Ray Tracing Through The Graphics Pipeline
EEC 277 Final Project

Ahmed H. Mahmoud and Yuxin Chen

March 16, 2017

1 Very Brief Introduction to Ray Tracing

The reason we can see objects because photons are reflected into our eyes basing on the observation: firstly, without light we cannot see anythiny and secondly, withou object in our environment, we cannot see light. If we were to travel in intergalactic space, if there is no stuff around us, we cannot see anything but darkness even though photons are potentially moving through that space. This gives us (the old researchers) the idea that we can produce a picture by simulating the behavior of photons emitted by light sources, then there is another physical phenomena which we need to be aware of. Comparing to the total number of rays reflected by an object, only a select few of them will ever reach our eyes. So we would potentially have to cast zillions of photons from the light source to find only one photon that would strike the eye. This approach is called forward ray-tracing. Thus forward ray-tracing requires a computation power which is impossible in that age maybe also impossible in this age. Then Turner Whitted[**?**] came, solving this problem together with reflection, refraction, shadows and hidden surface removal by the way by backward ray tracing which tracks the ray backwards from eye to the object. Then it saves tons of computation of photons which finally don't get to our eyes, making ray racing possible. Also because it integrates reflection, refraction, hidden surface removal and shadows into one model, ray tracing is thus regarded as a versatile rendering technique for light-object interaction. It is something value worth research's time.

2 Why GLSL

Recursion is one of the basic implementation approach used in ray tracing, especially for ray tracing implemented on CPU, though recursion is not efficient in terms of its stack operation overhead, it is conceptually simple to programmer and is easy to code. One reason for implementation on OpenGL is not popular is that graphics pipeline is fixed pipeline with limited programmbility. However the programmable stages of graphics pipeline are enhanced and allow more flexibility and longer program that basic ray tracing components can be loaded as a single program. However, the benefit of using GLSL in OpenGL is obvious: the program can be operation system and GPUs independent and is not constrained with certain vendor like CUDA. Also a program implemented in OpenGL can be supported on WebGL which is run on web browsers and becoming rapidly common. It is thus still making sense and practical to consider developing a rendering system using OpenGL for general computation. #TODO AHMED add more or expand it making it more convincing.

3 Use of Texture Memory

4 Acceleration Structure: Bounding Volume Hierarchy

In [?] Whitted estimated that a recursive ray tracer spends up to 95% of its time testing for intersections. This imposes a huge computational cost to the process. There are two aspect in terms of the high cost for intersection checks.

1. Intersection checks for every objects. If the number of objects are high, this process can be painful even if most of objects end up not intersecting with the ray.

2. Computational expense of each intersection test depends on the complexity of the object representation.

One approach resolving the second issue is using bounding volume with intersection checks. If a ray doesn't intersect the bouding volume of an object, then the object is ignored from further processing for that particular ray. We are using axis-aligned bounding boxes(AABBs) which only require simple (basically compare) and small number of operation for intersection check. However, one point worthy mention: usually the tighter the bounding box fits the object, them few of the rays intersecting this bounding volume are likely to actually miss the object inside. However, if the bounding box has a complex shape, then it increase the cost for intersection check. Obviously, the bounding volume that fits this criteria is the object itself. A good choice for a bounding volume is therefore a shape that provides a good tradeoff between tightness and speed.
To address the first issue, an intuitive method is to only check subset of object who are likely be intersected. We build our approach around a bounding volume hierarchy, where each group is associated with a conservative bounding box.

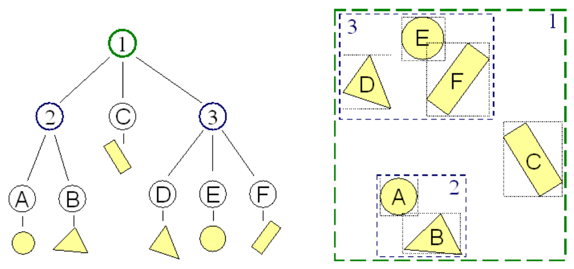

Figure 1: Example of bounding volume hierarchy

Effective use of BVH tree can be explain further into two aspects:

1. Efficient tree traversal ?? efficient or effective?
2. Effective construction of BVH tree

The explanation about first issue is discussed more in other Optimization since it is related more on how to minimize divergence on data and instruction. We will mainly talk about how to construct BVH tree in parallel in this section.
Why construction speed is of the essence? Suppose we use ray tracing in game or physics simulation, each frame, the objects can move, thus requiring a different BVH for each frame. So it is nontrivial to build the BVH tree fast.
The most promising parallel BVH construction approach is to use a so-called linear BVH(LBVH). The idea is to simplify the problem by first choosing the order in which the leaf nodes appear in the tree, and then generating the internal nodes in the a way that use this order. For data locality consideration, we prefer the objects that located close to each other in 3D space to also reside nearby in the hierarchy since objects nearby are highly checked together and data can happen to be preloaded in the memory because cache line is the base unit loaded. So a reasonable choice is to sort them using their Morton codes which encoding the space position information within the code. Then use parallel radix sort is the right tool for this job. A good way to assign the Morton code for a give object is to use the centroid point of its bounding box. Then we can sort those objects by their space position information and generate a node hierarchy where each subtree corresponds to a linear range of sorted primitives so that ones close to each other in 3D end up close to each other in the resulting tree.
Then we use the method describe in [**?**] to parallel construct the binary radix tree. One key insight in enabling parallel construction is to establish a connection between node indices and keys through a specific tree layout. The idea is to assign indices for the internal nodes in a way that enables finding their children without depending on earlier results. Using Morton code, we can easily know give a range how should we cut the range and make it into a tree. Then problem becomes how to find the range independently. We can take the advantage that the index of very internal node coincides with either its first or its last key. Then what you need to find out which direction it tends to be on the tree and its upper bound or low bound by comparing the length of the prefix with other objects (a larger difference on the prefix indicate it is likely to cut at this point). Nicely, the time complexity of this algorithm in the worst case is $\mathcal{O}(n \log_n)$.

5 Other Optimization