Ray Casting in Parallel: CUDA Optimizations of 3-dimensional Image Rendering

Julia Tucher
Williams College

Abstract

This paper regards a final project implementation of a ray casting algorithm in parallel. I attempt to optimize 3-dimensional image rendering via CUDA optimizations.

1 Introduction

In this final project, I plan to explore and implement an algorithm for ray casting. Ray casting is a means of depicting 3D images in two dimensions, through depicting three-dimensional input as a scene consisting of many, smaller, geometric shapes. The algorithm implements a series of ray-intersection tests where a ray is shot from each pixel in the desired two-dimensional image and is tested for intersection with every geometric shape in the scene. This algorithm solves many problems in 3D image rendering scenarios, including animation, game design graphics, and also computation geometry. It is an early version of a 3D-scene depiction algorithm, and the algorithm has evolved through many optimizations into rasterization algorithms (where instead of iterating over the pixels, the algorithm iterates over the triangles in the scene in order to reduce the region of intersection) and other ray tracing algorithms. Ray casting is used specifically to refer to the fact that the algorithm being implemented does not recursively cast rays. However, it is typically thought of as a high speed algorithm because it can depict reflections, refractions, and soft shadows through texture mapping rather than recursive ray tracing.

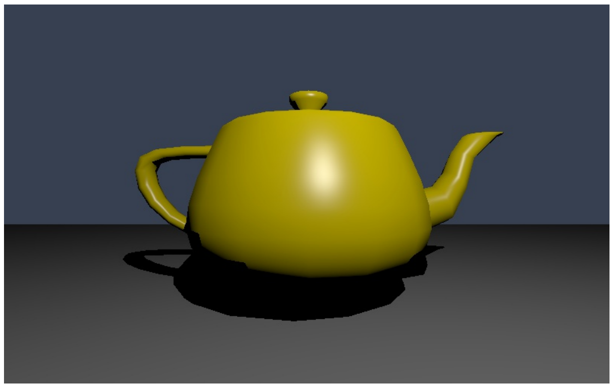

Figure 1: Figure 1. The canonical image of the Utah teapot, as rendered by the serial implementation of this algorithm

After implementing the algorithm in serial, several parallel and nonparallel optimizations were made using NVIDIA's CUDA programming environment to optimize for algorithm runtime. The serial algorithm was initially parallelized by mapping the computation for each pixel onto one thread in a CUDA kernel and various block sizes were compared to find the optimal number of threads per block [2]. Furthermore, CUDA's shared memory was utilized in an attempt to improve runtime. Shared memory was utilized to reduce runtime to minimize the number of calls made to global memory. The main use of memory in this algorithm is for the overall scene depiction and for the vectors used in each pixel's computations. When stored the vectors used in individual thread computations, three strategies were implemented: statically allocating arrays within the kernel, dynamically allocating memory within CUDA global memory, and dynamically allocating a large array for all of the vectors required for computation for each thread executing within a block. In regards to storing the scene data in shared memory, because of the self-iteration processes where each triangle must be compared to every other triangle in the scene, it is required that every triangle be stored in shared memory in order for all computation to occur. Overall, the shared memory optimizations and the parallel organization optimizations were compared within each category to each other and the serial implementation to find the optimal conditions.

2 High-level: 3-dimensional Input

In order to compute an image overall, the amount of light that arrives at each pixel of the image sensor in some eye or camera must be computed. Since all photons cannot be captured, a large number of them can be sampled by casting a ray from each pixel that the viewpoint camera perceives. In this algorithm, geometric rays (with origin point and direction) are traced from the eye of the observer or a camera and samples the light traveling towards the viewpoint from the ray direction, where the value of the light is referred to as its radiance. Note that radiance can later be converted into RGB values. This process is implemented as calculating the path of the ray through a shutter plane (that later becomes the two-dimensional image) and onto the geometric components of the scene to test for intersection. In this algorithm, the geometric components of the scene are exclusively triangles, with three vertices, a vector that is normal to the triangle, and a bidirectional scattering distribution function (BSDF) value which corresponds to the color and texture of each triangle. The scene will be represented in the input as a scene of triangles and a scene of light sources, as described in the next section, and the triangles are derived by an outside artists converting objects in a scene into a geometric mesh such that the surface of each object is completely represented. The smaller the triangles are, the more accurate the curvature of the image will be.

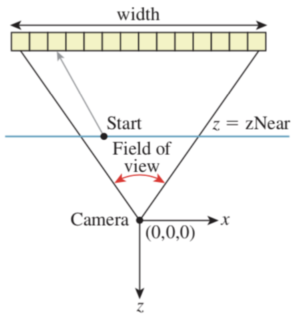

Figure 2: The zNear plane depicts the origin points of the rays (i.e. the plane containing the image pixels), the width is the width of the image, and the field of view (constant for each image) sets the angle of perception.

2.1 Setting the Scene

For this implementation, the orientation of the camera within the setting was kept constant, such that the images (of varying sizes) had their objects scaled, rotated, and shifted such that the input scene would be visible from the angle, position, and distance of the camera. In each image, the camera viewpoint is placed on the positive z-axis (at the point (0, 0, 1)) such that it faces down its negative axis. Thus, the two-dimensional image that is rendered is a section of a plane that is parallel to the xy-plane, with the y-axis increasing out of the top of the image, and the x-axis increasing to the right of the image. The height and width of the image are y- and x-dimensions, respectively, and traveling from left to right, top to bottom, traverse the x-axis negative to positive and the y-axis from positive to negative. This is depicted in Figure 2, although the camera is up one point on the z-axis.

2.2 Building Triangles

The vertices of each triangle are in three-dimensional coordinate space such that its position and angle relative to an incoming ray can be depicted as two vectors. This vector representation of triangles allows for easy computation of dot and cross products to compute the intersection of an incoming ray (which is also represented as a vector. The three normal vectors that are also components of the triangle are each perpendicular to a plane that is parallel to the triangle at that point. For example, if the triangle is contained within a single plane (i.e. the triangle is flat) all three of the normal vectors will be in parallel. In contrast, if any of the normal vectors diverge from one another or direct away from the triangle at different angles, the triangle will be somewhat curved in 3-space. These normal vectors are also referred to as shading normals, which indicates their ability to dictate the shading of the triangle. When combined with different sampling techniques, this results in a matte or glossy effect, respectively. The third component of the triangle, the BSDF value, is combined with the normal vectors for texturing.

In the sample used in collecting data for this paper, two different BSD functions were used to create two types of textures in each scene. The first, which created a matte effect, is called the Lambertian scattering or Lambertian reflectance and does not factor in the normal vector, instead computing a constant refraction (later, color) value for the whole triangle [1]. The other function used in this project for calculate the scatter density of the light upon a triangle was the Blinn-Phong scattering density method, which expands the Lambertian method for color, while using the dot product of the normal vector with the incoming ray to create a scalar for each radiance value. The overall effect of the Blinn-Phong method is the appearance of a glossy surface, where the normal vectors facing the light have a "shinier" appearance than those facing away [1].

2.3 Light Source

The third and final component of the scene that should be considered before discussing the algorithm is the light source of the image. In this algorithm implementation, the light source(s) of the image is represented by a location in three-space and a three-dimensional power vector where each dimension of the vector represents RGB radiance.[1] Light that is detected by the viewpoint's image is detected either because it was directly emitted or because it was scattered by some object within the scene.

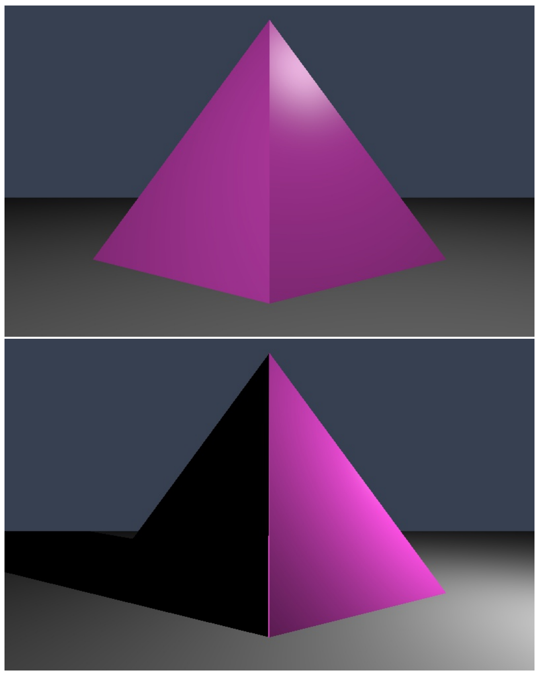

Figure 3: Figure 1. The same pyramid, constructed as a 4-triangle solid on a 2-triangle ground, as illustrated by two different light sources located at (1.0, 3.0, 1.0) (above) and (2.0, 0.0, 0.0) (below)

Figure 3 demonstrates how light bounces off of a scene by illustrating the major differences in reflection and shadow given the position of a light. Points along the ray that intersect with triangles in the scene, and angle one its way to the camera, the point will not be visible because it is cast in shadow. Thus, the algorithm must account for each triangle being cast in shadow by any of the other triangles.

3 Ray Casting Algorithm

Consider first the algorithm in its serial implementation. After filtering the input and storing it in the appropriate data structures (which are detailed in the code itself), the ray casting algorithm is passed an image array which will be filled with radiance values.[2]

[1]This means that different colors of light could be used, and multiple light sources could be placed around the scene at the artist's discretion

[2]Note that these radiance values are later converted to RGB values, but this is done after the image is rendered and when it is being converted to a .JPG image.

The other parameters for the ray caster function include the scene being depicted (again, as an array of triangles and light sources), the camera or eye's viewpoint, and the bounds of the image. The algorithm performs the same computation for each pixel in the image, iterating through all of the combinations of x- and y-values in the image range in typical row-major matrix traversal. First, it calculates the Ray: the origin point is a point in 3-space of the pixel in the image and the direction is a unit vector in the direction the eye is looking when that pixel is being considered. It initializes the distance between the point in the image as being infinity. This way, any triangle that the Ray intersects will be closer than the previous one. Then, for each triangle in the scene, test to see if the Ray intersects the triangle by updating the closest distance to correspond with the value of the closest triangle. Below is the pseudo code for the first section of the algorithm.

```
for each row of pixels y:
for each column of pixels x:
     let R = ray from eye to (x, y)
        distance =
        for each triangle T in scene:
         d = intersect(T, R)
```

Thus, if the current triangle is in fact closer than all previous, find the exact point of intersection through vector computation. Using the light source, the direction of the Ray, and the other triangles in the scene, the algorithm must check whether or not the light scattered from the first triangle is covered by any of the other triangles. This is performed by a helper method as a test of visibility. If the point is never visible, then it will be stored with a radiance of zero, and appear black like a shadow.

```
 if (d < distance):
   distance = d
   radianceSum = 0
   let P = intersection of R and T
   if P is visible:
    radianceSum += radiance w
```

Finally, after iterating through every triangle, the image pixel's radiance is updated based on the total radiance sum for that pixel, where radiance sum is a three-component radiance vector.

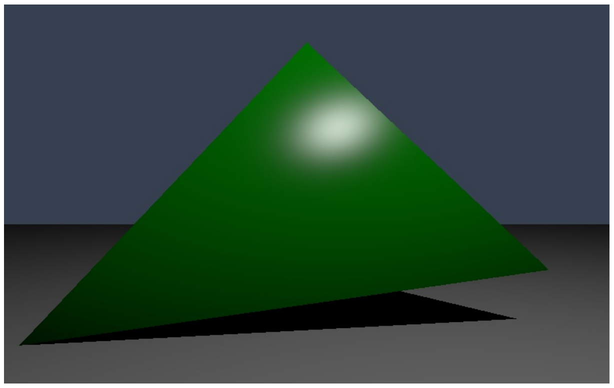
Figure 4: A simple, four triangle image that represents all modes of shading, shadow, and three-dimensionality at the simplest level.

```
image[x, y] = radianceSum
```

Before proceeding with serial and parallel implementation details, let's first consider what it means for a Ray to intersect a triangle and for a point to be visible so that all of the major components of the ray casting algorithm are evident.

3.1 Intersection and Visibility

The intersection of a Ray and a triangle depends on several conditions being false. The Ray should not be in parallel to the triangle, should not intersect the plane of the triangle but not the triangle itself, and it should not glance off the triangle at a non-visible angle. Furthermore, implemented as a helper function throughout this implementation of ray casting, the test of visibility is the most costly of computations in this algorithm because it must iterate through all of the other triangles in this scene to ensure that it is not cast in shadow. The function essentially generates many "shadow rays" from the triangle and tests for intersection with other triangles in the scene. Overall, vector computation in calculation of intersection and visible are the most computationally expensive but are not parallelized in this implementation because of their dependence on pixel-specific values including Ray vectors.

In terms of considering various parallel organization strategies, note that the limitations on the number of threads that can be sent to each Streaming Multiprocessor (SM) has previously been either the number of blocks (8 per SM) or the total number of threads (1536 per SM). However, because the memory demands of this

4 Translation to CUDA Environment

As an image processing algorithm, the ray casting algorithm lends itself to highly parallel computation. As seen in the above high level description of the algorithm, the most obvious area of parallel computation lies in the per-pixel iteration. Most image-based algorithms, including the ray casting algorithm, execute the same computation on each pixel in an image independently of all other pixels. Notice that the vector computations within the calculation for a single pixel's RGB value are all dependent on all of the steps preceding it because each dot and cross product that ultimately leads to a final radiance value is dependent on all of the steps before it. Furthermore, the most expensive operation in the algorithm proves to be the test of visibility, which is also Ray-dependent, meaning per-pixel is the most obvious organization method for parallelization. In addition, note that per-pixel computation is unreliant upon any other pixel for data, and as such, all data for each pixel will be read-only with no syncronization required in the initial parallelization.

Consider the instruction set of calculating a single pixel and the data required for each pixel's calculation as its image coordinates and the scene it is reflecting. As we can see, the ray casting algorithm is well equipped for SIMD architecture of the GPU, and thus exploration of parallel implementation in CUDA is likely to result in large improvements in runtime. CUDA, as a parallel computing platform, allows for GPU programming integrated into the existing C implementation.

4.1 Parallel Optimizations

For the first round of parallel optimizations, map each pixel in the ray casting algorithm to a thread in the CUDA kernel call. This included allocating global memory for all of the data used in the serial implementation, as well as allocating memory for

Block Dimensions	Thread/block	Threads/SM	Limiting factor
2x2	4	32 (8 blocks)	Only 8 blocks per SM
4x4	16	128 (8 blocks)	Only 8 blocks per SM
8x8	64	512 (8 blocks)	Only 8 blocks per SM
16x16	256	1536 (6 blocks)	Only 1536 threads per SM
32x32	1024	1024 (1 block)	Only 1536 threads per SM

Figure 5: In this figure are depicted the predicted limitations of the number of thread that will be sent to each Streaming Multiprocessor accounting for block and thread count limitations.

all of the local vectors used for each thread's computation. In the initial parallelization version, each of the 9 vectors required for each thread's computation were statically allocated within the CUDA kernel. However, after initial testing, it was found that even with a single thread per block, the kernel failed to launch. This is likely due to the limited local memory for each thread. Thus, the naive implementation dynamically allocates global memory for each local variable within the kernel using a call to malloc() from within the kernel.[3]

algorithm are higher than algorithms previously implemented, it is likely that the number of threads that can be sent to each SM will be lower. Thus, consider square block dimensions from 1x1 to 16x16, and increment each dimension by 2 in order to best pinpoint the best organization strategy. Before final testing, estimate the number of threads that will be able to execute on each SM based on the limitations of threads per Streaming Multiprocessor.

However, given the limitations of register memory per thread in a block (63 registers/thread) in CUDA, there may be unexpected limitations in the block dimensions. After some initial testing on small images, it appeared that the likely cap of block dimension would be around 16x16 threads per block, so proceed without the 32x32 block.

4.2 Non-Parallel Optimizations

Some of the possible non-parallel optimizations available to a parallel algorithm running on a SIMD

[3]In the serial implementation, vectors are dynamically allocated as well.

architecture like the GPU include utilizing shared memory and/or caching, memory tiling, and branch divergence. Since each thread requires access to all of the data available in the scene, it does not make sense to optimize based on spatial locality of threads or pixel-triangle combinations. Furthermore, because of the low number of thread that are likely to be sent in each block to an SM, it is likely that trying to reduce branch divergence would not have a very big impact on runtime because there are so few warps executing threads simultaneously. Thus, attempting to group nearby threads based on branch execution will have low impact. There are also not many opportunities to reduce branching in the algorithm because of the nature of each triangle's required comparison to every other triangle in the data.

Given the use of CUDA architecture and an algorithm that relies heavily upon accesses to global memory, in order to further increase runtime, consider utilizing shared memory. Within the CUDA architecture, each thread has direct access to a private register set, and accessing those registers takes around 1 instruction cycle. Furthermore, all of the threads that are executing on the same block (and within the same warp), have access to that same unit of shared memory, which has maximum capacity 48Kb and requires around 5 cycles to read from. The lowest level of memory in the GPU is global memory, which requires around 500 clock cycles to access. The naive implementation of the algorithm in parallel utilizes global memory to store both the triangle and light source data of the scene depiction as well as the vectors specific to each thread (as static allocation caused the kernel to fail to launch). Thus, utilizing shared memory in CUDA is likely to drastically improve runtime, but may limit the complexity of the image that can be processed. For the first optimization using shared memory, store a set of 9 vectors for each thread corresponding to the 9 vectors required for radiance calculation (Ray origin, Ray direction, interpolated normal, incoming Light, scattered Light, radiance sum, Barycentric weights, etc). Because of the limitations of dynamically allocated shared memory in CUDA, this must be represented as a single vector that corresponds to the size of its components. Specifically, its size is passed as an argument in the kernel invocation

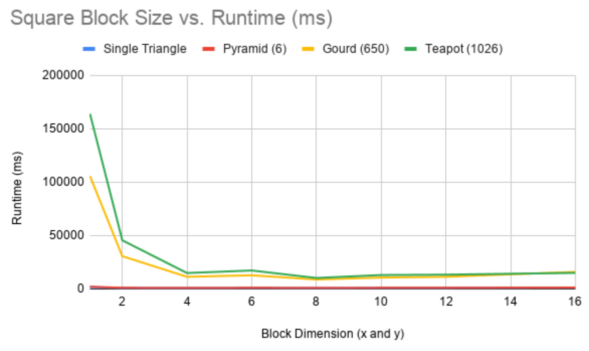

Figure 6: Depicted in this figure are the results of the naive implementation of the parallel version of the algorithm. Testing across 5 different scenes of different sizes, the number of triangles in each scene is indicated in parentheses next to the description of the image. On the x-axis is the square dimension of each block sent to the CUDA kernal, and on the y-axis is the runtime of the kernel.

and scales with the number of threads that are sent to each SM. The following is a sample calculation of the single shared memory array allocated for all threads' required vectors.

```
sharedMemorySize = (number of threads *
    9 * size of vector * sizeof(float))
rayTraceKernel<<<DimGrid, DimBlock,
    sharedMemorySize>>>
```

Then, within the kernel, the offsets of each vector region (and within that, each threads' vector) can be calculated using the size of vectors, block dimensions, and pointer arithmetic.

For the second optimization using shared memory, consider the number of triangles that can be pulled into shared memory by all of the threads. The limitation here is the capacity of shared memory itself, 48Kb. Furthermore, because each pixel-triangle combination must have access to every other triangle in the scene, either all of the triangles must be in shared memory or none of them can. Thus, this optimization will likely increase runtime greatly for simple images with low triangle counts. As above, create a large array as an argument to the kernel invocation, and within the kernel, have each thread contribute to reading in triangles from global

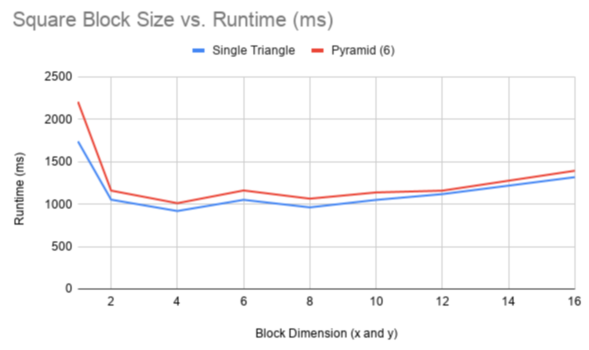

Figure 7: A continuation of results from Figure 4, this image demonstrates how runtime scales among the smaller images of fewer than 8 triangles (Single Triangle and Pyramid.

memory (since the triangle input must first exist in global memory before they can be read into shared memory).

5 Results

Overall, the resulting runtimes indicate that the maximum square block dimension that is feasible for images of <1026 triangles is 16x16 blocks (256 threads/block). Furthermore, in the naive implementation of the parallel algorithm, the optimal square block dimension is 4x4 threads, closely followed by 8x8 threads. As seen in Figure 6, the runtime increases drastically with the increase in block dimension from 1x1 to 4x4.

Figure 7 depicts a zoomed in duplicate of the results in Figure 6 so that the pattern of increased runtime can be seen for smaller images as well. After block dimensions of 8x8, the run time begins to increase, and the kernel launch caps at around 16x16 threads, depending on the size of the image.

Figure 8 demonstrates that both shared memory optimizations did increase runtime eventually, with the first optimization (pixel vectors) performing better than the second optimization (triangle data). However, once again, at higher thread counts per block, both optimizations perform about the same as the naive implementation. It should be noted that run time data was collected for all five of the original test images, but all images larger than 100

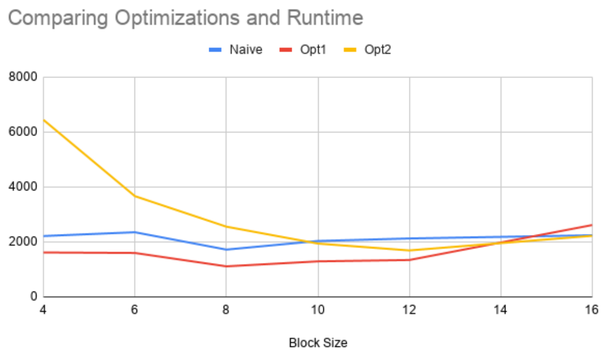

Figure 8: Depicted in this figure are the results from the first and second round of shared memory optimizations. For simplicity, only one of the five images tested is shown in the graph.

triangles had kernels that failed to launch for the second optimization. Of the three implementations, the best run time occurred with 8x8 block dimensions with the first optimization (up from 4x4 in the naive implementation). Furthermore, the second round of optimizations utilizing shared memory increased run time for low block dimensions.

6 Discussion

In conclusion, it is clear that the memory limitations of programming in CUDA make ray casting impractical for large and complex images that consist of many triangles. This implementation, however, seems practical and fast for simple, large scale images. The naive implementation's run time scaled with both image complexity and block dimension as expected, with the fastest dimension being 4x4 threads per block. It is likely that as block dimensions increased beyond what the total register memory of each block could handle, threads were executed serially within the same SM.

When comparing the two shared memory optimizations, it is interesting to note that the second optimization actually increased run time at small block dimensions. This is likely because when there were very few threads per block, those few threads had to iterate over every triangle at the very beginning of the kernel call and the operation of reading in triangle data from global memory was not

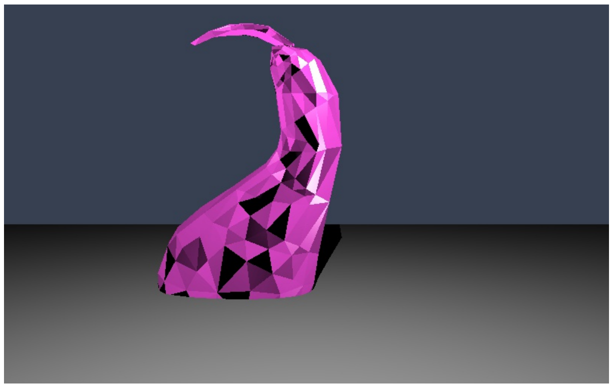

Figure 9: This image indicates that the success of the algorithm is reliant upon the creative ability of the artist designing the input, and despite having a reliable ray casting algorithm, the quality of the input is essential to the final image.

parallelized. Furthermore, having to read in data to shared memory slowed down the second optimization in comparison with the first optimization because the first optimization had no extra computational steps, it just provided threads somewhere in shared memory to store vector values.

Overall, the limitations of shared memory in CUDA meant that one optimization was implemented in exchange for the other. In order to have room in shared memory for the scene's triangle data, the threads' individual vectors had to once again be stored in global memory (and vice versa when storing pixel vectors in shared memory). So, the optimizations could not be compounded to have a larger, combined effect.

7 Next Steps and Takeaways

The most obvious next step for this algorithm is the transition from ray casting to rasterization. Instead of iterating over pixels, a rasterization algorithm iterates over the triangles in a scene, which allows for early elimination of large sections of plausible pixels whose Rays will intersect with the triangle. This largely reduces the number of threads that would be sent to a kernel and reduce runtime, which is why rasterization of computer graphics largely shifted toward rasterization algorithms. In terms of next steps specific to this ray casting algorithm, it would be interesting to further explore the memory limitation of the GPU through a mixed approach. Canonically in ray casting, input data is passed through .OBJ files that store a set of vertices and a set of normal vectors. From these sets, triangles are constructed as a set of three vertex indices and three normal vector indices. This reduces the amount of space required in most cases, and thus it may be interesting to examine storing one set in global memory and the other in shared memory such that calls to global memory are not omitted, but largely reduced.

Because this algorithm is in the domain of image processing algorithms, its per-pixel computation makes it inclined for pixel-to-thread mapping, and thus it seems like a prime example for CUDA programming. However, due to the high memory demands of this algorithm, there is a memory bottleneck when implementing.

When first considering the pros and cons of each programming environment, I initially was deciding between Pthreads and CUDA because having a high amount of data that is all read-only would allow for less memory copying and synchronization involved. However, the shared memory implementations, although useful only on a smaller scale, were very interesting to explore.

References

[1] John F. Hughes Andries van Dam, James D. Foley and Steven K. Feiner. *Computer Graphics: Principles and Practice*. 1.00 edition, 1982.

[2] Wen mei Hwu and David Kirk. *Programming Massively Parallel Processors: A Hands-on Approach*. 1.00 edition, 2010.