10s	29
6.3	Visualization of rays traced per chunk	30
6.4	Visualization of chunk loads	31
6.5	26x Moana Island scene. Total render 1h:27:10s. 15m:23s partitioning, 1h:11m:47s rendering	32
6.6	26x Moana Island scene. Stopped early and displayed with the rays that had already terminated within the first 5 minutes of the render stage	32
6.7	FLIP image comparison between full render and progress at 5 minutes. See Figures 6.5, 6.6	33

Chapter 1

INTRODUCTION

In recent years, rendering has become increasingly photo-realistic, with many renders being nearly indistinguishable from a photo. However, this detail comes at a performance cost. It can take minutes or hours to render a single frame on complex scenes. Traditionally, complex renders are performed on powerful CPU systems with a large amount of memory. Recently, GPU hardware-accelerated raytracing has greatly decreased render times, making GPUs a viable option for some scenes [5]. However, despite advances, GPU rendering is not yet practical for large production scenes. Production scenes can contain billions of polygons and millions of instances, and continue to grow in size [20]. Even with instancing, which is using multiple copies of the same 3D mesh, scenes use many gigabytes just for the geometry data. Furthermore, texture data sets are huge, limiting the amount of GPU memory available for geometry. Out-of-core rendering is crucial for final-frame rendering in computer animation and visual effects. Therefore, we propose a method for geometry streaming of large, heavily-instanced scenes that is compatible with hardware-accelerated raytracing.

Our method addresses two main obstacles in the use of hardware-accelerated raytracing:

- The memory requirements of the scene exceeding the memory of the GPU
- Efficiently using in-memory acceleration structures supported by hardware-accelerated raytracing

Efficient raytracing cannot operate on the raw geometry data itself, and instead requires the use of acceleration structures, such as bounding volume hierarchies (BVHs) in the OptiX raytracing framework [15]. OptiX is a library provided by NVIDIA for utilizing raytracing cores on modern GPUs [15]. However, these acceleration structures can be costly to build. It is not practical to rebuild a monolithic acceleration structure as geometry is streamed in and out of core. Instead we use multi-level acceleration structures to minimize re-building costs when streaming.

Given these constraints, we refine the challenges for our approach and our proposed solution as the following:

- How can the scene be divided into partitions small enough to individually fit into GPU memory? *Solution:* Our algorithm divides the scene into chunks (of instanced geometry) small enough to fit into GPU memory.
- How can re-building of accelerations structures be minimized? *Solution:* Our algorithm loads chunks as needed with a priority-based system using ray heuristics, to keep frequently used chunks resident.

To test our system, we ran benchmarks on the Moana Island scene, a publicly available scene released by Disney that contains 82.4 billion total quads and 39.3 million instances [20, 30]. The scene depicts a lush island island from Disney's Moana with many detailed bushes, trees, and shells scattered across the shores and mountain peaks.

This paper presents our algorithm that takes advantage of hardware accelerated rendering and includes the following contributions:

- A method of streaming instanced geometry to a single GPU

- An accompanying method for partitioning the scene
- Benchmark data for our system on the Moana Island scene, and a proof of concept rendering of 26-times duplicated Moana Island test scene.

Chapter 2

BACKGROUND

2.1 Rendering

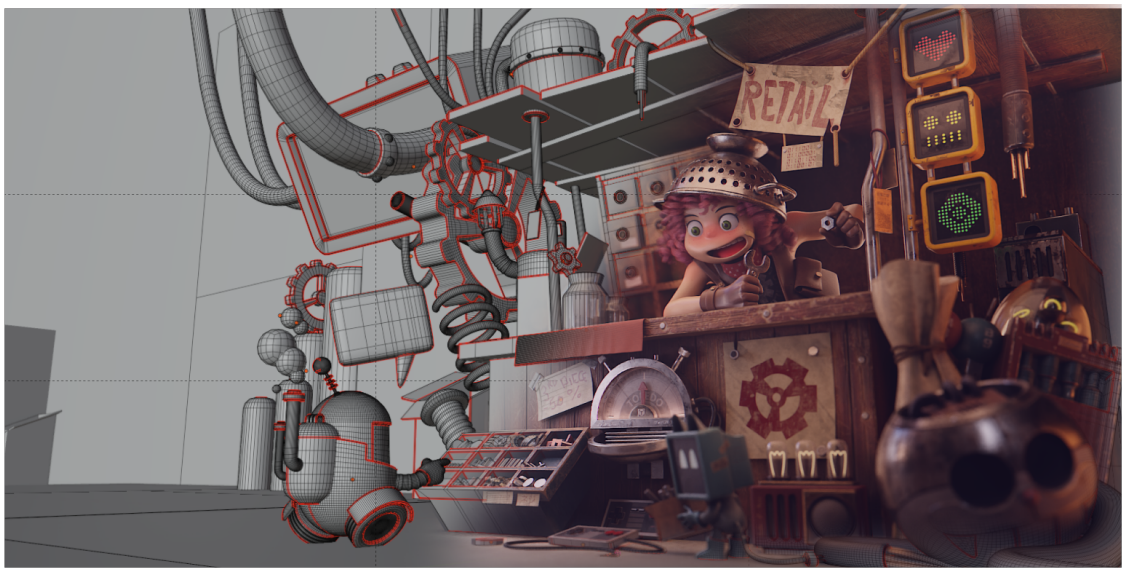

Figure 2.1: A rendered scene. Left: 3D geometry data. Right: Final image. Scene credit: Alex Treviño [25]

Rendering is the process of converting 3D geometry into a 2D image. This process involves determining which geometry is visible within the 2D image, and shading the geometry with the appropriate lighting. There are several methods for rendering, each with varying speed and quality trade-offs. See Figure 2.1 for a visualization of the raw geometry data and the rendered results.

There are a few common rendering techniques, with different performance and image quality trade-offs [6].

- **Rasterization** is fast and typically used in games and typically uses a local lighting model such as the Phong model, but more realistic lighting requires more advanced algorithms and data structures.

- With **Raytracing**, individual rays of light are traced throughout the scene. This allows for more accurate shadows and reflections, but is typically slower than rasterization. Raytracing in games usually on casts a few rays per pixel per frame to keep steady frame rates.

- **Path Tracing** is a type of raytracing in which many light paths are traced throughout the scene. These paths can bounce multiple times, allowing for indirect lighting. Additionally, many paths are sent through the same pixel and are randomly scattered according to the physically-based material models.

Additionally, there are a few other rendering concepts that are important to understanding our paper.

- The **Rendering Equation** defines how most light is reflected for every surface point, given the incoming light distribution and surface material properties [1].

- **Monte Carlo Sampling** is a technique for randomly sampling a distribution that is difficult or impossible to solve analytical [2].

- **Acceleration Structures** are hierarchical structures, such as bounding volume hierarchies (BVHs), used to traverse geometry data faster when tracing rays [17].

- **Instanced Rendering** uses one copy of mesh geometry data and renders it many times with different transformations to allow for complex geometry with far less memory usage [26]. This is commonly used for rocks and leaves.

2.2 GPU Compute

GPU architecture is fundamentally different from CPU architecture. The goal of CPUs is to run a handful of very fast cores to achieve high performance. On the other hand, GPUs rely on using thousands of cores. While each individual core is slower, GPUs can acheive much higher performance when the work executed in parallel. For example, graphics processing tasks often involve the same computation run over millions of pixels, making it an ideal use case for GPUs [4]. Additionally, GPUs have much stricter memory requirements. CPU system memory can be swapped out and expanded as needed, whereas GPUs have limited memory that is soldered directly to the device. This makes memory a precious resource when writing software for GPUs.

2.2.1 Optimizing GPU Performance

The difference in architecture from CPUs also means different strategies must be used when optimizing GPU code. The following summarizes the key methods to maximizing GPU performance.

- **Maximize the number of active threads**. To ensure optimal performance, keep the GPU fed with work so it is not idle.
- **Minimize branching and incoherent memory accesses**. GPU threads are organized in groups of 32 called warps. Branching causes warp divergence, causing threads to stall. Incoherent memory accesses hurt cache performance.
- **Minimize global synchronization**. GPUs are highly parallel and perform best when queued up with enough work. Synchronization operations stall the GPU and limit the speed at which it can perform work.

- **Minimize the memory transfer between the CPU and GPU**. Memory transfers over the CPU and GPU occur over a PCIe memory bus which is comparatively very slow.

Maximizing the number of active threads and minimizing memory transfers are the most important aspects for our project as it deals with out-of-core methods that involve many memory transfers.

2.2.2 RTX and OptiX

Recent generations of GPUs have support for hardware-accelerated raytracing. This provides a fast hardware implementation for common ray traversal operations and acceleration structure building. This is much faster than software approaches for many scenes. OptiX is a library for utilizing hardware-accelerated raytracing [15].

2.2.3 Streaming

Video streaming has become increasingly popular. The core principle being, data is only sent to the user when requested. Rendering systems can use a similar streaming concept. Instead of sending all the data to the GPU at the start of the render, the GPU can request only what is necessary. This saves precious GPU memory and can save on load time as well. Existing systems such as Redshift use streaming with textures [24]. Instead of loading all textures at once, texture tiles are only sent when requested by ray intersections, and only at the requested mip levels. At the start of this project, our system supported texture streaming using the Demand Loading library [15]. See Figure 2.2. Due to time constraints, we were not able to integrate texture streaming into the final Moana Island render due to Disney's Ptex per-face texture format [21], but we plan to support it in the future.

Figure 2.2: Scene rendered with an earlier version of our system using texture streaming. Scene credit: NVIDIA [19]

Chapter 3

RELATED WORK

Software implementations of raytracing on GPUs are not new. However, recently GPUs have been designed to contain dedicated hardware for raytracing. Hardware-accelerated raytracing is much faster than software implementations, and is the focus of this paper [3, 5].

Production scenes have always been large and have grown over time, demanding out of core methods to address rendering their large scale geometry [14, 23, 31]. For example, in 2014 a scene from Guardians of the Galaxy contained 1.2 billion unique triangles and used 30 GB of memory using the Arnold path tracer [10].

Recently, there have been several attempts to render one specific large scale scene as representative of these large scenes. Specifically, there has been exciting recent work on rendering the Disney Moana Island scene, which contains 39.3 million instances, 261.1 million unique quads, and 82.4 billion instanced quads [30, 27, 32, 12, 13].

Wald was able to load and render the Moana Island scene in its entirety on an RTX 8000 (48GB VRAM) [27]. This implementation is capable of rendering the Moana Island scene at a resolution of 2560x1080 at 25 frames per second, using progressive refinement with the image converging in just a couple seconds. Additionally, this work publicly provides data wrangling and file format utilities for interacting with the Moana Island scene. The pbrtParser library, created in part for this work, supports a much faster binary format for loading this scene [28]. This library and format is used by our implementation before converting to a custom format with chunk data. Wald's work is similar to ours in that the goal is to render the Moana Island scene.

However, our approach is distinct because we target GPUs with less memory, only 24GB versus 48GB, and scale to more complex scenes.

Zellman et al. partitioned the scene using various partitioning methods [32]. The most important insight from this paper for our application was that better partitioning can be achieved by combining spatial and data density metrics. The target application for the Zellman et. al. work was a distributed or multi-GPU system. As a result, ray transfers were an important metric that was minimized. There are similarities in the partitioning schemes; however our approach optimizes for geometry streaming instead of ray transfers. Ray transfers do not affect performance as much when both partitions are resident on the same GPU, and are therefore less of a concern for our application.

Hellmuth was able to render the Moana Island scene on an RTX 2070 (8GB VRAM). This was achieved by first partitioning the scene by mesh type. For each mesh, the geometry and accompanying instances are loaded. The rays are then traced through the loaded geometry. The next mesh is then loaded and traced. This process was repeated until all rays had finished. This technique uses hardware-accelerated raytracing and was able to render the Moana Island scene at a resolution of 1024x429 with 1024spp in 5h:10m [12]. Hellmuth's system also loads geometry dynamically to handle larger scenes; however our method is distinct because we use additional memory to track ray heuristics for loading assets using a priority based system.

Jaroš et al. implemented a path tracer capable of rendering the Moana Island scene, partitioning across multiple GPUs [13]. The paper targets an NVIDIA DGX-2 server, containing 16 Tesla V100 GPUs. The system uses CUDA Unified Memory, a form of virtual memory management, to address the data across the multiple GPUs [11]. The implementation described in the Jaroš et al. paper was able to render the Moana Island scene at a resolution of 5120x2560 with 1000spp in 3m:1s [13]. While there are

some similarities, our approach focuses on a single GPU with hardware-accelerated raytracing. Our method uses streaming to address the fact that hardware-accelerated raytracing does not support CUDA Unified Memory.

One key difference in our method compared to previous out-of-core approaches is that we prioritize loading of partitions to maximize overall ray throughput. Our system first traces rays through a top-level structure of partition bounding boxes. Then a histogram containing counts of rays intersecting each partition is calculated on the GPU. The partitions are then sorted by priority using the ray counts, with a bias towards already resident partitions. Once the most important partitions are loaded, the relevant rays are traced through the loaded geometry. This continues until all rays terminate.

Other GPU approaches use small pages and sort rays to achieve cache hits, but still evaluate all rays in the ray buffer every pass [9]. Instead, we allow rays intersecting less requested geometry to be stalled until other more coherent rays are processed. This allows our system to wait to load geometry until enough rays are requesting it, ensuring there is enough work to achieve adequate GPU utilization. This is especially helpful for larger partitions which are more costly to load. Instead of a traditional cache that pages in smaller regions at once [9, 29], we instead divide the scene into larger partitions and prioritize which to load before each tracing step.

We need larger geometry partitions to efficiently utilize hardware ray tracing. Compared to a software BVH implementation, we have limited flexibility. We cannot modify the BVH build and trace to insert custom logic or paging since it is implemented in hardware. Additionally, the BVH layout is internal to the hardware and driver so we cannot modify it. Therefore, we partition at a larger scale to still have optimized BVH build and trace times on the hardware, but insert an additional top-level BVH to determine which geometry to load. Previous approaches that use

fine-grain page-level caches would not be able to fully utilize hardware-acceleration for BVH building or tracing [9, 29, 18, 16].

Previous CPU approaches do include some priority-based loading, being the use of 8 priority buckets in Wald et al. and cost and benefit estimates for voxel scheduling in Pharr et al. [29, 18]. However, given that our work runs on a GPU, we use different methods. On modern GPUs, calculating ray counts per partition is a fast operation, and is even faster with larger and fewer partitions. Therefore, we utilize exact ray counts per partition into our priority-based loading scheme. Furthermore, to fully utilize the GPU, we load multiple chunks at once, then partition the ray buffer to run one large raytracing launch. Compared to a CPU system, our GPU needs this increased workload per step to ensure the GPU is not idle. Additionally, data transfers from system memory to GPU memory are very costly. Therefore, we minimize the data we send, using 48 bytes per 3x4 matrix instance transform and raw vertex and index buffers for triangle meshes. This saves memory bandwidth which saves overall time because the hardware BVH build is very fast and the BVHs are usually 3x the memory usage of the raw vertex and index buffers.

Chapter 4

ALGORITHM

Our approach to rendering large scenes that do not fit in GPU memory while taking advantage of hardware-accelerated raytracing, is composed of two stages: scene partitioning and the raytracing pipeline. Scene partitioning involves dividing the scene into chunks that fit into GPU memory. The raytracing pipeline includes the calculation of which chunks are most beneficial to stream, streaming the chunks, and tracing the rays through the loaded geometry. We describe these phases in more detail here.

Our benchmark scenes exceed the memory of our current GPU. We used the Moana Island scene, as well as a modified version of the scene with 26x the number of instances (∼230GB of GPU memory if all resident) [20]. This greatly exceeds our test GPU, the RTX A5000 with 24GB of memory.

4.1 Scene Partitioning

Our approach is intended for scenes that exceed the memory of the GPU and thus cannot all be loaded at once. Our method addresses this problem by dividing the scene spatially into chunks that can be loaded independently. First, the axis-aligned bounding box of the entire scene is calculated. Then this bounding box is subdivided into eight smaller bounding boxes. This process is repeated recursively until each bounding box contains less than a fixed-value of instances. See Figures 4.1, 4.2 for an example diagram of scene partitioning. This allows for more detailed regions of the scene to contain more chunks while less detailed areas contain fewer chunks. As

a result, all chunks have roughly the same memory requirements. Chunks containing no instances are discarded.

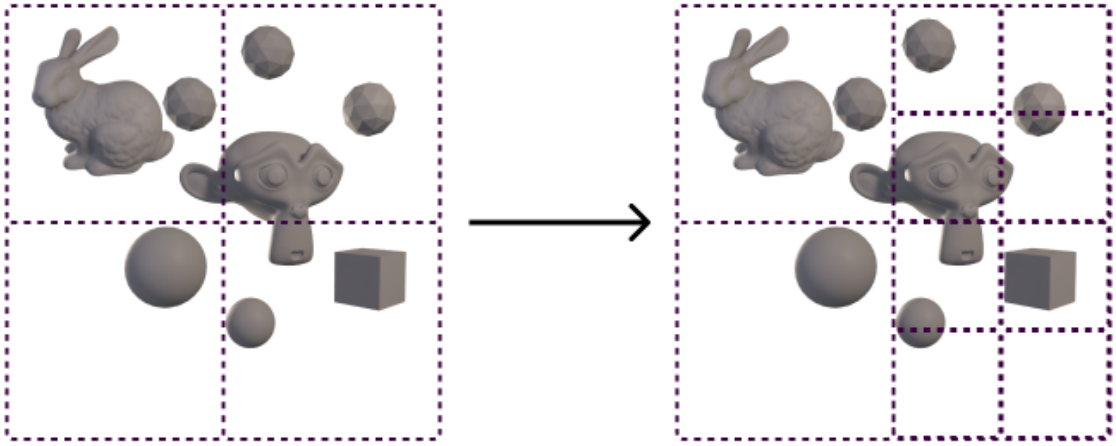

Figure 4.1: An example of a toy scene, recursively divided until the instance count reaches a given threshold

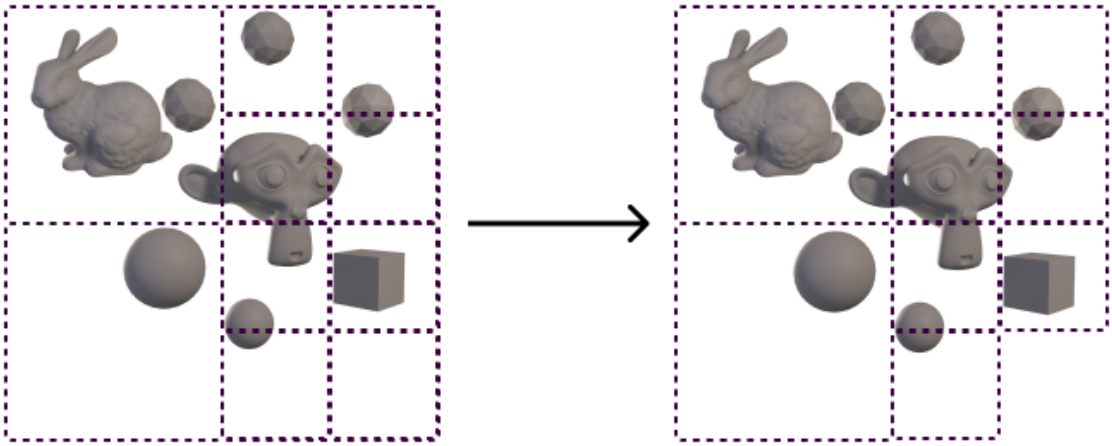

Figure 4.2: Empty chunks are discarded

Chunks are stored as a list of instance transforms per mesh ID. The chunk also has a list of dependencies, including the meshes it references, that need to be loaded along with the chunk. This method works well for most cases, however, for highly complex individual meshes this can cause challenges. Any meshes requiring more than 250 MB, (14 unique meshes, 421 instances in the Moana Island scene), are kept in their own individual bounding box (non-axis aligned in world space). Our methods

to handle these potentially overlapping boxes are handled in a re-tracing step. See Section 4.5 for details.

4.2 Raytracing Pipeline and Streaming

Once the scene has been partitioned into chunks we are ready to render with streaming support to manage geometry resident on the GPU. Well managed streaming of the geometry is the core contribution of our method. The main goals of our streaming method are:

- Minimize chunk loads and evictions to reduce data transfer overhead
- Maximize the number of rays that can be evaluated at once, allowing for higher GPU utilization

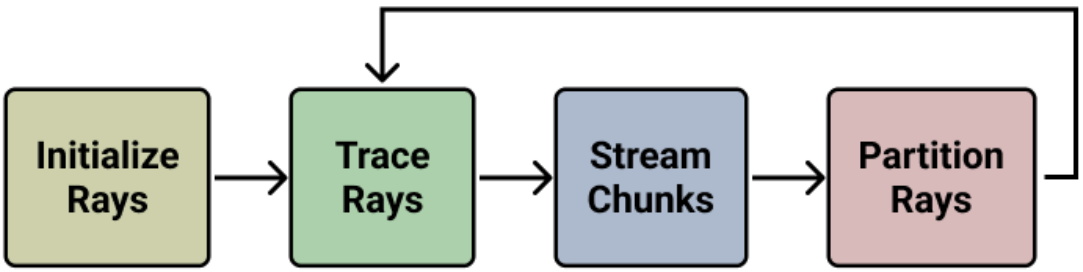

Figure 4.3: Overview of ray tracing pipeline

The raytracing pipeline starts with an uninitialized ray buffer for each pixel in the framebuffer. The algorithm then proceeds as follows:

1. **Initialize** the rays
2. **Trace** rays for a single wavefront step
3. **Stream** chunks based on priority system

4. **Partition** the rays by tagged asset residency

5. **Repeat** steps 2-4

To initialize the ray buffer, rays are **initialized** starting from the camera. For each pixel in the framebuffer, a ray is generated in a random direction through the pixel. All rays are explicitly tagged with no chunk dependency, to indicate they should be traced through all the chunk bounding boxes.

Rays are then **traced** through the chunk bounding boxes or the loaded chunks, depending on the ray's tag. Rays are traced one bounce at a time using a wavefront approach, and there are further re-tracing steps for overlapping chunks or ray misses within a chunk. See Section 4.5 for details on the re-tracing process.

The chunks are **streamed** using a priority system. The priority of chunks is determined using the function $P(A) = 256n$ if the chunk is resident, or $P(A) = n$ for non-resident assets, where n is the number of rays intersecting the chunk bounding box. Resident chunks are assigned higher priority because it is more efficient to trace rays through a chunk that has already been loaded. However, resident chunks are not always preferred, as a non-resident chunk with many rays hitting it could be more important to load. See Figure 4.4 for a diagram of the ray buffer operations.

Ray counts per asset are determined by using a histogram count on the GPU. The counts are then sent to the host for the calculation of priorities. For our benchmark of the Moana Island scene, the number of assets is less than 1,500 thus a CPU sort is not a bottleneck. However, the number of active rays may be much larger, for example, the best performance on the Moana Island scene used over 2 million active rays. Therefore, it is essential to calculate the ray counts per asset on the GPU.

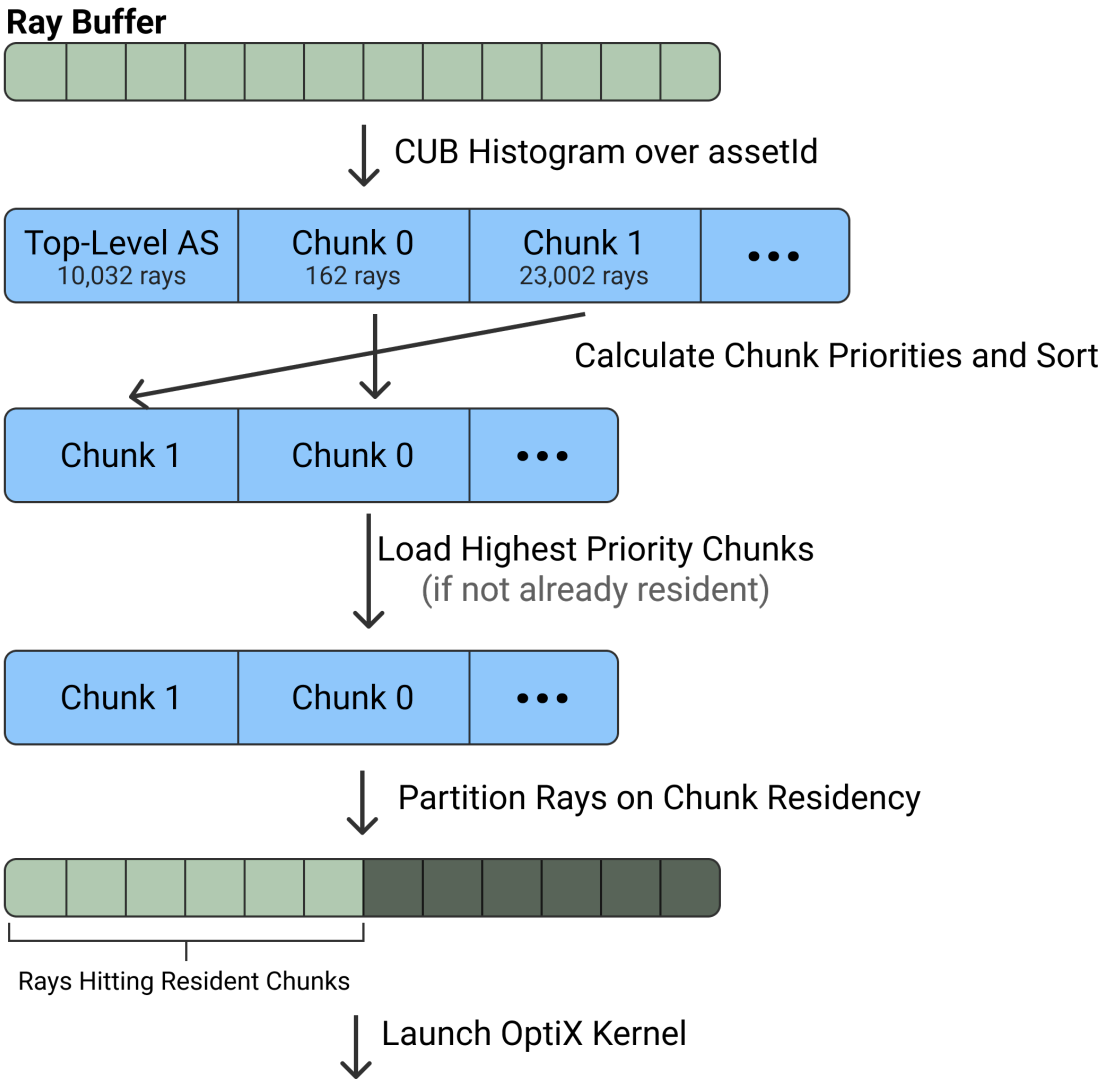

Figure 4.4: Ray buffer operations

Chunks are loaded starting with the highest priority chunk. Chunks are evicted starting with the lowest priority chunk. Chunk loading stops when the available memory is full. While iterating over the chunks in order of decreasing priority, if there is a significant decrease in priority, $P(C_{current}) < \frac{P(C_{previous})}{10}$, chunk loading is stopped early to prevent stalling for low-priority chunks.

Rays are **partitioned** by the residency of their tagged asset. Rays tagged with chunk bounding box or material tags are currently treated as always resident because the

memory requirements of those assets are very small. The end result is the ray buffer containing all rays tagged with resident assets contiguous in memory at the start of the ray buffer. The next trace step operates only on this portion of the buffer because rays intersected with non-resident chunks cannot be traced at the moment. Over time, the rays hitting the highest priority chunks will all be terminated, allowing for loading of lower priority chunks and evaluation of all rays. As an added benefit, waiting longer for lower priority chunks allows for more rays to hit the chunk. This gives the system more rays to process, at the same cost of a single chunk, therefore improving performance.

When a ray hits the environment or reaches the maximum bounce count, the ray is terminated. The color is accumulated into a pixel buffer. If the pixel has not reached the target sample count, a new ray is started from the camera to replace the terminated ray. Otherwise, the pixel value is written to the final image buffer.

All raytracing uses OptiX 7.4 [15]. Ray buffer histogram counts, sorting and filtering use the CUB library [8]. Starting the rays, re-starting the rays, and writing the terminated ray color values out to the framebuffer use custom CUDA kernels.

4.3 Asset Loading

When an asset (chunk or mesh) is requested, our system first checks if the asset is already resident in GPU memory, in which case the request is already fulfilled.

Otherwise, the asset must be built. This process is divided into two stages: preparing the OptiX build input, and building the acceleration structure.

Chunks: First, all the meshes referenced by this chunk are loaded. Then, for each mesh referenced by the chunk, a list of affine transforms (3x4 matrix) is sent to the

GPU and space is allocated for the chunk build input (which contains a buffer of OptixInstances). PCIe bandwidth is saved by using a smaller representation for the transforms, and then expanding on device because each OptixInstance is 80 bytes, while our affine transforms are 48 bytes each [15]. A CUDA kernel is used to expand each transform into an OptixInstance, which includes populating the mesh acceleration structure handle and the shader-binding table (SBT) offset for the instance.

Meshes: The vertex and index buffers are transferred to the device for the build input. Additionally, any applicable attribute buffers for the mesh are transferred as well (e.g. normals, texture coordinates).

Once the build input has been created, the OptiX API is used to build and compact the acceleration structure. This requires additional temporary space in GPU memory, so the system allocates extra space to leave sufficient memory available for build operations. Pre-built acceleration structures are not cached to host memory because the acceleration structure is multiple times larger than the build input. This improves the speed of memory transfers. Additionally, for chunks, the acceleration structure must be rebuilt regardless because the meshes may be in different locations in device memory since the last time the chunk was built.

4.4 Asset Cache

The algorithm relies on an asset cache to manage which assets are resident in GPU memory. For our method, we use an asset cache that is a fixed size, smaller than the total memory available on the GPU because extra space is needed for ray buffers and temporary buffers for OptiX builds. The asset cache handles loading and eviction of chunks and meshes given the requests from the raytracing pipeline.

Loading Assets: To load assets, the asset cache first checks if there is enough memory available in the cache. Each asset's memory usage can be estimated with an upper bound using the OptiX API. This estimate is later corrected with the exact memory usage of the asset after it is built. For chunks, acceleration structures for the referenced meshes are built first as the chunk must reference their acceleration structure handles.

Evicting Assets: If there is not enough space in the cache, chunks are evicted, starting with the lowest priority chunks. A reference count is kept to track the number of resident chunks referencing a mesh. When evicting a chunk, the meshes referenced can only be evicted if the reference count reaches zero.

For all memory allocations, this project uses the Rapids Memory Manager library. The sub-allocators provided by this library reduce the number of *cudaMalloc* and *cudaFree* calls, which avoids overhead from expensive memory allocations and frees.

4.5 Re-tracing of Overlapping Chunks

The inclusion of large meshes (greater than 250 MB) into a normal axis aligned chunk has the potential for inefficient streaming. This is due to the fact that axis-aligned chunks will be hit by intersecting rays that may miss the large mesh. For highly complex meshes, our algorithm isolates the large individual assets into tighter individual bounding boxes to optimize loading these large assets on the GPU for cases when they are more likely hit. As a result of this optimization, chunk bounding boxes can overlap. To solve for the correct intersection, a re-tracing approach that uses a step wise traversal of chunks is used. See Figures 4.5, 4.6.

1. Start with $nearT = 0$ and $hitT = \infty$, for a given ray

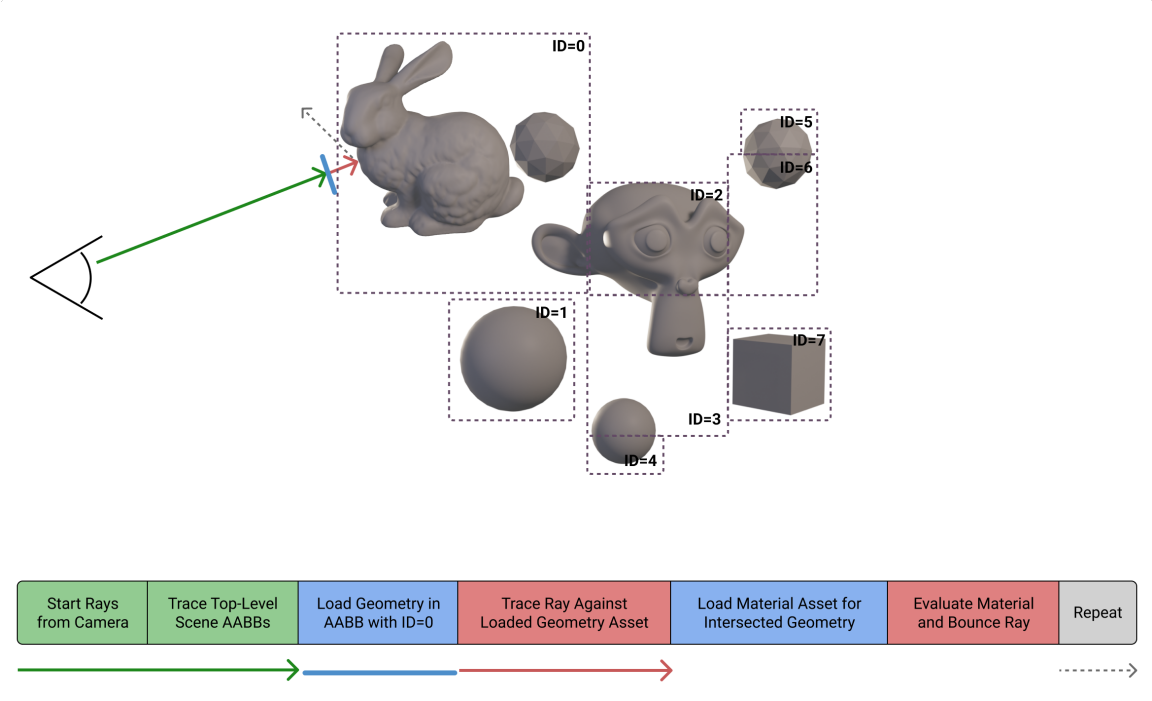

Figure 4.5: Re-tracing method: Ray hit

2. Trace to find the nearest chunk bounding box intersection with a t-value in the interval $[nearT, 0)$

3. When the chunk is loaded, trace the associated geometry. If there is a hit and $t < hitT$, update $hitT := t$

4. Repeat steps 2-3 until $hitT > nearT$

The ray origin may be inside one or more chunk bounding boxes. In this case, our approach first traces through all bounding boxes that contain the ray origin, recording the smallest t-value as $hitT$. Then it proceeds normally.

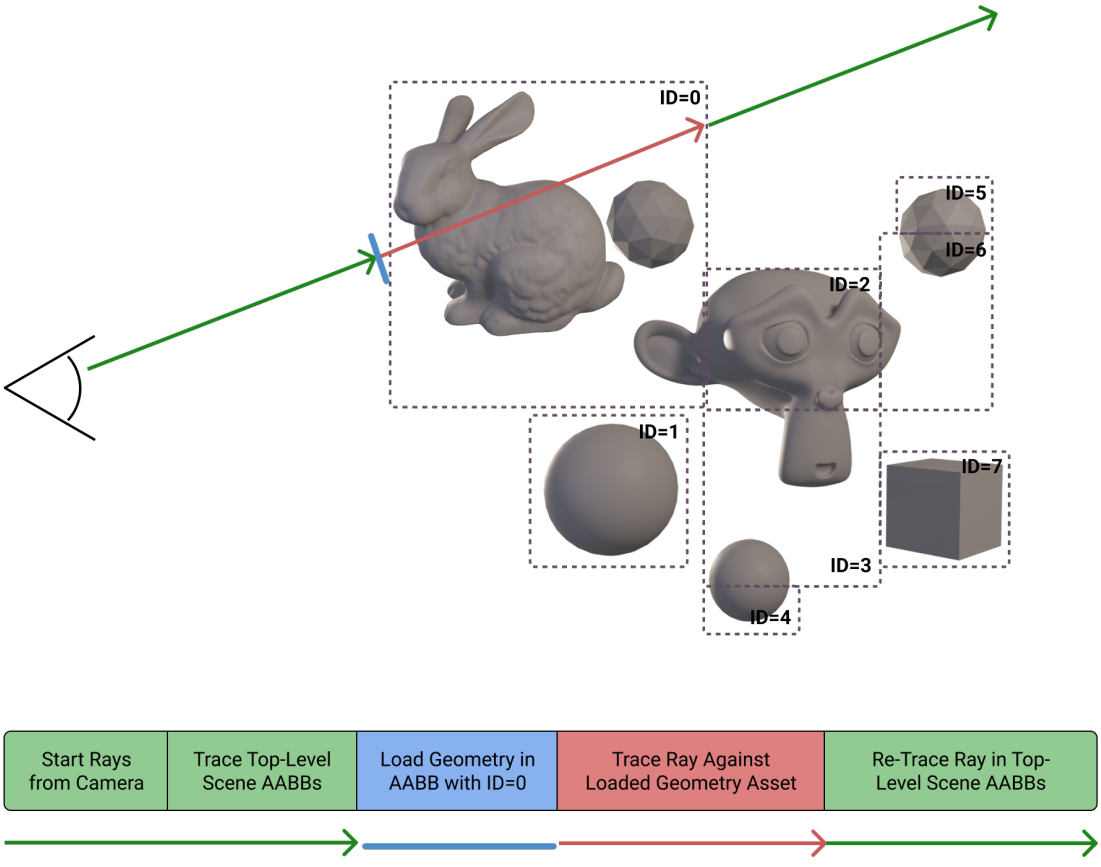

Figure 4.6: Re-tracing method: Ray miss

Chapter 5

SUPPORTING IMPLEMENTATION DETAILS

5.1 System Overview

This project uses a mix of CPU code (C++) and CUDA. CUDA is an extension of the C++ language for GPU programming. To run code on the GPU, the CPU calls a CUDA kernel function with varying dimensions of the work. For example, a kernel that processes each pixel in an image may be launched with the width and height of the image as the work dimensions. These dimensions are divided into block and grid dimensions for sub-grouping of the work. OptiX kernels are a subset of compiled CUDA code with special entry point functions for raytracing operations and events. Additionally, this project uses the CUB library built on top of CUDA for optimized general operations such as histogram counts and buffer partitioning.

5.2 OptiX Kernel

An OptiX kernel has several entrypoint functions.

OptiX supports hardware-acclerated raytracing of in-memory acceleration structures. Our project leverages OptiX for hardware acceleration, but uses custom intersection programs to stall the rays when intersecting unloaded chunk bounding boxes. See Table 5.1 for explanations of each entrypoint function type.

Table 5.1: OptiX raytracing entrypoint functions [15]

Function	Purpose	Execution Event
Ray Generation	Launching rays	Every thread in launch dimensions
Closest Hit	Evaluating intersections	Every closest hit from ray launches
Any Hit	Evaluating intersections, including determining light occlusion	Every hit from ray launches
Intersection	Intersection test for custom primitives	Every ray launch on custom primitives

5.3 CUDA Kernels

We use CUDA kernels for all other GPU operations that do require hardware-accelerated raytracing.

To save memory bandwidth, we send raw transforms stored as 3x4 float matrices (48 bytes each) for each instance. However, for OptiX to build an instance acceleration structure IAS, it requires the data to be packed in an 80-byte OptixInstance struct which includes the 3x4 matrix transform as well as a reference the instance's acceleration structure. Therefore, we allocate space for a buffer of OptixInstance structs and launch CUDA kernels to expand all the transforms efficiently into corresponding structs. This conversion to OptixInstances must occur every re-load of a chunk because the referenced mesh geometry acceleration structures may have been relocated since the last load. Performing this on the GPU rather than the CPU will also save compute time in addition to memory bandwidth.

Additionally, we use a mix of CUDA and CUB (a library on top of CUDA) for our general ray buffer operations. We use a custom CUDA kernel to initially populate the ray buffer with rays starting from the camera. Then we use a CUB histogram count

to count the rays intersecting each chunk bounding box. Next, chunks are loaded using a combination of CPU priority logic and the building of acceleration structures outlined above. Then, we use CUB to partition the ray buffer so the start of the ray buffer contains a contiguous array of rays all intersecting resident chunks. The main ray tracing using our OptiX kernel is performed next. After the rays have been traced an additional step, we use a CUDA kernel to write-out terminated rays to an image buffer and replace them with new rays from the camera if the pixel has not yet converged. Next, we use CUB to filter out any leftover terminated rays for converged pixels so we don't perform unnecessary work.

5.4 Memory Management

We use the Rapids Memory Manager (RMM) library for our GPU memory management. RMM provides an abstraction above raw memory management with 'cudaMalloc' and 'cudaFree' (similar to 'malloc' and 'free').

The down-side of 'cudaMalloc' and 'cudaFree' is that it requires a full GPU synchronization, which can be very costly. RMM supports sub-allocators which use a single call to 'cudaMalloc' to pre-allocate a memory pool. When memory allocations are requested, this large pool is divided into smaller regions without requiring a full GPU synchronization.

Sub-allocators can be much faster because of fewer full GPU synchronizations. However, our use case can cause sub-allocators to be severely fragmented. Our system has many large allocations for geometry, but relatively few small allocations. Our small temporary storage buffers can be pre-allocated. In our use case, sub-allocators often required more geometry to be evicted than strictly necessary because of memory fragmentation. As a result, the direct 'cudaMalloc' and 'cudaFree' performed

better than the sub-allocators used (pool and arena), despite the synchronization overhead. For example, our system rendered the Moana Island scene in 18m:33s with the standard 'cudaMalloc' and 'cudaFree'. However, due to increased eviction due to fragmentation, the pooled sub-allocator rendered the scene in 22m:39s. Although each allocation was faster, the increased eviction slowed the system slightly. These results may be different on a GPU with more memory available so we have left both raw 'cudaMalloc' and sub-allocators supported in our system.

In addition to using RMM, we use a custom class for managing GPU memory allocations. Our 'DevicePtr' class follows the RAII (Resource Acquisition Is Initialization) pattern. When we create a 'DevicePtr' the requested memory is allocated. When the 'DevicePtr' goes out of scope the memory is freed. We also wrap 'DevicePtr' in C++ smart pointers when necessary to transfer or share ownership.

Chapter 6

RESULTS

We have presented our system to render scenes that exceed the memory of the GPU. Our method divides the scene spatially into chunks that can be loaded independently and then streams those chunks to the GPU, minimizing chunk loads and evictions, to prevent memory transfer stalls and maximizing the number of rays that can be evaluated at once, allowing for higher GPU utilization.

Our system was able to render Disney's Moana Island scene in under 21 minutes, and a 26 times larger Moana Island scene in under 1h:28m. All renders use the following parameters: 1024x429 pixel resolution, 1024spp, with a maximum of 5 ray bounces. All materials are diffuse, and there is a solid color assigned to each mesh. Curves were not supported in this implementation, but only account for about 5% of the scene data (all instance counts and scene size estimates in this paper do not include curves) [20]. Our 26x Moana Island stress test shows that our system would be capable of rendering the extra curve geometry of a single Moana Island scene.

All timings were performed on a system using an RTX A5000 (24GB VRAM, 22GB used, 13GB asset cache, with the remainder for temporary build storage and ray buffers). The test system also used an Intel i9-10900K @ 3.70GHz and 64 GB DDR4-3200 MHz RAM.

Figure 6.1: Moana Island scene. 39.3 million instances. 1024spp, 1024x429. 18m:33s

6.1 Moana Island Scene

Disney's Moana Island scene contains 39.3 million instances, 261.1 million unique quads, and 82.4 billion instanced quads [30]. Our system was able to render this scene in 18:33s minutes. See Figure 6.1 for an image of the rendered Moana Island scene.

6.2 26x Moana Island Scene

As a stress test, a larger scene was created that contained 26 duplicate Moana Island scenes, without multi-level instancing. The scene contained over 1.02 billion instances, 2.14 trillion instanced quads, and would require ∼230GB of GPU memory if entirely resident. The number 26 was chosen to reach 1 billion instances, but is not an inherent limit in our system. See Figure 6.2 for our render of the 26x Moana Island scene.

The system rendered this scene in 1h:27:10s. The scene partitioning completed in 15m:23s. The raytracing and streaming completed in 1h:11m:47s. The focus of this

Figure 6.2: 26x Moana Island test scene. No multi-level instancing. 1.02 billion instances. ~230GB memory usage if all resident at once. 1024spp, 1024x429. 1h:27m:10s

work was on the raytracing throughput, and the current partitioning implementation is a single-threaded CPU version. Additional system memory and a multi-threaded implementation would increase performance for the scene partitioning stage of the algorithm.

6.3 Asset Cache Effectiveness

To verify the effectiveness of our asset cache, we recorded the number of loads per chunk and the total number of rays traced through each chunk.

For each chunk, we measured the rays traced through the chunk and the number of times the chunk was loaded into the cache. Overall, partitions with more ray intersections were loaded less frequently, showing our system effectively prioritizes important partitions. For example, in the 26x Moana Island test scene, the most requested chunk was loaded 114 times and 300,874,269 rays were traced within it over the full render. There were 2,231 occluded partitions that were never loaded out of 8,048 total. The most loaded chunk was loaded 980 times and 5,816,734 rays

were traced within it. Ideally, every chunk would be only loaded once. However, with a scene nearly 10 times the memory of the GPU (230GB scene vs. 24GB GPU memory), and many rays only become available after multiple bounces, some partitions must be loaded more than once. These metrics show that our system is working correctly and prioritizing loading partitions for higher overall ray throughput.

Additionally, we visualized this output. Each box represents an axis-aligned bounding box of a chunk. Smaller boxes indicate denser geometry due to our adaptive partitioning scheme.

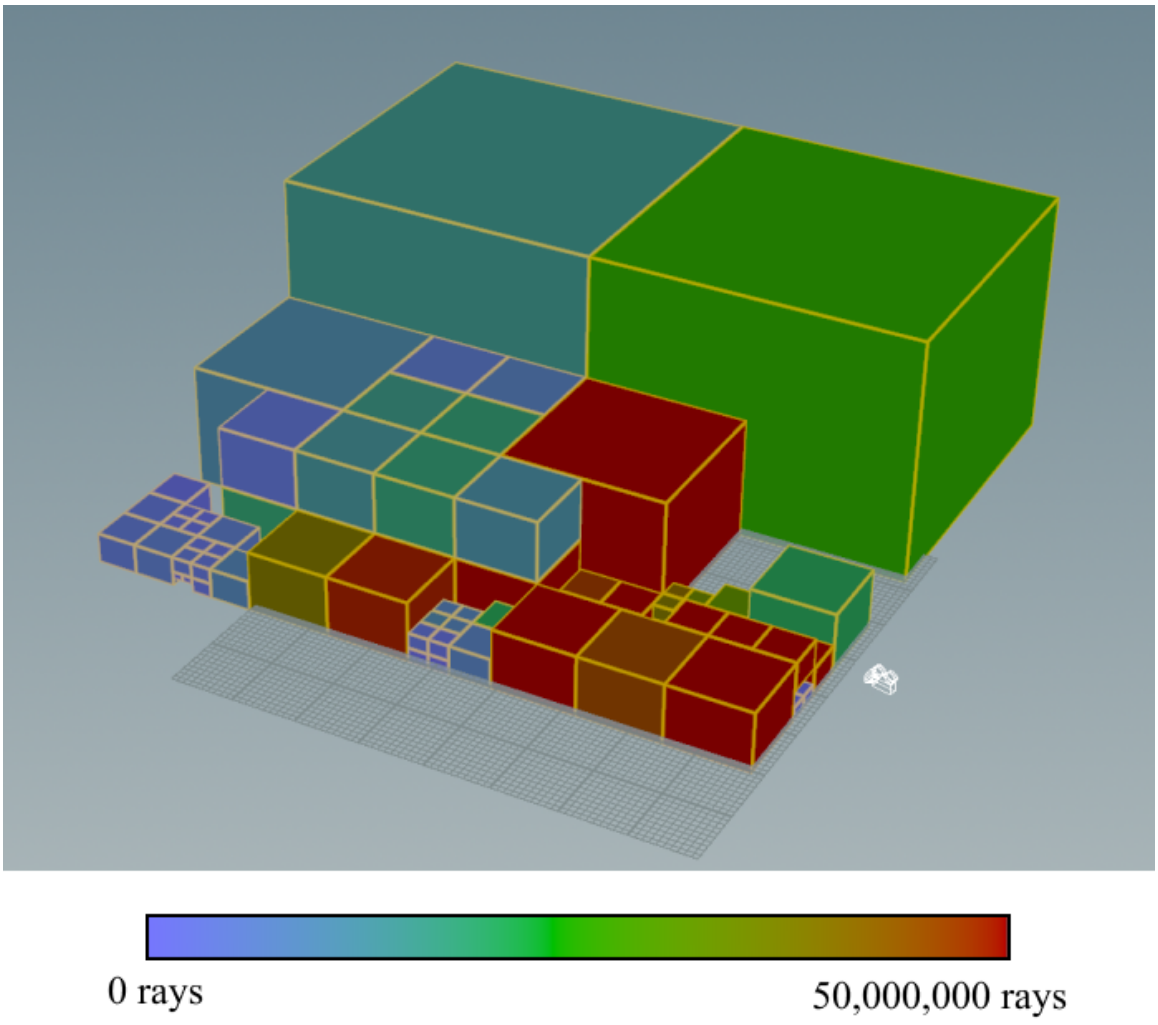

Figure 6.3: Visualization of rays traced per chunk

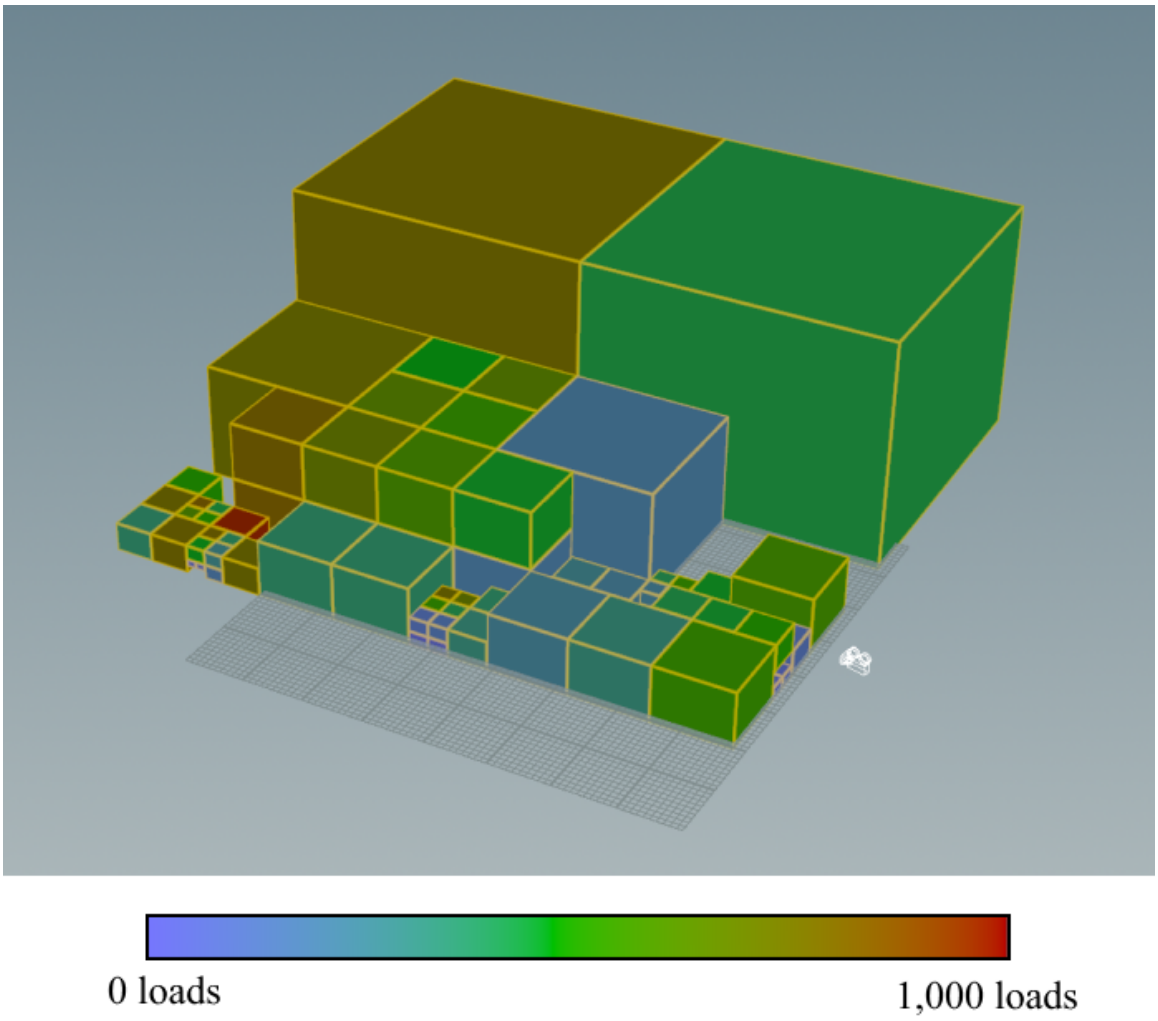

Figure 6.4: Visualization of chunk loads

As shown in Figures 6.3, 6.4, the regions of high ray throughput are roughly inverse to the regions with high load counts. This verifies that our system is working correctly and effectively prioritizing keeping important chunks in memory to ensure overall higher ray throughput of the entire system.

6.4 Straggling Path Cut-off

At the start of the render the ray buffer contains many coherent rays intersecting similar spatial regions. As the render progresses, the ray buffer becomes increasingly

divergent. This is mitigated by stalling rays until the priority of a chunk is high enough which increases coherency. However, at the end of the render all rays must be processed, so diverging rays will eventually need to be processed. This is a problem in raytracing in general because of incoherent memory accesses, but is even more of an issue for out-of-core approaches.

Figure 6.5: 26x Moana Island scene. Total render 1h:27:10s. 15m:23s partitioning, 1h:11m:47s rendering

Figure 6.6: 26x Moana Island scene. Stopped early and displayed with the rays that had already terminated within the first 5 minutes of the render stage

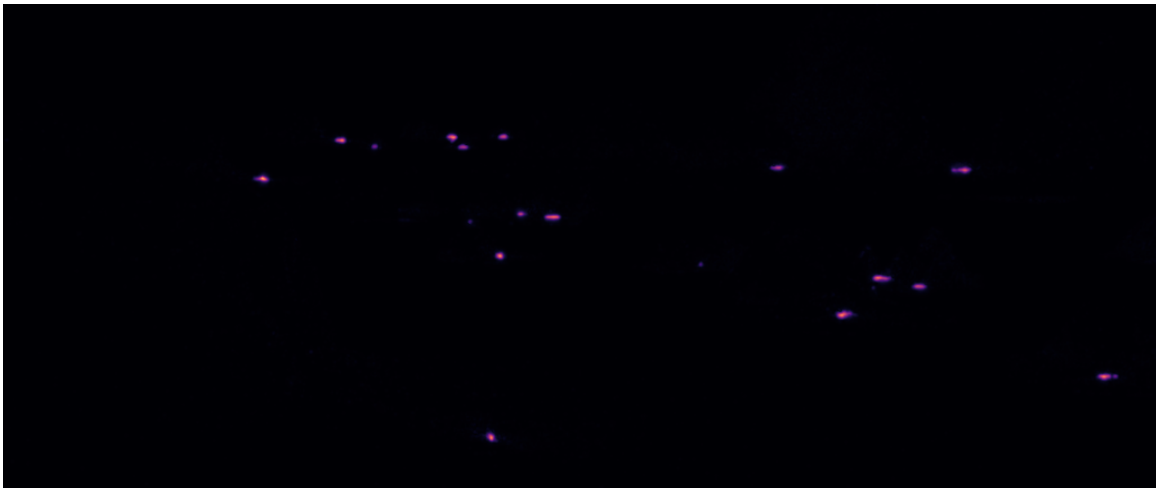

Figure 6.7: FLIP image comparison between full render and progress at 5 minutes. See Figures 6.5, 6.6

The 26x Moana Island render completed in 1h:11m:47s in the render stage (not including scene partitioning). In the first 5 minutes, 27,985,156 out of 28,114,944[1] pixels have converged, leaving only 129,788 pixels remaining. See Figures 6.5, 6.6, and Figure 6.7 for an image comparison to the final render.

As shown in Table 6.1, our system is able to very quickly render many rays when the ray buffer is large. However, as the number of active rays decreases, our system slows down because it needs to load more chunks, each with fewer rays. This is a problem with particularly dense areas of geometry. In the case of the 26x Moana Island scene, there are millions of triangles rendered within a single pixel. This shows that for the bulk of the image, we are rendering efficiently. However, very dense regions take a disproportionate amount of time to render with respect to their final contribution to the image. In future work we propose an LOD-based approach to mitigate this.

[1]The final rendered image has a resolution of 1024x429 but is rendering at 8x scale in each dimension to give the GPU more work. This allows 64 samples per pixel to be active at once which increases work done while each chunk is loaded.

Table 6.1: Render time versus non-converged pixels remaining in the rendered image

Time	Pixels Left
0m	28,114,944
5m	129,788
10m	61,655
15m	36,875
20m	24,832
25m	16,787
30m	11,611
35m	8,379
40m	6,054
45m	4,221
50m	2,990
55m	1,991
1h	1,067
1h:5m	356
1h:11m:47m	0

This analysis is for performance analysis only. Using the cut-off is biased and would produce too much noise in a final render. Instead, an unbiased approach to straggling ray cut-off is proposed as future work.

Chapter 7

CONCLUSIONS & FUTURE WORK

Hardware-accelerated raytracing is able to massively speed up rendering [3, 5]. However, production scenes continue to grow in complexity, outstripping the memory capacity of most GPUs. Geometry streaming is vital for production quality rendering of large scenes.

We presented an out-of-core rendering method capable of rendering arbitrarily large scenes on a single GPU using hardware-accelerated raytracing and geometry streaming.

There are a few major aspects we would like to focus on in future work:

- More advanced material models and sampling.
- Texture streaming.
- Chunk-based LODs.
- Unbiased straggling path cut-off.

First, we would like to make our system more usable for final production renders. We decided to focus on the main out-of-core approach for this paper due to time constraints, but would like to expand our feature set to match existing path tracers in the future. For simplicity of implementation, we used a simple diffuse Lambert BRDF and an environment light. In the future, we would like to implement the Disney material model and more light types, along with multiple importance sampling.

Second, we would like to implement texture streaming. Texture streaming already exists in several production renderers [24]. Texture streaming is similar to our geometry demand loading system. The Demand Loading library [15] is an open source implementation compatible with OptiX and is likely what we will use in the future.

Third, we would like to optimize regions of dense sub-pixel geometry. There are millions of triangles per pixel in some regions. This detail is not necessary in the final image. Several triangles per pixel should be sufficient. Therefore, and level-of-detail (LOD) solution would be useful here. However, it is not as simple as it seems. Existing level-of-detail approaches work on a single triangle mesh. In our case, the bulk of the memory usage does not come from raw triangle data, but instead the vast number of instance transforms. Simplyfing instances is a much more complex problem because you can not just remove small instances. Many small instances, such as grass or leaves, can combine to form a dense volume.

One idea to solve this issue is to use a voxel-based level-of-detail approach. Compared to more exact mesh decimation approaches, a rougher voxel re-meshing could be sufficient and would be fast since it is easier to parallelize. Normally, mesh decimation approaches, such as edge decimation [7], are preferred since they produce more visually accurate results. However, with millions of triangles per pixel a rough voxel approximation may be unnoticeable. The voxel mesh could only have thousands of triangles, using very little memory, but may still be sufficient since thousands of triangles per pixel is still fairly dense. Furthermore, at this ratio of triangles to pixels, using per-vertex attributes instead of baked textures may be sufficient for the level-of-detail approximation. Normals, roughness, and other material parameters could be stored and averaged while rasterizing each polygon to voxels. When averaging, varying normals for a voxel would contribute to a higher roughness value, because at this scale high-density normals are essentially microfacets in a roughness model.

A similar idea of converting normal information to roughness when zoomed out has been pursued in the cited blog post [22]. A stochastic approach to swap between level-of-detail approximations and the real geometry could help improve temporal coherence. Overall, level-of-detail approximations could greatly speed up the render and help mitigate the memory pressure of ray divergence.

Lastly, an unbiased variant of the straggling path cut-off could help save time. Russian roulette sampling randomly terminates rays based on their luminance and overall contribution to the image. This introduces noise but saves overall computation time. A similar approach for straggling path cut-off could be to randomly terminate rays based on their expected memory cost for processing. This could introduce much more noise as we do not have the guarantee that these rays will contribute little to the final image as we do with Russian roulette. But if we sample randomly, and when the sample is not terminated we weight it accordingly, it still could help save computation time.

REFERENCES

[1] Nefi Alarcon. *Ray Tracing Essentials Part 6: The Rendering Equation*. en-US. Apr. 2020. URL: https://developer.nvidia.com/blog/ray-tracing-essentials-part-6-the-rendering-equation/ (visited on 06/04/2022).

[2] Jason Brownlee. *A Gentle Introduction to Monte Carlo Sampling for Probability*. en-US. Nov. 2019. URL: https://machinelearningmastery.com/monte-carlo-sampling-for-probability/ (visited on 06/04/2022).

[3] John Burgess. "RTX on—The NVIDIA Turing GPU". In: *IEEE Micro* 40.2 (Mar. 2020). Conference Name: IEEE Micro, pp. 36–44. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2971677.

[4] Brian Caulfield. *CPU vs GPU: What's the difference?* en-US. Dec. 2009. URL: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (visited on 06/04/2022).

[5] Brian Caulfield. *What's the Difference Between Hardware- and Software-Accelerated Ray Tracing?* en-US. June 2019. URL: https://blogs.nvidia.com/blog/2019/06/07/whats-the-difference-between-hardware-and-software-accelerated-ray-tracing/ (visited on 04/15/2022).

[6] Brian Caulfiend. *What's the Difference Between Ray Tracing, Rasterization?* en-US. Mar. 2018. URL: https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/ (visited on 06/04/2022).

[7] *CGAL 5.4 - Triangulated Surface Mesh Simplification: User Manual*. URL: https://doc.cgal.org/latest/Surface_mesh_simplification/index.html#Chapter_Triangulated_Surface_Mesh_Simplification (visited on 06/04/2022).

[8] *CUB: Main Page*. URL: https://nvlabs.github.io/cub/index.html (visited on 04/15/2022).

[9] Kirill Garanzha et al. "Out-of-core GPU ray tracing of complex scenes". In: *ACM SIGGRAPH 2011 Talks*. SIGGRAPH '11. New York, NY, USA: Association for Computing Machinery, Aug. 2011, p. 1. ISBN: 978-1-4503-0974-5. DOI: 10.1145/2037826.2037854. URL: https://doi.org/10.1145/2037826.2037854 (visited on 06/03/2022).

[10] Iliyan Georgiev et al. "Arnold: A Brute-Force Production Path Tracer". In: *ACM Transactions on Graphics* 37.3 (Aug. 2018), 32:1–32:12. ISSN: 0730-0301. DOI: 10.1145/3182160. URL: http://doi.org/10.1145/3182160 (visited on 06/10/2022).

[11] Mark Harris. *Unified Memory for CUDA Beginners*. en-US. June 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (visited on 04/15/2022).

[12] Chris Hellmuth. *Render blog · by Chris Hellmuth*. Oct. 2020. URL: https://www.render-blog.com/ (visited on 04/07/2022).

[13] Milan Jaroš et al. "GPU Accelerated Path Tracing of Massive Scenes". In: *ACM Transactions on Graphics* 40.2 (Apr. 2021), 16:1–16:17. ISSN: 0730-0301. DOI: 10.1145/3447807. URL: https://doi.org/10.1145/3447807 (visited on 04/15/2022).

[14] Janne Kontkanen, Eric Tabellion, and Ryan S. Overbeck. "Coherent Out-of-Core Point-Based Global Illumination". In: *Computer Graphics Forum* (2011). ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2011.01995.x.

[15] *NVIDIA OptiX 7.4*. URL: https://raytracing-docs.nvidia.com/optix7/index.html (visited on 04/15/2022).

[16] Matt Pharr and Pat Hanrahan. "Geometry Caching for Ray-Tracing Displacement Maps". en. In: *Rendering Techniques '96*. Ed. by Xavier Pueyo and Peter Schröder. Eurographics. Vienna: Springer, 1996, pp. 31–40. ISBN: 978-3-7091-7484-5. DOI: 10.1007/978-3-7091-7484-5_4.

[17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Bounding Volume Hierarchies*. URL: https://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies (visited on 06/04/2022).

[18] Matt Pharr et al. "Rendering complex scenes with memory-coherent ray tracing". en. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*. Not Known: ACM Press, 1997, pp. 101–108. ISBN: 978-0-89791-896-1. DOI: 10.1145/258734.258791. URL: http://portal.acm.org/citation.cfm?doid=258734.258791 (visited on 06/03/2022).

[19] *Pixar Universal Scene Description (USD)*. en-US. July 2019. URL: https://developer.nvidia.com/usd (visited on 06/04/2022).

[20] Heather Pritchett and Rasmus Tamstorf. *Walt Disney Animation Studios - Moana Island Scene*. en. 2018. URL: https://disneyanimation.com/resources/moana-island-scene/ (visited on 04/15/2022).

[21] *Ptex*. URL: http://ptex.us/ (visited on 06/04/2022).

[22] *Roughness mip maps based on normal maps?* en. Sept. 2018. URL: https://kosmonautblog.wordpress.com/2018/09/17/roughness-mip-maps-based-on-normal-maps/ (visited on 06/10/2022).

[23] R. Schregle, L.O. Grobe, and S. Wittkopf. "An out-of-core photon mapping approach to daylight coefficients". In: *Journal of Building Performance Simulation* 9.6 (2016), pp. 620–632. DOI: 10.1080/19401493.2016.1177116. eprint: https://doi.org/10.1080/19401493.2016.1177116. URL: https://doi.org/10.1080/19401493.2016.1177116.

[24] *The world's first fully GPU-accelerated, biased renderer — Redshift...* en. URL: https://www.maxon.net/en/redshift/features (visited on 06/04/2022).

[25] Alex Treviño. *The Junk Shop*. en. Nov. 2019. URL: https://cloud.blender.org/p/gallery/5dd6d7044441651fa3decb56 (visited on 06/04/2022).

[26] *Tutorial 33 - Instanced Rendering*. URL: https://www.ogldev.org/www/tutorial33/tutorial33.html (visited on 06/04/2022).

[27] Ingo Wald. *"The Elephant on RTX" – First Light. (or: "Ray Tracing Disney's Moana Island using RTX, OptiX, and OWL")*. en. Oct. 2020. URL: https://ingowald.blog/2020/10/26/moana-on-rtx-first-light/ (visited on 04/07/2022).

[28] Ingo Wald. *Ingowald/PBRT-parser: A simple parser for the PBRT file format*. URL: https://github.com/ingowald/pbrt-parser/tree/master.

[29] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. "An interactive out-of-core rendering framework for visualizing massively complex models". In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05. New York, NY, USA: Association for Computing Machinery, July 2005, 17–es. ISBN: 978-1-4503-7833-8. DOI: 10.1145/1198555.1198756. URL: https://doi.org/10.1145/1198555.1198756 (visited on 06/03/2022).

[30] Ingo Wald et al. "Digesting the Elephant – Experiences with Interactive Production Quality Path Tracing of the Moana Island Scene". In: *arXiv:2001.02620 [cs]* (Jan. 2020). arXiv: 2001.02620. URL: http://arxiv.org/abs/2001.02620 (visited on 03/08/2022).

[31] Rui Wang et al. "GPU-Based out-of-Core Many-Lights Rendering". In: *ACM Trans. Graph.* 32.6 (Nov. 2013). ISSN: 0730-0301. DOI: 10.1145/2508363.2508413. URL: https://doi.org/10.1145/2508363.2508413.

[32] Stefan Zellmann et al. “Finding Efficient Spatial Distributions for Massively Instanced 3-d Models”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Steffen Frey, Jian Huang, and Filip Sadlo. The Eurographics Association, 2020. ISBN: 978-3-03868-107-6. DOI: 10.2312/pgv.20201070.