Photon Mapping in the TRAX Environment

Brig Bagley
University of Utah
CS 6965 – Parallel Ray tracing
Final Report and Analysis
Instructor: Erik Brunvand
16 December 2011

Abstract This paper describes ray tracing implementation of photon mapping on the TRAX environment. Photon mapping is a form of global illumination that captures both direct and indirect lighting in a scene. Following the implementation of photon mapping on the CPU, this paper also describes the process of re-writing the program to function on TRAX. A short analysis between four scenes and two sets of photons illustrated the performance of the photon mapping algorithm.

1 Introduction and Goals

Photon mapping is a global illumination algorithm that casts “photons” from the light sources in the scene and stores them in an acceleration structure for quick access. These photons are gathered when calculating the color at a point in the scene to illuminate otherwise darker parts of the scene with colors coming from other locations. This process of “color bleeding” makes for a much more realistic image. Initially, the goal of this project was to successfully implement an efficient photon mapping ray tracer to apply to the TRAX environment. It was also a goal to run several tests and create a useful analysis of how well photon mapping would work on the real TRAX hardware. Although initial goals weren’t met completely, enough was completed to give a good idea what photon mapping would be like on TRAX.
2 Work Plan

The first step to this project was researching and implementing a software version of a photon mapping ray tracer. This was initially supposed to take about two weeks to tweak in preparation for work on the TRAX environment. To help in testing on TRAX, it was suggested to write the program using the TRAX coding restrictions. By continually testing the program on the simulator, it would be little to no work to run tests with it after the software version was tweaked enough. After thoroughly understanding how an efficient photon mapped ray tracer works, the plan was to make as many optimizations as possible and begin running renderings on the TRAX simulator. These tests would be compared to see what type of hardware and software considerations to take when running the ray tracer on the hardware itself.

3 TRAX Considerations

TRAX, although very simple compared to other parallel coding environments, still has many limitations and restrictions. The most common issues are avoiding the unsupported primitives, passing variables by reference, and the lack of any native dynamic memory managers. The biggest consideration for this project was managing global memory for the casting and gathering of photons. Photons in the implementation I used required a great deal of memory (about 12 words per photon, and up to 300,000 photons). It would be impossible to manage this much memory in local space. Another consideration was making sure that the photon map KD tree was write-only. The multiple cores in TRAX would be impossible to coordinate when reading and writing simultaneously at the same places in global memory. It is also imperative that there are no recursive algorithms, since TRAX doesn’t support it.
4  Solution and Algorithm

There are two important aspects to my solution. The first is casting the photons efficiently and uniformly across the scene and storing them in a KD tree to be accessed later when shading. The second aspect was to quickly gather the closest photons and weight their radiance by a few factors to add to the direct lighting.

Casting the photons was the simplest part of the project, although tweaking the algorithm took many hours. After settling on architecture for the photons, I looped through emitting \( N \) photons and bouncing them around the scene between 1 and 4 times. The direction of the bounce was determined by the properties of the object intersected. A Russian roulette technique was used to determine whether a photon was reflected, refracted, diffused, or absorbed. Reflection, refraction, and diffuse were constants that summed to 1.0. Whatever range a random number between 0 and 1 landed determined the next direction of the photon. Diffuse bounces had to occur at least once, and reflective or refractive bounces were absorbed as soon as exiting the medium and entering a diffuse surface. The absorbed photons were stored to a place in global memory.

Part of the photon structure was used to help in sorting them into the KD tree. Each photon was a splitting point for its sub-trees, or a leaf node. The splitting plane was in the dimension with the greatest extent and at the median photon after sorted by that dimension. It had an index range stored from the beginning of its left sub-tree to the end of its right sub-tree, as well as indices for the median of the sub-trees, acting as children. Each photon also had an index for its parent. This made for easy sorting and traversing of the tree.
Finally, photons had to be gathered for each ray intersection during the traditional ray casting and shading algorithm. This was first done by gathering all of the photons within a certain radius of the intersection point. However, to be more accurate, I later implemented a way to gather the N closest photons with a varied radius. After traversing the tree by split points to a leaf node, the tree was traversed backwards to see if any other photons were within a beginning radius. If the parent splitting plane was inside the radius, the sibling tree was traversed similarly. This ensured that all of the photons within the beginning range were found. If the root node was reached before enough photons were found, the process was restarting with a larger initial radius. If more than N photons were found, closer ones replaced further ones, and the radius shrunk to the greatest radius of the N closest photons. Photons were also filtered by their direction dotted with the normal of the surface and direction dotted with the direction of the view ray.

5 Problems Encountered

There were many problems throughout the entire project process. The first was getting a software version of the program to work efficiently and accurately. The KD tree structure was very difficult to set up for sorting and traversing. There were many hours of debugging to make sure these algorithms were accurate.

Once the casting, sorting, and searching for photons was working at some level, the next problem was trying to scale the indirect lighting to match the scene itself. Sometimes it was much too strong or weak, not distributed well, or non-existent. I initially used the photons within a certain radius. This produced interesting results, but the pictures were very blotchy.
After implementing the N closest photons algorithm, some things improved, but rendering time dramatically increased. A few other sampling techniques were used, but even on the CPU photon gather had to be either be fast and blotchy, or smooth and take hours at a time. In order to finish enough work I opted with the fast and blotchy algorithm.

Initially, my shading algorithm for reflections and refractions was recursive. It took a lot of thinking and a couple of days to write an iterative algorithm for the code work on TRAX. It used a lot more local memory, but nothing too substantial for a thread to handle.

Before understanding how the global memory worked, I was using local memory (for each thread) for the photon map. This, I found out, was a very big issue. I spent nearly a week re-writing all of my code to access only global memory when storing and reading the photons from the KD tree. I was able to get the algorithm to work, but for some reason, the pictures haven’t turned out anywhere near as accurate as they did before the conversion. On the upside, I was able to emit over 10 times the amount of photons very quickly without memory issues, and I was able to ensure uniform distribution of the photons in the scene. I was also able to parallelize the emitting of photons simply by adding 3 lines of code.

The final issue was getting the whole program to run and complete on the simulator. I had numerous errors that I had to re-write code for in order for the simulator to run. Unfortunately, I didn’t write my code for the simulator or start with as well as while I was working. This ended up being a big challenge to recover from, since the point of this project was to see how this rendering technique was to work in the TRAX environment. It was a little much to take on learning photon mapping, plus parallelizing it on a new architecture all at once.
6 Results and Images

I didn’t have much time to run many tests or complete a thorough analysis. Unfortunately my only runs that completed were not in the simulator. Simulator runs would disconnect from my computer before finishing. However, I was able to run four scenes with two different sets of photons. Below is a chart displaying some of the stats from the runs:

<table>
<thead>
<tr>
<th>Scene</th>
<th>Diffuse Photons</th>
<th>Caustic</th>
<th>CPU Run time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified Cornell</td>
<td>20,000</td>
<td>10,000</td>
<td>59.4</td>
</tr>
<tr>
<td>Modified Cornell</td>
<td>2,000</td>
<td>1,000</td>
<td>21.3</td>
</tr>
<tr>
<td>Cornell Box</td>
<td>20,000</td>
<td>0</td>
<td>35.8</td>
</tr>
<tr>
<td>Cornell Box</td>
<td>2,000</td>
<td>0</td>
<td>12.7</td>
</tr>
<tr>
<td>Conference</td>
<td>20,000</td>
<td>0</td>
<td>116.8</td>
</tr>
<tr>
<td>Sponza</td>
<td>20,000</td>
<td>0</td>
<td>224.3</td>
</tr>
</tbody>
</table>

Below are a number of image renderings throughout the project process.

Figure 1: Early stages of photon emitting and gathering before implementing functionality on TRAX.
Figure 2: Photon emitting on Cornell Scene read from memory - 200,000 Photons (left) and custom Cornell scene with caustics - 300,000 Photons (right) after adapting code to TRAX and global memory.

Figure 3: Full global illumination rendering of Cornell Scene read from memory - 20,000 Photons (left) and full global illumination rendering of custom Cornell scene with caustics - 30,000 Photons (right).
Figure 4: Direct illumination rendering of Conference Scene read from memory - 20,000 Photons (left) and full global illumination rendering of Conference Scene with caustics - 20,000 Photons (right). There were some obvious issues with this scene.

Figure 5: Direct illumination rendering of Sponza Scene read from memory - 20,000 Photons (left) and full global illumination rendering of Sponza Scene with caustics - 20,000 Photons (right).
7 References

