Real-Time Ray Tracing using CUDA

Ankit Agarwal*[1] and Prateek Malhotra†[1]

[1]Department of Computer Science, IIIT-Delhi

Abstract

One of the most common problems in Computer Graphics is to generate realistic looking images in real-time. Ray tracing is a well-known technique used to create high-quality images from geometric representations of 3d scenes. However, ray tracing is a computationally expensive technique, and the cost increases as more advanced illumination effects are added.

Graphics hardware architecture is tuned to provide high performance when performing data-parallel tasks. Modern-day GPUs have enormous compute capabilities and these can be harnessed for faster ray tracing. Through use of GPGPU techniques, significant speedups can be made in ray tracing. In this project, we attempt to build a CUDA-based ray tracer that utilises such hardware to perform fast ray tracing.

Citations can be done this way [**?**] or this more concise way [**?**], depending upon the application.

Keywords: ray tracing, cuda, k-d trees

1 Introduction

Ray tracing is a well established technique for generation of highly realistic scenes. Ray tracing is also a computationally intensive task. Image generation using ray tracing consists of 3 basic tasks: ray generation, ray intersection and shading.

1.1 Ray Generation

A camera model is used to define rays that go into the scene. The image plane is kept at a fixed distance from the camera. A ray (or sometime multiple rays) is generated for every pixel on the image plane.

1.2 Ray Intersection

The ray intersection stage is used to find the closest object in the scene that intersects the ray. As scenes are represented geometrically, there are usually thousands or millions of primitives that need to be tested to find the closest object. This is the most costly task in the ray tracing pipeline.

1.3 Shading

After finding the closest object, and subsequently the hit point, the object is shaded using an illumination model. Shading, also, commonly involves shooting secondary rays into the scene for reflective and refractive materials to give more realism to the final image.

*Ankit11020@iiitd.ac.in

†Prateek11077@iiitd.ac.in

2 Parallelism and Ray Tracing

Even though ray tracing has high computational demands, it is highly parallel in nature. The operations discussed in the previous section can be performed independently for each ray. This makes ray tracing a good candidate for being executed on a many-core processor like a GPU.

In our implementation, we assign the task of computing the color of a pixel to one thread. These threads are organized into a 2d block, which are further organized as a 2d grid. The total number of threads in the grid equals the number of pixels in the resultant image.

In the CUDA kernel, we compute the ray corresponding to the pixel for that thread. This ray is then shot into the scene to find the closest hit object. Initially, we used a naive, brute-force method to find the closest hit object. However, after analysing the execution of the kernel, we found that the ray intersection stage required the most time. So, we proceeded to improve our implementation in that aspect.

3 Basic Implementation

In our first attempt, the goal was to have a CUDA-based ray tracer in place. The ray tracer only supported triangle primitives. The ray tracer used the Phong Illumination model for shading, and Phong shading to compute the color at different points on the same surface. It also supported shadowing effects. Support for reflections and refractions was added through use of recursion which is available for CUDA 3.1+ on devices with compute capabilities 2.0 and above.

To read large scenes into memory, we wrote an obj file-format parser. The parser could read vertices, faces and normals to produce triangular meshes that represented the scene.

The CUDA kernel consisted of all three stages of the ray tracing pipeline. Ray creation was done depending on the pixel position in the image. As a 2d block, 2d grid configuration was used, threads mapped nicely onto the pixels.

The scene information was stored as a linear array in the global memory. To find the closest object that intersected the ray, we used a brute-force search on the array of all triangles. Intersection points were computed using Mller-Trumbore algorithm. The equations were solved using Cramer's rule to calculate the intersection point. The closest intersecting object for each ray was then used for shading. When reflective or refractive surfaces were hit, the ray was traced further to give the final result. Recursive tracing procedures were used for these secondary rays.

3.1 Performance Analysis

We used a publicly available Batman model that had approximately 17,000 triangles to test our implementation. To test secondary rays, this batman model was put inside a reflective box that was open

only from one side. A *1920x1080* resolution of the rendering can be seen in Figure 1.

Figure 1: *Batman model inside a reflective box*

The kernel took 165 seconds to render the results. We proceeded to profile the kernel execution in order to improve its performance. Firstly, we found that computing the ray-triangle intersections was the costliest operation in our kernel and took more than 90% of the entire thread execution time. This helped us identify the key area that we need to address in order to make our ray tracer faster.

Firstly, we recorded the time taken by each thread to complete its own execution. This allowed us to prepare a load distribution graph for the entire kernel. The graph can be seen in Figure 2. Red(higher) bars represent threads that took more time than others to complete their execution. As is evident from the graph, the load is very unevenly distributed across the threads. This can lead to significant performance drops due to stalling of threads in a warp.

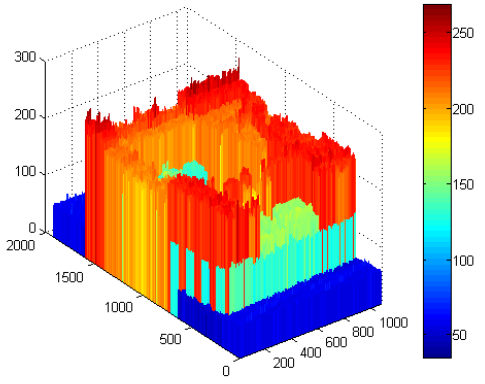

Figure 2: *Load distribution across the grid*

We explored ways to improve on both fronts. To improve on the ray-triangle intersection tests, we came across various acceleration data structures that can be used to minimise on the number of intersection tests performed. This can lead to speedups of up to 2 orders of magnitude. To address the issue of uneven load distribution, we looked up various load balancing schemes. These usually involve using work queues to hold all the tasks in a single, global queue. These tasks are fetched by worker threads. This way, threads do not stall and the load is better distributed. These can lead to speedups of up to 2-3x and are difficult to implement on the GPU where synchronization impacts their performance heavily.

4 K-d tree Implementation

To improve our ray tracer's performance, we decided to use spatial data structures that can bring down the number of intersection tests required to find the closest object by a significant number. After looking up various data structures, analysing their potential gains and considering the relative ease of implementation, we decided to use K-d trees to speed up our ray tracer.

K-d trees are a space-partitioning data structure. They are a special case of Binary Space Partitioning trees. In k-d trees, space is partitioned into two halves using splitting planes that are aligned to the co-ordinates axes. Using axis-aligned splitting planes leads to the axis-aligned bounding boxes that are easy to deal with in the construction and traversal phases.

A k-d tree node is an axis-aligned bounding box that contains many triangles. Intersection tests are performed on the triangles only if the ray intersects the bounding box. This helps to avoid intersection tests for all triangles whose bounding box is not intersected by the ray. In our implementation, the entire scene is put inside a bounding box. This box is then split recursively at each level of tree to give new, smaller bounding boxes. This way, the triangles are only stored in the leaf nodes. Intersection tests are only performed on leaf nodes that are intersected by the ray. During the k-d tree construction, the splitting axis is chosen in a round-robin manner when going down to the leaves. There are various ways to find the split point. We have chosen the median object along the axis as the split point for a node. Triangles that lie on the splitting plane are added to both child nodes. The k-d tree is constructed on the CPU as a flat array to ease processing on the GPU.

On the GPU, the k-d tree is traversed by every thread to find the closest intersection object for the ray. We have used an iterative, stack-based approach to traverse the tree. Nodes that are intersected by the ray are put in a stack. For non-leaf nodes that are intersected by the ray, we check if only one or both children nodes are intersected by the ray. This is done by comparing the entry and exit points of the ray with respect to the box, and the intersection point at the splitting plane. If both children are intersected by the ray, the far child is put in the stack first so that it is processed later. This ensures that nodes at the same depth are traversed in the order that the ray intersects them. Hence, at the leaves, finding an intersecting triangle means that no nodes in the stack need to be processed as they are bound to be farther away than all triangles in the current leaf.

5 Results

Using our k-d tree implementation for the same scene as before, we could achieve a speedup of 3x over the brute-force search algorithm. These tests were performed with no secondary rays involved.

The scene consisted of about 17,000 triangles. Because of duplication in the case of triangles on the spltting plane, the number of triangles to be processed went to 53,000. In the brute-force algorithm, all triangles are tested for every ray. That is, about 17,000 tests for each ray. Using k-d trees, the average number of tests performed for a ray was about 2,000. This is only 3.7% of the total triangles, and 11.5% of the triangles in the scene. The maximum number of intersection tests performed by a thread were found out to be 8926.

6 Conclusion

K-d trees reduce the number of intersection tests by a huge number and therefore give a good improvement in the overall performance. However, the speedup isn't as high when compared to the factor by which this number goes down. This is due to the following reasons. The number of duplicates does affect the quality of the k-d tree. In our case, duplication increased the total number of triangles by three times. Also, since the threads traverse the tree independently of each other, the memory requests made by them are quite arbitrary. Hence, the requests are not coalesced. However, in case of the basic implementation, since the threads perform intersection tests with all triangles in the same order, the requests are well coalesced. Apart from this, k-d trees suffer from divergence as well. Since, there may be cases where certain threads do not intersect with the bounding box at a level, their execution ends there, while the other don't, hence resulting in divergence within a warp.

7 Bibliography

Aila, Timo, and Samuli Laine. "Understanding the efficiency of ray traversal on GPUs." Proceedings of the conference on high performance graphics 2009. ACM, 2009.

Wald, Ingo, and Vlastimil Havran. "On building fast kd-trees for ray tracing, and on doing that in O (N log N)." Interactive Ray Tracing 2006, IEEE Symposium on. IEEE, 2006.

Wende, Florian. "Dynamic Load Balancing on Massively Parallel Computer Architectures." (2013).