Point Based Color Bleeding with CUDA and Caching

Nick Feeney

California Polytechnic State University

San Luis Obispo

Abstract - The main goal of this project was to explore the possibility of applying CUDA to the Point Based Color Bleeding global illumination algorithm. This project tackled the creation of surfels, the storage of surfels in an octree, representation of an octree in CUDA, and the transversal of an octree in CUDA. Future work will include the rasterization step of the Point Based Color Bleeding algorithm done in CUDA, spherical harmonics calculations for the intensity of surfels, and the development of an efficient caching method for surfels.

Keywords: CUDA, Point Based Color Bleeding, Caching, Surfels, Micro-polygons, Rasterization, Spherical Harmonics

1 Introduction

Global illumination is a well researched topic of computer graphics. Currently one of the best algorithms for achieving fast and accurate global illumination in a rendered scene is the Point Based Color Bleeding algorithm. This algorithm is extensively used by both Pixar Animation as well as Dreakworks Animation. This method had been used in many feature films including "Pirates of the Caribbean".[1]

Point Based Color Bleeding(PCB) is a two pass method that utilizes a point cloud to approximate global illumination at given points in a scene. The first pass generates the point cloud of surfels(explained later) that stores position, direct illumination, and light intensity also known as radiosity for a micropolygon. The surfels are stored in a octree representation that allows for efficient transversal. The second pass is the rasterization step. A raster cube is formed a every point where global illumination is needed. The surfels are then rasterized onto the faces of the cube. Once the rasterization is complete, global illumination is approximated for that point using the spherical harmonic representation of light intensity from the surfels that where rasterized onto the cube.

The rest of this paper is as follows. Section 2 is related work and background information. Section 3 is accomplished work and the current state of the project. Section 4 is results from the CUDA and octree speedups. Section 5 is on future work and finally, Section 6 is the conclusion.

2 Related Work

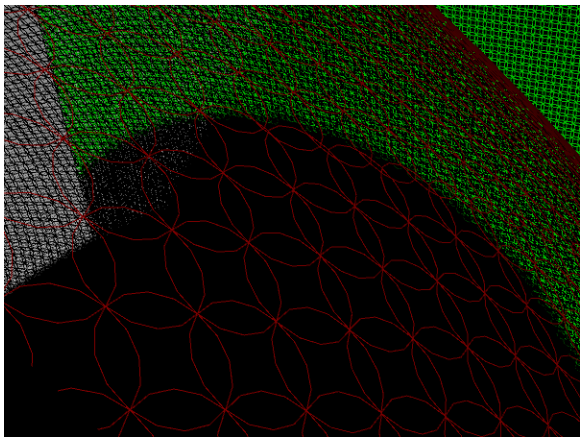

Figure 1: This is an example scene comprised of surfels. This image was taken from Christensen[1].

This Point Based Color Bleeding algorithm was inspired by the work done by Per H. Christensen[1]. His algorithm is based on two distinct steps, surfel generation and rasterization. The first step utilizes a REYES type render to subdivide all the surfaces into micropolygons. An example of a this is seen in Figure 1. Then these micropolygons also known as surfels are transferred into an octree(explained next) which allow for efficient surfel transversal. After the octree is created, spherical harmonics are calculated to estimate the projected power of every node in the octree. This estimation is used later to determine the amount of energy a surfel or group of surfels contributes to the color bleeding effect. The next step is rasterization. Christensen's algorithm creates a raster-cube at every point that global illumination needs to be calculated and rasterizies all of the surfels stored in the octree onto the faces of the cube. This is seen in Figure 2.

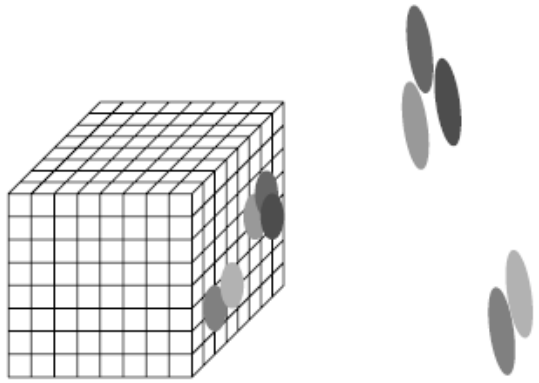

Figure 2: This is an example of a raster-cube. The circles are surfels being rasterized to the cube. This image was taken from Christensen[1].

Octrees are a tree data structure where all internal nodes contain exactly eight children. Octrees are used to subdivide space into an efficient tree structure which can greatly reduce the number of intersection tests needed to render a scene. By testing the bounding boxes of the tree nodes large amounts of the tree can be ignored, which reduces the number of intersection tests required. Octrees are easiest to understand by examining Figures 3 and 4.

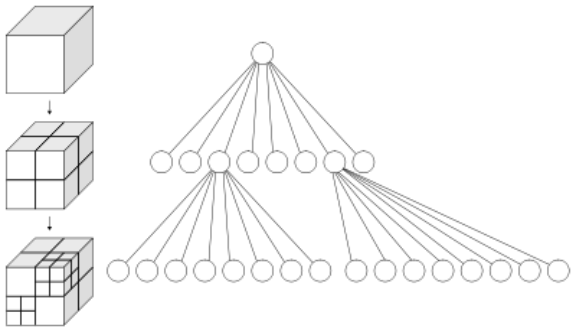

Figure 3: This is a tree structure used to illustrate the creation of a octree with 3 layers.

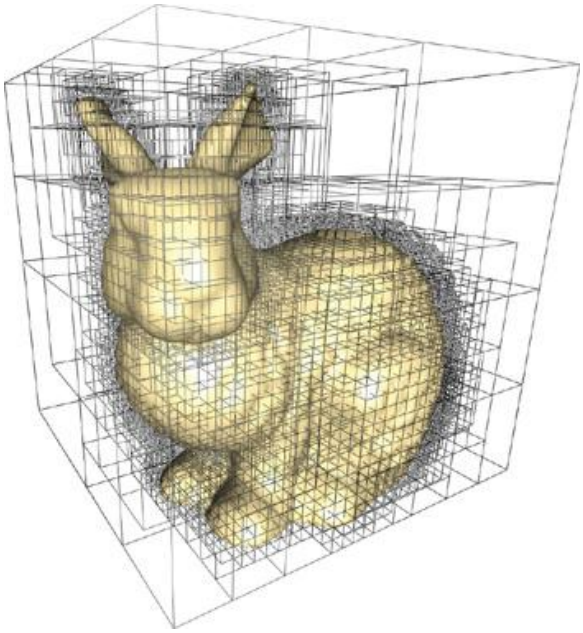

Figure 4: This is an example of a triangular mesh that has been stored into an Octree.

Spherical harmonic functions are essentially solutions to the Laplace's equation that can be used to model different effects with different levels of detail. They are used in many applications in both computer graphics and other fields and explain in great depth in Sloan[2]. It is important to understand that they are a series of floats, 27 in this case, that accurately model the flux of a given substance. The substance for this application is light intensity being absorbed and emitted from a surfel for set of surfels.

3 Implementation

This project is precursor to a full thesis about Point Based Color Bleeding(PCB) with CUDA and Caching. This project tackled four main aspects of the PCB algorithm, surfel generation, octree generation, CUDA efficient octree, and CUDA efficient transversal of an octree.

The central goal of the surfel generation step is to accurately sample the scene and pre-compute the direct illumination of all the sample points. To accomplish this a ray-casting based approach was used. First, the viewing frustum was enlarged to insure that important objects that were not directly in view would be sampled. Next, rays were cast from the camera directly into the scene. For every point in the scene that the rays intersected a surfel was generated. Rays can intersect the scene many times. When a surfel is generated at an intersection point, direct lighting is calculated and stored within the surfel. This method is demonstrated by Figure 5[3].

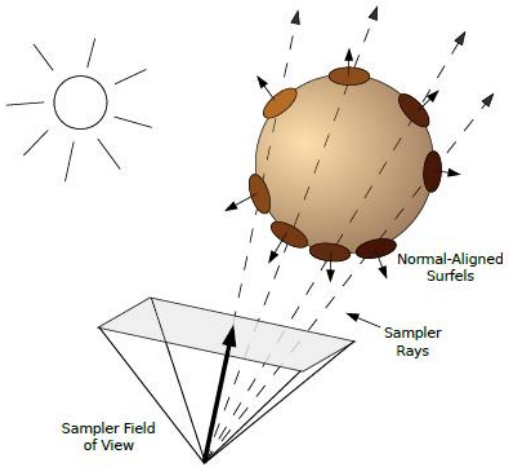

Figure 5: Rays are cast from camera through the enlarged view frustrum and a surfel is generated at every intersection point for every ray. It is imporant to note that a ray can intersect the scene many times.

After surfel generation, there are a relatively large amount of surfels. It is

extremely inefficient to do intersection tests against all surfels. This means that a spatial data structure is needed to manage the surfels. Octrees are one of the most efficient spatial data structures that can be implemented. This implementation creates and octree by first creating a node and a bounding box that contains all surfels. The space contained in this node is then recursively split into eight equal size sections until each node contains 32 or less surfels. This octree is represented using a tree structure where nodes contain pointers to their sub-nodes. This is similar to Figure 3, where surfels are only stored at leaf nodes.

An octree comprised of pointers and nodes is relatively easy to create and works well for a CPU based implementation. Unfortunately, this structure is almost impossible to transfer to CUDA. Also this structure is extremely inefficient to use in CUDA. A CUDA efficient octree is a condensed octree that is has been flattened into an array structure. Essentially, all node in the octree that are interior nodes or leaf nodes with data, are stored in depth first order into an array. Leaf nodes instead of containing data contain indices into a common array containing all of the surfels. This structure allows for the minimum amount of data to be used to fully represent the octree. This also allows for one direction octree transversal. This means that to find an intersection it is only necessary to increment across the octree once. One directional octree transversal great reduces the seek cost that can be incurred by a global read in CUDA because you are only seeking forward. An example of an quadtree flattened into an array can be seen in Figure 6. A quadtree is the 2D equivalence of an octree.

Lastly, this project required a way of visualizing surfels to verify that they were correct as well as verify the transversal method was efficient. The method that was chosen was a simple ray tracing method that required rays to be cast from the camera into the scene and intersect the surfels. The direct illumination stored in the closest surfel was then drawn into a color buffer. Each ray was cast using CUDA parallelism that decreased the overall run time by a factor of 21. It was a simple method for drawing the surfels but it verified that the surfels were both correct and efficiently stored.

4 Results

The original plan for this project was to

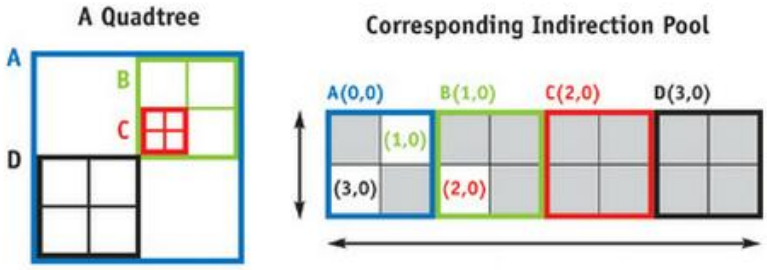

Figure 6: Quadtree compressed into an array. This is similar to how an octree is stored except that an octree is 3D not 2D.

decrease the time required to create surfels by using CUDA to generate the surfels on the GPU. This proved not be entirely necessary because the surfel generation phase was much shorter than expected. Instead work was done to efficiently represent the surfels in CUDA. The original non-octree, non-CUDA, implementation ran so slowly that timing unobtainable. It ran for over 2 hours for a 2000x2000 simple image. It also ran for 20 seconds to render a 100x100 image. The non-CUDA, octree implementation brought the total run time down to 5 minutes for a 2000x2000 image. The non-octree, CUDA implementation significantly improved performance bringing the total run time to 10 minutes. The final implementation utilizing both CUDA and the CUDA efficient octree runs in 14 seconds for a 2000x2000 image. This is a CUDA speedup factor of 21.

The rasterization step for Point Based Color Bleeding will require a similar amount of transversals of the CUDA octree so this speedup will most likely apply to the final implementation.

5 Future Work

This project was only the start of a much larger project. There is still much work to be done for the final project. First, the rasterization step of PCB needs to be written and adapted to use CUDA. Second, I want to try and implement a caching method that will hopefully speed up the octree transversals and surfel fetches.

Also, I want to continue working on speeding up the surfel creation phase for extremely large scenes. Currently, if a large scene is used, the surfel creation time can become relevant. It was not in the scope of this project to have large scenes but it is in the scope of the final project. One way of achieving this scene complexity would to utilize an octree for the scene objects as well as utilizing CUDA to do object intersections.

There are a few CUDA optimization tweaks that need to be researched as well. Work needs to be done to optimize the block and grid layouts as well as try and eliminate some of the branch divergence in the current implementation.

Finally, I believe that the new Nvidia Kepler architecture could greatly increase the speedup experienced by this project and the next. Unfortunately, this architecture is currently not release to the general public so development for this aspect of the project is on hold.

6 Conclusion

The Point Based Color Bleeding global illumination method is an extremely popular rendering method. The work done so far for this project shows that CUDA can greatly affect the run time of this algorithm. This initial project has seen great success exposing many problems and solutions that the final thesis project will experience and utilize. The CUDA efficient octree was a great success although there is still some

optimizations to implement. This has been a great first step for the final thesis project.

7 References

[1] Per H. Christensen. Point-based approximate color bleeding. 2008.

[2] Peter-Pike Sloan. Stupid spherical harmonics (SH) tricks. 2008. Microsoft Corporation.

[3] Christopher Gibson. Point-based color bleeding with volumes. 2011. Cal Poly San Luis Obispo

[4] *Sylvain Lefebvre, Samuel Hornus, Fabrice Neyret. GPU gems 2. Chapter 37. Octree Textures on the GPU. 2005.*

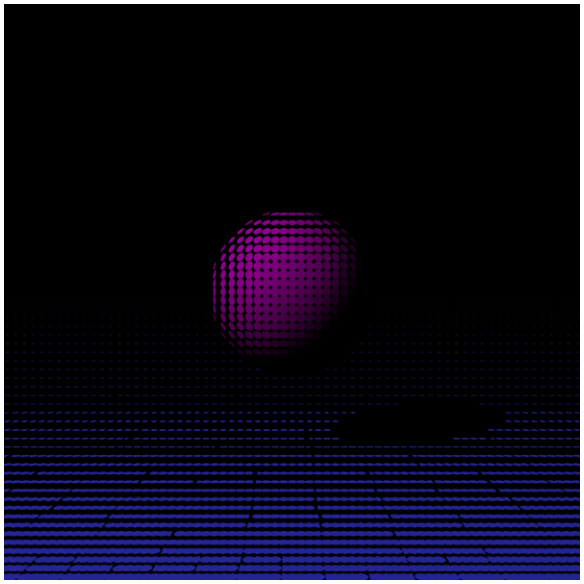

Figure 7: This is an extremely low surfel count image that is meant to prove that surfels are in fact created.

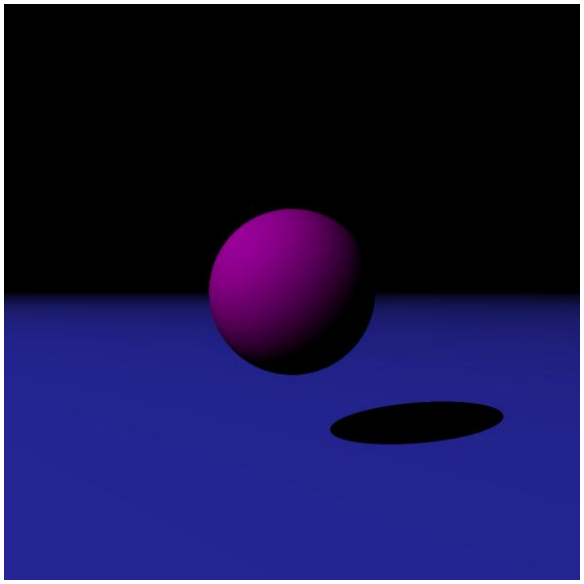

Figure 8: This is an example of an image with the correct number of surfels.