REAL-TIME GPU PHOTON MAPPING

SHERRY WU

Abstract. Photon mapping, an algorithm developed by Henrik Wann Jensen [1], is a more realistic method of rendering a scene in computer graphics compared to ray and path tracing. Unlike ray tracing, photon mapping first emits lots of photons from light sources in a scene and then uses the photon map to calculate the radiance of each pixel in the rendered scene. Previous work in the area include a GPU photon mapper written by Stanford Graphics [1], but there do not exist any real-time renderers yet. For this project, a real-time GPU photon mapper using Nvidia CUDA will be implemented.

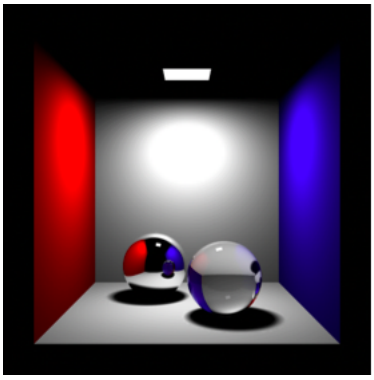
1. Introduction

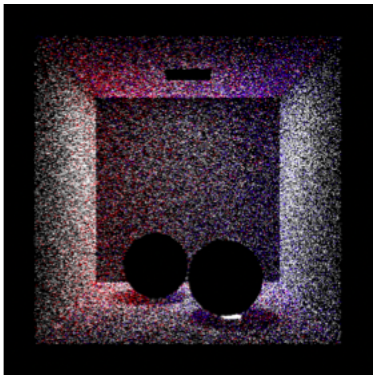
Photon mapping is an algorithm that builds on a more primitive algorithm, ray tracing, used for rendering scenes. This algorithm works by discretizing the image plane, shooting a ray through each discretized unit of the image plane, and testing whether each ray intersects with some object in the scene. If the ray does hit an object, then for each light source, the direction from the intersection to the light source and the color of the light source at the intersection are noted, and are used by the material of the intersected object to compute shading. This process solves the Rendering Equation:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, w_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t)(\omega_i \cdot \mathbf{n}) d\omega_i$$

The ray tracing algorithm can be extended to support visual effects to make the resulting image more realistic, such as shadows, reflection, refraction, and depth of field. It is also an *embarrassingly* parallel algorithm, but is still computationally intensive, depending on the visual effects desired in the result.

Photon mapping builds on ray tracing. As can be seen in the juxtaposed images below, the right shows the photon map and the right shows the rendered image, which utilizes the photon maps to calculate the extra brightness on the red, white, and blue walls.





The photon maps are built in the first step. Two maps are produced, one for diffuse photons, another for caustics. The diffuse photons are generated as follows. A photon starts at a light source and follows a ray predetermined to the light source's distribution. We restrict the discussion to a point Lambertian source, which uniformly samples a ray on a unit hemisphere. The photon follows that ray until it hits an object. If the photon hits a diffuse object, then it is stored in the diffuse map. We then decide whether the photon bounces off the object or absorbed by a simple sampling algorithm, Russian Roulette, which decides between a pair of events based on a random number. The caustic photons are similarly generated.

In the second step, the image is rendered similarly to the ray tracing algorithm. The main difference lies in getting the color from the light: in addition to computing the shaded color from the material, the colors of the k nearest photons are averaged and included. After the calculation is done for each pixel, the result is written to a file on disk or displayed.

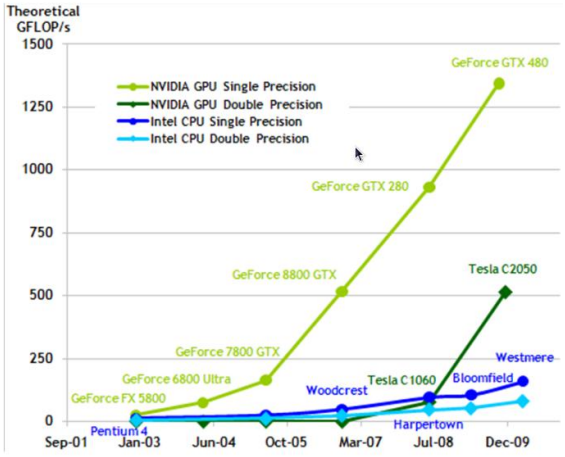
2. Background and Previous work

Several groups have built photon mappers for the GPU, including Wann Jensen himself [1], but few have utilized spatial hashing [2], or simply hashing in higher dimensions. One advantage the spatial hash has over the octree, the more conventional photon storage data structure, is that its structure is linear: that is, no pointer indirections are required to traverse the data structure. This linear structure also makes memory management easier, it can be copied back and forth from the GPU with one call to `cudaMemcpy` and similarly can be allocated and freed with one call to the respective instructions.

Several researchers at Microsoft Research have worked on a perfect spatial hash given a scene [3]. However, this implementation aims to be flexible and have a spatial hash that performs well for arbitrary scenes.

3. Implementation

The photon mapper is written in C++ with CUDA, Nvidia's framework for general-purpose GPU work. We chose GPUs instead of x86 CPUs because GPUs offer way more compute power than CPUs for the same price, as can be seen in the following chart.



In particular, GPUs are able to achieve this kind of performance because they have hundreds of shaders, analogous to cores, on one die. These shaders, while primitive, are incredibly fast at simple tasks, such as math, which is the bulk of the work in ray tracing and photon mapping.

3.1. **CUDA.** CUDA, or Compute Unified Device Architecture, has a collection of functions to interface with the GPU. It supports a subset of C and C++ functionality on the GPU. In particular, recursion is not supported due to a limited stack on the GPU. Since the ray tracing algorithm takes as input a fixed number of bounces, that can be a constant and recursion can then be simulated using template metaprogramming.

3.2. **Spatial Hashing.** The spatial hash implemented in this version of the photon mapper is quite naive. In the scene space, we arbitrarily bound the space into a cube and then partition it into some number $k^3, k \in \mathbb{N}$ of identical smaller cubes. The hashes of photons that lie within the cube are computed based on the position (x, y, z) of the photon as follows:

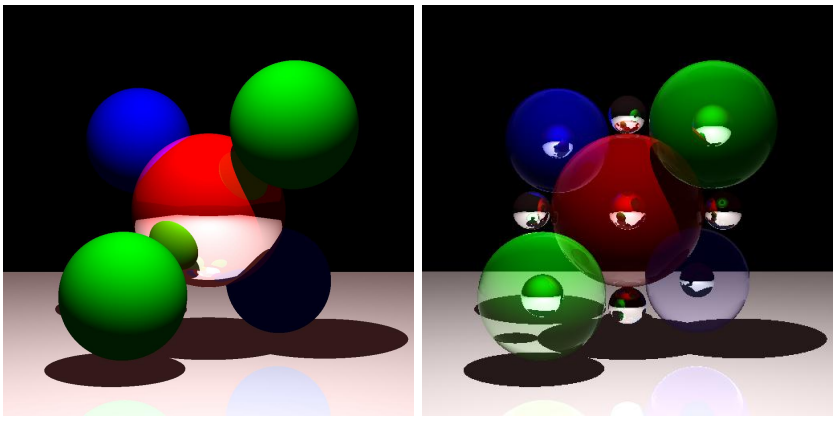
$$h(x, y, z) = k^2(\lfloor x \rfloor + k) + k(\lfloor y \rfloor + k) + (\lfloor z \rfloor + k). \tag{1}$$

For the few photons that are outside of the cube, the coordinates are clamped to be within the cube before feeding the values into (1). The spatial hash keeps track of the photon IDs that are in each of the subcubes.

During the global illumination and caustic illumination computation, given the intersection position x, the renderer will compute the luminance by averaging the power of the photons in the subcube that contains x. Although this is not a true solution to the k-nearest-neighbors problem, for a sufficient number of photons in the subcube, this strategy gives a sufficiently good estimate of the average of the k nearest photons, assuming the colors of the extra photons do have spatial locality and are not completely random.

4. Results

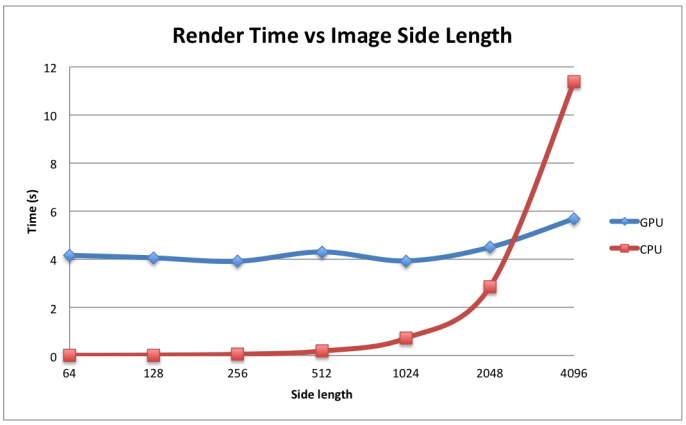
Currently, the ray tracing component has been tested and benchmarked. In particular, three scenes have been used for testing, ranging from simple to complex:



The left scene contains four smaller diffuse balls. Again, reflections can be sceen in the central red ball, which now reflects the green ball and the arrangement (from the reflection on the plane).

The right scene contains nine additional small specular balls, which were set up to be like mirrors. In addition, the larger balls are now refractive. This was designed to both tax the GPU in addition to test refraction.

In addition, benchmark results were obtained for the left scene:



The graph indicates that there is a large overhead in setting up the GPU, but the actual compute time is minimal once the data is on the GPU. Note that the exponential scaling is much less apparent on the GPU than on the CPU, which shows that distributing work over the shaders is quite effective. The larger input sizes are of interest for further developing the program into a feasible real-time renderer, especially for high screen resolutions such as 1920x1080, 2560x1600, and possibly 3840x2400.

The photon mapping portion has not been tested, although preliminary dumps of the diffuse map have shown its associated code to be functional.

5. Conclusion

The goal of the project was to write a GPU photon mapper that implements spatial hashing as an in-depth introduction to GPU programming and photon mapping. The project was implemented in stages, first a primitive ray caster, and then a ray tracer with no special effects, and at this time of writing, a ray tracer that supports reflections and refractions. While coding, much emphasis was placed on getting the code correct the first time since there was relatively little time to debug and that debugging giant swaths of code is painful, so unit tests were also developed in parallel to ensure correctness.

With this project, we hope to demonstrate the sheer power of GPUs, which are relatively inexpensive per teraflop compared to high-end multicore x86 chips. In addition, we hope to revolutionize graphics in games by taking advantage of the additional GPU power.

6. Future Work

Because of the short time given for this project, there are several optimizations and extensions that can be made. Two immediate optimizations that can be done are twiddling the block and grid sizes (work distribution parameters for the GPU) to maximize performance. Calls to `malloc` can be replaced with

`cudaHostMalloc`, which does more housekeeping, but has been shown to reduce copy times from host to device.

Instead of storing the color as a `vector3`, which is 16 bytes, we can store the wavelength of light. Since the visible spectrum is from 400nm to 700nm, we can store the wavelength as an `unsigned char`, which is 1 byte, at the cost of omitting 15% of the spectrum. This not only conserves memory on the host and the device, but also cuts down time spent transferring data back between the CPU and the GPU, a costly operation.

Furthermore, instead of using `shade` and `shadeAmbient` methods for shading, we can use bidirectional reflectance distribution functions, which leads to a more general implementation, which can be extended to more interesting materials, such marble, water, and skin.

Acknowledgements

I would like to thank Alan Edelman and Jeff Bezanson for giving me the opportunity to research photon mapping for 18.337. I would also like to thank Charles Leiserson, Saman Amarasinghe, and Bayley Wang for exposing me to photon mapping, ray tracing, and high performance computing.

References

[1] Timothy J. Purcell, Craig Donner, Mike Cammarano, et al., *Photon Mapping on Programmable Graphics Hardware*. Stanford University and University of California, San Diego, 2003.

[2] Sanket Gupte, *Real-Time Photon Mapping on GPU*. University of Maryland Baltimore County, 2011.

[3] Sylvain Lefebvre, Hugues Hoppe, *Perfect Spatial Hashing* Microsoft Research, 2006.