Accelerated Stochastic Progressive Photon Mapping on the GPU

Ishaan Singh Yingting Xiao Xiaoyan Zhu
CIS565, University of Pennsylvania

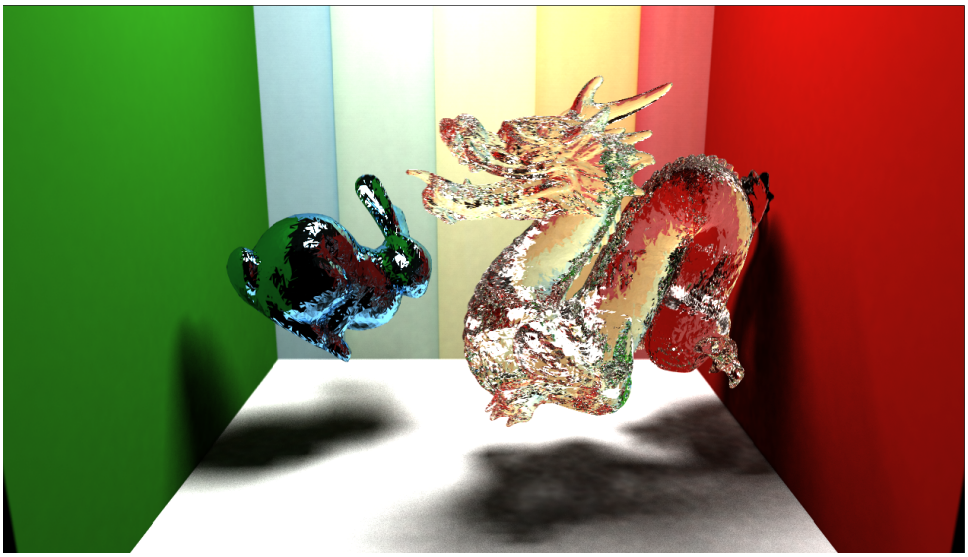

Figure 1. Stochastic Progressive Photon Mapping on the GPU

Abstract

This paper describes the implementation of a stochastic progressive photon mapper on CUDA5.5 using a spatial hash grid as an acceleration structure for speeding up density estimation and using a kd-tree to speed up ray intersection testing.

1. Introduction

Photon mapping is an advanced rendering technique based on irradiance caching and bidirectional path tracing. It speeds up global illumination computation for scenes by caching photons on the non-specular geometry and using them in a ray trace step to get global illumination effects without the noise associated with path tracing. This is similar to splitting a monte carlo path tracer into direct and indirect illumination

 ISSN 2331-7418

stages. Since photons stored have a certain size, it is not feasible to store an extremely large number of photons. We bypass this limitation by implementing progressive photon mapping where every iteration shoots a certain number of photons in the scene and the irradiance is averaged across multiple iterations. By including a stochastic camera position like in distribution ray tracing, we are able to bring back in all the effects of depth of field, motion blur, anti-aliasing etc. Thus our stochastic progressive photon mapping is faster as well as less noisy than a forward monte carlo path tracer albeit a little biased. To use the GPU optimally and improve our photon look up step, we implement the photon map as a spatial hash grid. This hashing technique allows us to have an O(P) photons where P is the number of photons in that particular grid cell. Additionally, we support large OBJ files with the support of our implementation of a stack-less KD-Tree using ropes.

2. Related Work

Real-time ray tracing is a hot topic in graphics and has been called the "future of computer graphics". But, it has also been said that ray tracing will "always be the future of computer graphics". This is because it involves a large compute to memory ratio. With massively parallel processors, i.e. modern GPUs, being made more and more easy to program upon, this future goal seems to be coming closer and closer. A large amount of research has gone into making ray tracing available on GPU and with multiple CUDA implementations available all over in the research community. The next step is to port path tracing on to the GPU and in the last few years multiple people have been involved in that as well.

A new approach to global illumination was proposed by Henrik Wann Jensen called photon mapping[0] which simulates photons being shot out from a light source and being stored on non-specular surfaces. These photons carry flux around with them and the color and flux of the photons is affected by the material properties of the surface that it hits off. Photons are stored or reflected or absorbed based on a probabilistic model related to the bidirectional reflectance distribution function. The first implementations of this were on the CPU and restricted to static scenes with the intention of reusing this information in subsequent frames. Since each photon computation is independent of the other photons, it is an embarrassingly parallel problem. Thus, it is a problem quite fit for the GPU in the same manner that ray tracing or path tracing is. GPU implementations of photon mapping have also been attempted in the recent past successfully. To make sure that the density estimation stage of photon mapping is fast, a spatial acceleration structure is essential. Without it, one would be traversing all the photons per ray intersection to find the photons that are contributing and considering the fact that photons are in the order of millions, this linear check is an expensive operation. A KD-tree is the suggested acceleration structure of choice

suggested by the creator of photon mapping, Jensen. While a kd-tree on the GPU is definitely possible, it is not an easy or trivial data structure to implement. This is because a tree traversal is intuitively recursive and needs a stack. Stackless KD-tree structures[2] have been suggested and used in modern GPU architectures successfully. But since the density estimation is a k-nearest neighbors problem a kd-tree will be not the most efficient data structure. A hash grid would be the ideal data structure to use as explained by Fleisz[1].

An extension to photon mapping involves removing the limit on the number of photons that can be stored by shooting a new set of photons each iteration and averaging the results of each of the iterations. This helps in getting a better estimate of indirect illumination, especially at caustic regions and regions that are completely in shadows with respect to ray tracing. This progressive photon mapping[3] technique allows for millions and millions of photons contributing but only requires storage for far fewer photons. Extensions to photon mapping have also been introduced to handle more complicated light paths (Specular-Diffuse-Specular). These light paths can be rendered using standard path tracing or bidirectional path tracing but the noise in such cases is unacceptably high and letting the algorithm converge takes far too long considering that these paths are not uncommon in real life (a light source within a refractive coating which is occluded partially : a light bulb which has a plastic holder). Using stochastic progressive photon mapping[4], such paths are handled adequately compared to even bi-directional path tracing since the density estimation handles these “caustic“ rays quite well.

Since a photon is a point, the spatial hash grid works quite well. But for accelerating triangle intersection tests, a stackless kd tree is still one of the most optimal data structures as it helps avoid large volumes where the ray doesnâĂŹt intersect with anything.

3. Method

We implemented stochastic progressive photon mapping with a spatial hash grid for photon look up and GPU kd-tree for ray triangle intersection testing. This section will describe each of these in full detail.

We work off the base code for project 2 with a fully integrated path tracer and ray tracer to start with. The code was extended to support obj files as well as textures.

3.1. Photon Mapping

For photon mapping, we do a light trace from the light source after choosing a random point on the light source. Each photon‘s initial position is chosen randomly on the surface of the emitting object. Currently, only cubes and spheres can be chosen as emitting sources. The photon is bounced around in a similar manner to path tracing

by intersecting it with all the objects in the scene. If the object that it intersects with is a diffuse object, the color carried by the photon is multiplied with the color of the object (similar to path tracing) and the photon is marked as stored and then bounced further by choosing an arbitrary direction the in hemisphere. If the surface is reflective or refractive, the photon's color is modulated but it is not marked as stored. The new direction to bounce the photon into is decided based on the BRDF or BTDF and it is passed along. These photons are the photons that contribute to the caustics. We are not storing a separate caustic photon map and global photon map, but using the global photon map for the caustics as well since the number of photons being used goes is quite high.

The number of photons to be sent out is pre-determined and they are distributed amongst light sources based on the energy of the light source. The energy of the light source is equal to the emittance of the light source multiplied by its surface area. This allows us to store exactly the same flux in each photon.

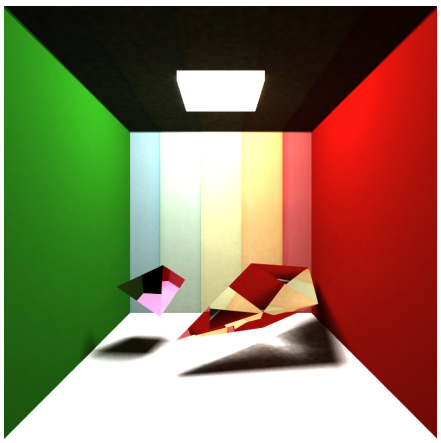

Figure 2. SPPM

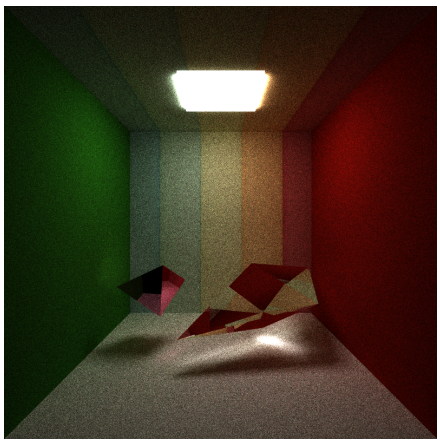

Figure 3. Path traced

Now, for density estimation using photons stored, we ray trace an image to collect direct illumination. Once a ray hits a surface, the k nearest photons are gathered and their weighted flux is used to estimate the global illumination at that point. We do not do a final gather stage and assume that the photons give a good enough estimate.

3.2. Spatial Hash Grid on the GPU

For finding the k-nearest neighbors on the GPU, we could loop through all the photons but that takes for ever (5 minutes vs 3 seconds). Thus, we decided to implement a spatial hash grid based on prior work. The hash grid works by assigning a hash value to each photon, sorting the photon list by this hash value and then storing the index of the first photon with a certain grid cell's hash in that grid cell. This hash grid is stored for the frame and then when the ray hits a surface, the grid index of the intersection

point is calculated. The starting photon index is taken out from here and we traverse the photon list till the grid index of the photon being tested is not the same as the current grid index.

This is done for the surrounding 26 grid cells as well and the nearest k photons are stored. For finding the k nearest photons, we fill an array of k photons with the first k photons we find per thread. Then onwards, if we find a new photon and its distance is lesser than the photon with maximum distance amongst those k, we swap the new one in and find the next photon with the maximum distance from the intersection point.

This use of the hash grid gives us O(1) look up time for the first photon and then we loop over P photons per grid cell to get the closest k and that is considerably faster than using a kd tree.

3.3. Stackless KD-Tree with Ropes on GPU

While KD-Tree‘s aren‘t the best for finding the k-nearest neighbors, it is still excellent for accelerating ray traversals through a scene. The scene is divided into a kd-tree on the CPU using a Surface Area Heuristic by trying out N bins. The KD-Tree is then processed and ropes are added to each leaf node. A rope is a connection to another node in the tree which is adjacent to the current node. Thus, each leaf node has six ropes; one for each direction in each of the axii; x,y,z. Once the ropes have been created, the tree is linearized for moving to the GPU. This is done by a DFS traversal and giving each node an index. Once this is done, for each node, the indices of the ropes and children are stored. This data structure is sent over to the gpu and can be traversed in a stackless manner.

The traversal is similar to the Popov traversal where each ray finds the leaf node that it is at, then finds all triangles that it intersects with. If it doesn‘t intersect with any triangles, the ray‘s starting point is set to the exiting intersection with this current leaf node. Thus, we march through the entire cell if there is nothing to intersect with or the ray doesn‘t intersect with any of the geometry there.

4. Results

We were able to get a CUDA-Enabled ray tracer with photon mapping working. It was extended to stochastic progressive photon mapping which uses a spatial hash grid as an acceleration structure. The images rendered have definitely visible global illumination effects of caustics and color bleeding. This can be seen in the following images. We tried to implement a GPU-KD Tree and got it working for the most part. This definitely helped speed up our renders, both on the GPU as well as the cpu version of it we made for testing.

The photon mapped images converge faster than path traced ones though the noise is apparent as blur. This blur can go be lowered by lowering the radius but that reduces

the energy calculated from the photons. Lowering the radius also helps speed up the gather step since there are fewer photons per grid cell.

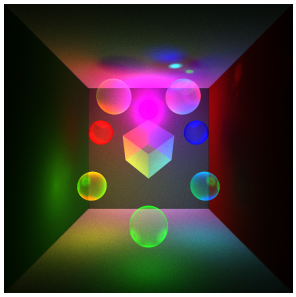

Figure 4. Photons in a scene with 6 light sources

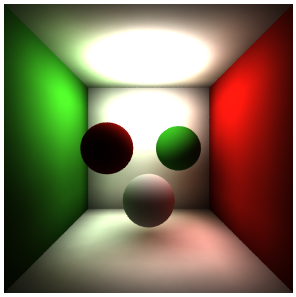

Figure 5. Image lit directly by photons

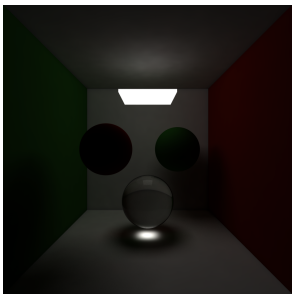

Figure 6. Combined Illumination

5. Performance

We first test how long it takes to gather photons for a large number of photons per iteration with and without using the spatial hash grid. As we can see, the grid makes the look up time almost constant while the brute force search scales about linearly.

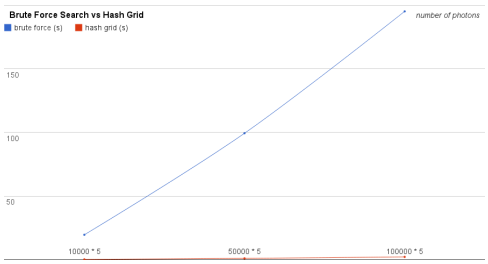

Figure 7. Photons with and without has grid

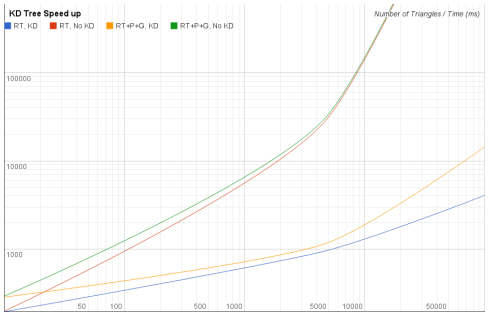

Figure 8. Speed up with KD Tree

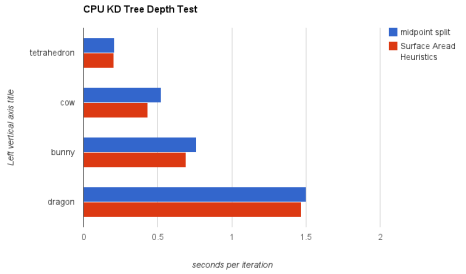

Figure 9. KD Tree: Mid point split vs SAH split

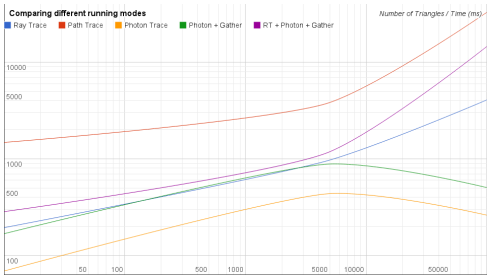

Figure 10. Different technique comparison (log,log)

We tested how much speed up was achieved by using a KD tree. From the values below, we can see that for meshes with only a few triangles, KD tree traversal is slower than performing intersection testing on all triangles in the scene due to the computation time required for traversal. However, with bigger meshes such as the Stanford dragon, a KD tree is necessary to rende the scene. Infact, the dragon doesn't render in even more than 10 minutes on simply ray tracing mode.

A cpu side comparison of kd tree traversal is described which shows that the KD Tree built with a surface are heuristic is better than a mid point split tree but marginally so. We believe this might be due to a non-optimal implementation of the surface area heuristic.

And finally, we compare times for rendering scenes of various complexities with just ray tracing, path tracing, photon tracing, photon trace and gather and lastly ray trace, photon trace and gather steps; all in log scale. We notice that the photon trace step actually peaks out a certain number of triangles, only taking that long at maximum. The gather step seems to be a constant overhead on top of the photon trace step. While the photon trace, gather and ray trace is more expensive than simlpy ray tracing, it is still far lower than the path tracing cost while giving out images of better quality most of the time.

6. Future Work

To extend this work, we would need to first to scale the energy of the photons vs the ray trace in a physically correct manner. This would ensure that the images for the path tracer and photon mapped match more correctly. More extensions would include building a better KD-Tree as well as changing the density estimation radius based on the largest of distance of k-neighboring photons. More BRDF and BTDF models can be included. Use of better random numbers would definitely be an area to explore.

Performance for the project has been on our minds since we started but a lot of our optimizations have been more algorithmic and higher level. But since we are on a GPU, a lot of more optimizations could be added, particularly with respect to the global memory accessing. Modern GPUs have a small cache but using shared memory during KD tree traversal would help speed things up significantly. A comparison with a short stack KD-Tree would be interesting to see. Both papers came out around the same time and seeing which works faster or better would be very useful.

7. References

[0] Henrik Wann Jensen, Realistic image synthesis using photon mapping, A. K. Peters, Ltd., Natick, MA, 2001.

[1] Martin Fleisz. Photon Mapping on the GPU. Master's thesis, School of Informatics, University of Edinburgh, 2009.

[2] Popov S., Gunther J., Seidel H.-P., Slusallek P.: Stackless kd-tree traversal for high performance GPU ray tracing. Computer Graphics Forum 26, 3 (2007).

[3] Toshiya Hachisuka , Shinji Ogaki , Henrik Wann Jensen, Progressive photon mapping, ACM Transactions on Graphics (TOG), v.27 n.5, December 2008

[4] Claude Knaus , Matthias Zwicker, Progressive photon mapping: A probabilistic approach, ACM Transactions on Graphics (TOG), v.30 n.3, p.1-13, May 2011

Singh I., Xiao, Y., Zhu ., Accelerated Stochastic Progressive Photon Mapping on the GPU, *Journal of Computer Graphics Techniques (JCGT)*, vol. 1, no. 1, 1–1, 2013
https://github.com/ishaan13/PhotonMapper/

Received:	Dec 12, 2013		
Recommended:	Dec 12, 2013	Corresponding Editor:	
Published:	Dec 12, 2013	Editor-in-Chief:	Morgan McGuire