POINT BASED APPROXIMATE COLOR BLEEDING WITH CUDA

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Nicholas DeFay Feeney

June 2013

© 2013

Nicholas DeFay Feeney

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Point Based Approximate Color Bleeding
With CUDA

AUTHOR: Nicholas DeFay Feeney

DATE SUBMITTED: June 2013

COMMITTEE CHAIR: Zoë Wood, Ph.D., Computer Science De-
partment Professor

COMMITTEE MEMBER: Chris Lupo, Ph.D., Computer Science De-
partment Professor

COMMITTEE MEMBER: Aaron Keen, Ph.D., Computer Science De-
partment Professor

Abstract

Point Based Approximate Color Bleeding With CUDA

Nicholas DeFay Feeney

Simulating light is a very computationally expensive proposition. There are a wide variety of global illumination algorithms that are implemented and used by major motion picture companies to render interesting and believable scenes. Every algorithm strives to find a balance between speed and accuracy. The Point Based Approximate Color Bleeding algorithm is one of the most widely used algorithms in the field today.

The Point Based Approximate Color Bleeding(PBACB) global illumination algorithm is based on the central idea that the geometry and direct illumination of the scene can be approximated by using a point cloud representation. This point cloud representation can then be used to generate the indirect illumination. The most basic unit of the point cloud is a surfel. A surfel is a two dimensional circle in space that contains the direct illumination for that section of space. The surfels are gathered in a tree structure and approximations are generated for the different levels of the tree. This tree is then used to calculate the appropriate color bleeding effect to apply to the surfaces in a rendered image.

The main goal of this project was to explore the possibility of applying CUDA to the PBACB global illumination algorithm. CUDA is an extension of the C/C++ programming languages which allows for GPU parallel programming. In this paper, we present our GPU based implementation of the PBACB algorithm.

The PBACB algorithm involves three central steps, creation of a surfel point cloud, generation of the spherical harmonics approximations for the point cloud,

and using the surfel point cloud to generate an approximation for global illumination. For this project, CUDA was applied to two of the steps of the PBACB algorithm, the generation of the spherical harmonic representations and the application of the surfel point cloud to generate indirect illumination.

Our final GPU algorithm was able to obtain a 4.0 times speedup over our CPU version. We also discuss future work which could include the use of CUDA's Dynamic Parallelism and a stack free implementation which could increase the speedups seen by our algorithm.

Contents

List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Computer Graphics	1
1.2 Parallelization	2
1.3 CUDA	2
1.4 Ray Tracing	3
1.5 Direct Illumination vs. Indirect Illumination	4
1.6 Point Based Approximate Color Bleeding	5
1.7 Our Contribution	6
2 Related Work	8
2.1 Octree	8
2.2 Phong Model	9
2.3 Monte Carlo Integration	11
2.4 Monte Carlo Ray Tracing	11
2.5 Radiance	12
2.6 Spherical Harmonics	13
3 Point Based Approximate Color Bleeding Algorithm	17
3.1 Overview	17
3.2 First Stage: Point Cloud Generation	18
3.2.1 Surfel Generation	18
3.2.2 Octree Creation	19

3.2.3	Spherical Harmonic Approximation Calculations	20
3.3	Second Stage: Image Rendering	23
3.3.1	Direct Illumination	23
3.3.2	Rastercube Creation	24
3.3.3	Rasterizing the Scene to the Rascube	25
3.3.4	Integrating Over the Rastercube	30
4	GPU Implementations	33
4.1	Full Surfel Rasterization	33
4.1.1	Navie Full Surfel Rasterization	33
4.1.2	Shared Memory Full Surfel Rasterization	35
4.2	GPU Full PBACB Algorithm	37
4.3	GPU Octree Transversal Pseudo Code	39
5	Results and Conclusions	42
5.1	Test System	42
5.2	Speed Analysis	43
5.2.1	GPU Full Surfel Rasterization Algorithms Analysis	43
5.2.2	GPU PBACB Algorithm Analysis	44
5.3	Future Work	48
5.3.1	Dynamic Parallelism	48
5.3.2	Stack Free Implementation	49
5.3.3	Multiple GPU Implementations	49
5.3.4	Hybrid Solution or Pipelining	50
5.4	Conclusions	50
	Bibliography	52

List of Tables

5.1	This table shows the times seen for the GPU Full Rasterization version and the GPU Shared Memory Full Rasterization version compared to the CPU threaded Full Surfel Rasterization	44
5.2	This table shows the times seen for the CPU threaded version of the PBACB algorithm compared to the Non Threaded CPU times	46
5.3	This table shows how the times seen for the GPU version of the PBACB algorithm	46
5.4	This table shows how many times faster the GPU PBACB algorithm was to the CPU PBABC algorithm	46

List of Figures

1.1	Ray Tracing casts a ray into the scene to determine lighting for each pixel	4
1.2	Graphic to illustrate the differences between indirect illumination and direct illumination	5
2.1	Octree Graphic Example	9
2.2	This is a graphical representation of the Phong model showing the view vector v , the normal n , the light vector l , the point p , and the reflection vector h	10
2.3	Sampling Graphic Example	14
2.4	This graphics shows of an example of Spherical Harmonics being applied to model lighting seen by a sphere	15
2.5	This graphics shows of an example of Spherical Harmonics being applied to environment mapping	15
2.6	These are a graphical representation of the first nine spherical harmonic basis functions	16
3.1	To sample the scene we generate surfels rays that are cast from the camera and for each intersection made with the scene a surfel is generated	19
3.2	This is an under sampled rendering of the point cloud	20
3.3	These are the spherical harmonics basis functions used to calculate Spherical Harmonics	21
3.4	This is an example illustrating the rastercube with surfels being rasterized to a side	25
3.5	The three Rasterization methods	26

3.6	This is an example of the error seen when only the center of the octree nodes where used to evaluate the spherical harmonics . . .	31
5.1	This is an example output of our Cornell Box that we used for timings	43
5.2	This is figure is used to show the difference between our GPU output and CPU output	47
5.3	This is figure is another scene rendered with our PBACB algorithm	48

Chapter 1

Introduction

1.1 Computer Graphics

Computer Graphics is a general term used to describe any image creation or manipulation task performed by a computer. One of the most interesting topics of computer graphics, is that of rendering. Rendering is the computer process of generating images from models. Rendering is done for games, movies, and different artistic applications. The goal of rendering is to generate images that are either artistic or realistic. Most rendering algorithms tends to breakdown into a series of operations that are done repeatedly over every pixel of the rendered image. This repeating characteristic of rendering algorithms often attracts solutions that allow for parallelization.

1.2 Parallelization

Parallelization is the concept of performing many operations at the same time. Parallelization relies on hardware to allow for multiple operations to happen simultaneously and thus much faster. The two types of Parallelization open to programmers working on a single machine, are CPU(Central Processing Unit) parallelization, and GPU(Graphical Processing Unit) parallelization. CPU parallelization utilizes the multiple processing cores found in modern CPUs. There are usually anywhere from four to sixteen extremely fast processing cores found in modern CPUs. GPU parallelization utilizes the GPU's thousands of moderately fast processing cores. In NVIDIA's new Tesla K20X GPU, for instance, there are 2688 processing cores capable of doing parallel operation. When used effectively, GPU parallelization can be many times faster the CPU parallelization.

1.3 CUDA

CUDA is a an extension of the C/C++ programming languages which allows for GPU parallel programming. CUDA is a product of NVIDIA and requires an NVIDIA GPU. CUDA works primarily on a thread, block, and grid system. A grid is a two dimensional array of blocks and a block is a three dimensional array of threads. This system allows for direct control of the GPU and its various components.

There are two major concerns to keep in mind when programming for the GPU, branching and memory. In a GPU, threads are bound together in small groups called warps. Warps are threads that are all executing the same exact instruction on different sets of data. Threads in a warp cannot execute different

instructions. This constraint leads to the first major concern, branching. Branching is the speed penalty attained when threads in a warp go down different paths of execution. What essentially happens is every thread travels down both paths of execution, but when a thread is on the wrong path the thread is ideal. The second concern is memory. The GPU has its own finite memory. This means there is a cost to moving memory over to the GPU and taking memory off the GPU. Usually, this cost is significant but worthwhile because of the major speedups the GPU provides.

These are the two most important concerns about GPU parallelization that relate to this paper. When programming for the GPU it is also important to keep in mind topics like, memory access patterns, memory usage, block and grid dimensions, shared and global memory usage, and many more things. If more information is required see NVIDIA's CUDA Zone [9].

1.4 Ray Tracing

Ray Tracing is one of the most basic yet adaptable rendering algorithms. The general idea behind the ray tracing algorithm is ray casting. In ray tracing, a vision ray is generated for each pixel in the final rendered image. These rays represent the inverse path light travels in a simulated scene. The rays are individually cast into the scene. For the closest intersection, that a vision ray makes with the scene, lighting is calculated and then stored into the corresponding pixel. When every ray has been cast and all lighting calculated the image has been rendered. Figure 1.1 shows a simple ray tracing scene.

Figure 1.1: Ray Tracing casts a ray into the scene to determine lighting for each pixel. Ray Tracing can be modified by many additional algorithms. This examples show the inclusion of shadow rays which determine if an intersection is in a shadow [7].

1.5 Direct Illumination vs. Indirect Illumination

There are two types of illumination in global illumination, direct and indirect. Direct illumination represents the light coming from the primary light sources in the scene. Direct illumination, in its simplest form, can be calculated using only scene data. That means that direct illumination only needs scene geometry and lighting information to be calculated. Direct illumination, in its simplest form, is relatively simple to calculate and does not require much computation in comparison to indirect illumination. The simple ray tracing example given in Section 1.4 is an example of rendering using only direct illumination.

Direct illumination is only half of any real rendering algorithm. Indirect illumination represents the ambient light bouncing off of all the surfaces in the scene. Light does not only come directly from primary light sources but it also bounces

Figure 1.2: Graphic to illustrate the differences between indirect illumination and direct illumination. Left is a rendered image using only direct illumination. Right is an image rendered using direct and indirect illumination.

off other objects in the scene. Indirect illumination is extremely computationally expensive. When computing indirect illumination, every point in the scene is technically its own light source so there are an infinite amount of light sources which have the possibility of reflexing light onto any receiving point. To simulate indirect illumination exactly, requires using the rendering equation to integrate over all incoming light seen by the receiving point [8]. To exactly integrate over the hemisphere of incoming light requires an infinite amount of processing. Since computers cannot process an infinite amount of data, this has lead to the development of many indirect illumination approximation algorithms. Figure 1.2 show the difference between a scene with and without indirect illumination.

1.6 Point Based Approximate Color Bleeding

The Point Based Approximate Color Bleeding(PBACB) global illumination algorithm was developed by Per H. Christensen at Pixar [2]. The algorithm was

designed to calculate indirect illumination much faster than previously done. The algorithm utilizes a two pass method to calculate indirect illumination. The first pass generates a point cloud representation of the scene which stores the direct illumination in elements known as surfels. A surfel is a two dimensional circle that is represented by a position, a normal, a radius, and a color. The whole scene is broken down into these surfels. Then the surfels are sorted into an octree(see related works). Next, spherical harmonics representations of the light emitted from, and projected area of, the nodes of the octree are calculated and stored.

The second pass actually calculates the indirect illumination for a given surface in the scene. To calculate the indirect illumination for a given point on a surface a rastercube or hemicube is generated at the given point and the point cloud is rasterized to the sides of the cube. This creates a very simplified representation of all the incoming light received by the given point. Using that information the indirect illumination can be simply calculated by integrating over the incoming light. This whole process is explained in more detail in Chapter 3.

1.7 Our Contribution

The goal of this project was to assess the possible speedups that could be obtained by incorporating CUDA into the Point Based Approximate Color Bleeding algorithm. We developed a number of GPU based algorithms to test the possible speedups which could be attained. In the end we found that the PBACB algorithm has inherent characteristics that do not port perfectly to the GPU. However, we were still able to get around a 4.0 times speedup over the CPU version. It is also important to note that the direct speedups obtained from us-

ing the GPU are not the only possible benefits. Utilizing the GPU to calculate indirect illumination frees up the CPU to do other calculations while the GPU is busy. In our opinion, inclusion of GPU parallelism is a great improvement over a strictly CPU version of the Point Based Approximate Color Bleeding algorithm.

Chapter 2

Related Work

This project builds on several important concepts from computer graphics related to, spacial data structures, radiance, spherical harmonics, and other indirect illumination algorithms. A basic understanding of these topics will help to understand the coming chapters so it is presented here.

2.1 Octree

An octree is a hierarchical spatial data structure in which every node in the tree except for the leafs contains exactly eight children [15]. An octree groups objects together based on spatial locality which allows quaries to eliminate large section of the octree for spatial reasons. Octrees can greatly reduce the qurry time for rendering algorithms. Octrees are typically generated by recursively splitting nodes into eight equal sized children until a threshold is reached. Figure 2.1 show an example of an octree.

Figure 2.1: Octree Graphic Example. Left is an example showing the tree structure of an octree. Right is a graphical example of an octree applied to a bunny model [17, 15].

2.2 Phong Model

The phong model is a numerical model of the direct illumination of a surface [10]. The model describes how light is reflected off a surface through its three central light components, ambient, diffuse, and specular. The phong model is represented by the following equation [10, 18]:

$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (L_m \cdot n) i_{m,d} + k_s (R_m \cdot V)^\alpha i_{m,s}) \quad (2.1)$$

- I_p is direct illumination for a given point
- k_a is the ambient reflection constant for the surface which the point resides
- k_d is the diffuse reflection constant for the surface which the point resides
- k_s is the specular reflection constant for the surface which the point resides
- m is the current light source
- i_a is the ambient light constant for the scene
- $i_{m,d}$ is the diffuse component of the light source m

Figure 2.2: This is a graphical representation of the Phong model showing the view vector v , the normal n , the light vector l , the point p , and the reflection vector h [16].

- $i_{m,s}$ is the specular component of the light source m
- L_m is the direction vector from the point to the light source m
- n is the normal of the surface for which the point resides
- α is a shininess constant for the surface which the point resides
- R_m is the vector given by $R_m = 2(L_m \cdot n)n - L_m$
- V is the direction between the point and the camera

The three components of this model are used to represent different forms of light that are present in all scenes. The ambient component is an approximation for the indirect illumination for a scene. It is an extremely coarse representation of indirect illumination because it is essentially just a constant for all points. The diffuse component models the light scattered by the receiving point. Not all light is reflexed perfectly, some is scattered in a hemisphere over the receiving point. The final component, specular, models the shininess of a surface by reflecting incoming light more heavily along the reflection vector. A graphical example of the Phong model is shown in Figure 2.2.

2.3 Monte Carlo Integration

Monte Carlo Integration is a numeric sampling method that is used to estimate the value of integrals. It is based around the central idea that an integral can be approximated by randomly sampling the function and summing the result. The main equation for Monte Carlo Integration is [5]:

$$\int f(x_i)ds = \frac{1}{N} \sum_{j=1}^N f(x_i)w(x_i) \quad (2.2)$$

In this equation, $f(x_i)$ is the function you are integrating over, N is the number of samples which you are taking of the function, and $w(x_i)$ is the weighting function that is being used. Monte Carlo Integration can vary in accuracy by changing the number of samples that are taken of the function. Perfect accuracy requires an infinite amount of samples.

2.4 Monte Carlo Ray Tracing

Monte Carlo Ray Tracing is one of the more popular indirect illumination algorithms because it is an extension of ray tracing. Monte Carlo Ray Tracing uses simple ray tracing to calculate the direct illumination for all pixels. However, it also includes a secondary ray cast for every pixel to calculate the indirect illumination. As stated before, to calculate indirect illumination, for a given point, the algorithm must integrate over the hemisphere of incoming light. Monte Carlo Ray Tracing uses Monte Carlo Integration to numerically estimate the integral by casting Monte Carlo sampling rays into the hemisphere above the receiving point.

Usually it requires about 256 Monte Carlo sampling rays to accurately calcu-

late the first bounce of indirect illumination, for a single pixel. This means that for a 1024x1024 size image there are 1,048,576 primary vision rays used to calculate direct illumination as well as 268,435,456 Monte Carlo rays used to calculate the indirect illumination. Most Monte Carlo implementations however do not stop at the first bounce of light. This means that there is another 256 rays per ray for the second bounce, or 68,719,476,736 Monte Carlo Rays. As you can see Monte Carlo Ray Tracing can quickly becomes overwhelming the more bounces that are simulated.

2.5 Radiance

Radiance is the measure of energy emitted from or passing through a surface in a specific direction. Radiance in terms of rendering is used to measure the power and color of a single ray of light [16, 13]. Radiance is defined by the equation:

$$L = \frac{d^2\phi}{dA \cos\theta dw} \quad (2.3)$$

Where ϕ is radiant flux or the measure of energy per second and $\frac{d\phi}{dA}$ is flux density, the instantaneous transfer of radiant flux of an area. This terms can now let us define the term irradiance or the incoming radiance seen by a surface point. To define the irradiance of a given point x from a specific direction w' we use the equation:

$$E(x, w') = L(x, w') \cos\theta dw' \quad (2.4)$$

Now that we have an equation for the irradiance of a given point and direction we can apply it to all possible direction. This means that we integrate this

function over the hemisphere above the receiving point giving the equation:

$$E(x) = \int L(x, w') \cos\theta dw' \quad (2.5)$$

Equation 2.5 allows us to calculate the total radiance of a point. To use radiance in rendering application we need radiance emitted from the receiving point not the total radiance. This means that we need to take the total radiance equation and augment it by the surface characteristics of the receiving point. This leads us to the last term, the Bi-direction Reflectance Distribution Function(BRDF). The BRDF is a function that is used to model surface materials and is represented by $f(x, w, w')$. The BRDF is a function that converts incoming radiance from a given direction w' to emitted radiance in the direction w .

Adding the BRDF to the Equation 2.5 gives us our final equation for reflected radiance:

$$E(x, w) = \int f(x, w, w') L(x, w') \cos\theta dw' \quad (2.6)$$

2.6 Spherical Harmonics

Spherical harmonics are the solutions to Laplace's equation when restricted to the sphere. The benefit of modeling a function using spherical harmonics is that spherical harmonics have the ability to model complex functions in very compact forms. Spherical harmonics are essentially Fourier basis functions applied to a sphere. This means that spherical harmonics are excellent at modeling functions that can be represented by a sphere structure. Spherical harmonics have many different applications including, modeling heat transfer in physics, modeling electron configurations in chemistry, and modeling lighting in computer graphics.

The general process behind calculating a set of spherical harmonics can be bro-

Figure 2.3: Sampling Graphic Example. Left shows a two dementional sample grid. Right show the sample grid restricted to a sphere [5].

ken down into three parts. First, a set of sample points are created, like seen in Figure 2.3. Then the sample points are constricted to a sphere also seen in Figure 2.3. Then, a set of spherical harmonic coefficients are calculated by sampling the function that is being model using the sample points that are constricted to a sphere. Figure 2.4 shows the resulting spherical harmonic coefficients if this process is applied to modeling the light received by a sphere.

Spherical harmonics are comprised of basis functions which are small independent functions that can be scaled and combined to form an approximation of the original function [5]. Varying the number of spherical harmonic basis functions included in the spherical harmonic representation varies the accuracy of the representation. An example of high accuracy coefficients can be seen in Figure 2.5, which is an environment map of a complex scene. This example is meant to demonstrate the accuracy that can be achieved by spherical harmonics.

We use spherical harmonics in this algorithm as a way of approximating light emitted from, and projected area of, a spacial region. We utilize only the first nine spherical harmonics basis functions in our representation because it allows us to have sufficient accuracy with a very small amount of coefficients. These nine

Figure 2.4: This graphics shows of an example of Spherical Harmonics being applied to model lighting seen by a sphere. On the left, the lighting of this sphere is reconstructed from sixteen Spherical Harmonic coefficients. On the right, is a spherical plot representation of the same Spherical Harmonic coefficients[5].

Figure 2.5: This graphics shows of an example of Spherical Harmonics being applied to environment mapping. This example is meant to demonstrate the accuracy possible using Spherical Harmonics [5].

Figure 2.6: These are a graphical representation of the first nine spherical harmonic basis functions that we used in our spherical harmonic representations [14].

basis functions are graphically shown in Figure 2.6. The ability to approximate light emitted from a spacial region allows a reduction in the required calculations for indirect illumination. A more detailed explanation of how spherical harmonics are used in this application is presented in later sections.

Chapter 3

Point Based Approximate Color Bleeding Algorithm

This thesis is based on the original PBACB algorithm written by Per H. Christensen [2]. This chapter will explain that algorithm in detail and how we implemented slight changes to replicate his work. The PBACB algorithm is a two pass method that creates a point cloud approximation of the direct illumination and then uses that approximation to calculate the indirect illumination quickly. The combination of the direct illumination and the indirect illumination generates a final rendered image.

3.1 Overview

The PBACB algorithm is composed of two central stages which are each comprised of multiple parts:

First Stage: Point Cloud Generation

1. Surfel Generation
2. Octree Creation
3. Spherical Harmonic Approximation Calculation

Second Stage: Image Rendering

1. Direct Illumination Calculation
2. Rastercube Creation
3. Rasterization of the Point Cloud
4. Integration of the Rastercube

These stages are described in more detail below.

3.2 First Stage: Point Cloud Generation

3.2.1 Surfel Generation

The PBACB algorithm utilizes a point cloud representation of the scene to greatly reduce the work involved in calculating indirect illumination. To generate this point cloud the original algorithm suggests using a REYES based approach [2]. This approach essentially dissects every object in the scene into micro-polygons [12]. Direct illumination is then calculated for every micro-polygon and then the micro-polygons are then converted into surfels. The REYES based approach guaranties that all of the surfaces in the scene are fully covered. Having a scene that is fully covered is important for the direct illumination step of second pass.

Figure 3.1: To sample the scene we generate surfels rays that are cast from the camera and for each intersection made with the scene a surfel is generated[4].

Our implementation, utilizes a ray casting method for sampling the scene. The scene is sampled much like a ray tracing algorithm would sample for direct illumination. The primary difference is that instead of using just the closest intersection that a sampling rays makes, all intersections are used to sample the scene. Figure 3.1 demonstrates this process. This method is much simpler to implement but does not guarantee a complete covering of all surfaces. We saw that with minor adjustments to the direct illumination step in the second pass a complicated REYES based approach was not necessary for our needs.

3.2.2 Octree Creation

Once we have the surfels generated, we organize them into an octree representation by recursively splitting each node until a small number of surfels are

Figure 3.2: This is an under sampled rendering of the point cloud. The surfels are exaggerated and the sampling rate is low for effect. An actual sampling would have far fewer gaps between surfels.

in each leaf. We saw that having a maximum of 16 or 32 Surfels in each leaf provided the best results. We use a spacial data structure so that we can group surfels together based on what region of space they are in. After we have these groupings, we can form approximations for the light emitted by these regions of space. These approximations can then be use instead of the actual surfels which can greatly reduce the workload.

3.2.3 Spherical Harmonic Approximation Calculations

After the octree is created we calculate the spherical harmonic approximation for the power emitted from each node as well as the projected area of each node. These approximaitons will be used later in the algorithm in place some of the acutal surfels. These approximations will greatly reduce the number of individ-

$$\begin{aligned}
Y_{00}(\theta, \phi) &= 0.282095 \\
(Y_{11}; Y_{10}; Y_{1-1})(\theta, \phi) &= 0.488603 (x; z; y) \\
(Y_{21}; Y_{2-1}; Y_{2-2})(\theta, \phi) &= 1.092548 (xz; yz; xy) \\
Y_{20}(\theta, \phi) &= 0.315392 (3z^2 - 1) \\
Y_{22}(\theta, \phi) &= 0.546274 (x^2 - y^2)
\end{aligned}$$

Figure 3.3: These are the spherical harmonics basis functions used to calculate Spherical Harmonics. x, y, z are the components the d vector, $(\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\theta))$ [11]

ual surfels that will be actually need to be rasterized. The spherical harmonic representation for the power emitted and projected area of a single Surfel is give by the equations [2]:

$$p_{lm} = \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\pi} B_i A_i (d \cdot n_i)_+ Y_{lm}(\theta, \phi) \sin\theta d\theta d\phi \quad (3.1)$$

$$a_{lm} = \int_{\phi=0}^{2\pi} \int_{\theta=0}^{\pi} A_i (d \cdot n_i)_+ Y_{lm}(\theta, \phi) \sin\theta d\theta d\phi \quad (3.2)$$

- B_i is the RGB Radiosity of the Surfel
- A_i is the area of the Surfel
- n_i is the normal of the Surfel
- θ and ϕ are the spherical coordinates(the polar and azimuth angles)
- d is the vector $(\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\theta))$
- $(x)_+$ is short hand for $\max(x, 0)$
- Y_{lm} are the spherical harmonics basis functions listed in Figure 3.3 [11].

As the original algorithm suggests, we used the first nine harmonics which gave sufficient accuracy [2]. To fill out the spherical harmonics of an octree leaf, we simply calculate the spherical harmonic coefficients for each surfel in the node and

then added the coefficients together [2, 14]. To calculate the spherical harmonics of a non leaf node we simply add together the spherical harmonic coefficients of each of the eight children [2, 14]. Early on in the creation of our algorithm we designed and implemented a GPU kernel that calculated the spherical harmonics of all of the surfels. This kernel is a trivial kernel that we used to reduce the total runtime of our algorithm. The speedups over the CPU version are considerable but when compared to the runtime of the indirect illumination it is not significant.

Monte Carlo Integration Math

In practice, we use Monte Carlo Integration to evaluate the spherical harmonic integrals for a single surfel. Monte Carlo Integration is a numerical estimation method based on Equation 2.2. Applying the estimation equation to the double integral for power emitted by a surfel, given by Equation 3.1, gives us:

$$f(x_i) = B_i A_i (d \cdot n_n)_+ Y_{lm}(\theta, \phi) \quad (3.3)$$

And because we want our samples to lie evenly across the sphere that we are integrating over our weighting function is [5]:

$$w(x_i) = \frac{1}{p(x_i)}, p(x_i) = \frac{1}{4\pi}, w(x_i) = 4\pi \quad (3.4)$$

This gives the final equation:

$$p_{lm} = \frac{4\pi}{N} \sum_{j=1}^N B_i A_i (d \cdot n_i)_+ Y_{lm}(\theta, \phi) \quad (3.5)$$

where theta and phi are weighted random samples points calculated from:

$$\theta = 2\arccos(1 - \xi_x), \phi = 2\pi\xi_y, \xi_x = 0 \rightarrow 1, \xi_y = 0 \rightarrow 1 \quad (3.6)$$

Using Equation 3.5 we easily constructed an algorithm which could numerically calculate the spherical harmonics of a surfel which is shown in the next section. The area equation is identical to the power equation except does not include the RGB radiosity component.

Monte Carlo Integration Pseudo Code

```

for each sample point 1 -> N
{
  x,y = weighted random numbers 0->1
  phi = 2.0 * PI * y;
  theta = 2.0 * acos( sqrt( 1.0 - x ) );
  vector d = (sin(theta)*cos(phi), sin(theta)*sin(phi),
              cos(theta));
  d_dot_n = dotproduct(d, surfel.normal)
  if( d_dot_n > 0 )
  {
    Ylm = calculate spherical basis functions using d
    for each spherical harmonic coefficient
    {
      power += area * d_dot_n * Ylm * surfel.color
      area += area * d_dot_n * Ylm
    }
  }
}

```

3.3 Second Stage: Image Rendering

3.3.1 Direct Illumination

Once the point cloud representation is completed then the rendering of the actual image can begin. The first step in this process is to calculate the direct illumination for each pixel. If the point cloud representation fully covers all surfaces

in the scene then it is possible to simply cast vision rays into the point cloud and use the direct illumination stored in the point cloud. Our implementation does not guarantee full coverage of the scene. This means that we must cast vision rays into the original scene and recalculate the direct illumination. This is inefficient, however the total time to calculate direct illumination is much smaller than the time to calculate indirect illumination. The primary point of this project was to examine the indirect illumination calculation so this time penalty was deemed acceptable.

3.3.2 Rastercube Creation

Indirect illumination is the sum of all incoming light a point receives that does not originate directly from a light source. The point cloud representation is an approximation of the light emitted from all of the surfaces in the scene. Unfortunately, far too much light would be gathered if every surfel or octree node contributed to the indirect illumination of every point. Objects that are hidden behind other objects need to be excluded from the calculations since they do not radiate light onto the point in question.

The PBACB algorithm approximates the light that a point receives through the use of a rastercube. A rastercube is an axis aligned cube on which the entire point cloud is rasterized. The idea behind the rastercube is that if the point cloud is roughly rasterized to the sides of the cube then an approximation of the incoming light for the receiving point is generated. Using this approximation it is then possible to calculate a estimation of the indirect illumination.

For our algorithm, a rastercube contains six axis aligned sides with 8x8 pixels and a depth buffer per side. We generated a rastercube for each vision ray and

Figure 3.4: This is an example illustrating the rastercube with surfels being rasterized to a side [2].

center it at the intersection that ray made with the scene.

3.3.3 Rasterizing the Scene to the Rascube

To rasterize the point cloud to the sides of a rastercube we recursively iterate over the entire point cloud octree. Like the original PBACB algorithm, we use three methods to rasterize a surfel to the sides of a rastercube, Near-Field Ray Tracing, Mid-Field Individual Rasterization, and Far-Field Cluster Rasterization. We use the solid angle of the octree nodes to determine which field surfels belong to. Solid angle is a ratio of the projected area of an octree node versus the distance to the receiving point squared. The maximum solid angle, which is defined by the user, defines how accurately to render the image. Increasing the solid angle increase the approximations that are used. Pseudo code for the octree traversal is given below.

Figure 3.5: The three Rasterization methods: Near-Field Ray Tracing(right). Mid-Field Individual Surfel Rasterization(middle). Far-Field Cluster Rasterization(left) [2].

Octree Traversal Pseudo Code

```

traverseOctree( node , position , normal )
{
    if( node is a leaf ){
        for each surfel in node
        {
            if( distance( surfel.pos , position ) < surfel.radius )
                raytrace surfel to cube
            else
                rasterize surfel to cube
        }
    } else if ( position is in node bounding box ) {
        for each child node
            traverseOctree( child , position , normal )
    } else if ( node bounding box below horizon ) {
        return;
    }
    else {
        area = Eval Spherical Harmonics Area
        solidangle = area / distance( position , node.center )
        if( solidangle < MAX_SOLID_ANGLE ) {
            clusterPower = Eval Spherical Harmonics Power
            rasterize cluster to cube
        }
        else
            for each child node
                traverseOctree( child , position , normal )
    }
}

```

Near-Field: Ray Trace Surfels

If a surfel is less than a surfel's radius away from the receiving point, a ray tracing method is used to project the surfels onto the rastercube. A ray for each pixel in the rastercube, that is above the horizon, is generated and a surfel intersection test is performed to determine if each raster pixel should be filled in. This method is necessary because it accurately determines distance. This is important because multiple surfels can overlap at the receiving point and if accuracy is low then too many pixels will be filled in, which skews the results at sharp edges. This method is rather slow in comparison to the mid-field method but there are so few times ray tracing is needed it is not an issue.

Mid-Field: Rasterize Surfels

Surfels that are close to the receiving point but farther than their respective radii, are rasterized individually to the sides of the rastercube. We determine if a surfel is close to the receiving point by using the solid angle of octree nodes. The mid-field and near-field methods are how octree leaf nodes are rasterized to the rastercube. The octree traversal pseudo code above, shows this process in more detail. To rasterize a surfel, we first determine if the surfel is above the horizon of the receiving point. If it is, we then project the area of the surfel to the six sides of the rastercube. The estimated projected area of a surfel to a side is given by: $p_a = A_i(n_i \cdot c_i)(w_s \cdot c_i)$

- A_i is the Area of the surfel
- n_i is the normal of the Surfel
- c_i is the connecting vector between the receiving point and the surfel position

- w_s is the depth axis of the side of the cube you are projecting onto

Once we have the projected areas of the surfel, we rasterize the surfel to each side of the ratercube. To rasterize the surfel to a single side of the ratercube, we replace the surfel with an axis alined square with the calculated projected area for the given side and rasterize. This is a crude approximate for the surfel but because of the small number of pixels in the ratercube the lack of accuracy does not effect the final result in a significant way. This is also an extremely fast method for rasterize a surfel to the ratercube.

Far-field: Surfel Clusters

Surfels that are far away from the receiving point are clustered together and rasterized. To do this, we utilize the spherical harmonic representations stored in the various levels of the point cloud octree.

First we calculate the projected area and power of the surfel cluster from the spherical harmonics. Then, we rasterize an axis aligned square using the same method as the mid field surfels. The difference being that the power and area of this axis aligned square represents a cluster of surfels as apposed to a single surfel.

Evaluating Spherical Harmonics

To evaluate a set of spherical harmonics for a given direction you mush take the weighted sum of the basis functions. The evaluation of a set of spherical

harmonics for a given direction is given by the function [2, 5]:

$$P(\theta, \phi) = \sum_{l=0}^2 \sum_{m=-l}^l p_{lm} Y_{lm}(\theta, \phi) \quad (3.7)$$

This function translates in to the provided pseudo code in the next section.

Evaluating Spherical Harmonics Pseudo Code

```

evaluateSphericalHarmonicsPower( node, direction )
{
    dx = direction.x
    dy = direction.y
    dz = direction.z
    Ylm[0] = 0.282095;
    Ylm[1] = .488603 * -dy;
    Ylm[2] = .488603 * dz;
    Ylm[3] = .488603 * -dx;
    Ylm[4] = 1.092548 * dx * dy;
    Ylm[5] = 1.092548 * -dy * dz;
    Ylm[6] = 0.315392 * (3*dz*dz - 1);
    Ylm[7] = 1.092548 * -dx * dz;
    Ylm[8] = .546274 * (dx*dx - dy*dy);

    color = 0
    for each spherical harmonic coefficient
    {
        color.r += node.harmonics.red[i] * Ylm[i];
        color.g += node.harmonics.green[i] * Ylm[i];
        color.b += node.harmonics.blue[i] * Ylm[i];
    }
    return color
}

```

It was found that evaluating the spherical harmonics for projected area of a node had a slight complication. We found that for flat surfaces the octree did a poor job at accurately representing the projected area of close nodes. This was

caused by the fact that the center of the octree node was offset from the actual position of the flat surface. The center of the node was what we used originally to calculate the direction vector for evaluate the spherical harmonics. The incorrect direction vector, which was caused by the offset, resulted in a much smaller area value then there sould have been. To solve this problem we simply evaluated the projected area using the direction to all the corners and the center of the node and took the maximum. This change meant that for far away nodes the nine direction vectors we used would vary slightly but for close nodes the direction vectors could vary widely. The effect of this change essentially insured that close octree nodes would not be thrown away too early in the algorithm because the center of the node did not accurately represent the location of the surface. This does not negatively effect the resulting image because if an octree node is greatly effected by the different possible directions vectors then the solid angle will insure that the children are used which is the proper solution. An example of this error is show in Figure [3.6](#).

3.3.4 Integrating Over the Rastercube

After the point cloud has been rasterized to the rastercube it is time to approximate the indirect illumination for the receiving point. To do this we must integrate over the rastercube pixels. Our algorithm simulates diffuse surfaces so we scale each pixel by the dot product of the normal and the pixel direction. The pseudo-code is shown in the next section.

Figure 3.6: This is an example of the error seen when only the center of the octree nodes were used to evaluate the spherical harmonics. You can see it clearly where the back wall and the right wall meet. It looks like strange patterns on the back wall.

Rastercube integration Pseudo Code

```
integrateRastercube( cube )
{
    color = 0
    s = (linear distance component)
    for each pixel in the rastercube
    {
        dotProd = dot( pixel direction , normal );
        color += cube.pixelcolor * dotProd /
                (s + cube.pixeldepth^2)
    }
    pixels = (brightness effect) / # pixels above horizon
    color /= pixels
}
```

In this model, we found it helpful to include to variables shown in the above pseudo code as *linear distance component* and *brightness effect*. The *linear distance component* controls the effect distance has on the color bleeding effect. If a linear component is not added to the distance attenuation calculations for the pixels then extremely close pixels can drastically skew the resulting image. We found that a good value for *linear distance component* was 1.0. The *brightness effect* was included to counteract some of the effect that this *linear distance component* had on distant pixels. We found that a good value of *brightness effect* was 3.0.

After the direct illumination has been calculated via ray casting and after the indirect illumination has been calculated via the rastercubes, the combination of the two creates the final rendered image which can be seen in Figure 5.1. Figure 5.1 demonstrates nice global illumination effects such as color bleeding that can be seen in all of our result images. This project began with a CPU implementation of the PBACB algorithm and we then moved on to a parallelized GPU version, described in the next chapter.

Chapter 4

GPU Implementations

The GPU is a powerful tool that can add impressive levels of parallelism to an existing algorithm. We used CUDA, NVIDIA’s extension to the C/C++ programming languages, to add GPU parallelism to the PBACB global illumination algorithm. This chapter talks about the different algorithms that we created. We also discuss the characteristics of the PBACB algorithm which presented difficulties for us when we were adding GPU parallelism.

4.1 Full Surfel Rasterization

4.1.1 Navie Full Surfel Rasterization

The first GPU implementation that we tried was the simplest possible solution. We simplified the rasterization algorithm, by rasterizing all surfels as if they fell into the mid-field distance range. This meant that every surfel needed to be rasterized to the rastercubes. The first step in this algorithm was to collect all of the intersections that the primary vision rays made. For each intersection,

we recorded the position and normal. We then designed a CUDA kernel that parallelized across all of the intersection points. This meant that every thread on the GPU was responsible for the indirect illumination calculation for a single pixel.

The original naive implementation gave poor results, orders of magnitude slower than the CPU version. In CUDA, resources are shared between threads. If a thread uses too many resources it will limit the number of threads that can run in parallel. Limiting of threads that can run in parallel, causes an inefficient utilization of the GPU. We found that register usage for each thread was 63 which meant our kernel was running very few threads in parallel. The problem was compounded by the number of times global memory was accessed. Global memory accesses are often masked by switching to other threads and executing calculations until the memory accesses are complete. The effect of having very few threads running in parallel meant that most of the memory accesses the kernel made were not being masked.

We were however able to fix the register problem. In CUDA, it is possible to limit the number of registers used by a thread. Limiting the number of registers that a thread uses however increases memory swaps. We found that limiting the registers to 32, was well worth the swap time. After we limited the register usage we were able to render an image that was almost identical to our CPU version in much less time. For a scene containing 1 million surfels, the CPU version ran in 2020.08 seconds where our GPU version ran in 592.345 seconds. More statistics are shown in Table 5.1 and pseudo code for this algorithm is shown in the next section.

Full Surfel Rasterization Pseudo Code

The following is the pseudo code for the GPU kernel that rasterized every surfel to every rastercube. This kernel utilized a parallelization scheme across the pixels of the final image. The variables `blockIdx`, `threadIdx`, and `blockDim` are variables that are defined for CUDA to distinguish threads from each other.

```
--global-- void kernel_SurfelRasterization( surfels ,
      num_surfels , positions , normals , out_buffer ,
      num_pixels )
{
    index = blockIdx.x * blockDim.x + threadIdx.x;
    if( index >= num_pixels )
        return;
    position = positions[index];
    normal = normals[index];
    cube = init Rastercube;
    for( i = 0; i < num_surfels; i++ )
    {
        current = surfels[i];
        rasterize current to cube
    }
    color = integrate over cube to get color bleed

    out_buffer[index] = color
}
```

4.1.2 Shared Memory Full Surfel Rasterization

The next implementation that we tried rasterized every surfel to the rastercubes, like the first algorithm, but this algorithm utilized CUDA's shared memory to speedup the memory accesses. CUDA has three types of memory that a thread can utilize, global memory, shared memory, and local memory. Global memory is the slowest memory cache, but it is the largest being around 1 gigabyte. Global

memory is also the memory cache that the CPU can write to. The size of global memory is dependent on the specific GPU that being used. Shared memory is a memory cache that is shared across a single block. Shared memory is much faster then global memory but is much smaller. Shared memory can be 16 kilobytes or 48 kilobytes in size. Local memory is the fastest memory but it is on the thread's stack and is not shared between threads.

To utilize shared memory, our algorithm shuffled surfels from global memory into a shared memory buffer. It then used the surfels in the shared memory to rasterize before pulling another batch of surfels into the same shared memory buffer. Each thread was responsible for moving only one surfel per batch into the shared memory buffer. This meant that every thread reduced the number of global memory reads by a factor of the number of threads in a block. We found that with 512 threads in a block we had the best results. Pseudo Code for this algorithm is shown below.

Shared Memory Pseudo Code

The following is the pseudo code for the GPU kernel that rasterized every surfel to every ratercube and utilizes shared memory. This kernel utilized a parallelization scheme across the pixels of the final image. The variables `blockIdx`, `threadIdx`, and `blockDim` are variables that are defined for CUDA to distinguish threads from each other. `THREADS` is a define that specifies the number of threads in a single block. This define is necessary so that the compiler knows how big to make the shared memory buffer at compile time.

```

__global__ void kernel_SurfelRasterization( surfels ,
    num_surfels , positions , normals ,
    out_buffer , num_pixels )
{
    index = blockIdx.x * blockDim.x + threadIdx.x;
    if( index >= num_pixels )
        return;
    position = positions[index];
    normal = normals[index];
    cube = init Rastercube;
    __shared__ Surfel s_block[THREADS];
    loop_count = ceil( num_surfels / THREADS );
    for( i = 0; i < loop_count; i++ )
    {
        s_index = i*THREADS + threadIdx.x;
        if( s_index < num_surfels )
            s_block[threadIdx.x] = surfels[s_index];
        __syncthreads();

        for( int j = 0; j < THREADS; j++ )
        {
            current = s_block[j];
            rasterize current to cube
        }
        __syncthreads();
    }
    color = integrate over cube to get color bleed

    out_buffer[index] = color
}

```

4.2 GPU Full PBACB Algorithm

The final GPU algorithm that we designed implemented the full Point Based Approximate Color Bleeding algorithm on the GPU. We essentially ported the algorithm directly over to the GPU. The major change that we had to make was to modify the octree traversal code to use a stack method as opposed to a recursive

method. Utilizing the same pixel parallelization method as with the previous to GPU algorithms we were able to fully implement the PBACB algorithm on the GPU.

There are some important notes about this algorithm however that are not immediately apparent. First, the octree representation used on the GPU is significantly different than the standard octree used in CPU code. The octree on the GPU was implemented with an array structure rather than the standard pointer method. This was necessary for two reasons. First, it is impossible to move the standard pointer octree over to the GPU because only arrays can be moved to the GPU. Second, the array based version of the GPU octree provided a beneficial memory access pattern which helps to speedup the memory accesses into the octree. The arrangement of the nodes in the octree guarantees a one direction transversal of the Octree array. This one direction transversal greatly reduces the seek times for global memory accesses.

The other important aspect of this algorithm to note is memory usage. This algorithm requires that the entire point cloud be moved onto the GPU for the indirect illumination to be calculated correctly. During testing we saw that this is not a problem for us but could present a problem if an extremely large octree or surfel count is needed. For our 1 million surfel test we saw a GPU memory utilization of about 130 megabytes. GPUs usually have over a gigabyte of global memory so there is still a lot of extra room to work with. It is also possible to get GPU with up to 3 gigabytes of global memory if that is needed.

4.3 GPU Octree Transversal Pseudo Code

The following pseudo code is the device function written to traverse the octree on the GPU. This function is meant to be called by a CUDA kernel that is parallelized across the pixels in the final image. The variable MAXOCTREEDEPTH is a constant that represents the maximum depth allowed in the octree. This is needed because there is no dynamic memory allocations in CUDA. The variable MAXANGLE is a constant that represent the maximum solid angle for this calculation. The two variables position and normal represent the receiving point.

```

__device__ void gpuOctreeTraversal( cube, gpu_octree,
    surfels, position, normal, )
{
    stack[MAXOCTREEDEPTH * 8];
    push root onto stack
    while(stack != empty) {
        node = pop node off stack
        if(node.leaf == true) {
            for each surfel in node {
                dis=distance(surfel.pos, position);
                if( dis < surfel.radius )
                    raytrace surfel to cube
                else
                    rasterize surfel to cube
            }
        }
        else
        {
            if( position in node's boundingbox ) {
                push all 8 children onto stack
                continue;
            }
            else if( node's boundingbox below horizon )
                continue;
            else {
                center = get node's center
                area = evaluate Spherical Harmonics area
                dis = distance( center, position );
                solidangle = area / (dis*dis)
                if( solidangle < MAXSOLIDANGLE ){
                    evaluate Spherical Harmonics power
                    rasterize cluster to cube
                    continue;
                }
                else {
                    push all 8 children onto stack
                    continue;
                }
            }
        }
    }
}

```

This chapter presented the implementations that we explored while working on this project. All three of our algorithms out performed the CPU based PBACB algorithm for our 1 million surfel test. There are still optimization methods that could be applied to the PBACB GPU algorithm which are discussed in the next chapter.

Chapter 5

Results and Conclusions

In this section we demonstrate that the GPU PBACB algorithm that we created out performs both the CPU PBACB algorithm and the other GPU algorithms that we created. The timings for these scenes are run on the Cornell Box containing two spheres unless otherwise specified as seen in Figure 5.1. The outputs of the GPU algorithms all matched the CPU counterparts to within an acceptable difference as seen in Figure 5.2.

5.1 Test System

The test system that all of these algorithms were run on is California Polytechnic State University Server, Tesla. Tesla contains two Intel Xeon E5504 CPUs which run at 2.00GHz, four NVIDIA Tesla C2050 GPUS, and twelve gigabytes of RAM. The GPU times given in the following section are utilizing a single GPU in Tesla.

Figure 5.1: This is an example output of our Cornell Box that we used for timings. 1 million surfels, 512x512, CPU output

5.2 Speed Analysis

Our analysis is based off of the time it took the indirect illumination to be calculated. All of the algorithms that were created shared the same octree creation code and direct illumination calculation code so it has been removed from the timings to better highlight the differences between the algorithms.

5.2.1 GPU Full Surfel Rasterization Algorithms Analysis

The GPU Full Surfel Rasterization and the GPU Shared Memory Full Surfel Rasterization algorithms both outperformed the CPU version as seen in Table 5.1 and Table 5.2. These algorithms were originally only meant to be a test but their ability to outperform the CPU version demonstrates the power of GPU paral-

	Full Surfel	Shared Memory	CPU Full Surfel
66871 Surfels in Octree	51.685s	46.203s	1256.243s
267603 Surfels in Octree	159.65s	138.947s	4988.457s
1070263 Surfels in Octree	592.345s	508.947s	N/A

Table 5.1: This table shows the times seen for the GPU Full Rasterization version and the GPU Shared Memory Full Rasterization version compared to the CPU threaded Full Surfel Rasterization. There were 8 cores used for the CPU threading in this example.

lelism. It is important to note that with these algorithms scaling is a big issue. These algorithms do not utilize any spacial data structures or approximations for regions. This means that the time it takes to calculate the indirect illumination scales directly with the number of surfels in a scene. These methods do not scale as well as the CPU version because the CPU version uses approximations for surfels where these methods use all the surfels. If the number of surfels is increased by a large amount, like is seen in industry, the speedups obtained by using the GPU will be reduced significantly.

We do not recommend implementing this version of the algorithm if there are going to be a very large amount of surfels. The major draw back of this version of the algorithm is also one of it’s best qualities, however. It calculates the same image the PBACB algorithm would if a maximum solid angle of 0.0 is chosen. If accuracy is the final goal then this version of the GPU algorithm is the best choice. This version will run much faster then the PBACB algorithm if a maximum solid angle of 0.0 is chosen.

5.2.2 GPU PBACB Algorithm Analysis

The GPU PBACB Algorithm out performs the CPU version significantly. We were able to obtain a 4x speedup over our fastest CPU version using a scene that

was comprised of over 1 million surfels. Our fastest CPU version utilizes OpenMP to add 8 core threading. OpenMP is a threading library that is very simple and can add CPU threading quickly. We utilized this library to speedup our PBACB algorithm as seen in Table 5.2. As stated previously, the GPU algorithm’s only major limitation is that of memory on the GPU. The memory usage for the GPU for the 1 million surfel scene was 130 MB out of a possible 1 gigabyte.

The reason we only see a 4x speedup is because of some inherent characteristic that the PBACB algorithm has. The PBACB algorithm has a significant branching problem. The three paths that an octree node can take are to rasterize the cluster, add the children to the stack, and rasterize all of the surfels to the cube. The path that rasterizes the surfels individually is significantly more work than the other two paths. This means that all the threads in the block are slowed down when any thread needs to rasterize the surfels individually. The other reason that we only see a 4x speedup is because of the size of the kernel. The PBACB kernel that was designed is rather large. The stack that we need to keep track of the current octree nodes is a none trivial size. This means that there are a large amount of needed registers. We limit the registers to 32 which increases the parallelism but also increases the local memory swaps. Local memory swaps slow down all of the threads. In the end, we get a very good speedup but not as good as we were hoping for. The exact timings of the CPU and GPU versions and the comparison between the timings are listed in Table 5.2, Table 5.3, and Table 5.4.

	Threaded SA-0.03	Threaded SA-0.06	Non Threaded SA-0.03
66871 Surfels	743.514s	257.710s	1956.872s
267603 Surfels	988.705s	527.985s	6664.859s
1070263 Surfels	2020.08s	1091.63s	N/A

Table 5.2: This table shows the times seen for the CPU threaded version of the PBACB algorithm compared to the Non Threaded CPU times. This version uses 8 CPU cores on Tesla. SA means Solid Angle.

	Max Solid Angle 0.03	Max Solid Angle 0.06
66871 Surfels in Octree	119.894s	64.036s
267603 Surfels in Octree	264.326s	121.936s
1070263 Surfels in Octree	498.331s	255.976s

Table 5.3: This table shows how the times seen for the GPU version of the PBACB algorithm.

	Threaded SA-0.03	Threaded SA-0.06	Non Threaded SA-0.03
66871 Surfels	6.201x	4.024x	16.321x
267603 Surfels	2.812x	4.330x	25.214x
1070263 Surfels	4.053x	4.264x	N/A

Table 5.4: This table shows how many times faster the GPU PBACB algorithm was to the CPU PBACB algorithm.

Figure 5.2: This figure is used to show the difference between our GPU output and CPU output. The difference originates from, the randomness associated with our sampling method, the difference between GPU and CPU math function accuracies, and the difference between our GPU and CPU implementation of rasterization.

Figure 5.3: This is figure is another scene rendered with our PBACB algorithm.

5.3 Future Work

5.3.1 Dynamic Parallelism

Dynamic Parallelism is a relatively new NVIDIA addition to CUDA. Dynamic Parallelism is essentially the ability of a kernel to spawn more kernels. Dynamic Parallelism also enables recursion in CUDA kernels. This is a new tool which can be used to eliminate branching problems. Dynamic Parallelism may be able to eliminate the branching problem that the current GPU PBACB algorithm had with the rasterization of individual surfels to the rastercube. If an algorithm could be designed to reduce the amount of time that it took to rasterize the individual surfels in the nodes we predict a much faster implementation would be the result.

It may be possible for Dynamic Parallelism to completely remove the need for a stack object in the traversal code. This implementation would heavily leverage Dynamic Parallelism but it may map better to PBACB algorithm than the current stack implementation. The algorithm however would have to take special note not to overflow the function stack frame with too many recursive calls which would be the case if the octree went too deep.

5.3.2 Stack Free Implementation

The current implementation is heavily dependent on the stack object that is created for each thread. This object contributes to the large register count of the threads. It may be possible to design an octree representation that eliminates the need for a stack variable entirely. If the register count is lowered then the amount of memory swaps per thread could be reduced increasing the overall speed of the algorithm. Extensive testing is required to test this hypothesis but it may be worthwhile.

5.3.3 Multiple GPU Implementations

The current algorithm only makes use of a single GPU. It is possible for CUDA to utilize multiple GPUs. It would require additional `cudaMallocs` and additional `cudaMemcpy`s but it is possible to use two GPUs. An additional GPU could reduce the time it takes to calculate the indirect illumination by as much as half because you are doubling the compute power. This option is only limited to how much money is invested into the computer that is running the algorithm.

5.3.4 Hybrid Solution or Pipelining

Another possible speedup that could be implemented is to utilize the CPU while the GPU is busy. It is possible to split the work between the GPU and the CPU to be more effective. The current implementation lets the CPU go idle while the indirect illumination is being calculated on the GPU which is not ideal. It may be possible to reserve a portion of the indirect illumination calculations to be done by the CPU while the GPU is working on the rest.

A different solution which keeps the CPU busy could be implementing a pipeline. The idea behind pipelining is to do work on the CPU while the GPU is busy. For example, it may be possible to start the indirect illumination calculations before doing the primary ray direct illumination calculations which would in effect hide the direct illumination calculation items, which reduces the total computation time of the algorithm. Another example is if multiple frames need to be calculated then the next frame could be started on the CPU while the current frame is using the GPU. The GPU and the CPU are separate and to optimize the system fully both must be kept fully utilized.

5.4 Conclusions

In this project we have presented our design for a GPU based Point Based Approximate Color Bleeding global illumination algorithm. We have shown that we obtained an impressive 4x speedup over the corresponding CPU algorithm. We have also presented other similar color bleeding algorithms that run on the GPU. These algorithms also out performed the CPU version of the PBACB algorithm. We believe the GPU is a definite asset to the PBACB algorithm.

We also did testing to determine the limitations of our algorithm. We found a number of issues that the current implementation suffers from that stem from characteristic of the original algorithm. These issues include branching problems, extensive register usage, and CPU utilization. We have also presented possible solutions to these problems to be explored in future works. It is our opinion that the most likely optimal solution will be a stack free implementation that leverages the characteristics of the octree spacial data structure. However, it would be worthwhile to attempt a Dynamic Parallelism version because the Dynamic Parallelism tool is so untested the results could be very promising. It is our final conclusion, that the GPU is a strong tool that could greatly reduce the runtime of any Point Based Approximate Color Bleeding Algorithm.

Bibliography

- [1] Harrison Lee McKenzie Chapter. Textured hierarchical precomputed radiance transfer. Master's thesis, California Polytechnic State University San Luis Obispo, June 2010.
- [2] Per H. Christensen. Point-based approximate color bleeding. Technical report, Pixar Animation Studios, 2008.
- [3] Per H. Christensen and Dana Batali. An irradiance atlas for global illumination in complex production scenes. Technical report, Pixar Animation Studios, 2004.
- [4] Christopher J. Gibson and Zoe J. Wood. An approach to point based approximate color bleeding with volumes. Master's thesis, California Polytechnic State University San Luis Obispo, June 2011.
- [5] Robin Green. Spherical harmonic lighting: The gritty details. Technical report, Sony Computer Entertainment America, January 2003.
- [6] Jeroen vanBaar Hanspeter Pfister, Matthias Zwicker and Markus Gross. Surfels: Surface elements as rendering primitives. Technical report, Microsoft Corporation, March 2000.

- [7] Henrik. Ray tracing. [http://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](http://en.wikipedia.org/wiki/Ray_tracing_(graphics)), April 2008.
- [8] James T. Kajiya. The rendering equation. Technical report, California Institute of Technology, 1986.
- [9] NVIDIA. Nvidia cuda zone. <https://developer.nvidia.com/category/zone/cuda-zone>, 2013.
- [10] Bui Tuong Phong. Illumination for computer generated pictures. Technical report, University of Utah, 1975.
- [11] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. Technical report, Stanford University, August 2001.
- [12] Loren Carpenter Robert L. Cook and Edwin Catmull. The reyes image renderingarchitecture. Technical report, Pixar, P. O. Box 13719, San Rafael, CA 94913, July 1987.
- [13] Ryan Schmitt. Gpu-accelerated point-based color bleeding. Master’s thesis, California Polytechnic State University San Luis Obispo, June 2012.
- [14] Peter-Pike Sloan. Stupid spherical harmonics(sh) tricks. Technical report, Microsoft Corporation, February 2010.
- [15] Samuel Hornus Sylvain Lefebvre and Fabrice Neyret. *GPU Gems 2*. April 2005.
- [16] Eric Haines Tomas Akenine-Moller and Nathaniel Hoffman. *Real-Time Rendering*. A K Peters, Ltd, 3rd edition, 2008.
- [17] WhiteTimberwolf. Octree. <http://en.wikipedia.org/wiki/Octree>, 2010.

- [18] Wikipedia. Phong reflection model. http://en.wikipedia.org/wiki/Phong_reflection_model, 2012.
- [19] Tomoyuki Nishita Yoshinori Dobashi, Tsuyoshi Yamamoto. Radiosity for point-sampled geometry. Technical report, Hokkaido University, The University of Tokyo, 2004.