GPU Accelerated Ray-Tracing on NVIDIA Hardware: Using object-based KD-Trees to accelerate image generation

Giuseppe DiNatale[1]
Schuyler Kylstra[1]
[1]University of North Carolina at Chapel Hill, Computer Science. Chapel Hill, NC, 27599

This paper presents a summary of our work in the development of a GPU enabled ray-tracer written in CUDA for the fall 2014 UNC Comp 770 class. We will cover our development of the GPU pipeline as well as our normalized representation of object based KD trees. We were interested in this topic because generating film quality frame rates using a ray-tracer is an open problem in computer graphics research. Furthermore, we were interested in learning the basics of CUDA GPU programming. Specifically, we wanted to address the problem "What constraints need to be included to develop a high speed graphics engine via ray-tracing?" To address this problem, we developed a framework for the scene environment where there is a upper bound on the number of objects in the scene and each unique-object is described via a local *normalized* KD-tree. We managed to implement a CUDA ray-tracer with an interactive geometry that renders close to one frame every few seconds.

Index Terms—**Graphics, GPU, CUDA, Ray-Tracing, KD-tree, NVIDIA**

I. Introduction

As it stands right now, there are no gaming systems that use a ray-tracer as the bedrock of their graphics engine. All of the available systems instead use a rasterizer to visualize the scene along with a handful of useful hacks to simulate real world lighting effects. Shadows, reflections, and more are all tacked on to image generation as a side effect. Because of this, the quality of a rasterized image is limited and of lower quality than a ray-traced image. However, they greatly outperform modern ray-tracers in the time it takes to generate a single image. This is because each object in an scene is constrained to occupy a small piece of the screen where as a ray-tracer needs to evaluate the every pixel against potentially every object. To combat this data structures are employed that contain reflect the spatial layout of the scene. These structures are deployed to minimize the number of triangles that have to be investigated. The canonical example of such a data structure is called a kd-tree. A kd-tree breaks up all of space into disjoint regions. These volumes are then grouped into larger regions to form a tree structure. Each leaf node has a collection of triangles that are at least partially contained in its interior. By utilizing these data structures a ray-tracer dramatically improves its performance.

However, problems arise when you are interested in dynamic scenes. If there is any change in the scene there is a good chance that the KD tree is no longer correct. Rapidly updating the KD-tree to maintain utility is a major open problem in graphics research today. In this paper we will not try to tackle this problem. Instead we will try to avoid it entirely. Tree upkeep is especially important when there are a large number of spatially independent objects. This means there is no overall structure to the scene and no assumptions about location can be used to construct the KD-tree. The high number of independent objects makes it impossible to consider all the objects in parallel. In this paper we focus on the scenes where this is not the case - scenes with a limited number of independent objects.

Organization: The rest of this paper is organized into the following sections:

1) *Related Work* - We discuss the work that informed our project and the cutting edge of ray-tracing.
2) *Methods* - This section contains our main contributions and will be broken down further.
   a) *Scene Construction* - We describe our heuristics for the organization of scene data.
   b) *Normalized KD-Tree* - We explain the features and implementation of our object centric data structure.
   c) *CUDA Pipeline* - We discuss how we take our formulation and implement it on a NVIDIA GPU.
3) *Results* - We report on what we implement, the quality of our images, and the frame rates we achieve.
4) *Conclusion* and Future Work - We explain the implications of our work and propose avenues for future development.

II. Related Work

Ray-tracing has a long history in computer graphics, dating back to the 1980's at the latest [6]–[9]. These renderers were known to produce high quality images but were limited by the capabilities of the time period. As scenes became more complicated, researchers began developing data structures to space and minimize the number of ray-object intersection tests that were needed. This led to the development of a number of space partitioning techniques. The canonical example is the kd-tree which is a 3D binary search tree over a closed volume. KD-trees and similar data structures produced far more efficient rendering algorithms but are still too slow for many real-time applications. Developing methods further

accelerate image rendering is an ongoing field of study. In recent years, ray-tracer performance has improved with the utilization of GPU technology to enable massively parallel computation [1], [4], [5], [13], [14].
One of the methods utilized to accelerate tree traversal on a GPU is back-tracking [5]. Back-tracking is a simple algorithm that has a significant impact on the number of nodes visited during tree traversal. Each time we reach a leaf node of a tree and don't produce an intersection we return to the most recent ancestor of that leaf that we have not fully explored. In our project we implement a compressed version of the algorithm covered in the 2006 Foley paper. In our case we limit the calculations to a single GPU kernal. We believe this choice ended up being detrimental to the efficiency of our renderer.

The main limitation of modern ray tracers is that the acceleration structures do not handle dynamic scenes effectively. In fact, a standard kd-tree of low granularity used for a dynamic scene would need to be reconstructed for almost all movements. This leads to the development of efficient and parallelized methods for tree reconstruction [2], [3], [10]–[12], [15]. However, we are going to try to completely avoid this necessity by utilizing object-hierarchies. This technique dates back to the 80's as a way to limit the number of ray-triangle intersection calculations required. It achieves this by performing intersection calculations on the space occupied by and object before investigating the triangles that compose those object [9]. If a ray does not pass through the spcae occupied by an object then there is no way the ray interacts with that object.

III. Methods

In this section we explain our contributions. The following are the definitions for the terminology we will be using in this paper:

- *Mesh* - The triangular representation of an *object*.
- *Ray* - The mathematical representation of vision, light, and shadow in a 3D scene.
- *Object* - A static geometry constructed from a *mesh*. Represented by its own *normalized kd-tree*.
- *Normalized KD-Tree* - A data structure that used to describe the spatial layout of a mesh and to efficiently evaluate a ray-object intersection.
- *Character* - A collection of pairwise spatially dependent *objects*. The set of possible locations for two objects in a character are representable in a finite volume. Two different *characters* are spatially independent.
- *Atomic Character* - A character composed form a single object.
- *Scene* - The total set of *characters* that are represented at any given time.

For this paper a building or a landscape could be considered a character. This information could conceivably be used to accelerate collision detection but that is not the focus of this paper.

A. Scene Construction

We wanted to enable the construction of dynamic scenes but were trying to avoid the costly process of tree reconstruction. As a scene evolves, the triangular meshes move relative to each other. In the worst case, every single triangle could move independently. An example of this chaotic geometry would be the shattering of a dish or glass. Trying to implement this shattering as a feature would require a tree reconstruction as the scene geometry at time t_0 would be completely different from the geometry at time t_f. However, when you look at the state of the art in game engine's this feature is never - or at the most rarely - included. In fact, much of the geometry rendered in modern computer games adheres to a high degree of interdependence.
Consider the game Halo as an example. In that game there are incredibly sophisticated environments that the player can explore. However, much of the structure of the environment is locked in place - the mountains do not shatter as you run through them. The number of completely independent objects can be relatively low for a very detailed scene. Using this intuition, we decided to describe the geometry of the scene based on the independent object. Each character relative geometry of a collection of objects and each object is described with its own kd-tree. One requirement is that the triangle mesh for a given object is *static* allowing us to pre-compute every kd-tree.

B. Normalized KD-Tree

When we treat each object as an independent geometry we can scale each mesh independently within our scene. Furthermore, it becomes possible to present a scene with two separate instance of the same object but at vastly different scales. This distinction between scene geometry and model geometry implies that the absolute positions of points in a model are not important but rather the locations of the points relative to the rest of the model. We reject the notion of absolute space!

To take advantage of this freedom, every object's bounding box is the unit cube. Internally, this causes the object geometry to be scaled but by recording relative scales for $\hat{x}$, $\hat{y}$, and $\hat{z}$ the scene view of the object is unchanged. Containing the object geometry in the unit-cube allows us to:

1) Represent an object's geometry using fixed point encoding.
2) Scale objects independently in the scene space.

3) Convert from scene space to object space very easily.

The advantage of using fixed point numbers in our objects is that the math is computationally cheaper so we can conceivably achieve better frame rates than by using floating point arithmetic.

Each object in the scene is now represented by a point and three orthogonal vectors of various lengths. These represent the axes of a given object's frame of reference. The bounding box described by these vectors is the scene space bounding box of the object and *is not necessarily a unit-cube*.

Each kd-tree is contained in a bounding volume. Ray bounding volume intersection can be evaluated using the Möller-Trumbore algorithm with a minor amendment. Change $\alpha + \beta \leq 1$ to $\alpha + \beta \leq 2$ and both are less than or equal to 1.

C. CUDA Pipeline

In order to implement our ray tracer, we had to break down the process of ray tracing into stages which could be implemented as individual CUDA kernels. In order to remove the overhead of memory allocations, all buffers for various data structures (i.e. rays, scene objects, lights, etc.) are preallocated on the device. The only data transferred from device to host is the final frame buffer at the end of the pipeline. The stages of the pipeline are listed and described in order below. Below, let W the width and H be the height in pixels of our image. Let L denote the number of lights in our scene and let N denote the number of objects in our scene.

Ray Generation

As the first step, a ray must be generated for each pixel of the viewing plane. The camera is a perspective camera, therefore each ray generated will start from the origin of the camera and will pass though the center of a pixel. When calling the ray generation kernel, we will have CUDA partition our data into W blocks with each block having H threads. The block index and thread index will denote the x and y locations of our pixels respectively. The output of this kernel is a 1 dimensional array which is WxH in length that will contain all rays generated.

Object Intersection

Once rays are generated per pixel, object intersection must be performed. Each ray must be tested with every object in the scene to determine if an intersection occurs. Intersection detection is accomplished by traversing a pre-generated kd tree for each unique scene object. We implemented a stack based kd traversal algorithm since CUDA does not handle recursion very well. The object intersection kernel expects a W x H block array with N threads per block and will perform an intersection test for every ray object combination. The output of this kernel is a W x H x N array of intersection data. The intersection data contains the time value of the intersection, a pointer to the object with which the ray collided, the index of the triangle of the mesh that the ray intersected, and a point representing the point on the surface of the intersecting triangle.

Intersection Reduction

The next step of the process is to determine the closest intersection, if there was one, for each ray. Our intersection reduction algorithm is a basic stride based reduction. The intersection reduction kernel will accept the full intersection data from the previous kernel and will reduce the set of collisions for each object by finding the minimum time value for each intersection of a particular ray. The kernel expects a W x H block grid which will have N number of threads to perform the reduction for each block. The output is an array which is WxH in length which contains the data for the closest intersection.

Ambient Processing

Now that the closest intersection has been determined, the ambient intensity per pixel can be determined. Using the object pointer contained within our intersection data, we can determine the surface properties of the object and assign each pixel the ambient intensity of the surface. Also, the calculation of the surface point occurs here since that will be useful for the following steps of the pipeline. The inputs for this stage of the pipeline are the rays and the reduced intersection data. The output of the stage is the intensity buffer with ambient color set for the pixels with rays that intersected with an object. The intensity buffer is W x H in size. The kernel is expecting W blocks with H threads per block with each block thread pair performing the ambient processing for a single ray.

Shadow Processing

Using the surface points calculated from the prior stage, we can now determine if the surface points are in shadow. This will require rays to be shot from the surface point to every light in the scene. If the "light rays" intersect with an object before intersecting with the light source, the surface point is in shadow. The shadow processing kernel expects a 2D, W x H block grid with LxN threads per block. Each thread in the block is responsible for calculating the light ray between the surface point and its assigned light and determining the ray intersects with its assigned object. Intersection is again determined by using our stack based kd tree traversal. The shadow processing kernel expects a 2D, W x H, block array with LxN threads for each block. The kernel produces a W x H x L array of booleans which indicates if the light ray was obstructed for the given light and camera ray.

Color Processing

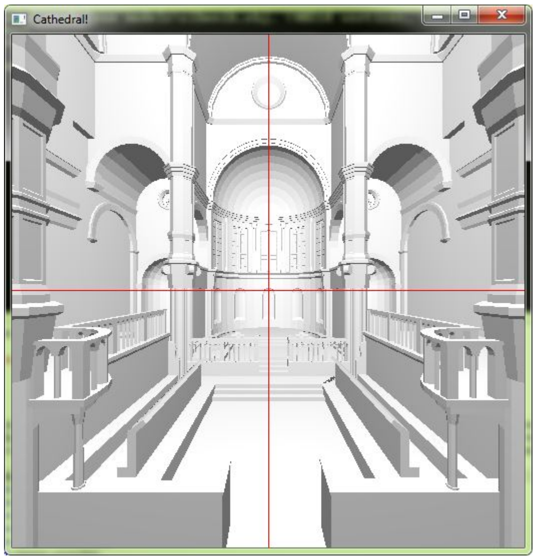
Once we determine which lights are obstructed for each surface point, we can add each unobstructed light's contribution to the color of the pixel in the viewing plane. This portion of the pipeline is another stride reduction to sum all light contributions. This is where the lighting model is implemented. For our ray tracer, we simply implemented the diffuse portion of the Phong Shading model. This kernel expects a 2D, W x H, block gride with L threads per block. The output of this stage is the total intensity per pixel.

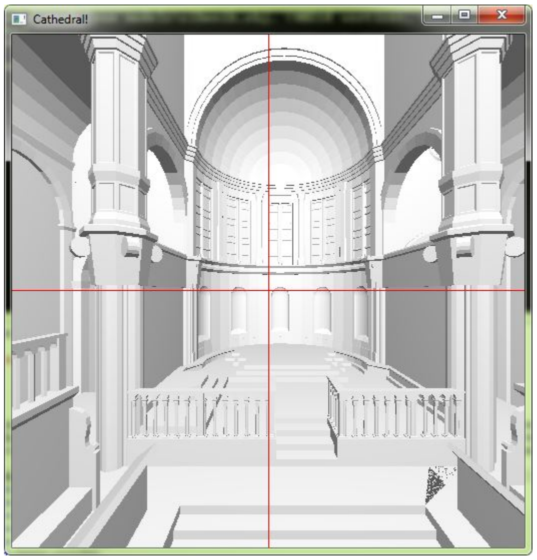
Gamma Correction

Finally, gamma correction must be performed. This kernel simply applies the basic gamma correction with a gamma value of 2.2. The kernel expects a 2D, W x H, block grid with one thread per block. The output of the kernel is simply the final frame buffer to be used to display on the monitor.

IV. Results

Given the time constraints, we were unable to implement the object based scene structure. However, we did produce a working CUDA ray-tracer using a stack based traversal and backtracking. We only generated one scene base on a precomputed KD-tree of the Sibenik Cathedral benchmark. The cathedral is the only object in our scene and is static. Additionally, there is a single light source in the scene located at the origin. These are not necessary constraints imposed in our generator but rather a product of time to include various features in our scene. It would be simple to add additional lights of various colors. Our scene is dynamic in that the location of the camera can change and is controllable by a user. Images generated by our system are displayed below:





As you can see, we still have some bugs but it is generally functioning. Overall, we can generate a new frame every 2 - 3 seconds. However, we only tested on a machine that had a single GTX 780 which was pushing both the display ouput and performing CUDA operations. This means our program was competing with screen image generation and every other image producing process. Problems porting our code meant we only were able to test on a single machine but we would expect a dramatic improvement in render time when we are not competing with other processes.

Overall, our code has the capability to represent *i*) multiple objects, *ii*) multiple light sources, *iii*) ambient coloring, *iv*) diffuse coloring, *v*) camera motion, and *vi*) gamma correction.

V. Conclusion and Future work

When we chose to do this project we did not appreciate the scope of the problem we were tackling. Our goals were to implement a ray tracer with very advanced scene characteristics in a framework that neither of us had worked in before. The devils of parallel programming quickly appeared and limited our implementation. We failed to render in real time but achieved much higher frame-rates than when rendering on the CPU alone.

In the future we would change multiple aspects about our implementation. First, it is important that we simplify our kernal functions. After a review of the literature, the papers we examined spent much more energy reducing the size of their individual kernals. For instance, the 2005 Foley paper from Stanford uses as many kernals to calculate ray intersection as we use in our entire pipeline.

Additionally, we were never able to explore the possibilities of incorporating normalized kd-trees. Any benefits from this approach remain unknown at this time.

References

[1] Carr, Nathan A., et al. ”Fast GPU ray tracing of dynamic meshes using geometry images.” Proceedings of Graphics Interface 2006. Canadian Information Processing Society, 2006. APA

[2] Havran, Vlastimil, Robert Herzog, and H-P. Seidel. ”On the fast construction of spatial hierarchies for ray tracing.” Interactive Ray Tracing 2006, IEEE Symposium on. IEEE, 2006.

[3] Hunt, Warren, William R. Mark, and Gordon Stoll. ”Fast kd-tree construction with an adaptive error-bounded heuristic.” Interactive Ray Tracing 2006, IEEE Symposium on. IEEE, 2006.

[4] Huss, Niklas. ”Real Time Ray Tracing.” (2004).

[5] Foley, Tim, and Jeremy Sugerman. ”KD-tree acceleration structures for a GPU raytracer.” Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. ACM, 2005.

[6] Fujimoto, Akira, Takayuki Tanaka, and Kansei Iwata. ”Arts: Accelerated ray-tracing system.” Computer Graphics and Applications, IEEE 6.4 (1986): 16-26.

[7] Glassner, Andrew S., ed. An introduction to ray tracing. Morgan Kaufmann, 1989.

[8] Glassner, Andrew S. ”Space subdivision for fast ray tracing.” Tutorial: computer graphics; image synthesis. Computer Science Press, Inc., 1988.

[9] Goldsmith, Jeffrey, and John Salmon. ”Automatic creation of object hierarchies for ray tracing.” Computer Graphics and Applications, IEEE 7.5 (1987): 14-20.

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

[11] Nah, JaeHo, and Dinesh Manocha. ”SATO: Surface Area Traversal Order for Shadow Ray Tracing.” Computer Graphics Forum. 2014.

[12] Popov, Stefan, et al. ”Experiences with streaming construction of SAH KD-trees.” Interactive Ray Tracing 2006, IEEE Symposium on. IEEE, 2006.

[13] Popov, Stefan, et al. ”Stackless KDTree Traversal for High Performance GPU Ray Tracing.” Computer Graphics Forum. Vol. 26. No. 3. Blackwell Publishing Ltd, 2007.

[14] Singh, Jag Mohan, and P. J. Narayanan. ”Real-time ray tracing of implicit surfaces on the GPU.” Visualization and Computer Graphics, IEEE Transactions on 16.2 (2010): 261-272.

[15] Wald, Ingo. ”On fast construction of SAH-based bounding volume hierarchies.” Interactive Ray Tracing, 2007. RT’07. IEEE Symposium on. IEEE, 2007.