Real Time Cuda Ray Tracing

Liam Tyler
tyler147@umn.edu

1. INTRODUCTION

1.1 Motivation

Ray tracing is a technique for rendering images that is often highly desired in computer graphics. The technique revolves around trying to realistically simulate how light travels through a scene. This inherently allows it to achieve a much higher degree of realism than rendering with traditional rasterization. Effects such as reflection, refraction, and soft shadows are difficult to simulate with rasterization. However, these effects happen naturally through ray tracing, however, making it easy to accomplish. The ease of implementing these effects and the high quality results have motivated a lot of research to be done in speeding up ray tracing.

1.2 Cost of Ray Tracing

Ray tracing is not usually used because of its computational cost. Since rays can reflect and refract off each surface, each pixel of the image can require dozens or even hundreds of rays to be traced through the scene. For a 1080p image, there are over two million pixels. This means that there can be hundreds of millions of rays needed to produce the whole image. In addition to that, each ray needs to be tested against all of the geometry in the scene to see what it hits, which can be an expensive operation when there are millions of shapes in a scene. To shoot all of these hundreds of millions of rays and test each one against millions of shapes is extremely difficult to do in less than 16 milliseconds, which is the standard benchmark for interactive games. This is the primary reason why video games do not use ray tracing today. With GPUs getting faster, this is slowly starting to change.

1.3 Parallelism

Ray tracing is often called "embarrassingly parallel." This is because there are millions of pixels and each pixel's color can be calculated independently. These combine to make ray tracing a perfect problem for the GPU to handle. To parallelize this, launch a 2D grid of threads that covers the image plane and assign each pixel to one thread. This potential speedup enables a lot of new goals that were not realistic with just a CPU.

1.4 Goals

There were six specific goals for this project:

1. Render each frame in less than 16ms (60 FPS)
2. Have up to 1 million triangles and spheres at once
3. Have Blinn-Phong shading including reflection and refraction
4. Support shadows
5. Support static and dynamic objects
6. Handle 1000 objects in a single scene

Of these six goals, the first five were accomplished. The reason for the last one not being supported is explained in Section 3.2.3.

2. DESIGN OVERVIEW

2.1 General Process

Ray tracing always follows the general process seen in Figure 1 below.

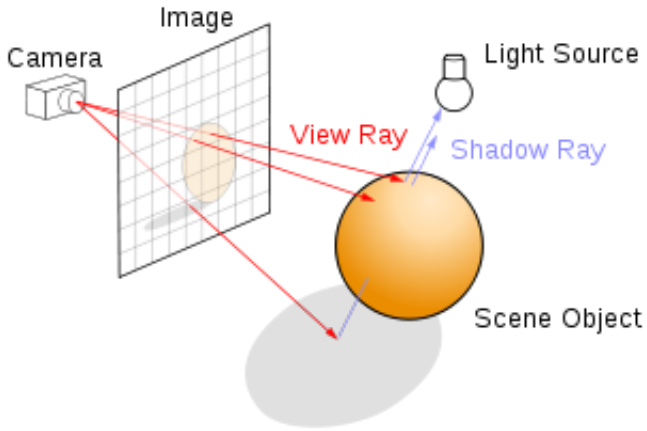

Figure 1. Illustration of the ray tracing process

The ray tracing starts by shooting one ray for each pixel on the screen. All rays start at the camera's position, but to calculate the ray directions an imaginary image plane is constructed in front of the camera. This image plane will have the same resolution as the screen being rendered to, such as 1920 x 1080. Using the camera's forward vector, up vector, and field of view, the positions of every pixel on the image plane can be calculated. The ray directions are then found by subtracting the camera position from all of the pixel positions. Each ray is shot into the scene and tested to see if it hits an object. If no object intersection occurs, then set the color of the pixel to be the background color of the scene. Otherwise, if the ray does hit an object, then the color needs to be calculated using the object's material properties. This can be used in the Blinn-Phong shading model to compute the color of the object at that specific point and viewing direction. If the object is reflective, then the reflective ray must be computed and traced through the scene as well. Whatever color object that ray hits will be added onto the pixel's final color using the reflectance model again. The same thing applies to the refracted rays for objects like glass. This is a naturally recursive process, since the reflected and refracted rays can also hit objects that reflect or refract. For higher quality images, the rays can be allowed reflect and refract many times, but the user typically sets some max depth. Once all of the rays have been traced to their maximum depth and the final colors calculated, the image will be complete.

2.2 Project Design

2.2.1 Setup

There were three main areas of design, starting with the setup. Before a scene is rendered, all of the objects must be loaded from

disk and formatted for the GPU. Figures 2 and 3 describe this process.

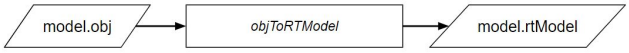

Figure 2. Shows the model conversion that is run on all OBJs

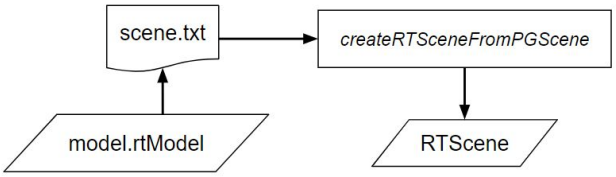

Figure 3. The scene file and models get loaded and converted to a GPU ready scene format

The first issue is that 3D models often come in OBJ format. OBJ files are plain text, and need to be processed and converted to the format the ray tracer is expecting. This is done offline using a C++ executable I created as seen in Figure 2. Then a scene file is created which describes what objects, lights, cameras, and models are present in the scene and where they are located. I previously created a C++ graphics library called "Progression" that parses scene files into an object called "PGScene". I extended that code to parse a few extra pieces of information specific to the ray tracer, and the rtModels. Once the ray tracer application starts, it uses this process to parse the scene files. Upon completion, the PGScene is converted into an "RTScene" object, which has the specific data layout that the ray tracer needs, as seen in Figure 3.

2.2.2 Ray Tracing

The overall design of the Cuda ray tracer very closely matches the process described in Section 2.1. In particular, there are four main functions that can be seen in Figure 4 below.

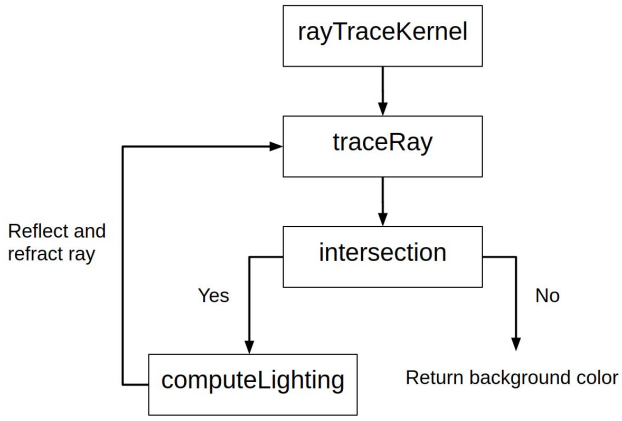

Figure 4. The flowchart of function calls in the ray tracer

The *rayTraceKernel* function calculates the image plane and assigns each thread one pixel to process. The *traceRay* function is what actually shoots the ray through the scene and calls *intersection* to see which object was hit, and it calls *computeLighting* to calculate that object's color. It then computes the reflected and refracted rays to be traced as well. In terms of data handling, each function has access to the RTScene object described in Section 2.2.1. Once the kernel is complete, every pixel color is finalized, and the image is ready to be displayed.

2.2.3 User Interaction

In order for this to be a real time application, the camera and objects have to be updated each time step, the updated scene has to be ray traced, and the result has to be displayed on the screen for the user. This process is illustrated in Figure 5.

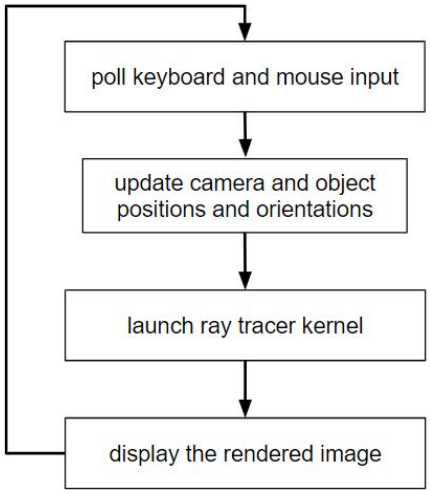

Figure 5. Flowchart of the main tasks each frame

Cuda doesn't handle any of the windowing, keyboard and mouse input, or drawing images. To accomplish these tasks, I used the same library, "Progression," described in Section 2.2.1. This creates a window to draw on, and allows the user to query input events. To display the image, I used OpenGL. The Cuda kernel writes to an OpenGL texture, which is then drawn on the screen once the kernel is finished. The rest of this paper will focus on the implementation and results of the ray tracing aspect of this project.

3. IMPLEMENTATION

There were several key areas of implementation that had to be considered for functionality as well as performance. These are described in the following subsections.

3.1 Eliminating Recursion

One of the problems that quickly arose was the recursive nature of the *traceRay* function. GPUs with a compute capability of 2.0 and higher do support recursion, but there is a limited amount of stack space for each thread. The recursive approach was working when the max recursion depth was very low and only reflections were supported. As soon as the max depth was increased and refractions were added, the threads would run out of stack space. In order to keep the stack space bounded, I had to switch to an iterative approach.

To make this process iterative, a queue of rays was used. Instead of making a recursive call to *traceRay* for the reflected and refracted rays, the rays are added to the queue. Along with the ray, the current depth must also be saved, as well as the current color multiplier. The color multiplier comes from the lighting equation. An object's color is computed using the following equation:

finalColor = *object.color* +

*object.reflectivity * traceRay(reflectedRay)* +

*object.transmissivity * traceRay(refracedRay)*

With the iterative approach, the first ray's multiplier is initialized to one. For each successive ray, its multiplier equals the current ray's multiplier, multiplied by the object's reflectivity or transmissivity accordingly. This allows the rays in the queue to be processed in any order. The following code in Listing 1 is the direct code that processes the queue inside of the *rayTraceKernel* function.

```
RayQ Q;
Q.push(item);
float3 color = make_float3(0, 0, 0);

while (Q.pop(item)) {
    color += traceRay(Q, item, scene);
}
```

Listing 1. Code snippet for how rays are processed

As long as the queue is large enough, this allows for both reflection and refraction for an arbitrary max depth. The next major implementation detail was in the intersection tests.

3.2 Bounding Volume Hierarchy

3.2.1 Problem

The first approach I chose for doing ray shape intersection tests was to loop over all of the objects in the scene, and test the ray against each one. This was an O(n) process. It worked fine at first, but many models have a very large number of triangles. The Stanford Happy Buddha statue is 1.09 million triangles. With the O(n) approach, this took over 12 seconds to render a single frame. Since the goal is to render a frame in under 16 ms, a speedup of several orders of magnitude was needed. One common method to achieve this is to use a spatial data structure.

3.2.2 Concept

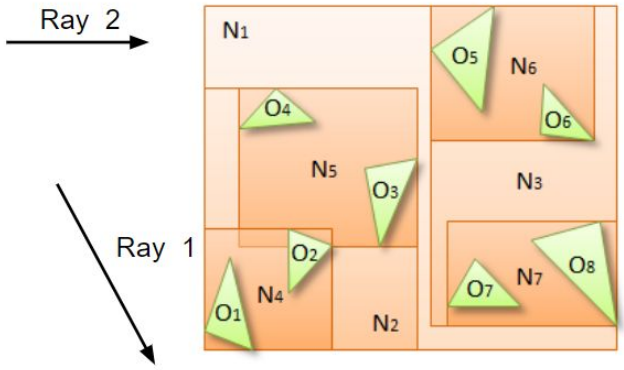

Figure 6. A visualization of a BVH and two example rays

The spatial data structure I chose to use was a bounding volume hierarchy (BVH). An illustration of a BVH can be seen above in Figure 6. The idea is to first encompass all of the triangles in a bounding box (N1 above). Then, split the box into two smaller boxes that each have a portion of the triangles (N2 and N3 above). This repeats until each box only has a small number of triangles in it. The reason this is a massive speedup can be seen by walking through the two example rays in Figure 6. For Ray 1, we test it against the bounding box N1. Since it doesn't hit the box at all, it is impossible for it to hit any of the triangles inside it, and we don't have to test anything further. This saves seven intersection tests. For Ray 2, it would be tested against N1, then N2 and N3, then N6 and N7, then finally the triangles O5 and O6. This saves one intersection test. With a large number of triangles, this is on average an O(log(n)) number of intersection tests, which gives very large speedups. In order to use a BVH, I had to implement BVH creation for models.

3.2.3 Construction and Traversal

When I first implemented BVH construction, I used the following algorithm:

1. Loop over all of the triangles to find the overall bounding box and average centroid
2. If there are two or fewer triangles, it is a leaf node in the tree, so save the list of triangles in that node
3. Otherwise, designate the longest axis of the bounding box to be the splitting axis
4. Loop through all of the triangles again, placing any that are less than the average centroid value for that axis in the left child, and any that are greater in the right child
5. Recursively partition the left and right children by going back to Step 1 for each child

Since models are not being edited while the program is being run, I do this offline at the same time that the OBJ models are being converted to rtModels, as described in Section 2.2.1. The construction can be done offline since it takes over 20 seconds for the Buddha statue. This was also the reason why I was not able to achieve the last goal I had of 1000 objects in a scene at the same time. To accomplish this, a BVH would need to be constructed for all of the objects each frame, since objects can move around. Real time BVH creation is a topic in and of itself, and not the focus of this project. Once the offline BVH tree is created, it is flattened, the pointers are converted to integer indices of the flattened array, and saved alongside the vertex data for the model in the rtModel format. At the ray tracer startup, it can just be copied straight into the GPU, ready for traversal.

Typically BVH traversal is done through recursion on the CPU. Just like the ray queue, however, we can manage the recursion with a stack of BVH nodes and a while loop. If a ray intersects a BVH node that isn't a leaf, then add that node's children to the stack. If it intersects a leaf node, then loop over the triangles in that node, doing a ray-triangle intersection test for each one. Once the stack is empty, the closest triangle (if any) along that ray will be found. This gave a large speedup as detailed in Section 5, but the choice of splitting along the average centroid was somewhat arbitrary, and it turns out there are better ways to make splits.

3.2.4 Surface Area Heuristic

The goal while constructing a BVH is to choose splits in a way such that the average number of intersection tests needed to be done by any given ray is as low as possible. The splitting method called the "surface area heuristic" (SAH) is an alternative way to choose a split, that turns the split decision into a cost optimization problem according to the following equation:

*splitCost = (left surface area) * (number of left triangles) +*

*(right surface area) * (number of right triangles)*

According to a paper from Nvidia, if you choose the split that has the lowest cost using that equation, then a more optimal BVH will

be formed [1]. Instead of splitting along the centroid now, we loop over many possible values for the split for all axes, not just the longest one. At the end, choose the best split found (which could be not splitting at all) and make that node a leaf node.

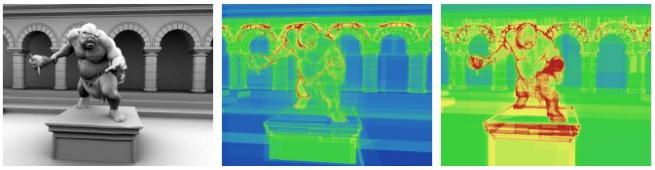

Figure 7. Heatmaps describing the number of tests needed. Scene left, SAH heatmap middle, centroid heatmap right

The results of this technique were illustrated in the same Nvidia paper with Figure 7 above. On the left is the original scene, and the middle and right images are heat maps showing how many intersection tests that ray had to perform. On the right is the heatmap for the centroid method, and in the middle is using SAH. The rendering with SAH very clearly did not have to perform as many ray intersection tests. This directly translates into improved performance, since intersection tests are the main bottleneck of this application. In addition to reducing the number of intersection tests, I tried to implement optimized ray-shape intersection functions.

3.3 Ray Shape Intersection Tests

I wrote several versions of the intersection tests for axis aligned bounding boxes (AABB) and triangles. The goal for both of them was to reduce divergence by eliminating if statements where possible, and reducing the number of floating point operations needed.

3.3.1 Ray AABB Test

The original ray AABB intersection test code had several if statements in it. Some of those were due to the fact that at some point, the ray's direction is used as a divisor, which can be zero for some axes. To avoid the division by zero, an if statement was used. It turns out that the if statement is unnecessary. As described in Tavian Barnes' Ray AABB article [2], doing comparisons with the resulting infinities still produces the correct results. In addition to that, Cuda functions 'fminf' and 'fmaxf' are implemented in hardware, so it is faster to use those than a corresponding if statement.

```
float tmin = (min.x - p.x) * invDir.x;
float tmax = (max.x - p.x) * invDir.x;
float tmp;
if (tmin > tmax) {
    tmp = tmax;
    tmax = tmin;
    tmin = tmp;
}
```

➡

```
float tx1 = (min.x - p.x)*invDir.x;
float tx2 = (max.x - p.x)*invDir.x;

float tmin = fminf(tx1, tx2);
float tmax = fmaxf(tx1, tx2);
```

Listing 2. Code snippet using the min and max functions

Using these two changes, all of the if statements in the test can be removed as seen in Listing 2, so the test is branchless. A noticeable performance increase resulted from this.

3.3.2 Ray Triangle Test

The original ray triangle intersection test algorithm I used worked, but was not the most computationally efficient algorithm. There are several other algorithms for this that are more efficient. The one that I switched to is called the "Möller-Trumbore" method. I will not describe how the algorithm works here, but it is more computationally efficient, and did result in a performance increase as seen in Section 5.

3.4 Data Representation

Traditionally in computer graphics, model data has a lot of duplicate vertices. This is because neighboring triangles share at least one vertex. Since GPU memory has traditionally been very limited, a method for avoiding this data duplication has been needed. The typical approach taken is illustrated in Figure 8 below.

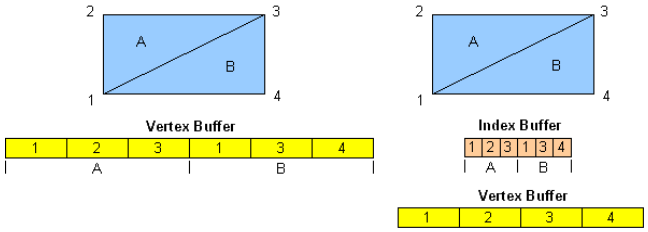

Figure 8. Mesh duplication left, and index buffer right

To avoid duplication, the unique vertices can be listed once, and then an "index buffer" is used to point to the vertices. This saves memory since each triangle is now just three integers, instead of eighteen floats (nine for the vertex information, nine for the normals). This however, introduces a slight performance issue on the GPU.

The unfortunate performance implication of this is that it now takes two global memory accesses to get the vertex data instead of one. Since global memory accesses take hundreds of cycles, this can slow things down. The approach I took was to store the model with the duplication. This meant that models typically used three times as much memory, but only one memory access was required to get the vertex information. This did result in a performance boost.

4. VERIFICATION

One nice aspect of computer graphics is that usually verification can be done just by looking at the output image. The correct shape of the models is known when exporting or downloading the model. As long as we see the correct shape of the model, then the next thing to verify is the pixel colors. This again is typically easy. If the material is red, then we expect the object to be red. If it is reflective, look in the reflected direction, and see if that is what is shown on the reflective object. Verifying refraction is the most difficult. Two common checks for this in the graphics community are to render a magnifying glass or a glass sphere. The first should magnify, and the second should show an inverted image of whatever is behind the sphere. The four main scenes used while verifying the correctness were: a red Buddha statue, a reflective sphere, a refractive sphere, and a teapot with a plane behind to test shadows. Below are the results of these verification scenes.

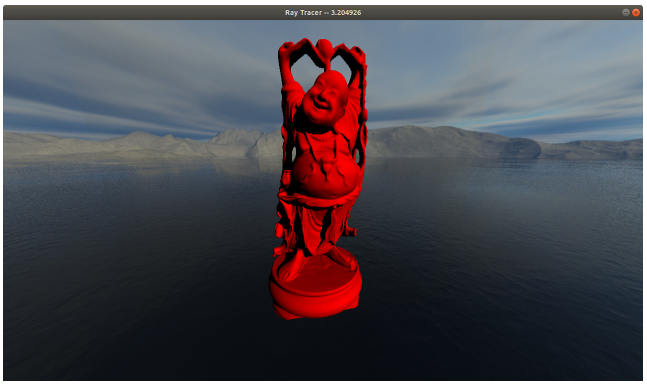

Figure 9. Buddha statue with diffuse red material

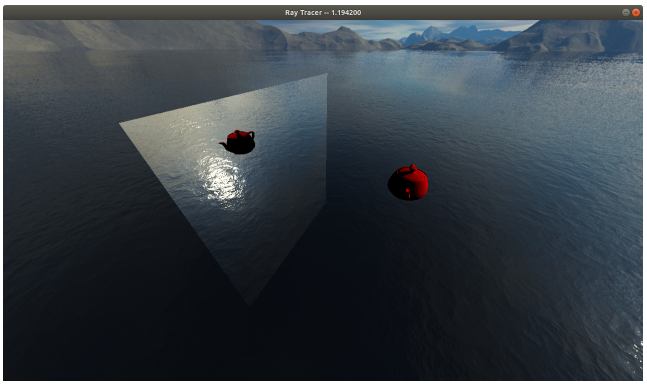

Figure 10. Teapot and environment reflecting on plane

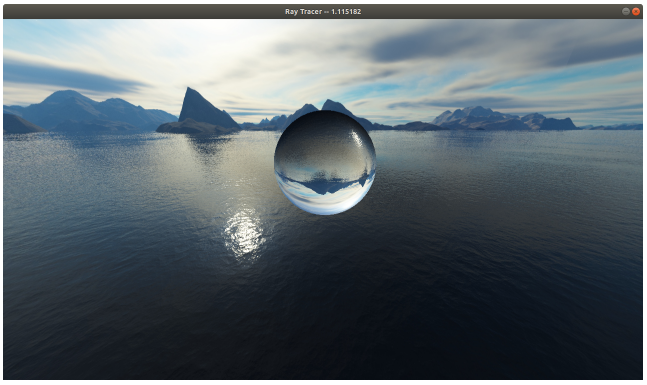

Figure 11. Refractive sphere

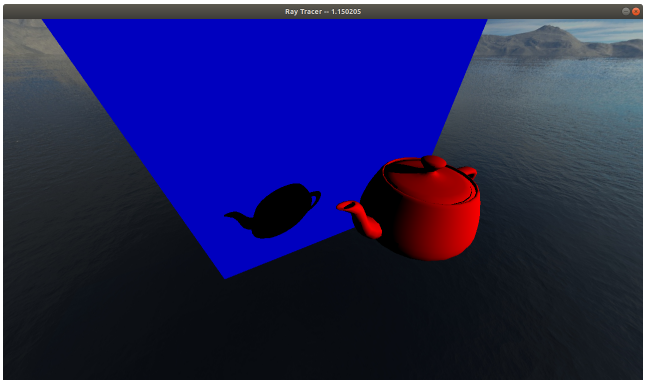

Figure 12. Teapot and environment reflecting on plane

In all of them, we see the expected image, so we can confirm that the ray tracer is working properly. The next step if to see what the performance is.

5. PERFORMANCE

To test the performance of my application, I used the following scene: a model at the origin with a red reflective material, the camera looking straight at it from ten feet away, and one directional light. I timed how long it took to render this scene with the four models seen in Table 1 below.

Table 1. The four models and their triangle counts

Model Name	Triangle Count
Cube	12
Teapot	2,256
Dragon	100,000
Happy Buddha	1,087,451

After each attempted optimization, I ran this benchmark again, and see what the new times were. All tests were run using an RTX 2080. Figure 13 below shows what the original times were, followed by the times after adding in the BVH.

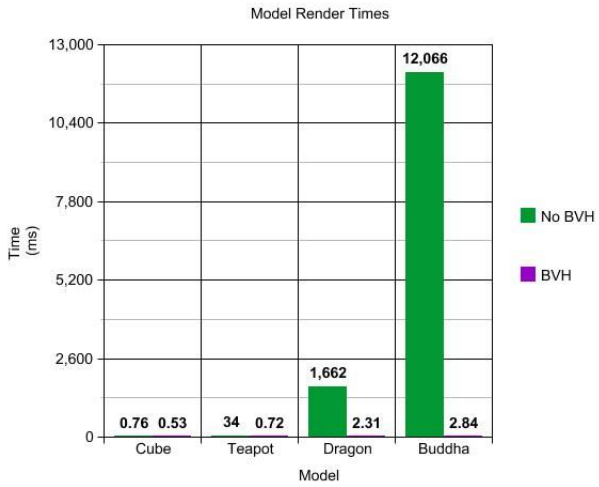

Figure 13. Initial render times vs render times with BVH

This was by far the largest optimization. This alone actually achieved the goals I had, since the Buddha statue was over one million triangles, and rendered in well under 16ms. I also tried over a dozen other optimizations. Some of them did not improve performance, but Figure 14 below shows six of the attempted optimizations.

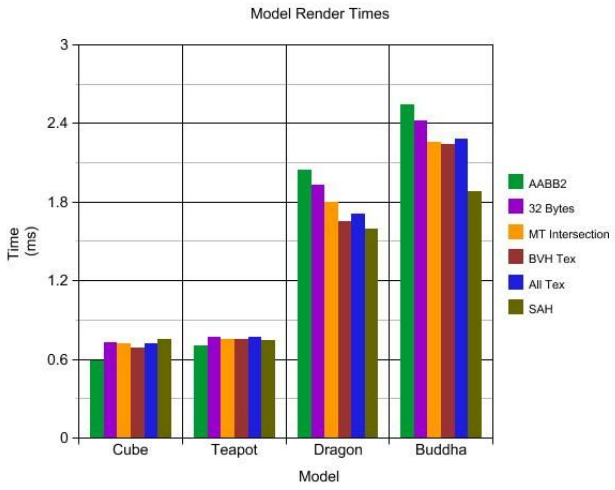

Figure 14. Six further optimizations and their timings

The first (green) was using the branchless AABB intersection test, which had a significant impact on the larger models. The next (purple) was reducing the size of a BVH node from 36 bytes to 32. This allowed more nodes to fit in cache and aligned them better, resulting in a small speedup for the larger models, but reduced performance for the smaller models due to a small overhead cost that was added with this. The third (orange) was using the more computationally efficient ray triangle intersection test, which again had a small speedup. The fourth (reddish-brown) was moving the BVH into textured memory. This had a slight improvement for all models. The fifth (blue) was moving the BVH and triangle data into textured memory, but this had worse performance for all models. The final one (murky green) was using the surface area heuristic for the bounding volume construction, which finally brought the Buddha statue down to 1.88ms.

Overall the performance gained was great. The ability to shoot tens of thousands of rays at the same time with the GPU opens up the doors of real time ray tracing, which was not possible with a CPU. I achieved all of the goals besides the 1000 objects in a scene simultaneously, which I decided not to focus on as described in Section 3.2.3. The main limiting factor is just the inherent computational cost of ray tracing. To shoot hundreds of millions of rays and test each one against millions of shapes is just an extremely large amount of work to do in less than 16 ms. That being said, there were several areas which could be improved upon with more time.

I think that with more development time, I could implement the fast BVH construction to achieve the last goal. In terms of individual model speeds, there were several techniques I did not use, such as fast math, shared memory, or persistent threads. I tried to use shared memory for a couple optimizations, but didn't see a performance increase in them. I am sure there are some areas that could benefit from this technique. Persistent threads is a technique that I didn't try, but has been mentioned to have potential applications in ray tracing. If I used these I could maybe get the larger models closer to 1.5ms or under.

6. CONCLUSION

The paper describes why ray tracing is a desired application and why it is naturally suited for GPUs. It describes the major design decisions I made between processing the models offline, as well as the processes for tracing a scene. The main implementation choices revolved around various optimizations in all parts of the *intersection* function, as well as converting the recursive sections to be iterative. The results are highlighted by rendering the 1.09 million triangle Buddha statue in 1.88ms, when it was initially over 12 seconds. This shows that ray tracing that real time ray tracing is no longer just a dream, but in fact possible with the current generation of GPUs.

7. REFERENCES

[1] Stich, M., Friedrich, H., & Dietrich, A. (2009, August). Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (pp. 7-13). ACM.

[2] Barnes, T. (2011, May). Fast, Branchless Ray/Bounding Box Intersections. https://tavianator.com/fast-branchless-raybounding-box-intersections/