Realtime GPU Raytracing

Minxuan Guo, Nicholas Kelly, Liang Xia

I. Introduction

In the field of realtime graphics, most scenes are produced through a method called Rasterization. During rasterization, the entire scene is transformed based upon the camera viewport. In this manner, all rays from the camera are parallel to each other and object intersection becomes fast and regular. While this method is sufficient for a 3D to 2D projection of the scene, raytracing could be used to achieve more complex effects such as dynamic shadows, reflection, and refraction. Raytracing is a method that approximates physical light transport by casting rays not only from the camera (primary rays), but also for reflection, refraction, and light visibility (secondary rays). It is heavily used in animation and movie production due to the achievable realism and flexibility for various effects. However, raytracing is not nearly as uniform rasterization due to the fact that rays can be cast from any location in the scene where they are only cast from the camera in rasterization. Additionally, since rasterization is a very uniform process, a pipeline with special-purpose hardware exists in modern GPUs which allows for realtime rendering of 3D scenes. Unfortunately, most GPUs do not have raytracing hardware and many raytracers are usually implemented in CPUs (or Many-Integrated-Cores such as the Xeon Phi) due to the non-uniform, parallel nature of the implementation. Although, due to the programmability and availability of modern GPUs, raytracers are recently being implemented in CUDA or GLSL (OpenGL). While these solutions attempt to utilize the inherent parallelism of raytracing, achieving realtime performance is often difficult due to how GPUs handle divergence. Thus, many techniques have been proposed for load balancing, collision detection, and reducing the overall work in order to reduce render times. In this paper, we aim to explore current methods for realtime raytracing and attempt to elaborate upon them in order to produce a near realtime implementation. Additionally, while some prior work only implement partial features (e.g. only primary and shadow rays), we aim to support primary and all secondary (refraction, reflection, light/shadow) rays as well as arbitrary triangle meshes and textures. We decided to implement the final raytracer in CUDA due to the availability of NVIDIA hardware and their toolset.

II. Related Work

Due to the irregular nature of raytracing and the need for efficient collision checking, spatial partitioning structures are usually utilized. These structures adaptively partition the geometry within a scene in a tree-like fashion. Thus, instead of testing every triangle for ray collisions, the tree is traversed with more efficient collision checks (e.g. bounding boxes) and only select leaf-nodes are tested for triangle-ray intersections. However, these structures contain an initial overhead to produce and the construction time is directly proportional to the intersection speedup (quality of the structure). Thus, in production-quality raytracers, more time is spent on constructing a balanced tree due to the long render times. Although, in our case we need fast tree construction if we intend to support near-realtime operation (with animation). Lauterbach, et al [4] introduce a fast BVH (bounding volume hierarchy) construction method which use a Morton code (Z-ordering) to quickly partition and construct the BVH. In addition, the method they introduce is completely parallel and fits the CUDA execution model well. Thus, we use this method for our BVH construction (which we elaborate on later).

The classic traversal for a raytracer follows a recursive or stack-based implementation. However, both of these approaches are prohibitive in a GPU where we aim for efficient memory bandwidth. Afra and Szirmay-Kalos [2] discuss a "stackless" traversal mechanism where a bit-vector is used instead of a full stack which contains information about the state of the traversal. Instead of having a full stack and performing a stack pop, they traverse back up the tree and inspect the bit-vector. Each time there is a node traversal up or down, the stack is shifted right or left, respectively. When visiting a child, if the other node will need to be traverse later, a "1" is shifted into the bit-vector. Then, when the ray traverses up the tree, it goes down the opposite child and resets that bit in the bit-vector. Additionally, the bit-vector indicates when there is no more work to be done as it will be zero in that case.

While raytracing is a non-uniform rendering solution, many of the primary and shadow rays are usually fairly consistent in terms of origin and direction. There has been much prior work focusing on ray-packetization where rays are combined into one and traced until they effectively diverge. Garanzha [1] uses ray-packetization as well as ray-ordering. In their implementation, they first compute "hashes" for each ray. A hash contains information about the direction and origin of the ray. In particular, they quantize the 3D direction and origin into a 32-bit hashed value. They then perform stream-compression in order to reduce the amount of elements which are sorted. After sorting and decompressing the ray hashes, frustums are created for each hash grouping. A frustum is similar to a cone but contains two axis-aligned rectangles which indicate the divergence or spread of the ray-packet. These packets are traced through the BVH, gathering possible leaf-nodes for later intersection testing. After all packets are traced, the triangles are tested for ray intersections in parallel in order of nearest distance (in order to minimize divergence). While there is possibly more divergence for reflection and refraction rays, eye and shadow rays will often benefit with improved parallelism.

Additionally, they are able to group arbitrary rays due to their parallel hashing and sorting method. For our purposes, we employ a similar hash sorting mechanism, but focus on a different divergence-avoidance method (Treelets) instead of packetization.

For more incoherent rays (reflection and refraction), Timo and Tero [3] discuss the usage of "Treelets" in their raytracer architecture analysis. Treelets are type of graph partitioning specific to trees where sub-trees or node groupings are created. This grouping is useful for creating work queues in spatial partitioning trees (such as a BVH). In their implementation (simulation), treelets are constructed using a bottom-up dynamic programming and a top-down, greedy approach which seeks to maximize the surface area of the treelet while targeting a maximum size. These treelets are used as work queues for the sections of the treelet and since rays can potentially be in any part of the scene during traversal, they are an unbiased method of improving parallelism and locality. Since the paper focuses on architectural additions and analysis, they also propose fine-grained scheduling, launching, and compaction of work to warps. Many of the suggestions of that they provide need architectural features for efficient implementation but we use the idea of treelets in order to minimize divergence and improve locality.

III. Design

A. Overview

In order to render animated scenes in realtime, we decided to re-generate the acceleration structure in realtime. We did not explore tree-update methods, since re-generation is simpler method of enabling animations instead. We decided to use a BVH tree for our acceleration structure due to the relative ease of construction compared to a KD-Tree. Our research into prior work indicated that choosing KD-Tree split planes was a more difficult algorithm to parallelize.

Our ray-tracer has two main parts, BVH construction and BVH tree traversal. We implemented the fast BVH tree construction technique described by Karras [5]. Our tree traversal algorithm utilizes treelets [3], ray sorting [1] and stackless BVH concepts [2]. The flowchart (Figure 1) illustrates our algorithm. The following sections describe each step in more detail.

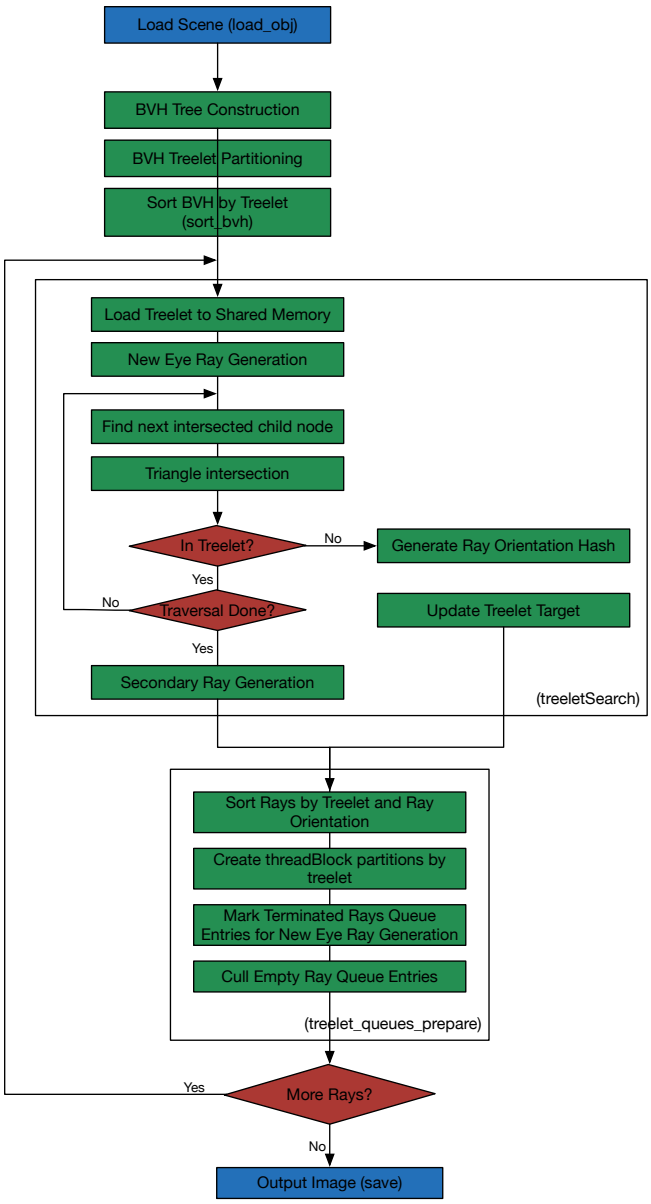


Fig. 1: Raytracer operation

B. BVH-Tree Construction

We base our BVH-Tree construction on Linear BVH (LBVH) algorithm [4] to take full advantage of parallelism. A triangle array is sorted in Morton code order to improve locality for further steps. BVH-Tree construction is conducted in three steps: primitive sorting, leaf-node construction, and internal-node construction.

1) Primitive Sorting: The first step of BVH construction is to calculate a bounding box array to start with. The Linear BVH algorithm computes this array by sorting primitives with morton code. Each primitive is assigned with a morton code, representing its position along the z-order curve. Since the morton code can be calculated using the geometric coordinates of the primitive, assigning morton code to primitives can be parallelized to N threads $(N = numberofprimitives)$. Parallel radix sorting is used to sort primitives. The bounding box of each primitive is sorted along with primitives for further usage.

2) Leaf-node Construction: Leaf-nodes wrap a small set of primitives in their bounding boxes. Since the morton code indicates primitive's position along z-order curve, we can group primitives with same morton code to a leaf-node. If the scene is sparse, the morton code can be truncated before grouping to keep number of leaf-nodes reasonable and to reduce leaf-node construction time. After previous primitive sorting, primitives are in a linear morton-code order, and all primitives in a leaf-node should sit in a linear range. Thus, we only need to find the start and end position of leaf-nodes to determine what primitives are warped in each leaf-node. Therefore we can still break the work to N threads, each thread

responsible for decide whether its primitive is a start point of a leaf-node. With such information, an inclusive scan is used to determine leaf-node number each starting point belongs to.

Getting bounding box of a leaf-node can be broke to the leaf-node granularity. Each thread take one leaf-node, reading the bounding box of primitives in the leaf-node and get the bounding box of that leaf-node.

3) Internal-node Construction: The goal of internal-node construction is to get each node's children and parent. internal-nodes enclose multiple leaf-nodes or internal-nodes according to its position in the hierarchy. Unlike leaf-nodes, internal-nodes has an inherent tree structure, posing great challenge to parallelize construction.

To solve this problem, we adapt a specific tree layout [5] to indexing internal-nodes. According to this layout, leaf-nodes and internal-nodes are stored in two separate arrays. leaf-nodes have index from 0 to $(leafNum - 1)$, internal-nodes have index from 0 to $(leafNum - 2)$. The index of internal-nodes are assigned with following rules:

1) internal-node 0 is the root node
2) if one internal-node splits at leaf-node i, its left child and right child would have index i and (i+1) respectively, as shown in Figure 2

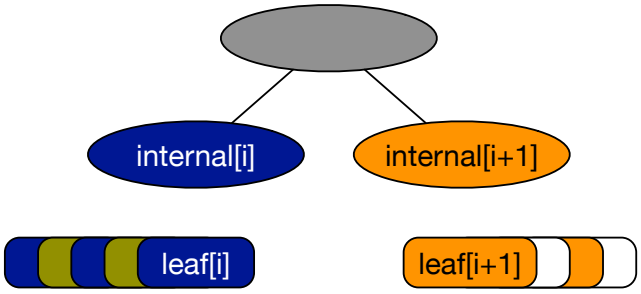


Fig. 2: Indexing rule for internal-nodes

With the rules above, we can construct the internal-node hierarchy if knowing each internal-node's split point. Given the consideration that morton code indicates spatial position, we split the internal-node at the point where morton code changes most significantly in the range. Hence we break internal-node construction into two task: find coverage range for all internal-nodes and find the split point of each range.

Because of the linear organized primitives, each internal-node warps consecutive leaf-nodes. Like leaf-node construction, we only need to know the starting leaf-node and ending leaf-node to get what's covered by each internal-node. From the indexing rules above we can draw the conclusion that leaf-node i is either the starting leaf-node or the ending leaf-node of internal-node i. To get the other end of internal-node's range, we start from the knowing end and look at its neighbour to find less significant change in morton codes. After getting coverage range, we only need to read morton code of leaf-nodes in that range to find a point where adjacent morton bits varies most significantly.

In the process described above, each internal-node would only need to read leaf-node array to decide its split and children. Thus the process of each internal-node is independent to each other despite of the recursive tree structure. Therefore, we break the process of getting internal-nodes hierarchy into the granularity of internal-nodes.

Unlike the process of getting internal-nodes hierarchy, getting bounding box is inherently sequential since the bounding box calculation of children need to be finished before bounding box calculation of parent starts. However, bounding box calculation of nodes without parental relationship can be parallelized. To take advantage of this parallelism, we starts with $leafNum$ threads. Each threads starts with a leaf-node and climb up the hierarchy after finishing current calculation. Since each node could only calculate its bounding box after its two children are done, the first thread to enter a node should terminates right after entering. The second thread to enter performs the calculation and continues climbing up. To implement such scheme, we introduce an atomic variable in internal-node object. Each thread would do an atomic compare and set operation on the atomic variable to decide whether it should calculate bounding box or terminate.

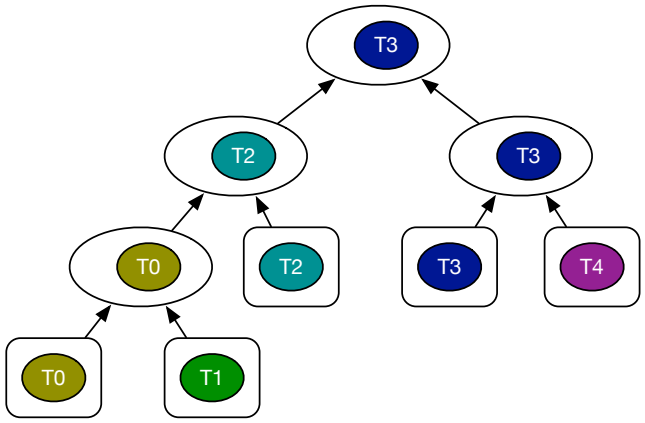


Fig. 3: Bounding box calculation: two threads trying to enter a internal-node, the first one terminates; the second one performs bounding box calculation and trying to entering parent node.

C. BVH-Tree Traversal

BVH-tree traversal was the largest component of our ray-tracing work. It was frequently over 50% of the runtime of our frame and over 95% for a majority of our project's lifetime. As such, it received the most optimization focus at both a macro (ray/tree organization) and micro (algorithm) level. This section will focus on the algorithmic optimizations to the treelet traversal code.

We break our full BVH-tree into smaller "treelets" to take advantage of the parallel nature of the GPU [3]. Each of these treelets is sized to fit within the shared memory allocation of a threadBlock. Eye rays are generated in Z-pattern order in order to make sure each thread is performing a similar treelet traversal. Each thread owns a single ray and passes these rays between treelets as the ray traversal takes place. The block of work is organized as a list of treelets and the rays

```
while in treelet
  intersect current node
  if node is leaf
    intersect primitives
```

```
while in treelet
  while inner node
    go to next child
if node is leaf
    intersect primitives
```

that are passing through them. Each threadBlock is responsible for a chunk of the full list of in-flight rays. Our algorithm borrows "stackless" traversal concepts from Afra [2] to reduce memory resource requirements. We eschew a full tree stack, which would allow more efficient tree traversal, in favor of a light weight stack vector. This reduction in resources allows us to process more rays in parallel and maximize the thread-level parallelism of the GPU.

Our initial implementation of the treelet traversal algorithm was very naive. Each thread operated on a single node in lock-step before proceeding to the following node. Our code structure came from CPU programming roots and looked like the following:

This algorithm structure traversed the treelet as shown in Figure 4 (a). Because each thread was executed in lock-step, the hard work (primitive intersection) was almost never executed in parallel. As you will see in our results, its very rare for threads within a warp to execute leaf-node traversal at the same points of the treelet. Analysis of the code execution in NVPROF revealed a lot of divergence within our warps. leaf-node traversals are the longest latency operation in the treelet search process. This is because the primitives are stored in global memory and there is no way to reliably store them in shared memory or improve the spatial locality of accessing them. Ensuring that leaf-node traversal happens in parallel is paramount to treelet search performance.

In order to reduce warp divergence, we implemented the while-while algorithm structure [2]. The goal here is to ensure that every thread has a chance to arrive at a point in the treelet where primitive intersection is done. This allows all threads that have primitive intersection work to do them in parallel. As you can see in Figure 4 (b), this method allows the longer latency work to be parallelized while the easier (inner node intersection) work is allowed to be executed out of sync across the threads. Our algorithm performs all node-traversal within a single while loop as illustrated in the pseudo code below.

Restructuring the code in this way fixed our warp divergence issue. However, warp occupancy was still extremely low (max occupancy ¡ 50%). This was because we were using so many register resources that we could not fit the maximum number of thread blocks into a core. A lot of work was done to reduce register usage to try to get more thread blocks to fit within a core. We tried using volatile and __restrict__ keywords, code optimizations and the -maxrregcount compiler option. While we did improve register usage slightly (from 63

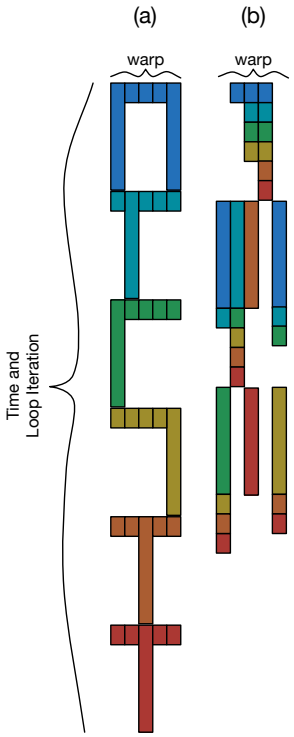


Fig. 4: Divergence Reduction

per thread to 52), it was not enough. Additionally, most of our efforts, especially the -maxrregcount option, dramatically increased register spills. When data cannot fit in register space, the compiler extends the register space with "local" memory space. This space is local in the sense that each thread has its own area, however, the data is treated as if it were stored in global memory. Register spills increase cache contention and global memory traffic, hurting overall performance. The treelet traversal and triangle intersection algorithm is simply too large to reduce. In the end, global memory usage increases nullified any gains from register usage reduction.

Instead, we took a different approach. We focused on reducing global memory usage via register and shared memory usage. In addition to storing the current treelet, we also stored any leaf-nodes and treelet roots that would be necessary to the traversal of the current treelet in shared memory. Even though this tripled our shared memory usage, it also eliminated the need for global memory pointers throughout the bulk of our treelet search algorithm. With the bulk of our global memory accesses happening at the start and end of the treelet search, our performance increased dramatically. We were able to amortize long-latency global memory accesses with larger amounts of computation work. This was the final optimization that allowed us to hit frame-rates that approach "realtime" status.

D. Load Balancing

Due to the incoherency of reflection and refraction rays and the non-uniform execution of ray traversal, we used the concept of treelets for load balancing. Treelets are constructed

in a simple, parallel manner. Each treelet effectively has a root node which can be determined independently using the depth of the node and the amount of nodes below it. Basically, root nodes are statically defined by a depth divisor, where every Nth depth is a root node. Additionally, in order to avoid small treelets, a minimum treelet size is specified which allows smaller treelets to be merged into the level above them. These treelets are used to complete the ray traversal process in a bulk-synchronous manner, with load-balancing and ray generation at each step. During traversal, each ray flags which treelet they will visit next in the ray treelet queue. Compared to Timo and Tero [3], we implement work queues using parallel primitives such as radix sort and exclusive scan. In current GPUs it is difficult to do the fine-grain scheduling without architectural support for efficient queue management. Thus rays are traversed until they hit a treelet-boundary, where they are then sorted and put into new treelet queues. The queuing process consists of (see Figure 5):

1) Ray treelet queue tagging (which treelet the ray is going to next)
2) Hash ray treelets by direction and origin
3) Sorting ray treelets and a ray mapping
4) Flagging queue boundaries
5) Exclusive scan of flags
6) Creation of work queues (treelet ID, starting index, length)

These work queues not only define ray-treelet assignment, but also the new generation of eye rays. It is important to note that each queue has a fixed size which is equal to the amount of threads in a thread-block. Thus, there are some inefficiencies since thread-blocks must have a constant size, but for most of the execution, queues will be saturated with rays. Since the traversal kernel is launched with the exact amount of queues each time, the only loss of utilization (in terms of launching) is leftover room in queue for a thread-block. In this case, the thread-block size can be set small enough so the losses are minimal.

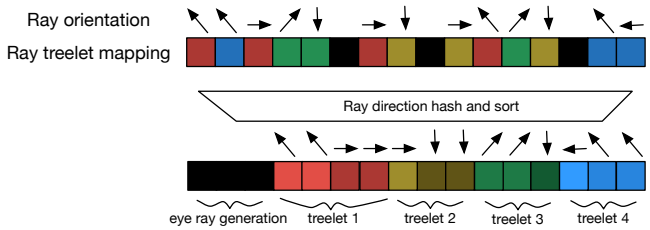


Fig. 5: Queuing process

Since we are doing a sort of treelets, we can take advantage of the sort to also group rays within a treelet by origin and direction. Similar to the approach taken by Garanzha [1], we create a hash based upon the 3D origin and direction of the ray. The hash is then shifted on the beginning of the treelet ID and then removed at the creation of the work queues. This technique aims to minimize divergence within a warp since the rays being processed are approximately in the same position and direction. Hashing is primarily for reflection and refraction rays; however, it can also prevent shadow and eye rays from drifting during sorting.

E. Secondary Rays and Shading

We handle secondary ray generation in a way that controls the non-deterministic problem of scattering rays and allows us to maximize the utilization of the ray queue. Each time a ray bounces, its ability to generate secondary rays is dependent on deterministic factors like ray bounce depth or light source count and non-deterministic factors like material properties. We wanted to avoid tracing rays that would not contribute to the overall image. The naive method is to create all reflection, refraction and shadow rays when a ray intersects an object and ignore ray intersections with no visible color contribution. This is inefficient. An alternative method is only generate relevant rays. However, determining this when an intersection occurs and allocating an appropriate number of rays is difficult when thousands of threads are attempting to allocate the same shared ray queue resource at the same time.

Our solution involves a ray stack that generates one secondary ray at a time and allows for a sufficiently large number of ray bounces. This makes the problem of secondary ray generation deterministic and also allows us to cull irrelevant rays. Each secondary ray is generated in place in the ray queue and a ray does not relinquish its ray queue entry until every secondary rays has terminated. This approach also has the added benefit of avoiding atomic operations on pixel blending operations.

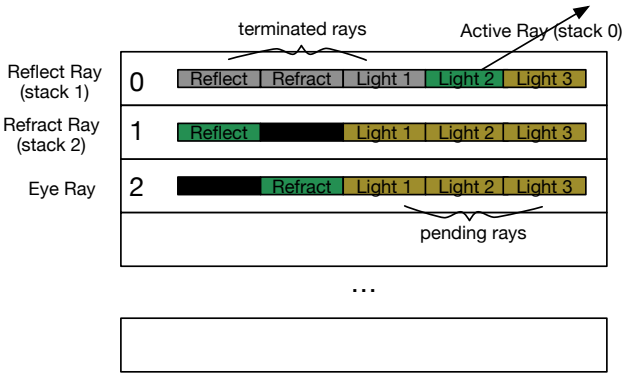


Fig. 6: Ray Stack

We applied the warp balancing concepts explored in the BVH traversal code to our ray generation function. The slightly cheaper algorithm to calculate secondary ray contribution is computed up-front for every possible secondary ray. Secondary ray generation depends on the material of the intersected triangle. A bit-vector is kept that tracks the number of secondary rays that this intersection has to produce. This bit-vector is used to determine which secondary rays go through the more expensive ray generation calculations. Secondary ray traversal occurs in a depth-first order as illustrated in Figure 6. Greyed out entries represent cleared bit-vector bits for secondary rays that have completed their traversals. The green entries represent the ray traversal path. The yellow entries represent rays that are awaiting processing.

IV. Results

The final raytracer was implemented in CUDA and tested on a Nvidia GTX-970. Most of the results were recorded in a single render; however we do support a realtime rendering option. In general the performance of the raytracer is very scene dependent. A scene with less reflections, refractions, and light sources will have less emitted rays and thus better (realtime) performance. We evaluate the performance across multiple test scenes (from the graphics community) which are described in Table I. Overall, depending on the scene and parameters, we can achieve realtime framerates (15 - 20 Fps); however, in cases of complex geometry and secondary rays the framerates drop (0 - 4 Fps).

Scene	Vertices	Triangles	Nodes	Treelets
Cessna	3.745K	3.897K	0	0
Bunny	2.503K	4.968K	0	0
Dragon	50K	100K	0	0
Cornell	8.461K	7.088K	0	0
Sponza	121.7K	40.2K	0	0

TABLE I: Test Scenes

A. Kernel Execution

The overall kernel execution across our test scenes is shown in Figure 8. Parsing is simply the time taken to parse the OBJ file which defines the scene geometry and texture mappings (CPU code). BVH construction is very important to realtime performance, mainly if the scene is changed. We can see that the construction method is very parallel and scales well across increasing scene complexity. As expected, the tree traversal dominates the overall execution time. This is ideal since most time should be spent traversing and rendering the image. However, the time spent in the queuing kernel should be minimal although in many cases it is nearly equal or greater to the traversal time. One reason for this is the long-tail of execution, as less rays are traversed and less time is spent in the traversal kernel, the overhead queuing becomes more noticeable. Another reason is the fact that depending on the ray buffer size, large amounts of data need to be sorted and parsed. Stream compaction is a possible addition to improve sorting and processing speed (discussed in future work).

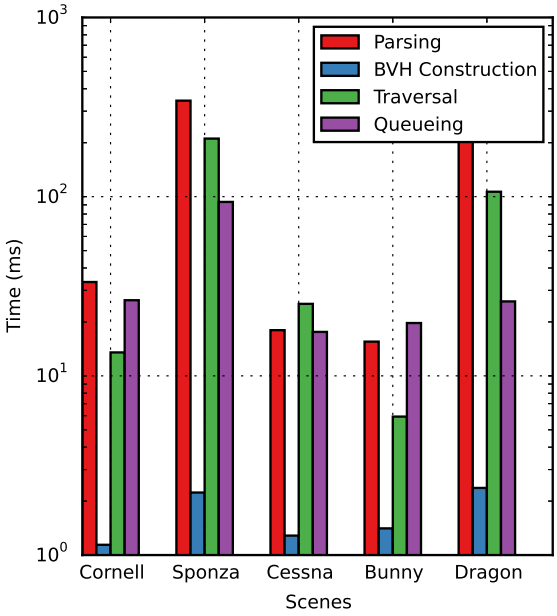


Fig. 8: Execution Breakdown

B. Tunable Parameters

Many aspects of our raytracer are tunable and configurable. These include recursive depth, ray batch-size, and treelet sizing. Recursive depth indicates the maximum amount of "bounces" for each ray which exponentially increases the amount of rays traversed (depending on the material parameters in the scene). Additionally, as discussed earlier, these rays cause further divergence due to their random nature. Figure 9 shows execution times for increasing maximum depth. Some scenes have little reflective/refractive materials; however, dragon is completely refractive. Thus we can see that our queuing and traversal kernels scale fairly well with refraction and reflection rays.

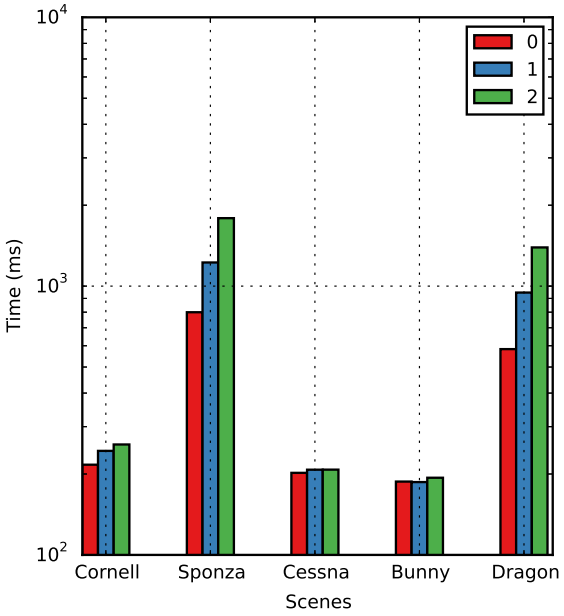


Fig. 9: Recurse Depth

In addition, the ray batch-size was modulated which is shown in Figure 10. The ray batch-size is size of the buffer which contains each ray being traversed. Intuitively, a larger buffer leads to more parallel work (more thread-blocks) within the traversal kernel. However, we can see that the performance of some scenes actually suffer from a larger buffer. Again, this is due to the long-tail effect where the rest of the buffer is currently not utilized for secondary ray generation. Thus, there is wasted space as well as increased processing time in the queuing kernel.

(a) Cessna (b) Dragon (c) Sponza

Fig. 7: Selected Renders

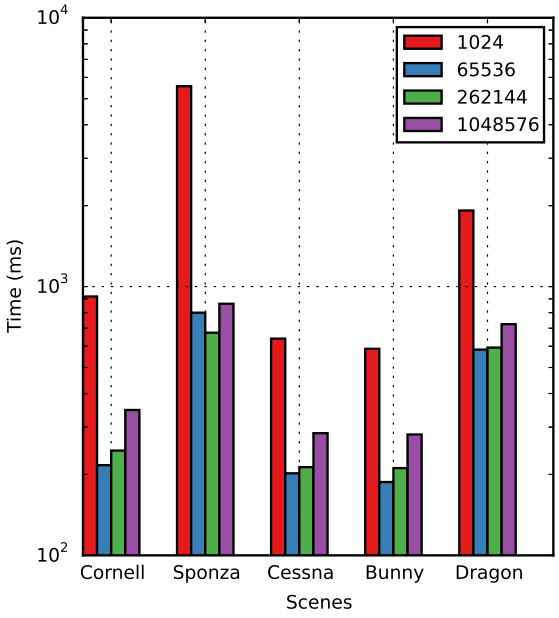


Fig. 10: Varying Ray Batch-Size (rays)

C. Ray Hashing and Ordering

We came up with a metric called warp occupancy to measure how balanced the treelet search work is in our kernel. Warp occupancy is measured as the number of executed leaf-node traversals divided by the number of leaf-node traversals possible if every thread performed as many as the thread that performed the most in that warp. We measured warp occupancy across two of our optimizations that aimed to improve thread balancing: ray sorting and Z-pattern ray generation.

As expected, the results show that generating eye rays in a Z-pattern greatly improves warp occupancy. This is because the rays can traverse together, leading to more similar tree traversal profiles. This improvement in occupancy directly translates to roughly 5% improved performance. Surprisingly, the results for ray-sorting are not promising. Warp occupancy actually decreased. We think this could be due to having an insufficient number of bits to hash ray orientation. Also, the scenes we are rendering may not have enough secondary rays to warrant sorting the rays by orientation.

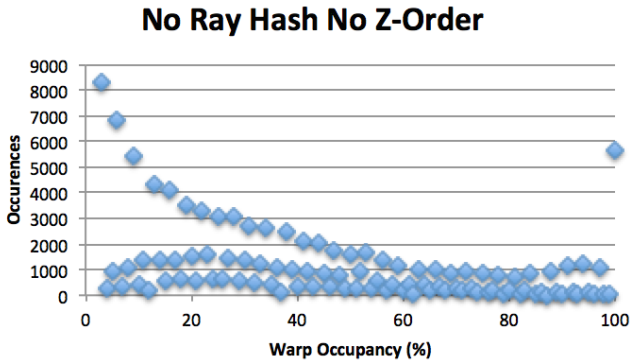
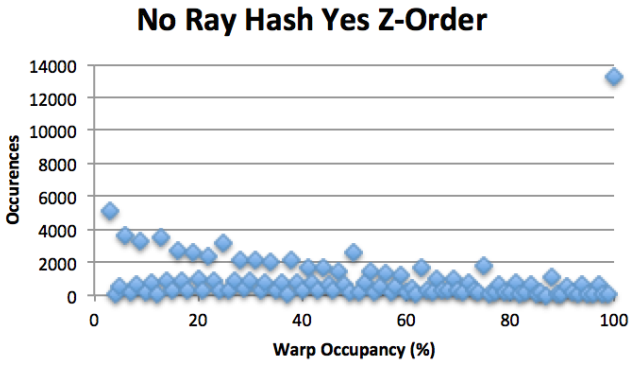
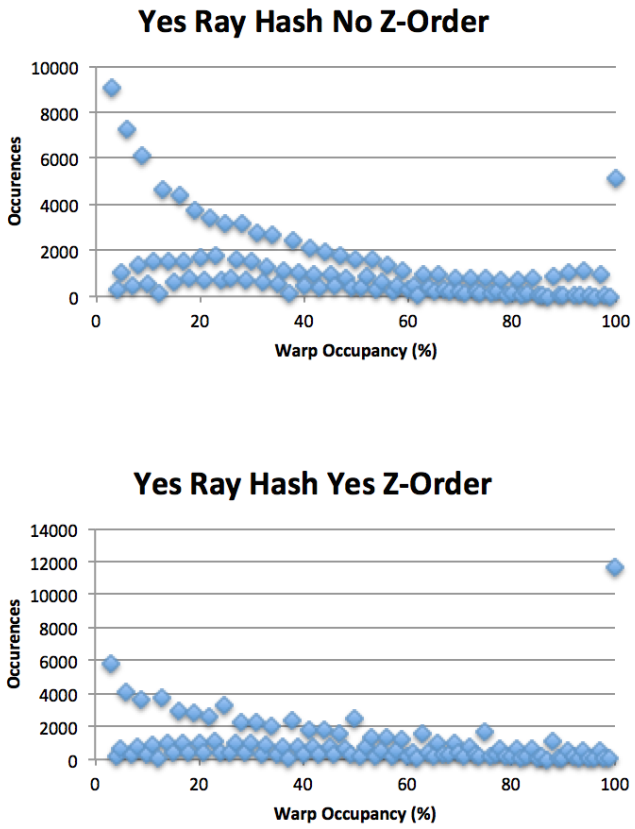


Fig. 11: Varying Ray Batch-Size (rays)



V. Debug Techniques

Working on personal machines with much weak (read NVidia GT-750M) graphics cards introduced limitations to the debug techniques available to us. During the course of the project we discovered a few methods around these limitations.



The capability of cuda kernels to print to the screen was an extremely helpful debug tool. We conditioned print statements to track a single ray execution through every kernel call. We also use print statements to investigate why some thread indexes behaved inconsistently. Lastly, print statements were used to track the status of ray queues and stacks during the execution of our raytracer.

A lot of our work was done on a graphics card that was also responsible for driving a local display. Systems place an internal time limit on cuda kernels that run on graphics cards that drive the display. This timeout is often hit when cuda-memcheck is used and sometimes even when debug flags are turned on. To get around this, we used cuda-memcheck to narrow down to the erroneous function as much as possible. Sometimes all we get is a kernel name. Then we used print statements conditioned on possible error conditions such as index out of range, loop iteration count exceeding some threshold, etc. We discovered that a race condition exists between cuda kernel print statements and cuda kernel exception detection. The kernel would exit with an exception before any print statements could be executed. Thread synchronization could not reliably cause the print statements to appear. We suspect this is because we frequently have print statements inside conditional blocks, causing inconsistent synchronization behavior. Another possibility is that threads deliver the print statement and pass the synchronization barrier much faster than a print statement can be processed. To get around this issue, we repeat the print statement multiple times when we have a strong suspicion on an erroneous line of code. This forces the system to show us the condition of the program before a kernel exception. This debug technique allowed us to close many of the nagging memory issues in our complex code.

VI. Future Work

Our analysis showed that there are large imbalances in the number of leaf-nodes checked between the threads of a single warp. Frequently, a small number of threads will do large numbers of leaf-node traversals while the other threads complete the treelet traversal without visiting any leaf-nodes. Since leaf-nodes are the longest latency computation, it would be valuable to rebalance the work across the idle threads in the warp. This functionality is facilitated by the __shfl functions in cuda. In order to avoid atomic writes, the result of the intersections will need to be stored and gathered by a single thread at the end of the treelet traversal function. We believe this will do a lot to rebalance the low warp occupancy numbers seen in our results.

A side-effect of our secondary ray generation stack is the long tail that appears at the end of our frames. Because each ray is confined to a single ray queue entry for the duration of its lifetime through the scene, it cannot take advantage of freed resources we run out of new eye rays to generate. This tail can be reduced by allowing secondary rays to be generated into empty ray queue entries at the end of the frame. This change will introduce inter-thread dependencies. To avoid atomic operations, the ray stack should track the completion of each secondary ray entry independently. This will make the secondary ray generation vector a multi-bit vector to track the state of each secondary ray. The original master ray queue entry will only perform pixel blending once all secondary rays have completed. This can be asynchronously passed between threads by a producer-consumer relationship between secondary ray threads and the master ray thread.

Furthermore, there is room for improvement in our scheduling and queuing kernels. Currently, the sorting and processing overhead can be prohibitive for some scenes and the end of execution. Thus, a method similar to stream compaction as discussed in [1] could be implemented which would decrease the amount of data sorted and scanned. Additionally, the sort space should be decreased towards the end of the execution (unless it is filled with secondary rays). There is also a need for more experimentation with treelet construction and sizing. Generating quality treelets is key for improved performance with secondary rays, thus more heuristic-based methods may perform better. Also, treelet sizes could possibly change dynamically depending on the scene or the stage in traversal (e.g. merging treelets).

VII. Conclusion

In this paper, we explored current methods of achieving realtime raytracing. We implemented the final raytracer in CUDA and demonstrated nearly realtime performance for certain scenes. However, as in results from past research, these scenes are more simplistic and contain less secondary

rays. Further work needs to be done to analyze our current implementation, not only for GPU utilization and divergence but also the effectiveness of our structures and methods. While we performed some initial utilization and occupancy analysis with Nvidia's tools and improved shared memory usage, there is still potential for kernel improvement (in terms of access stride, bank conflicts, register usage, and more). These problems are still active sources of research, especially with the increasing popularity of raytracing in the graphics community. While rasterization will still continue to be used for most realtime graphics, raytracing is becoming a more realistic alternative. Probably the most interesting option are hybrid solutions which either use rasterization for some part of the raytracing process (i.e. casting eye rays) or utilize special raytracing hardware. Such processes and devices (PowerVR) already exist but need further improvement in order for widespread adoption. As more raytracing techniques are accelerated, we will begin to see increasingly complex visual effects in the realtime graphics space.

REFERENCES

[1] Garanzha, K.; Loop, C., "Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing," Eurographics, 2010, vol. 29, no. 2, pp. 289-298, May 2010.

[2] Afra, A.; Szirmay-Kalos, L., "Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing," Computer Graphics Forum, 2013, vol. 0, pp. 1-11, 2013.

[3] Timo, A.; Tero, K., "Architecture Considerations for Tracing Incoherent Rays," High Performance Graphics (2010), pp. 113-122, 2010.

[4] Lauterbach, Christian, et al. "Fast BVH construction on GPUs." Computer Graphics Forum. Vol. 28. No. 2. Blackwell Publishing Ltd, 2009.

[5] Karras, Tero. "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees." Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics. Eurographics Association, 2012.