GPU-ACCELERATED POINT-BASED COLOR BLEEDING

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

Ryan Schmitt

June 2012

COMMITTEE MEMBERSHIP

TITLE:	GPU-Accelerated Point-Based Color Bleeding
AUTHOR:	Ryan Schmitt
DATE SUBMITTED:	June 2012
COMMITTEE CHAIR:	Zoë Wood, Ph.D.
COMMITTEE MEMBER:	Chris Lupo, Ph.D.
COMMITTEE MEMBER:	Alexander Dekhtyar, Ph.D.

Abstract

GPU-Accelerated Point-Based Color Bleeding

Ryan Schmitt

Traditional global illumination lighting techniques like Radiosity and Monte Carlo sampling are computationally expensive. This has prompted the development of the Point-Based Color Bleeding (PBCB) algorithm by Pixar in order to approximate complex indirect illumination while meeting the demands of movie production; namely, reduced memory usage, surface shading independent run time, and faster renders than the aforementioned lighting techniques [8].

The PBCB algorithm works by discretizing a scene's directly illuminated geometry into a point cloud (surfel) representation. When computing the indirect illumination at a point, the surfels are rasterized onto cube faces surrounding that point, and the constituent pixels are combined into the final, approximate, indirect lighting value.

In this thesis we present a performance enhancement to the Point-Based Color Bleeding algorithm through hardware acceleration; our contribution incorporates GPU-accelerated rasterization into the cube-face raster phase. The goal is to leverage the powerful rasterization capabilities of modern graphics processors in order to speed up the PBCB algorithm over standard software rasterization. Additionally, we contribute a preprocess that generates triangular surfels that are suited for fast rasterization by the GPU, and show that new heterogeneous architecture chips (e.g. Sandy Bridge from Intel) simplify the code required to leverage the power of the GPU. Our algorithm reproduces the output of the traditional Monte Carlo technique with a speedup of 41.65x, and additionally achieves a 3.12x speedup over software-rasterized PBCB.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

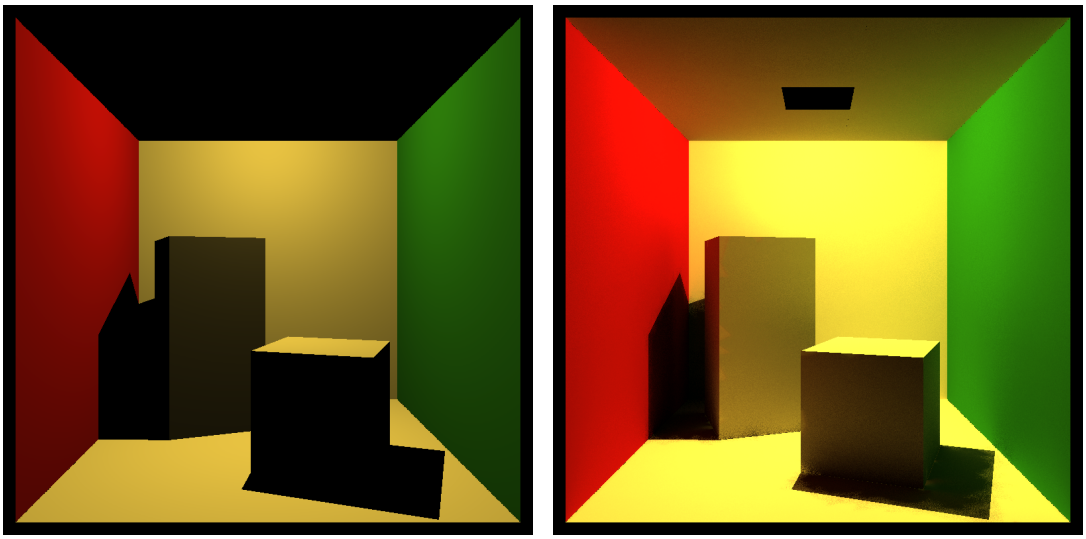
Introduction

1.1 Computer Graphics

Computer graphics is, in general, anything produced by a computer that is not plain text or sound. Although, perhaps a more fitting definition is using a computer to draw a picture; this is also called rendering. There are a vast array of different areas in which one may want the help of a computer to render an image, from the entertaining, like video games and animated films, to the scientific, like medical visualization and computer-aided design and drafting. Each of these disciplines can have varying requirements of computer graphics: some need real-time rendering in order to respond to user-input, and others may trade the real-time speed for precise simulation. The goal of computer graphics is to identify the requirements of the application and render the highest quality image given those restrictions.

1.2 Ray-Tracing and Global Illumination

In particular, this thesis is concerned with ray-traced rendering using global illumination algorithms, which are most commonly utilized to produce high-quality photo-realistic images. A ray-tracing algorithm can be classified as a global illumination algorithm when it incorporates not only direct illumination from light sources, but indirect illumination, or light that is inter-reflected between scene geometry from the same light sources (Figure 1.1).



(a) Direct Only (b) Direct & Indirect

Figure 1.1: Cornell Box rendered with both direct illumination and direct & indirect illumination.

Ray-tracing achieves photo-realism by simulating the physics of light using a scene comprised of light sources, mathematically-defined geometric surfaces, and a virtual camera (see Figure 1.2). It produces renders that are specifically not real-time in nature, but meant to take as long as is necessary to produce quality results. In this setting, we must render life-like images, so an accurate simulation of light physics is required, but oftentimes we can obtain a convincing result using approximations. Specifically, ray-tracing follows the opposite path of the light:

instead of tracing light rays from the light sources until they happen to hit the virtual camera, which is very physically accurate, we start at the camera and trace into the scene.

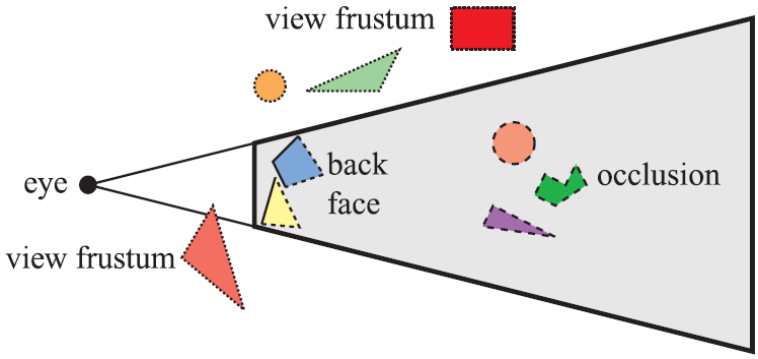


Figure 1.2: The eye and virtual camera are equivalent, geometry that lies in the viewing frustum is rendered [2].

Once these primary rays travel from the camera, into the scene, and intersect with the geometry, we can calculate the shading at that point in order to determine the color of the primary ray's associated pixel in the final rendered image. This shading calculation can vary from a simple direct illumination computation, to a complex global illumination calculation.

In this thesis, we focus on global illumination techniques 2. We split the calculation into direct illumination, which is the amount of light that leaves the source and directly intersects our shading point, and indirect illumination, which is the amount of light contributed from the diffuse inter-reflections of geometry in the scene; a phenomenon that is exemplified by the fact that the space under our desks is not completely dark, or that an illuminated red wall may reflect red light onto a nearby white box, causing it to appear reddish. These two components (direct and indirect illumination) are combined into one value that represents the incoming illumination at our primary ray's intersection point. We solve for the amount of this illumination that follows back along the ray to the camera, and

we write that value to the primary ray's associated final-image-pixel. Performing this ray-tracing algorithm on each such final-image-pixel generates a rendering of the scene geometry as defined by the light sources and virtual camera, and is classified as global illumination.

1.3 Monte Carlo Ray-Tracing

One of the most widely used methods for the indirect illumination calculation in ray-tracing is called Monte Carlo sampling, and it involves randomly and discretely sampling the hemisphere above an intersection point (Figure 1.3). This is performed by recursively tracing yet more rays into the scene in order to gather information about what geometry is nearby and what color it is shaded; this attempts to solve for the diffuse inter-reflections incident at an intersection point.

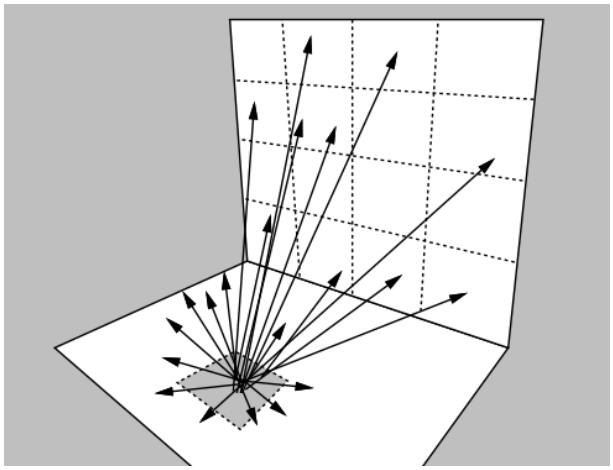


Figure 1.3: Monte Carlo sampling of a hemisphere above an intersection point [30].

In our experience, around 256 of these rays are required, and are traced from each primary ray intersection point, into the scene, in order to gather enough shading information about the adjacent geometry to calculate a believable and

accurate indirect illumination value. The number of rays that require intersection calculations, and shading calculations, can quickly escalate into the tens and hundreds of millions. Renders requiring multiple hours to complete are not rare.

1.4 Point-Based Color Bleeding

Recently, the Point-Based Color Bleeding algorithm was developed at Pixar by Per H. Christensen [8] for indirect illumination. Instead of tracing rays, as in Monte Carlo ray-tracing, discretized surface elements (surfels) are rasterized onto a cube of eight-by-eight-pixel images, approximating the hemisphere used in the Monte Carlo ray-tracing (Figure 4.1). Once the surfels have been rasterized onto the pixels of the cube faces, the pixels are weighted and convolved into one value representing the indirect illumination at a point.

The surfels are comprised of a location, surface normal, surface area, and shaded color computed from the direct illumination (Figure 1.4). The benefit of this technique is that the surfels can be precomputed and stored in a point cloud, separate from the scene geometry, and reused. This lends itself well to reduced memory usage, surface shading independent run time, and faster renders than Monte Carlo ray-tracing, all of which are very useful properties for Pixar and movie production in general.

1.5 Our Contribution

This thesis is primarily concerned with performing the indirect illumination calculation faster than the Monte Carlo sampling method, without sacrificing render quality. We achieve this by extending PBCB to utilize the specialized

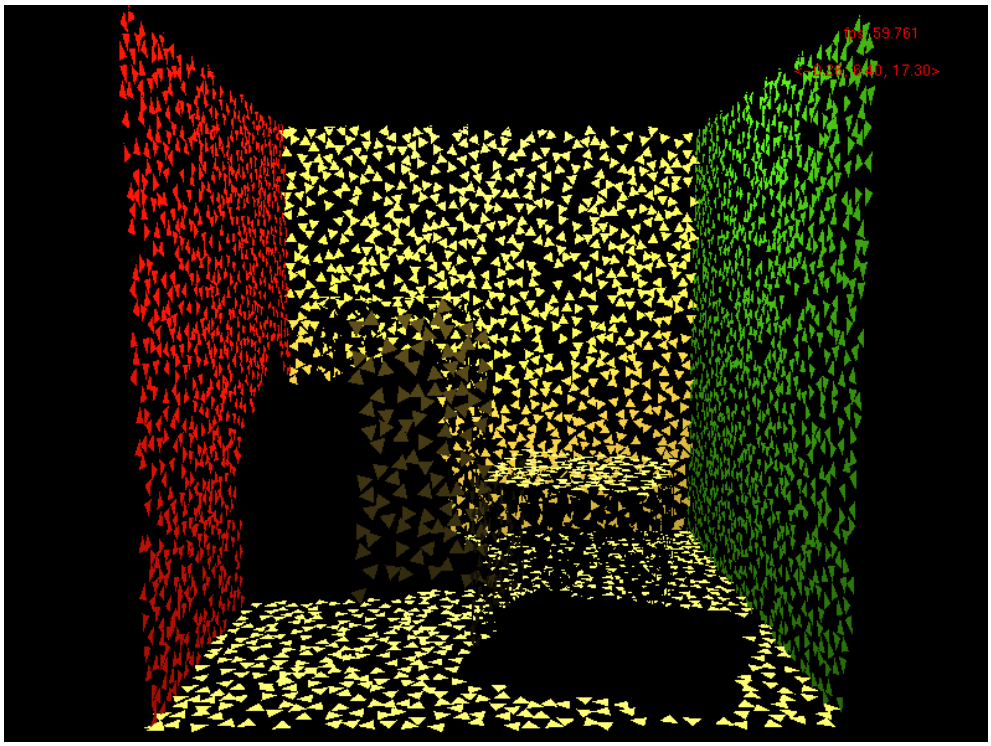


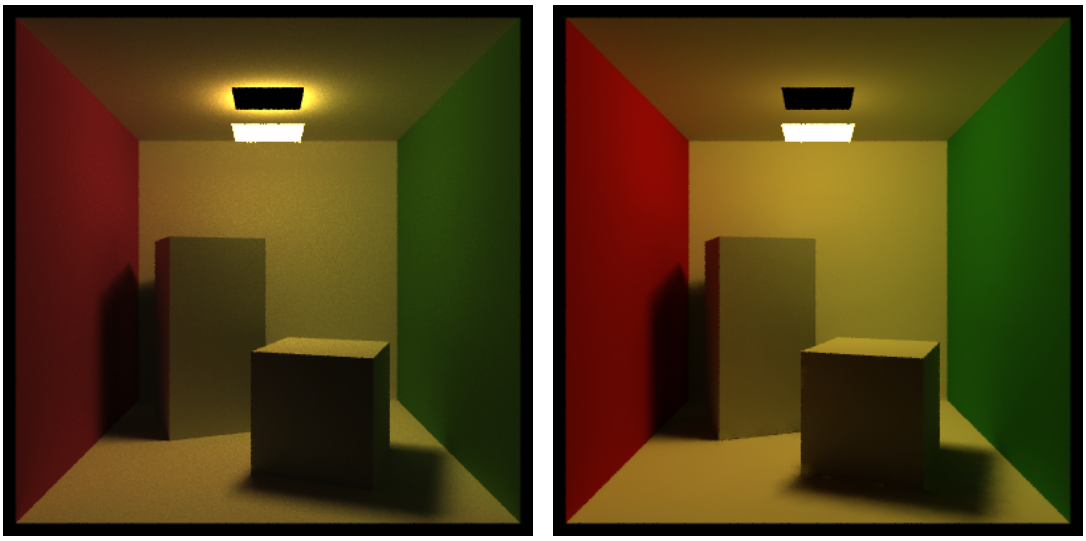
Figure 1.4: Surfels for the Cornell Box scene. Note that the surfel size has been reduced to exhibit the surfel shape and distribution.

rasterization capabilities of the modern heterogeneous architecture chips' graphics processing unit (GPU) to rasterize the point cloud onto five eight-by-eight-pixel images arranged into a cube above each primary ray intersection point; this technique approximates the hemisphere that is used in the radiance integral (see Section 2.1). Also, we contribute a preprocess where the scene's geometry is transformed into triangles (the preferred geometric shape for GPU-rasterization) and assigned color values based on direct illumination calculations evaluated per triangle vertex.

In this paper, we:

- review important rendering related equations and techniques,
- provide an in-depth description of the PBCB algorithm,
- present our PBCB extension and surfel generation preprocess,
- discuss and analyze our validation techniques and results.

Our contributions, leveraging the modern heterogeneous chips, realize much faster render times compared to Monte Carlo ray-tracing, while maintaining visually similar results (Figure 1.5). We achieve this by avoiding the numerous and costly intersection and shading calculations inherent in Monte Carlo Ray-tracing, and in some cases achieve an order of magnitude speedup.



(a) Monte Carlo (b) GPU PBCB

Figure 1.5: Cornell Box with indirect illumination using both Monte Carlo Sampling and GPU PBCB.

Chapter 2

Background

In our background section we will cover concepts that form the underpinnings of our thesis work. Specifically, we will discuss radiance in Section 2.1 in order to understand why sampling a hemisphere is integral to any global illumination algorithm. We discuss the Phong reflection model in Section 2.2 because we use this simplified lighting model in our algorithm. Section 2.3 describes the concept of anti-aliasing, which is a feature of our algorithm. We cover octrees in Section 2.4, which are used in Christensen's PBCB implementation [8] in order to store the surfel cloud. Monte Carlo integration is the basis for the Monte Carlo ray-tracer we have implemented for comparison against our GPU PBCB algorithm, and is discussed in Section 2.5. And lastly, we address the system architecture characteristics of the CPU and GPU, as well as how heterogeneous chips are changing them, in Sections 2.6 and 2.7, respectively.

2.1 Radiance

Radiance is a general term for the amount of light energy being transmitted through an area on a surface in a specific direction. Or more simply as the measure of brightness and color of a single ray of light [2].

In order to understand how to calculate radiance, we must first understand radiant flux and flux density. Radiant flux, ϕ, is a measure of energy (measured in joules) per second. Flux density is the instantaneous amount of radiant flux over an area, and is written as:

$$E = \frac{d\phi}{dA} \tag{2.1}$$

We can now define radiance as the flux density with respect to a projected area and a solid angle, or:

$$L = \frac{d^2\phi}{dA * \cos\theta * dw} \tag{2.2}$$

Incoming radiance, or irradiance, is the radiance arriving at a surface, or the flux density of arriving light. We can define this in terms of radiance, where x is a surface point and w' is the incoming ray direction, as:

$$E(x, w') = L(x, w') * \cos\theta * dw' \tag{2.3}$$

Further more, we integrate over a hemisphere to solve for the incoming radiance, at point x, in all directions:

$$E(x) = \int L(x, w') * \cos\theta * dw' \tag{2.4}$$

Ultimately we are interested in the reflected radiance: the amount of irradiance that is reflected by a surface back towards our virtual camera. This is based on the material properties of the surface, which is typically represented by a

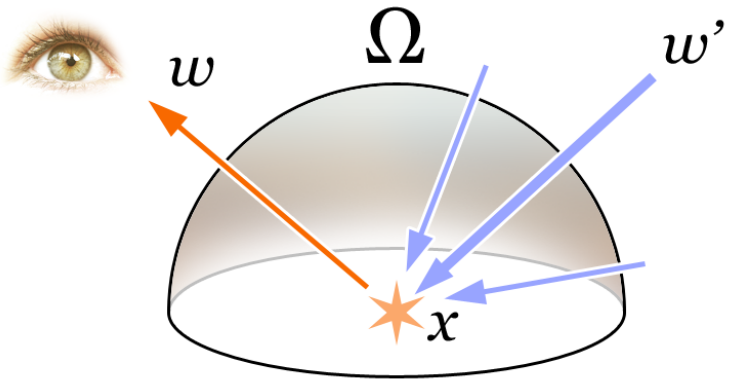


Figure 2.1: Incoming radiance in a hemisphere about a point, reflecting towards the camera [21].

BRDF (bi-direction reflectance distribution function). BRDFs are defined per-surface, and represented as $f(x, w', w)$. They solve for the ratio of radiance that is transmitted from an incoming direction, w', to an outgoing direction, w, at a surface point, x (see Figure 2.1). Incorporating the BRDF, the equation for reflected radiance is:

$$L(x) = \int f(x, w', w) * L(x, w') * \cos\theta * dw' \tag{2.5}$$

2.2 Phong Reflection Model

The Phong reflection model is a simplified shading model used to calculate reflected radiance, given one incident light ray. It is most commonly used in scenarios that require fast computation, such as real-time graphics applications. It was presented by Phong in his University of Utah Ph.D. dissertation in 1973 [25]. The equation calculates the reflected radiance using three color-vector terms: an ambient, diffuse, and specular.

The ambient component represents indirect illumination as a constant amount

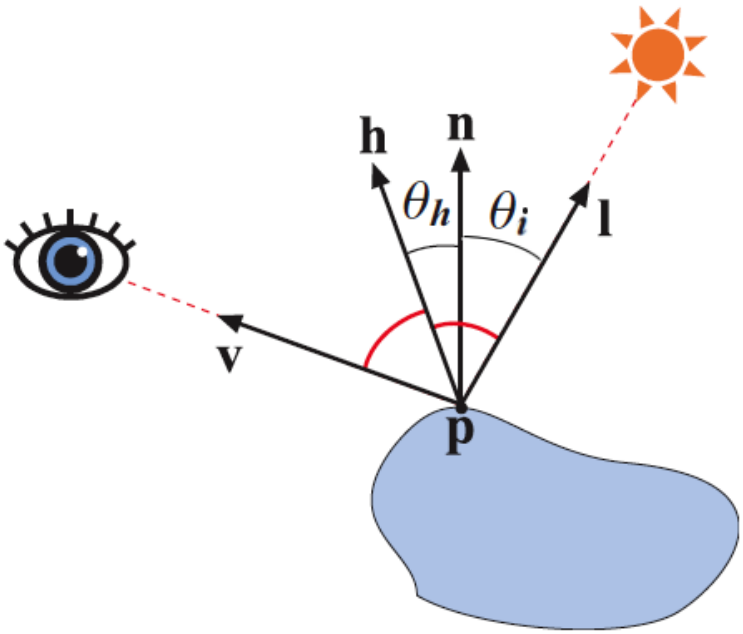


Figure 2.2: The vectors involved in the Phong Reflectance Model [2].

of reflected radiance added to all shaded points. This is the simplest term, but also poorly represents the subtleties of indirect illumination.

we're

The diffuse component represents direct illumination using Lambertian reflectance of the incoming radiance. Lambertian refers to Lambert's cosine law, which states that, for perfectly diffuse surfaces, the diffuse reflected radiance is proportional to the cosine of the angle between the surface normal and the light vector, θ_i. This is clamped in the range [0,1] to avoid subtracting light, and is commonly referred to as the $n \cdot l$ *factor* [2], a convention we shall adopt in this thesis. Figure 2.2 visualizes these components.

The specular component represents the shininess we can view from certain angles on highly reflective surfaces. Specular reflectance is calculated by raising the cosine of the angle between the surface normal and the half vector (the vector

halfway between the view and light vector), θ_h, to the shininess of the surface, s.

Each of these components represent the ratio of incoming radiance, C_{light} to reflected radiance, C_{out}, but the reflected radiance is also scaled by the surface material's color characteristics, C_{mat}, as well as ambient, diffuse, and specular characteristics (represented by the scalar ratios: M_{amb}, M_{diff}, and M_{spec}, respectively). Combining these components we get the final Phong reflection model equation:

$$C_{out} = (M_{amb} * C_{mat} * C_{light}) + ((n \cdot l) * M_{diff} * C_{mat} * C_{light}) + ((n \cdot h)^s * M_{spec} * C_{mat} * C_{light}) \tag{2.6}$$

2.3 Anti-aliasing

Anti-aliasing combats aliasing, the artifacting caused by under-sampling. A typical example of this in signal processing, where an analogue signal is sampled as some rate to determine its amplitude at that point in time. If the sample rate is too low (i.e. under-sampling) then the signal will not be accurately captured because the signal's characteristics between samples was lost. This can be seen in Figure 2.3.

This occurs in computer graphics due to the limited sample rate provided by our display devices. To illustrate this point, imagine attempting to display a sphere with four pixels. More typically we experience aliasing in computer graphics as jaggies, or lines that should be smooth but are jagged (see Figure 2.4).

In order to combat this phenomenon, techniques for anti-aliasing have been developed. These can take many forms [19], but generally it involves some form

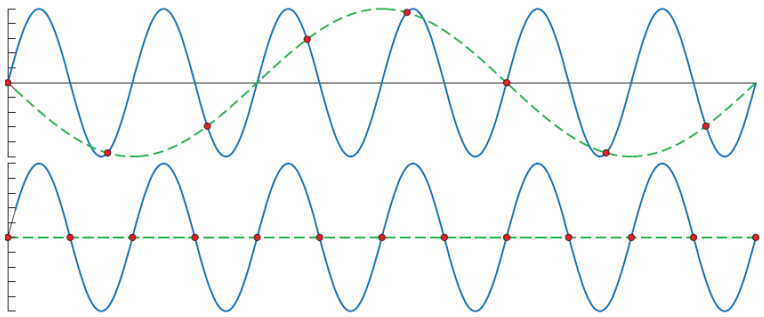


Figure 2.3: The blue signals are being undersampled by the red dots, which leads to inaccurate reconstruction as evidenced by the dotted green line [2].

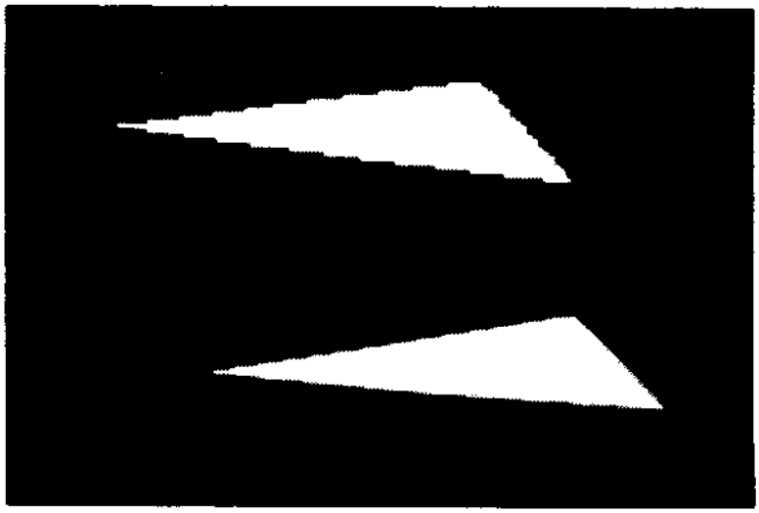


Figure 2.4: Jagged edges apparent along the silhouette edges of the top triangle are reduced via anti-aliasing techniques in the lower triangle [12].

of over-sampling (i.e. rendering the image at a multiple of the display resolution) to compensate for the fixed sample rate of our display devices. By over-sampling the rendering, we can capture the subtleties of the image in software, and can down-sample the image in a way that helps alleviate the fixed display sample rate.

The simplest form of this technique is called super-sampling: where we render an image at some multiple of the desired final resolution, and down-sample the

large image back to the desired final resolution by averaging the extra pixels into one value. For example, if we wish to render a 300 by 300 pixel image, we might render an intermediate image at 600 by 600 and then each final pixel is represented by 4 intermediate pixels, which can be averaged into one final pixel value; this is called 4x super-sample anti-aliasing.

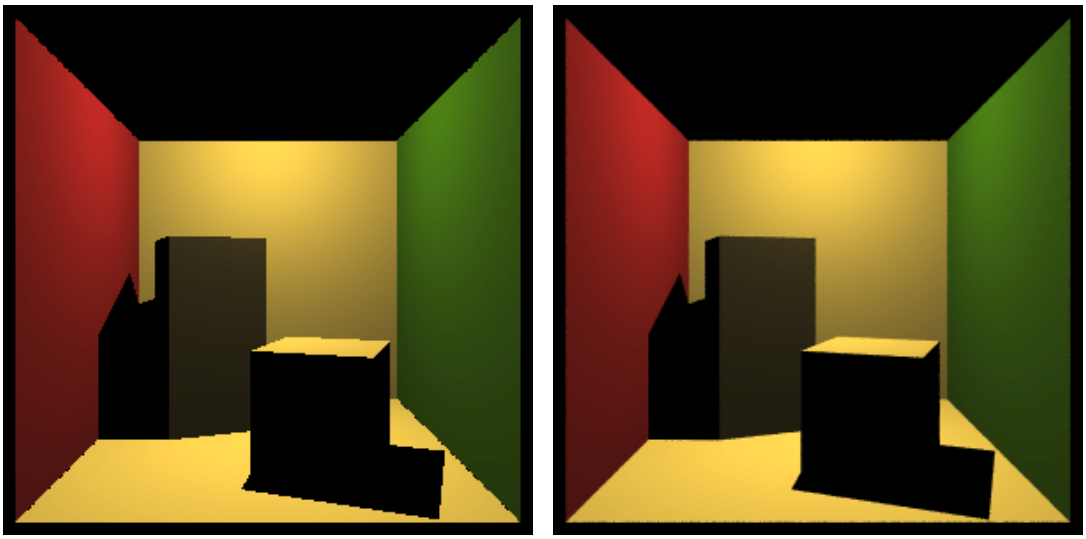
However, in order to reduce the aliasing artifacts to an acceptable level, we usually require a large super-sample rate. This can result in unacceptably long render times. The main reason we require high super-sample rates in ray-tracing is the regular pattern created by generating rays that pass through the center of each pixel. Generally in ray-tracing, the more random you can make a sample pattern (while not missing important features), the less aliased your final image will be. Therefore, we randomly jitter the direction of each ray within the bounds of the pixel. This facilitates lower super-sample rates.

An example image of anti-aliasing in our rendering algorithm can be seen in Figure 2.5.

2.4 Octree

Octrees are a type of spatial data structure. These data structures are useful in order to accelerate queries about what objects are adjacent to a point, or what objects overlap or are spacially close. Due to their hierarchical construction, they typically improve spatial queries from $O(n)$ to $O(\log n)$ [2].

Octrees enclose the entire scene in an axis-aligned bounding box, and recursively, as well as regularly, subdivide the the box in all three dimensions (x, y , and z). This produces eight equal-sized child boxes, hence the name octree. An



(a) No Anti-aliasing (b) 9x Anti-aliasing

Figure 2.5: Our simple Cornell box rendered with no anti-aliasing, and 9x box-filtered anti-aliasing.

illustration of this can be found in Figure 2.6.

2.5 Monte Carlo Integration

Monte Carlo integration is a technique used to evaluate integrals [24]. By repeatedly evaluating randomized discrete samples, we converge on the true evaluation of the integral. On average, these evaluations correspond to the correct solution, therefore we average multiple runs of the algorithm. We do not arrive at the correct solution in this way, but one that is statistically close.

We use Monte Carlo to evaluate the complex integral in Equation 2.5. It is almost impossible to create a closed-form representation of the terms in this equation, and therefore Monte Carlo lends itself to its evaluation. To evaluate the incoming radiance at a point, we can generate randomized sample vectors (rays) over the domain of integration, the unit hemisphere. We then average

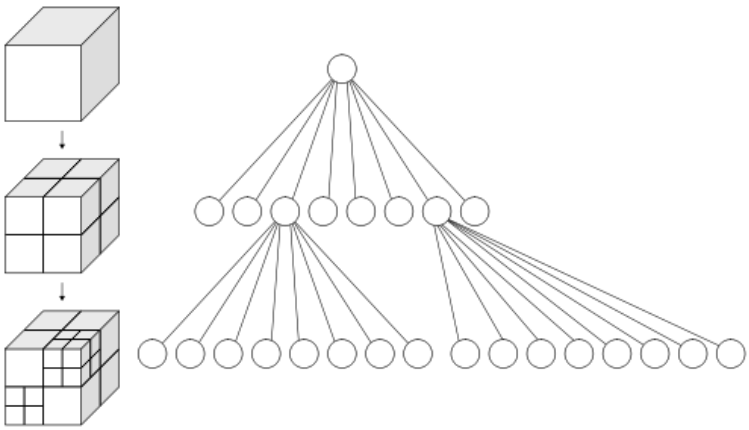


Figure 2.6: The hierarchical levels of an octree [37].

these samples in order to obtain a result that is close to the true evaluation.

The main drawback of Monte Carlo integration is that it only converges at a rate of $O(n^{-1/2})$, with n being the number of discrete random samples. This means that increasing the number of samples by 4 would only reduce the error by half [24]. And because each sample requires one or more rays to be traced against the scene geometry and shaded at their intersection point, it quickly becomes cost-prohibitive to obtain low-error results. Error in this technique is exhibited by adjacent pixels that have disparity in their brightness and color, or noise.

Most of the current research in Monte Carlo for computer graphics is aimed at reducing this error through methods other than increasing the sample count [24]. One technique we use to better distribute the sample rays across the unit hemisphere is to discretize the hemisphere into a grid of sample directions with higher density in areas of higher importance, namely the top of the hemisphere where the $n \cdot l$ factor is largest, and jitter each sample direction by a random amount. This is an example of stratified sampling, combined with importance sampling.

2.6 CPU versus GPU

The CPU and the GPU are both processors, but have quite different architectures due to their purpose. The CPU is considered a serial processor that handles general computation, which is typically thought of as instruction-driven, and involves executing unpredictable instructions on irregular data and likely includes branching. The GPU, on the other hand, is a stream processor optimized for data-driven graphics rendering of more regular data with predictable memory access [2].

Because of their differing purposes, the CPU and GPU have traditionally been divided into two physically separate components of a computer system's architecture connected by the PCI bus. The application running on the CPU will transmit rendering data and instructions over this bus to the GPU, which will process this input and produce a rendering. Depending on the purpose of this rendering, it can either be displayed on a monitor attached directly to the GPU, or transmitted back over the bus to the CPU for additional processing. As is common in multiprocessing systems, the two common problems we experience are load balancing and communication [2].

Load balancing can take place between the CPU and GPU, as well as internally within the GPU. The programmer is generally responsible for handing the load balancing between these two components, while the GPU itself leverages FIFO queues between stages to avoid stalling any part of the pipeline.

The latency and bandwidth of the bus connecting the CPU and GPU can cause communication problems as well. The theoretical maximum one-way bandwidth of the PCI Express 2.0 bus is 8.0 GB/s [28], while if we wanted to render 300 million triangles per second we would require a rate of 10.2 GB/s just to

transfer the triangles to the GPU [2]. If we want to saturate this bandwidth, we must be constantly streaming data across the bus, however this is not always the use-scenario. Many times we wish to transmit data to the GPU, process it, and send it back to the CPU. In this case, the PCI bus can become the bottleneck as it is designed for one-way streaming of data.

2.7 Heterogenous Chip Architectures

The traditional architecture for systems with both CPU and GPUs involves the PCI bus, which connects these two disparate components. However, recent advancements have allowed chip densities so high that both CPU and GPU can fit onto one silicon chip. These architectures, that combine CPU and GPU, are called heterogeneous chips.

Intel has released both Sandy Bridge and, more recently, Ivy Bridge architectures [17] that adhere to this ethos, and AMD has released Fusion [4]. By combining both the CPU and GPU onto one chip, the hardware can bypass the shortcomings of the PCI bus to more quickly communicate.

In systems utilizing the traditional architecture, rendering small scenes that require final data processing on the CPU would have been cost-prohibitive due to the latency required to transfer data from CPU to GPU and back again. However, with heterogeneous chip architectures, we can almost ignore this latency in our algorithms; allowing for new-found simplicity.

2.8 Review

In this section we have covered the important concepts that frame our exploration of GPU-accelerated Point-Based Color Bleeding. The concept of radiance gives us the motivation to sample a hemisphere, while Monte Carlo integration allows us to discretely sample in order to estimate the integral over that hemisphere. We use the Phong reflection model in order to calculate the reflected radiance after gathering the irradiance samples. Anti-aliasing is a feature of our renderer, although not essential to the functioning of our renderer. We addressed the octree spatial data structure used in Christensen's PBCB [8] for surfel storage. Lastly, we compared and contrasted the CPU and GPU in order to motivate our use of the GPU to accelerate our algorithm, and covered the wrinkles introduced by new heterogeneous chip architectures.

Chapter 3

Related Work

In this section we discuss the methods for computing indirect illumination relevant to our thesis. Although radiosity and photon mapping are other popular global illumination algorithms, we did not have freely available implementations that would accept our scene format, and in the interest of scope, we did not implement our own versions. We do not cover them here and leave it to the reader if further knowledge of the subject area is desired [13, 18].

We first cover the Monte Carlo gather method, which we have implemented for comparison with our GPU PBCB algorithm. This method produces highly realistic results, but at the cost of copious computation, which in our experience, can easily require multiple hours of rendering time on commodity hardware. Our GPU PBCB algorithm is meant to produce comparable results, much faster.

We also discuss the basic PBCB algorithm, which we have extended to leverage the rasterization power of the GPU. The PBCB algorithm as developed by Christensen at Pixar [8] utilizes a software-based rasterization. It is here that we feel leveraging the power of the GPU can accelerate the algorithm and produce

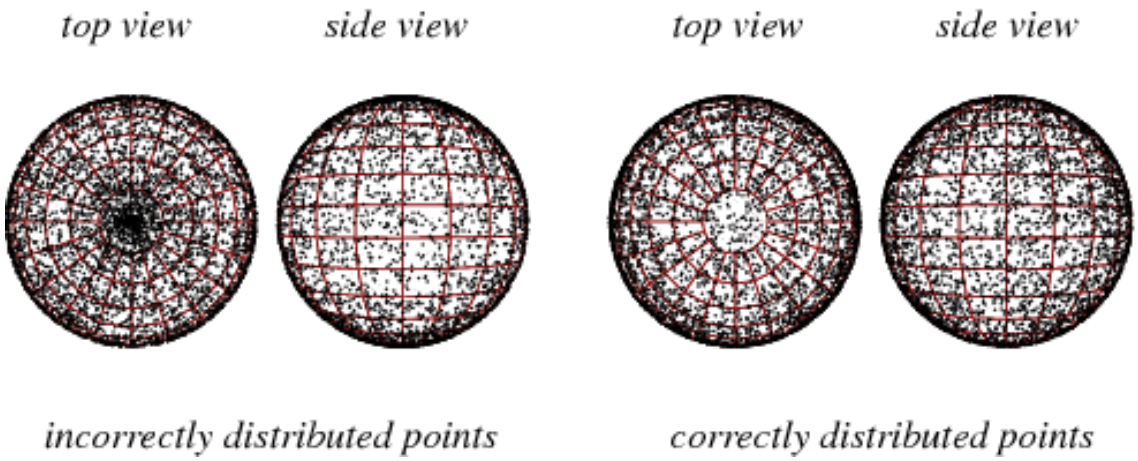


Figure 3.1: Point distribution on a sphere: left is naïve point sampling that results in higher point density near the poles, right is correct sampling [36].

faster render times.

Lastly, a promising area of research that we do not cover here is maximal Poisson disk point set generation on 3D surfaces [10]. The goal of this research is similar to our surfel generation goal; that is to say, they both strive for uniform distribution of points on 3D surfaces. Although the technique presented in [10] does not address transformed spheres or boxes, it may be possible to extend their ideas to cover these types of geometry.

3.1 Monte Carlo Gather

The Monte Carlo gather method is used in ray-tracing for indirect illumination, and receives its name from the Monte Carlo integration approximation discussed in Section 2.5. Instead of computing the true outgoing radiance integral, which is all but impossible except in the simplest of cases, we discretize the integral and compute its approximation. We begin our description of this method

at the time a ray-geometry intersection is found.

The first step is to generate the randomized sample vectors used to discretize the integral over the hemisphere. To this end, the algorithm leverages a hemisphere sampler: a function that takes two floating point variables as input, and produces a sample point on the hemisphere, see Figure 3.1 (it should be noted that identical input always results in identical output). Typically, samplers are designed to produce a uniform random distribution over the hemisphere when two uniform random variables, u_1 and u_2, are used as input. The following equations [24] describe this mathematically:

$$\begin{aligned} x &= \cos(2\pi u_2)\sqrt{1-u_1^2} \\ y &= \sin(2\pi u_2)\sqrt{1-u_1^2} \\ z &= u_1 \end{aligned} \tag{3.1}$$

However, in some cases a uniform distribution over the hemisphere is not desired. We have found that using a purely uniform random distribution over the hemisphere does not result in visually acceptable noise levels until we reach approximately 256 sample vectors (see Figure 5.1).

In the case of indirect illumination computation, sample points near the top are more highly weighted then those near the bottom due to the $n \cdot l$ factor used in the Phong Shading model (see Section 2.2). Therefore, a technique called importance sampling [24] is used to create a non-uniform sampling (even though the input variables are from a uniform distribution) of the hemisphere, where the sample density increases where the $n \cdot l$ factor is largest: the top of the hemisphere. We refer to this as a cosine-weighted hemisphere sampler because the likelihood of generating a given sample point is proportional to the cosine of the angle between the vector represented by the sample point and the unit hemisphere's

vertical vector $<0, 1, 0>$. This is described mathematically in equation 3.2 [24].

$$\begin{aligned} x &= \cos(2\pi u_2)\sqrt{u_1} \\ y &= \sin(2\pi u_2)\sqrt{u_1} \\ z &= \sqrt{1 - x^2 - y^2} \end{aligned} \tag{3.2}$$

An algorithm implementing equation 3.2 is used to generate a desired number of cosine-weighted sample points on a unit hemisphere. These sample points are treated as vectors to describe the direction of sample rays used to gather incoming radiance information. In order to convert from a vector in unit hemisphere coordinates to a vector in world space coordinates we use Algorithm 3.1. This is a standard coordinate system transformation between surface, or object, space and world space.

Algorithm 3.1 Transform from unit hemisphere to world space.

function HEMISPHERETRANSFORM($surfaceNormal$)
  if $|surfaceNormal.y| = 1$ **then**
    $tangent := <1, 0, 0>$
  else
    $k := <0, 1, 0>$
    $tangent := k - ((k \cdot surfaceNormal) * surfaceNormal)$ //Gram-Schmidt Process
    normalize($tangent$)
  end if
  $binormal := surfaceNormal \times tangent$
  $transform := \begin{pmatrix} tangent.x & surfaceNormal.x & binormal.x \\ tangent.y & surfaceNormal.y & binormal.y \\ tangent.z & surfaceNormal.z & binormal.z \end{pmatrix}$
  return $transform$
end function

The next step is to use these transformed vectors as ray directions to perform a recursive ray tracing computation. Each computation returns the incoming radiance, just as the primary rays return incoming radiance written to a final image pixel. These incoming radiance values are convolved into one indirect illumination value for the point we are attempting to shade by computing their

arithmetic mean. This results in an indirect illumination value used in the current shading computation.

Typically, the shading computations returned by the rays generated by the Monte Carlo method use direct illumination only, but because light can reflect off multiple surfaces in reality, multi-bounce Monte Carlo gather can be performed as well, to increase realism. This is where the Monte Carlo method is used recursively to calculate the indirect illumination contribution to the incoming radiance along one of the Monte Carlo sample rays. However, this is incredibly computationally expensive, and not used in this thesis.

Lastly, it is important to note that the number of samples used directly affects the presence of noise in the final image. Figure 5.2a illustrates how noise levels are reduced by increasing the number of Monte Carlo samples.

3.2 Point Based Color Bleeding

Point Based Color Bleeding is a technique for indirect illumination computation developed by Christensen at Pixar. It specifically addresses concerns relevant to movie production, namely: surface shading independent run time, reduced memory usage, and faster run times than Monte Carlo gather [8]. It leverages rasterization instead of ray-tracing to compute the indirect illumination component of a shading computation.

The PBCB algorithm approximates the radiance integral by gathering incoming radiance samples as rasterized cube-face pixels. When computing the indirect illumination at an intersection point, the hemisphere is approximated by a 5-faced cube, called a cube-map, where each face is an 8-by-8 pixel texture (see Figure

4.1). The primitives that are rasterized are called surfels (surface elements) and are generated as a preprocess.

Surfels are required due to the limitations and requirements of the rasterization step: rasterization cannot support complex geometry common in ray-tracing (e.g. non-planar surfaces, like spheres, or NURBS [26]), and in order to make it as fast as possible, it should not perform any complex shading. Each surfel is composed of a location, surface normal, radius, and color, and can be conceptualized as a disk (see Figure 3.2). In order to adequately capture the shading information with one color per surfel, the geometry is point-sampled at a desired density and a surfel is constructed per sample point. (As an aside: we should mention that [8] does not describe a point-sampling algorithm, and we have developed our own as described in Section 4.2.1) The surfel's location is defined by the sample point, its normal is defined by the source-geometry's surface normal at the sample point, its radius is defined such that there are no gaps between surfels, and its color is computed by shading the sample point using only direct illumination. In this way, a direct-shaded surfel cloud representation of a scene's geometry is generated, as in Figure 1.4, and stored in an octree (see Section 2.4). This spatial data structure is used in order to facilitate fast spatial lookups and provide a level of detail to the surfel representation when desired. That is to say: a hierarchical node in the octree can average all of its child surfels or nodes into an amalgamation and present them as a single surfel.

During the course of the standard ray-tracing algorithm, the PBCB technique is used to compute the indirect illumination component required to shade an intersection point. For one point, a 5-faced cube-map is constructed such that each face is composed of an 8-by-8 pixel texture. This approximates the hemisphere used in the Monte Carlo gather technique (see Figure 1.3). The surfels are then

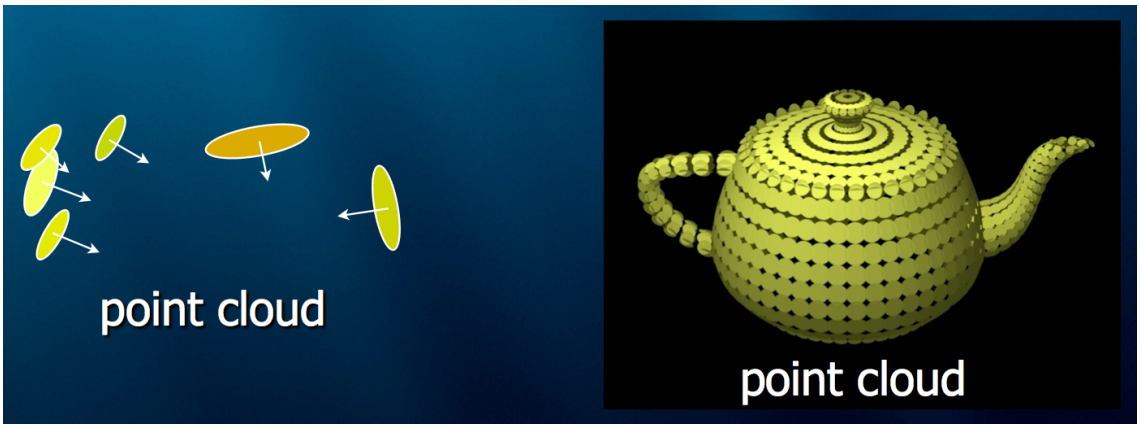


Figure 3.2: Surfels as disks, from Per H. Christensen's slides on the subject [9].

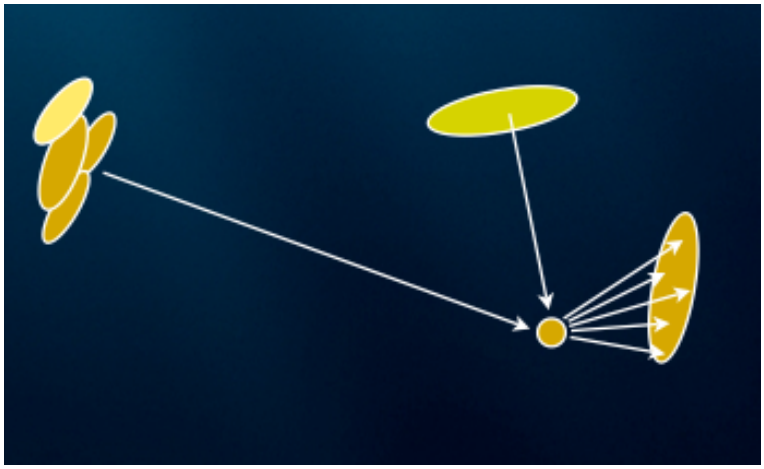


Figure 3.3: Close surfels are ray-traced, at average distance they are rasterized directly, and at long distance they are averaged [9].

convolved onto the cube-map using a proprietary algorithm [8]. What we do know about the algorithm is that, for very close surfels, rays are traced through the pixel to perform accurate shading, for average distance, the surfels are directly rasterized, and for distant surfels, a node from the surfel octree is used as an average representation of all the surfels in that area (see Figure 3.3). Figure 4.1 illustrates the result.

With the surfel cloud rasterized onto the cube-map, each pixel represents the

incoming radiance from its direction. The center of the pixel can be interpreted as if it were a sample point on the unit-hemisphere and the pixel's color can be weighted using the $n \cdot l$ factor to compute the incoming radiance value from that angle. These incoming radiance values are scaled by the surface color (see Section 2.2) and averaged to compute the final indirect illumination contribution to the shading computation for an intersection point.

Chapter 4

GPU Point Based Color Bleeding

4.1 Algorithm Introduction

Indirect illumination presents a computationally expensive problem. Potentially, the entirety of a scene's geometry could contribute illumination to any given point for which we try to compute indirect lighting. For Monte Carlo ray-tracing, intelligent randomness, spatial data structures, and attention to writing performant code can help alleviate the problem, but we can do better.

Our algorithm uses rasterization, rather than ray-tracing, to determine the radiance during the indirect shading computation. Rasterization has traditionally been used in real-time graphics due its high throughput, but we can leverage that same throughput to reduce our time spent calculating indirect illumination. However, we want to also gain hardware acceleration, and GPU rasterization algorithms require specific input, namely triangles. Thus, we preprocess our scene geometry to convert it into a triangle-based surfel cloud, and store it in GPU memory. At render time, we can leverage the GPU to quickly rasterize

these surfels, in parallel, onto cube-maps to capture the radiance at a point. Figure 4.1 visualizes the rasterization of surfels onto a cube-map.

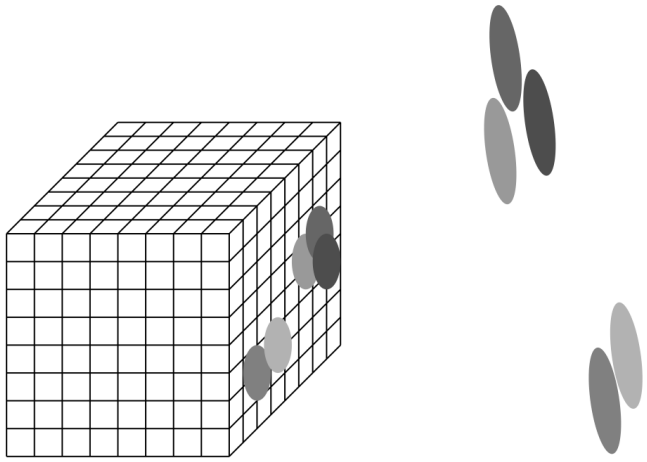


Figure 4.1: Surfels being rasterized onto an 8x8 cube [9].

In this way, each cube-map pixel represents the radiance at the location of the cube-map, in the direction of that pixel. This is analogous to the way Monte Carlo ray-tracing casts a ray and calculates shading at its scene intersection to represent radiance, in the ray's direction, at the ray's origin.

In this chapter, we further discuss our GPU Point-Based Color Bleeding algorithm: Section 4.2 explains our preprocess for generating triangle-based surfels, Section 4.3 follows with an explanation of our rendering pipeline. By combining our indirect illumination algorithm with GPU-hardware specifically designed for fast and parallel rasterization, we achieve quantitatively similar results in less time than Monte Carlo ray-tracing.

As an overview, the algorithm steps are as follows:

Algorithm 4.1 Psuedocode for our GPU Point-Based Color Bleeding algorithm.

```
function GPUPointBasedColorBleeding(scene)
    for all geomPrimitive in scene do
        surfels += GenerateSurfels(geomPrimitive, 500) //see alg. 4.6
    end for
    rays := GenerateRays(scene.imageDescription)
    for all ray in rays do
        intersection := Intersect(ray, scene)
        direct := DirectIllumination(intersection, scene)
        indirect := IndirectIllumination(intersection, scene)
        finalColor := direct + indirect
        image.writeToPixel(ray.index, finalColor)
    end for
    WriteToDisk(image)
end function
```

4.2 Surfel Generation

The PBCB algorithm relies on the abstract representation of a scene's geometry as surfels in order to leverage the power of rasterization. Real-time graphics applications also utilize rasterization, but do not use surfel representations. This is because they target specific framerates between 30 and 60 frames per second, allowing 16 to 33 milliseconds for the computation of Phong shading, texture mapping, bump mapping, and other techniques, such as dynamic shadows, on a per pixel basis. In the case of PBCB, rasterization must perform as fast as possible. To achieve this, additional computations of the kind referred to earlier (e.g. dynamic shadows) are avoided, and the rasterization algorithm simply uses the color of the surfel, computed as the direct illumination at the surfel's location on the source geometry, directly as the pixel's color.

The reason why the PBCB algorithm does not simply rasterize the geometric primitives as a whole is because each primitive would only be represented by one

color, resulting in inaccurate indirect illumination. Another benefit of surfels are that they are uniform in size and therefore are amenable to binning in spatial data structures. Our thesis does not incorporate a spatial data structure because the GPU does not require one itself and simply brute force rasterizes the entire surfel cloud in parallel. However, we discuss how a spatial data structure could be incorporated in our future work section on the topic (see Section 5.6.5).

In the traditional PBCB algorithm, surfels are disks, but our GPU PBCB algorithm cannot use this representation. The GPU is specifically designed to render images quickly by parallelizing the work, thus increasing throughput. However, we gain this performance at the cost of generality: the GPU can only process triangles as input data. Therefore, we must translate the representation of our geometric primitives into triangles (see Figure 4.2).

Each geometric primitive uses the same general algorithm to generate surfels, with differences in how the initial points are generated. We begin by generating a random distribution of points on the surface of the primitive, and use those points as the center for each triangular surfel. There has yet to be developed any algorithm with which to generate a uniformly random distribution of points on arbitrarily transformed geometry in linear time. Sampling untransformed geometry can be performed in linear time, but the subsequent transformation will result in a non-uniform distribution. In fact, Paul Bourke's methods in [6] and [7] require specific geometry that has not been transformed to generate a point distribution, and operate in polynomial time.

Our distribution algorithm is novel in its support for arbitrarily transformed geometry, but still requires polynomial time. However, because surfels are generated as a pre-process; the run time does not affect our results. In general, we calculate our point distributions by first generating random points on the sur-

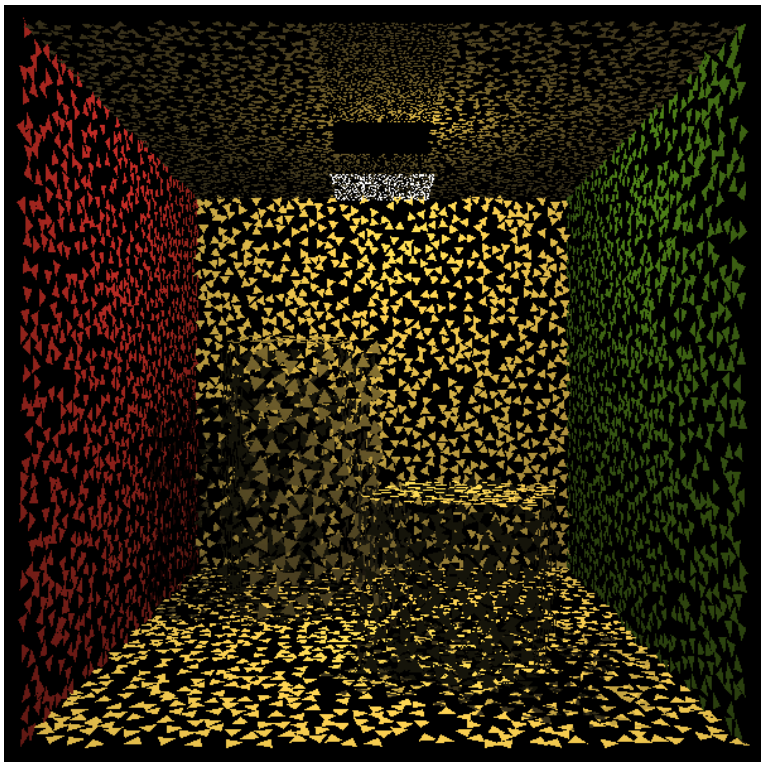


Figure 4.2: The surfels generated for our Cornell Box with area light. Note that the surfel size has been reduced to exhibit the surfel shape and distribution.

face of our untransformed geometry, transforming these points, then pruning the two closest, repeatedly, until we achieve the desired number of points. While in theory this algorithm does not achieve a mathematically uniform distribution, in practice it distributes the points well enough to achieve our goal of quantitatively similar results to Monte Carlo ray-tracing, is simple to understand and implement, and meets our requirement of supporting arbitrarily transformed geometry. One major limitation of this algorithm is that it sacrifices extra work at the cost of simplicity: in the case of untransformed geometry, we could generate exactly the number of points required, but instead we proceed with the oversampling and culling without it being a necessity.

Once a point, p, has been computed, the three triangle vertices are generated

by adding the point's surface tangent vector, v, to p to solve for the first vertex, then we rotate v by 120 degrees to solve for the second, and once more for the third. The length of v is determined by multiplying the distance between the two closest sample points on the geometry and a constant. The goal is to have coverage over the surface of the geometry without superfluous gaps or overlaps. We have found that doubling the distance results in satisfactory coverage.

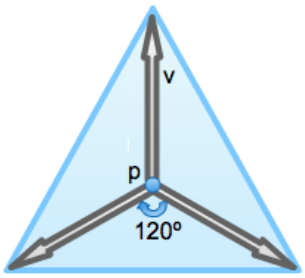


Figure 4.3: The construction of our triangular surfels; v is the surface tangent, at the sample point, p. Its length is determined by the two closest sample points on a given geometric primitive, multiplied by two. The other vertices are constructed by rotating the first by 120°.

We discuss the general surfel generation algorithm in Section 4.2.2. We also discuss the specific point generation algorithms for boxes, spheres, and triangles in Section 4.2.1. In Section 4.2.2 we discuss our algorithm for constructing triangular surfels, and in Section 4.2.3 we discuss our storage method for these surfels.

4.2.1 Point Generation

In order to generate surfels for any of our geometric primitives, we first generate a point distribution to act as center points for our surfels. The strategy we use for point distribution is stratified stochastic sampling.

Box

Points for the box primitive are generated per side. Each side is divided into a u-v grid, with a point for each intersection. We jitter the points to add randomness, which helps during the culling process explained later in this section, and to help mask any artifacting that can result from regular patterns. The pseudocode can be seen in Algorithm 4.2. Point distributions for a box can be seen in Figure 4.5.

Sphere

Point generation for a sphere is based on a uniform sampling algorithm [24] that requires two uniform random variables as input. Instead of using entirely uniform random variables, u and v, we step through u and v coordinates, which effective subdivides the sphere into a grid. Again we jitter the coordinates to help randomize our surfel cloud. The psuedocode can be seen in Algorithm 4.3. Point distributions for a sphere can be seen in Figure 4.7.

Algorithm 4.2 Generate stratified stochastic sample points for a box.

function SAMPLEGEOMETRY($box, numSamples$)
  $pointsPerSide := numSamples/6$
  $pointsPerDim :=$ sqrt($pointsPerSide$)
  for each *side* **in** *boxSides* **do**
    for $u := 0...pointsPerDim$ **do**
      for $v := 0...pointsPerDim$ **do**
        $\alpha := (u+\text{rand}(0,1))/pointsPerDim$
        $\beta := (v+\text{rand}(0,1))/pointsPerDim$
        if *side* = small z-plane **then**
          $x := (1-\alpha) * box.start.x + \alpha * box.end.x$
          $y := (1-\beta) * box.start.y + \beta * box.end.y$
          $z := box.start.z$
        else if *side* = large z-plane **then**
          $x := (1-\alpha) * box.start.x + \alpha * box.end.x$
          $y := (1-\beta) * box.start.y + \beta * box.end.y$
          $z := box.end.z$
        else if *side* = small x-plane **then**
          $x := box.start.x$
          $y := (1-\alpha) * box.start.y + \alpha * box.end.y$
          $z := (1-\beta) * box.start.z + \beta * box.end.z$
        else if *side* = large x-plane **then**
          $x := box.end.x$
          $y := (1-\alpha) * box.start.y + \alpha * box.end.y$
          $z := (1-\beta) * box.start.z + \beta * box.end.z$
        else if *side* = small y-plane **then**
          $x := (1-\alpha) * box.start.y + \alpha * box.end.y$
          $y := box.start.y$
          $z := (1-\beta) * box.start.z + \beta * box.end.z$
        else if *side* = large y-plane **then**
          $x := (1-\alpha) * box.start.y + \alpha * box.end.y$
          $y := box.end.y$
          $z := (1-\beta) * box.start.z + \beta * box.end.z$
        end if
        $point := box.modelTransform * <x, y, z>$
        $points \mathrel{+}= point$
      end for
    end for
  end for
  return *points*
end function

Algorithm 4.3 Generate stratified stochastic sample points for a sphere.

```
function SampleGeometry(sphere, numSamples)
    numPointsU := sqrt(numSamples)
    numPointsV := numSamples/numPointsU
    for i := 0...numPointsU do
        for j := 0...numPointsV do
            u := (i+rand(0,1))/numPointsU
            v := (j+rand(0,1))/numPointsV
            z := 1 − 2 ∗ u
            r := sqrt(max(0, 1 − z ∗ z))
            φ := 2 ∗ π ∗ v
            x := r∗ cos(φ)
            y := r∗ sin(φ)
            point := <x, y, z> ∗ sphere.radius + sphere.center
            point := sphere.modelTransform ∗ point
            points += point
        end for
    end for
    return points
end function
```

Triangle

We generate our triangle sample points by modifying a uniform sample pattern algorithm [24] in a similar manner to our sphere point generation. The original algorithm requires two uniformly random variables as input, and will generate a uniformly distributed random point on the triangle. By discretizing the u and v values and stepping through them, we effectively walk a uniformly distributed grid across the triangle's surface area. Again, we jitter the sample point location in order to eliminate any regular patterns. The psuedocode can be seen in Algorithm 4.4. Point distributions for a triangle can be seen in Figure 4.9.

Algorithm 4.4 Generate stratified stochastic sample points for a triangle.

```
function SampleGeometry(triangle, numSamples)
    numPointsU := sqrt(numSamples)
    numPointsV := numSamples/numPointsU
    v_{0..1} := triangle.vertex1 − triangle.vertex0
    v_{0..2} := triangle.vertex2 − triangle.vertex0
    for i := 0...numPointsU do
        for j := 0...numPointsV do
            u := (i+rand(0, 1))/numPointsU
            v := (j+rand(0, 1))/numPointsV
            // Inside or outside the triangle?
            if u + v < 1 then
                point := triangle.vertex0 + u ∗ v_{0..1} + v ∗ v_{0..2}
            else
                point := triangle.vertex0 + (1 − u) ∗ v_{0..1} + (1 − v) ∗ v_{0..2}
            end if
            point := triangle.modelTransform ∗ point
            points += point
        end for
    end for
    return points
end function
```

With the previous point distribution algorithms, we must address the issue of non-uniform scaling. For example, if we are generating points for a box scaled more in the y-axis than the x- or z-axis, then our computed sample points will be further spaced apart along that axis. This leaves us with a sampling pattern that is not uniform, but stretched in one direction.

Our solution to this problem is simple, easy to implement, and novel in the fact that it is geometry and transformation independent. We generate a multiple of the requested number of sample points on the untransformed geometry, jitter their location, transform them using the source geometry's model transformation, and repeatedly cull the two closest points until we arrive at the desired number of points. In this way, we work backwards to a more uniform distribution in a

way that supports arbitrary geometric topology and transformations. It is important to note that our algorithm discovers the two closest points using a $O(n^2)$ algorithm. This could potentially be accelerated with a spatial data structure (see Section 5.6.5). Figures 4.5, 4.7, and 4.9 show the result of this process for varying multiples of the requested number of surfels; notice the coverage issues caused by not generating more than the requested number of surfels (e.g. Figure 4.5a).

The pseudocode is listed in Algorithm 4.5.

Algorithm 4.5 Create points and repeatedly cull one of the two closest points.

```
function GeneratePoints(geomPrimitive, numberOfSamples)
    points := SampleGeometry(geomPrimitive, 2 * numberOfSamples) //alg. 4.2, 4.3, 4.4
    while points.length > desiredNumPoints do
        minPointIndex := FindAndRemoveClosestPoints(points)
        points.remove(minPointIndex)
    end while
    return points
end function
```

4.2.2 Surfel Generation

With the points generated, our surfel generation algorithm is quite simple: it consists of using the sample points as the center for triangular surfels. The question we must answer is what surface area will achieve adequate coverage of the source geometry. Too large of a surface area results in extending our surfel borders beyond the edges of our geometry as well as excessive surfel overlapping. Too small of a surface area results in gaps between surfels. Both of these issues will cause artifacts in our rendering by misrepresenting the source geometry.

Our algorithm uses the distance between the two closest sample points as the theoretical radius required for disk surfels to adequately cover the geometry.

In practice, we double this radius in order to ensure coverage. In order for our triangle surfels to match this surface area, we extrapolate the length of the vector from triangle surfel center to equilateral vertex using Equation 4.1.

$$vectorLength = \sqrt{\frac{\pi * radius^2}{\sqrt{3} * \sin^2(60)}} \tag{4.1}$$

However, the random jittering of sample points often results in initial sample points that are very close together. While the jittering produces the desired randomness, it also gives us too low an estimate of the required radius; the result of which can be seen in Figure 4.6a. But the aforementioned point culling solution (see Section 4.2.1) also solves this problem. By generating a multiple of the desired number of points, and culling the closest ones, we create a more uniform distribution with less variance in the distance between adjacent sample points. We have found that a multiple of two produces visually acceptable results, while greater multiples do not contribute further. The results of varying number of generated points (culled to 500) are illustrated in Figures 4.5, 4.6, 4.7, 4.8, 4.9, and 4.10.

For typical surfel generation times per geometric primitive, see Table 4.1. In psuedocode, our surfel generation algorithm is listed in Algorithm 4.6. Figure 4.4 is a rendering of the raw surfel cloud in OpenGL without any lighting.

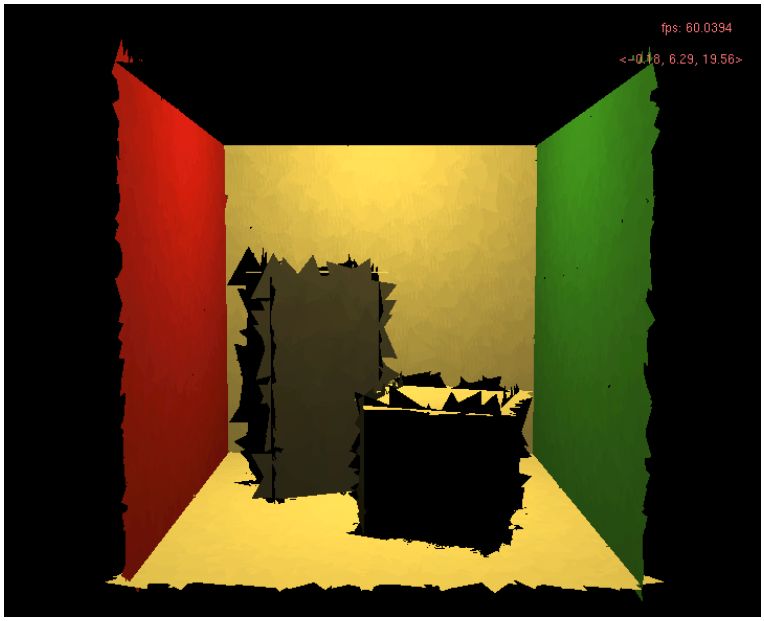


Figure 4.4: Surfels generated for the Cornell Box scene rendered, without lighting, in OpenGL using a VBO.

Algorithm 4.6 Generate surfels from points on a geometric primitive.

function GENERATESURFELS($geomPrimitive, numberOfSurfels$)
  $points$:= GeneratePoints($geomPrimitive, numberOfSurfels$) //see alg. 4.5
  $minD$:= FindClosestPoints($points$)
  $vectorLength$:= 2∗sqrt(($PI * minD * minD$)/1.299) //see eqn. 4.1
  for all *point* **in** *points* **do**
    $tangent$:= CalculateTangent($point, geomPrimitive$)
    $surfel$:= new surfel
    $surfel.v0$:= $point + vectorLength * tangent$
    $surfel.v1$:= $point + vectorLength$∗rotate($tangent$, 120)
    $surfel.v2$:= $point + vectorLength$∗rotate($tangent$, 240)
    $surfels$.add($surfel$)
  end for
  return $surfels$
end function

	500 pts.	1000 pts.	2000 pts.	4000 pts.	8000 pts.
Box	1ms	430ms	5s 927ms	43s 104ms	6m 24s 819ms
Sphere	1ms	714ms	6s 274ms	51s 286ms	6m 50s 309ms
Triangle	1ms	713ms	6s 341ms	51s 279ms	6m 49s 847ms

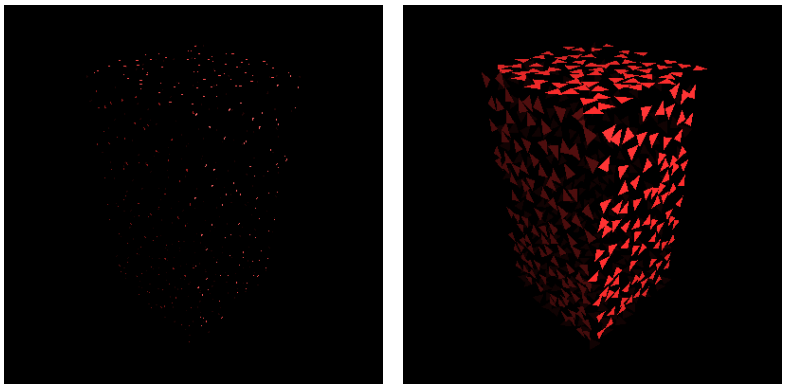
Table 4.1: Surfel generation times listed by geometric primitive versus number of initial points.

4.2.3 Surfel Storage

The surfels are generated on the CPU, but they must be rendered by the GPU. Therefore, it is logical to store them in GPU memory. This avoids the latency of transferring data over the CPU-to-GPU bus, which we discuss in Section 2.6. For this purpose we leverage the OpenGL Vertex Buffer Object, or VBO. This data structure is an array of vertices paired with colors and normals. Additionally, we never update the surfel vertex data, so we can store them on the GPU during our pre-process where they persist throughout our rendering.

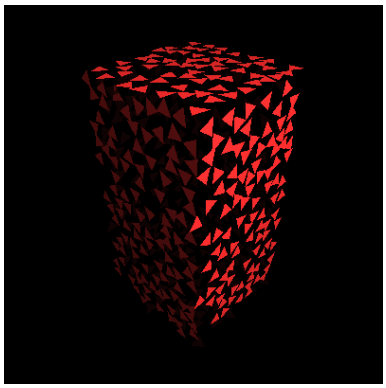
Additionally, it is important to note that in Christensen's PBCB implementation described in [8] he stores the surfels in an octree (see Section 2.4). This is because, if the surfels being rasterized are far away, it can be more efficient to group them together and treat them as one large surfel, with properties that are the average of the surfels it represents. The octree data structure lends itself to this practice as these large, coalesced, surfels can be constructed at octree nodes during the pre-process and used at render time with no additional computation. In fact, there can be quite a bit less computation, as one surfel is rasterized instead of many.

We do not implement this feature because our implementation renders it irrelevant. The purpose of the octree in Christensen's PBCB implementation [8] is to reduce the number of surfels rasterized. But our GPU implementation

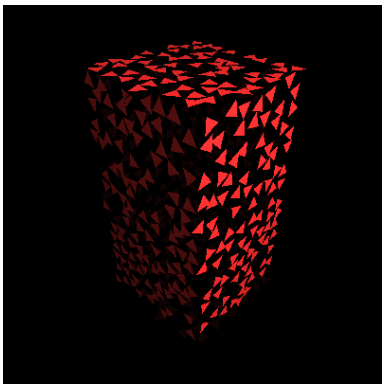


(a) 500 points (1ms)

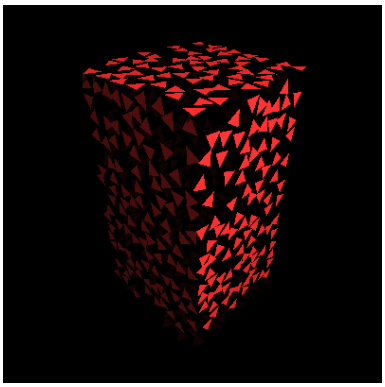
(b) 1000 points (430ms)



(c) 2000 points (5s 955ms)

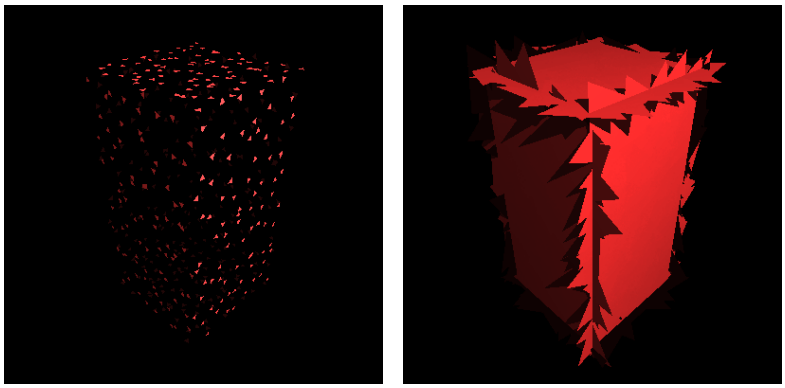


(d) 4000 points (43s 88ms)



(e) 8000 points (6m 24s 803ms)

Figure 4.5: Surfel generation for a box, scaled in the y-axis, varying from 500 to 8000 initial points at quarter size. Generation times included in parenthesis.



(a) 500 points (1ms)

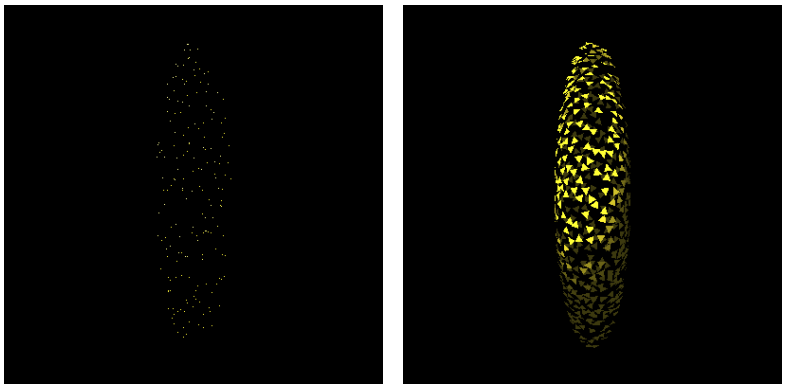
(b) 1000 points (430ms)

(c) 2000 points (5s 927ms)

(d) 4000 points (43s 104ms)

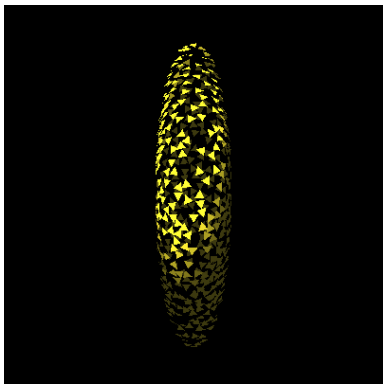
(e) 8000 points (6m 24s 819ms)

Figure 4.6: Surfel generation for a box, scaled in the y-axis, varying from 500 to 8000 initial points at full size. Generation times included in parenthesis.

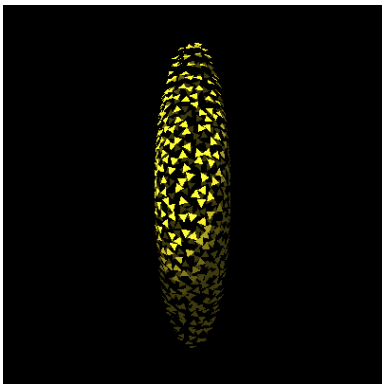


(a) 500 points (1ms)

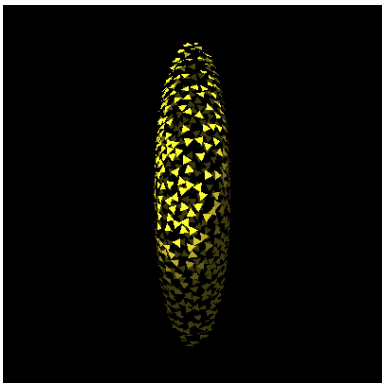
(b) 1000 points (708ms)



(c) 2000 points (6s 268ms)

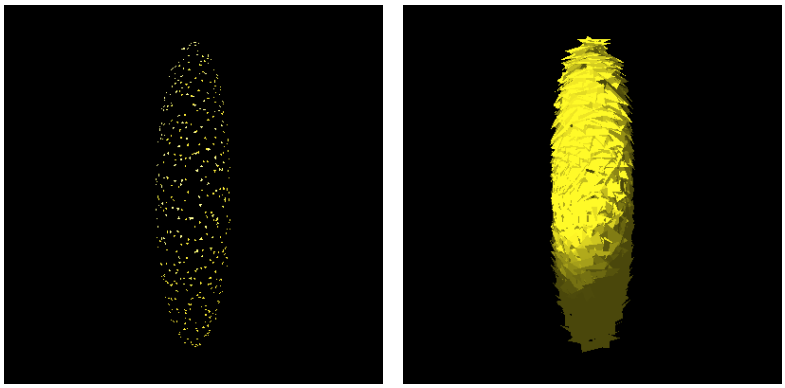


(d) 4000 points (51s 689ms)



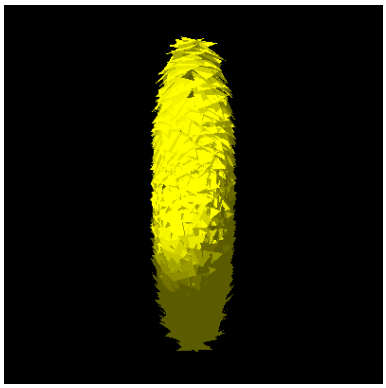
(e) 8000 points (6m 49s 391ms)

Figure 4.7: Surfel generation for a sphere, scaled in the y-axis, varying from 500 to 8000 initial points at quarter size. Generation times included in parenthesis.

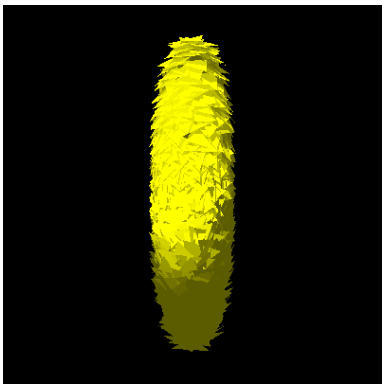


(a) 500 points (1ms)

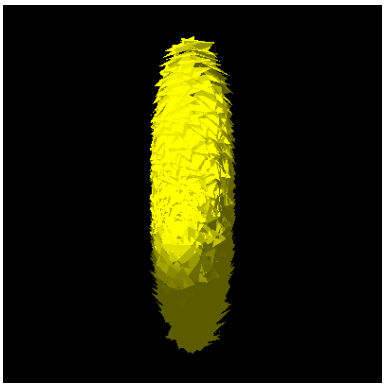
(b) 1000 points (714ms)



(c) 2000 points (6s 274ms)



(d) 4000 points (51s 286ms)

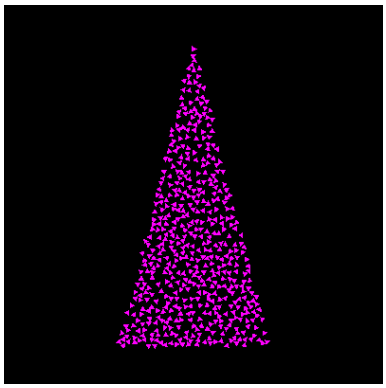


(e) 8000 points (6m 50s 309ms)

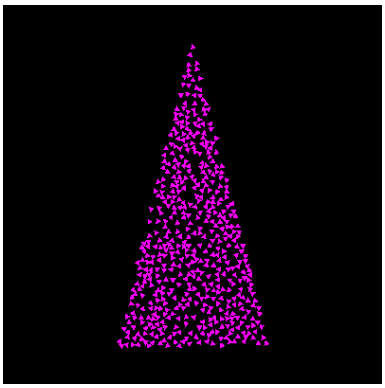
Figure 4.8: Surfel generation for a sphere, scaled in the y-axis, varying from 500 to 8000 initial points at full size. Generation times included in parenthesis.

(a) 500 points (1ms)

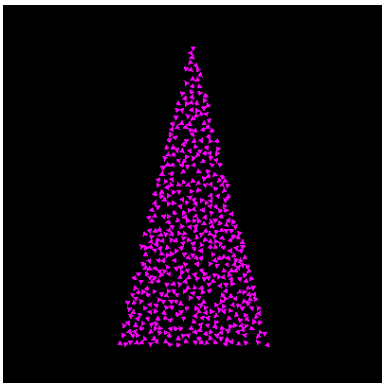
(b) 1000 points (703ms)



(c) 2000 points (6s 283ms)

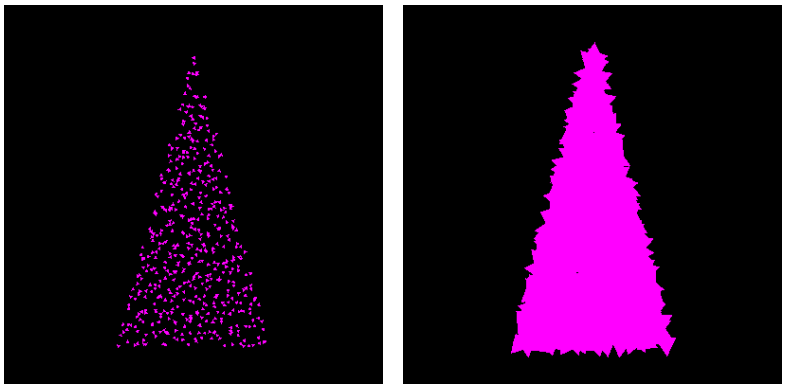


(d) 4000 points (51s 432ms)



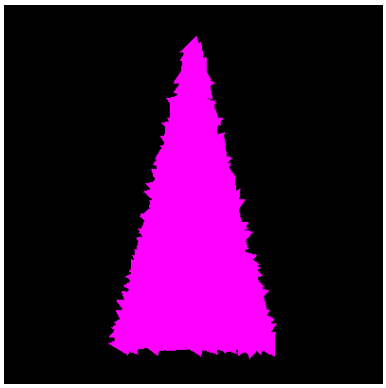
(e) 8000 points (6m 49s 630ms)

Figure 4.9: Surfel generation for a triangle, scaled in the y-axis, varying from 500 to 8000 initial points at 1/4 size. Generation times included in parenthesis.



(a) 500 points (1ms)

(b) 1000 points (713ms)



(c) 2000 points (6s 341ms)

(d) 4000 points (51s 279ms)

(e) 8000 points (6m 49s 847ms)

Figure 4.10: Surfel generation for a triangle, scaled in the y-axis, varying from 500 to 8000 initial points at full size. Generation times included in parenthesis.

rasterizes the entirety of the surfel cloud in one rendering pass, we do not require this optimization, and therefore avoid the system complexity it introduces.

Furthermore, it would actually diminish our performance to implement the octree feature as it stands in [8]. The VBO stored in GPU memory is static, is marked as such, and is therefore optimized by the OpenGL subsystem to remain on the GPU. If we update the geometry we intend to render (i.e. the VBO), then we incur the cost of at least one CPU-to-GPU transfer of data. We find this unacceptable for our goals, and therefore omit this implementation detail of PBCB.

4.3 Rendering

Our rendering algorithm follows standard ray-tracing as discussed in Section 1.2, with the exception of the indirect lighting computation. The pseudocode is listed in Algorithm 4.1.

The indirect lighting computation uses our GPU PBCB algorithm and is the focus of this thesis. We describe the algorithm in detail in Section 4.3.2, discuss the performance characteristics in Section 5, analyze our results in Section 5.3, and discuss future work in Section 5.6.

4.3.1 Ray-Tracing

Our ray-tracing algorithm follows the standards set in publications such as Peter Shirley's 'Fundamentals of Computer Graphics' [31]. First, we generate a list of rays, one per pixel, by iterating over all final-image pixels. As input we require a virtual camera definition, which provides a world-space location

and field of view characteristics such as width and height of the final image. A ray is composed of an origin point, inherited from the virtual camera's location, and direction, calculated as the vector between the origin and associated pixel's center.

We then iterate over the list of rays, tracing them against the scene geometry to calculate their intersection point. The intersection algorithms are based on concepts from [24], and [35].

Once we solve for the ray's intersection point with a geometric object, we calculate lighting. Because light calculations are additive [24], we can split the calculation into a direct and an indirect computation.

Our direct lighting calculation is quite simple: in the case of point lighting, we simply trace one shadow ray [24] and if the path to the light is unobstructed, then we calculate standard Phong shading (see Section 2.2), and in the case of area lighting we trace multiple shadow rays, in a uniform random distribution over the area light geometry, and scale the shading by the percent of unobstructed rays.

4.3.2 Indirect Gather via Rasterization (GPU PBCB)

By using the GPU's ability to rasterize triangles, we capture the incoming radiance at a ray intersection point by rasterizing our surfel cloud onto five 8 by 8 pixel textures arranged into a cube about the point. The cube represents the hemisphere used in the radiance integral (Section 2.1) . Using this technique, we attempt to realize a speedup over Monte Carlo ray-tracing as well as software-based PBCB. The psuedocode is listed in Algorithm 4.7.

Our implementation utilizes the OpenGL API, along with helper libraries:

Algorithm 4.7 Psuedocode for our indirect illumination algorithm.

```
function IndirectIllumination(intersection, scene)
    tangent := GramSchmidtTangent(intersection.normal) //see alg. 3.1
    for i := 0...5 do
        camera := ConstructCamera(i, intersection.point, intersection.normal, tangent)
        pixelData := Rasterize(camera)
        for j := 0...64 do
            x := j % 8
            y := j / 8
            right := −(camera.up×camera.direction)
            textureCenter := camera.location+(nearPlane ∗ camera.direction)
            a := −(7/8)
            b := 1/4
            //see fig. 4.14
            jPixelCenter := textureCenter + (a ∗ nearPlane + b ∗ nearPlane ∗ x) ∗ right +
(a ∗ nearPlane + b ∗ nearPlane ∗ y) ∗ camera.up
            sampleDirection := normalize(jPixelCenter − camera.location)
            weight :=clamp(intersection.normal·sampleDirection, 0)
            if weight > 0 then
                indirectColor += weight ∗ pixelData[j] ∗ intersection.surfaceColor
                totalPixels+ = 1
            end if
        end for
    end for
    indirectColor /= totalPixels
    return indirectColor
end function
```

GLUT and GLEW. Therefore, as a preamble to the algorithm, we must initialize OpenGL. This requires creating the OpenGL rendering context and associated buffers, and setting some constant state. In our case, we require an 8 by 8 pixel color texture and depth texture, and a vertex buffer with which we store our surfel data. Lastly, we set the constant rendering state by disabling OpenGL lighting such that the color we calculate in the surfel generation preprocess will be used directly, and by binding our textures and buffers, which informs OpenGL on how to use each buffer (e.g. render to color texture, or where vertex, color, and surface normal data is per triangle in the VBO).

As discussed in Section 4.3.1 and 1.2, the geometry intersection for each ray is computed, and then we calculate our indirect and direct illumination separately, combining later, because light is linear [24].

We begin our indirect illumination calculation for each intersection point by constructing five cameras that hold the render state required to rasterize each cube face. The cameras store their location, direction, up-vector, field of view, and near and far planes. The location for each camera is inherited from the intersection point. The direction is the intersection point's surface normal for the camera representing the top of the cube, and for the remaining cameras, we use the surface tangent, rotated in 90° increments. The up-vector for the cube-side cameras is the surface normal, and the top camera uses one of the side camera's direction vector. The field of view is set to 90° for each camera such that their viewing frustums fit together with no gaps (see Figures 4.11 & 4.12). The near plane is set to 0.01, and the far plane is set to 15, which we have found offers good results for our test scene because the height of the Cornell Box is 10 units.

Once the cube-face cameras are constructed, we iterate over each, and ras-

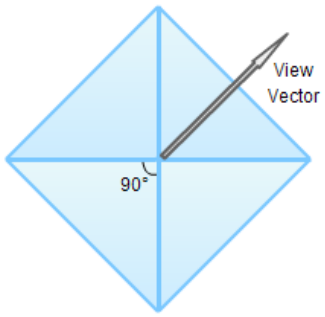


Figure 4.11: Top view of the cameras that comprise our cube map.

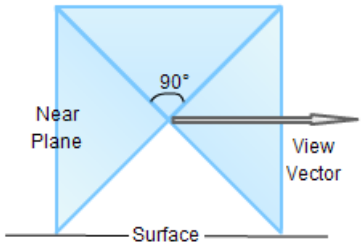


Figure 4.12: Side view of the cameras that comprise our cube map.

terize the surfel cloud onto an 8 by 8 pixel texture at the camera's near plane (see Figure 4.14). This texture data must then be read back from the GPU, and convolved into one indirect illumination value. We do this by iterating over each pixel, and adding its contribution, calculated using the diffuse component calculation of the Phong reflectance model (see Section 2.2), to the final color.

Because we calculate the pixel's weight using Lambert's Law, the bottom half of the cube-face sides are effectively not contributing due to their $n \cdot l$ factor being negative. This issue is visualized in Figure 4.13. This problem is unavoidable, however, because the cameras' view frustums must be equal in dimensions in order to properly emulate the hemisphere with uniform distribution.

Once each cube face has been rasterized and convolved, we return the final color as our indirect illumination value at the intersection point.

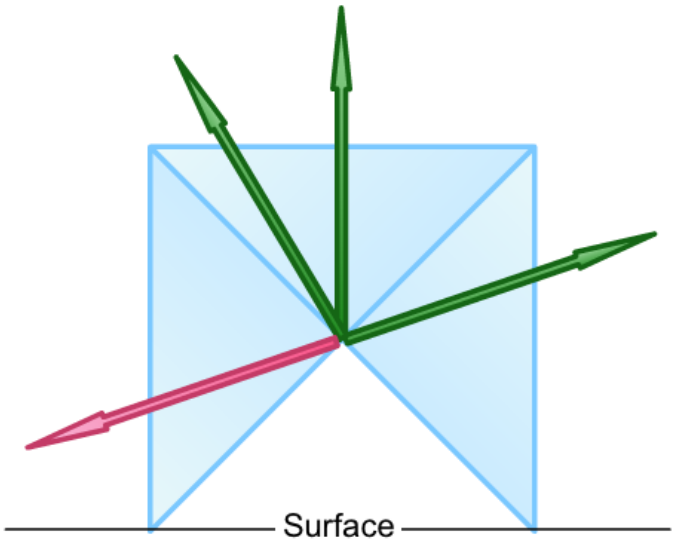


Figure 4.13: A few example rays, used as the l in the $n \cdot l$ weighting factor for the pixel color values, during cube-face convolution. The red ray designates that the pixel will not contribute due to a negative $n \cdot l$ factor.

4.3.3 Final Color Computation

With the direct illumination calculated using the typical Phong shading technique [25], and the indirect illumination calculated via our algorithm, the final color is computed by adding the resultant values. This can be done because of the fact that light is linear [24].

We then write the final color value to our image in memory. We store the final image in RAM until the rendering is complete, then write it to disk as an uncompressed TARGA image file.

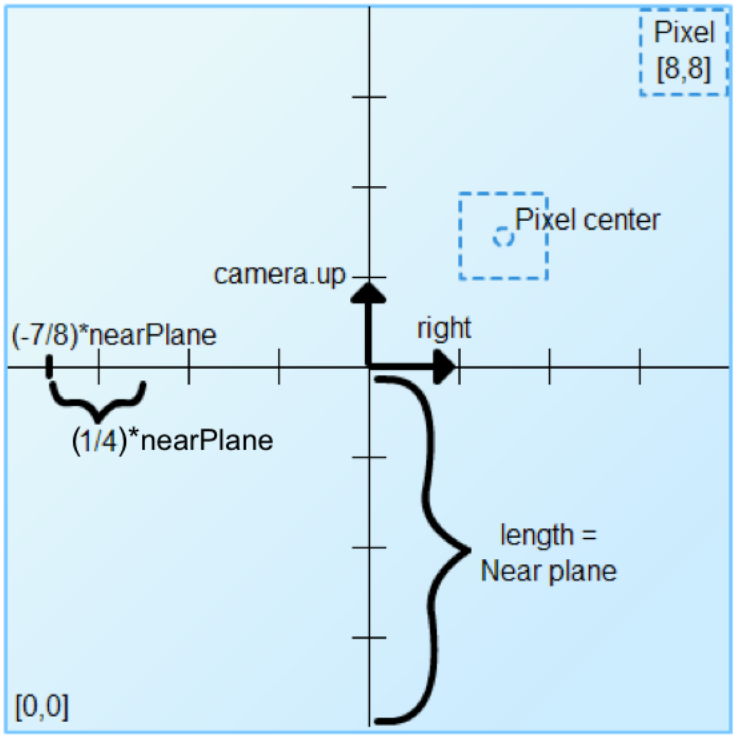


Figure 4.14: View of the texture that is rasterized at the near plane of the camera.

4.4 Review

We have presented a rendering algorithm that achieves an order of magnitude speedup over Monte Carlo ray-tracing on certain hardware, namely heterogeneous chip architectures, with the knowledge that the future work discussed in Section 5.6 would extend our results to all hardware.

As an integral step of the algorithm, we have discussed our novel surfel generation algorithms for each of our supported primitives. Our surfel generation algorithms not only generate the triangles required by GPU-hardware for rasterization, but produce reasonably uniform coverage for arbitrarily transformed geometry.

Lastly, we discussed our indirect illumination computation: how we store

our surfel cloud in GPU memory, rasterize it on to cube-map faces, and finally, convolve the cube-map pixels into one final color.

Chapter 5

Results and Discussion

In this section we demonstrate that our GPU PBCB algorithm performs well on two types of graphics hardware, rendering one of the most ubiquitous scenes in all of computer graphics: the Cornell Box [14]. We achieve faster render times at the cost of memory usage. Our rendered images display interesting global illumination effects such as color bleeding due to indirect illumination, and are comparable to those produced using the slow, but accurate, Monte Carlo ray-tracing algorithm.

5.1 Test Environment

Our test results were gathered on a 2011 MacBook Air [5]: 1.8Ghz Intel Core i7 (Sandy Bridge i7-2677M) [16], 4 GB 1333MHz DDR3 SDRAM, Intel HD 3000 Graphics (on CPU-die) with 384MB VRAM, and a solid-state (flash) hard disk. Our traditional architecture was represented by a 4.0GHz Intel i7-920 [15], with 6GB 1333MHz DDR3 SDRAM, ATI Radeon HD 5970 with 1GB VRAM [3], and a standard 7000RPM hard disk. For our software PBCB results, we forced

OpenGL into a software-only mode.

5.2 Test Scene

Our test scene is an adapted Cornell Box. The Cornell Box was originally developed to quantify the difference between a Cornell renderer [14] and a real photograph of the same scene. However, due to its simple and elegant design, it has been adopted by the field of computer graphics as a standard for global illumination comparison (see [11, 20, 32, 33]). Since its popularization circa 1985, it has become iconic and ubiquitous within the world of computer graphics.

This scene was chosen for its ability to showcase the subtleties of indirect illumination. The overhead area light casts clearly visible soft shadows around the cubes, and the color bleeding from the walls of the Cornell Box is prominently displayed in those shadows. Another demonstrative feature is that no direct light is cast on the ceiling; it is illuminated by purely indirect light.

Our Cornell Box is defined in a modified POV-Ray format [22], developed for the CSC 473 course at California Polytechnic State University, San Luis Obispo. It is comprised of 16 triangles, which make up the Cornell box, and 2 rotated and scaled cubes. This results in 14,000 surfels, following the algorithm in Section 4.2.

5.3 Analysis

Our analysis is threefold, based on the following metrics:

1. time to render one image

2. perceived image difference and quality

3. memory requirements

We also discuss scalability in terms of geometric and shading complexity.

5.3.1 Speed

To analyze the speed of our GPU PBCB algorithm, we render our Cornell Box scene and compare it against both a Monte Carlo algorithm, and a forced software-mode of OpenGL. We demonstrate a 41.65x speedup over traditional Monte Carlo method, as well as a 3.12x speedup over the software-based renderer used in traditional PBCB, represented by an OpenGL software rasterizer. These results are collected in Table 5.1.

Algorithm	**Time (traditional)**	**Time (heterogeneous)**
16 MC samples	525 sec	565 sec
32 MC samples	790 sec	862 sec
64 MC samples	1962 sec	2137 sec
128 MC samples	3682 sec	4063 sec
256 MC samples	7756 sec	8205 sec
Software PBCB	N/A	615 sec
GPU PBCB	236 sec	197 sec

Table 5.1: Render times for a 500x500 Cornell Box scene using our GPU PBCB algorithm compared against Monte Carlo with varying numbers of samples and traditional software-based PBCB. We include data for both traditional system architectures and heterogeneous architectures where the GPU and CPU share their die.

As evidenced by our results from image comparisons in Section 5.3.2, we do not reach a visually acceptable noise level in the Monte Carlo algorithm until 256 samples. Therefore, we will frame our analysis of run time by comparing our GPU PBCB algorithm against the 256 sample Monte Carlo render. In this

Algorithm	Time Diff. (traditional)	Time Diff. (heterogeneous)
16 MC samples	+123%	+187%
32 MC samples	+235%	+338%
64 MC samples	+731%	+985%
128 MC samples	+1460%	+1962%
256 MC samples	+3186%	+4065%
Software PBCB	N/A	+212%
GPU PBCB	baseline	baseline

Table 5.2: Differences in render times, as percent difference, for a 500x500 Cornell Box scene using GPU PBCB as a baseline compared against Monte Carlo with varying numbers of samples and traditional software-based PBCB. We include data for both traditional system architectures and heterogeneous architectures where the GPU and CPU share their die.

case, we attain a 41.65x speedup on our heterogeneous architecture system, and a 32.86x speedup on our traditional architecture system.

Another important feature of our work is the fact that we extend the basic PBCB implementation to leverage the GPU hardware to accelerate our algorithm. We were able to capture data for a software-based rasterizer by utilizing the ability to force a software-mode OpenGL context. This was not possible on our traditional architecture as it only supported a software-based OpenGL 1.0 context, while we require OpenGL 2.0 features. However, we were able to force our heterogeneous architecture system to use a software-based rendering context, and achieved a 3.12x speedup. We expect this speedup to be present on a traditional architecture system, but not to the same extent as it runs our algorithm 19.8% slower in our tests.

5.3.2 Image Quality

In order to compare image quality, we must determine a baseline for comparison. For this we use rendered images from our Monte Carlo algorithm. Specifi-

cally, we use an image generated using 256 samples in our Monte Carlo ray-tracer because we have found this to offer a visually acceptable level of noise. See Figure 5.1 for noise comparisons between varying numbers of samples.

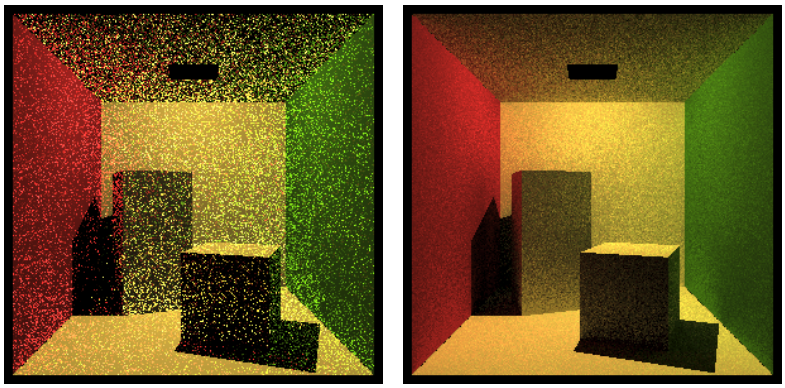
Using the mean absolute error image comparison algorithm implemented in ImageMagick compare [1], we can determine the difference between our image rendered using our Monte Carlo ray-tracer with 256 samples per primary ray, and the image from our GPU PBCB algorithm. The results of comparison with images rendered using varying numbers of samples in the Monte Carlo algorithm are collected in Table 5.3.

Algorithm	**MAE**
16 MC samples	3.22%
32 MC samples	2.78%
64 MC samples	2.12%
128 MC samples	1.83%
256 MC samples	1.76%
GPU PBCB	baseline

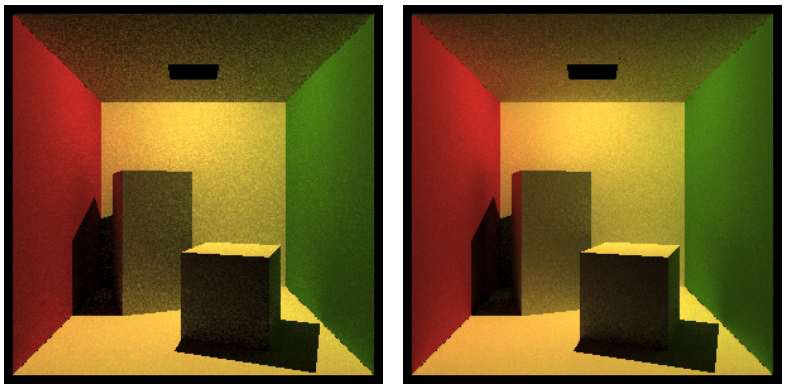
Table 5.3: Image difference for the Cornell Box scene with the emissive geometry area light using our GPU PBCB algorithm compared against Monte Carlo with varying numbers of samples and traditional software-based PBCB. The listed values are mean absolute error computed as the average over each pixel pair and each color channel.

The error percent values listed in Table 5.3 show us that we have produced very similar images. This can be verified visually as well in Figure 5.2. Our GPU PBCB algorithm produces renders that are most similar to the 256 sample Monte Carlo render, which is only 1.76% different on average. This makes intuitive sense as the noise in the lower sample rate renders causes a greater difference in appearance.

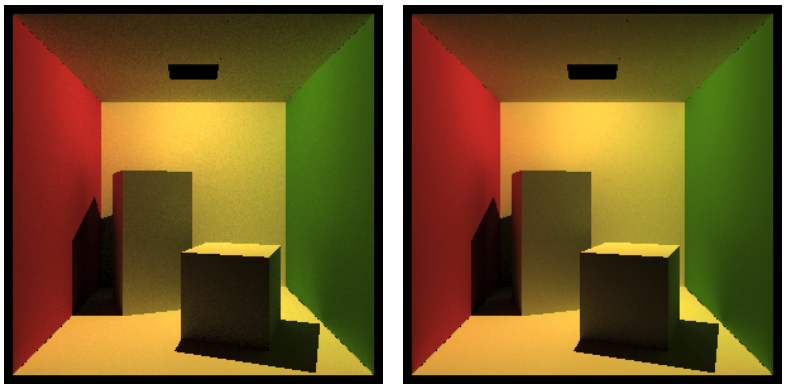
The error values are measures of mean absolute error. This is calculated by computing the difference between pairwise pixel's color channels and averaging



(a) 1 sample (b) 16 samples



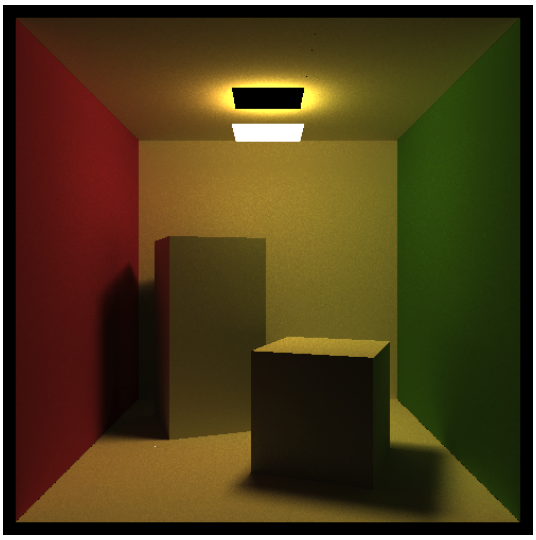
(c) 32 samples (d) 64 samples



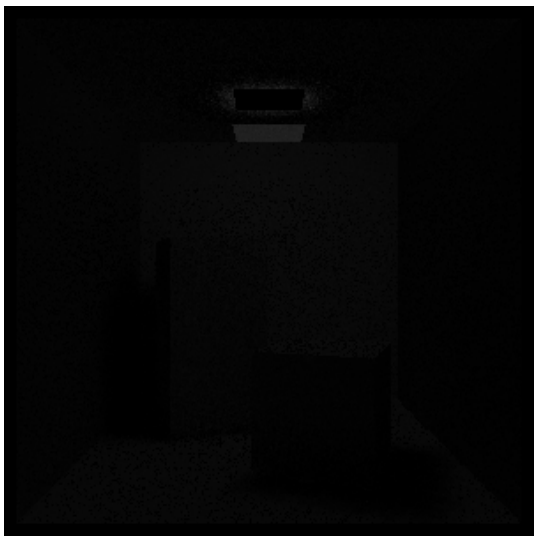
(e) 128 samples (f) 256 samples

Figure 5.1: Noise results from varying numbers of Monte Carlo samples in a Cornell Box with point light.

(a) GPU PBCB



(b) 256 sample Monte Carlo



(c) Image Difference

Figure 5.2: Our Cornell Box rendered with both 256 sample Monte Carlo and our GPU PBCB algorithms. Included is an image produced by using the distance between two colors for all color channels per pixel.

them over the entire set of pixels. For example, if our GPU PBCB algorithm produces a pixel with color values <0.45, 0.50, 0.45>, and the 256 sample Monte Carlo algorithm produces a pair pixel with values <0.1, 0.2, 0.1>, then the pairwise distance values are <0.35, 0.3, 0.35>, and the mean absolute error would be the average of these distances, i.e. 0.3333 or 33.33%.

Lastly, we have generated a difference image that was computed by using the absolute value of the difference between pairwise pixels in the two renders produced by our GPU PBCB algorithm and the 256 sample Monte Carlo render. This allows us to visualize the difference between renders.

It is important to note that the largest area of difference is the emissive geometry at the top of our Cornell Box. The difference is greatest here because of a limitation of our algorithm. Because we store direct illumination shaded surfels as triangles in a VBO on the GPU, we can only represent the color range from $<0, 0, 0>$ to <1,1,1>. This is actually a limitation of the fixed function graphics pipeline, as it only supports this range, clamping all color values outside of it. This is an issue for the emissive geometry, which has a greater than $<1, 1, 1>$ direct illumination value because it is emitting light. Our Monte Carlo implementation evaluates the direct illumination on demand without any clamping of values, and therefore receives the correct illumination values for the emissive geometry. However, the bright halo effect is not lost entirely in our GPU PBCB algorithm, so we find this acceptable.

5.3.3 Memory

Our memory usage was gathered by using the memusg script [29]. It polls the size reported by the unix utility 'ps' every 100 milliseconds and records the

largest instantaneous memory usage.

When comparing our Monte Carlo and GPU PBCB algorithms in terms of memory, it is important to note that they share most of their code. The only difference is the indirect illumination computation. In this way, Monte Carlo is our baseline, and our GPU PBCB algorithm adds additional memory requirements. These are summarized in Table 5.4.

Scene	**MC Mem.**	**GPU PBCB Mem.**	**Diff.**
1 object	17.81 MB	26.5 MB	+48.79%
2 objects	17.81 MB	26.6 MB	+49.35%
4 objects	17.81 MB	27.2 MB	+52.72%
8 objects	17.82 MB	28.5 MB	+59.93%
Cornell Box (18 Objects)	17.83 MB	31.8 MB	+78.35%

Table 5.4: The memory usage for our Monte Carlo and GPU PBCB algorithms for varying numbers of geometric objects. Included is the percent increase in memory usage.

The memory usage of our algorithm shows that we use a baseline of 48.79% additional memory for one object. This baseline increases linearly with the number of objects in the scene (see Figure 5.3). This is due to the fact that each object adds a fixed number of additional surfels.

The baseline increase is due to the fact that our algorithm requires the OpenGL and helper libraries (e.g. GLUT), as well as the user-mode graphics driver. These are loaded in to our process' address space and therefore increase its memory footprint.

It is also important to note that our Monte Carlo implementation increases its memory usage over time as well. However, it increases at a much slower rate than our GPU PBCB algorithm. This is because the Monte Carlo implementation only needs to store the base geometric object, but our GPU PBCB implementation stores that object as well as its associated surfels.

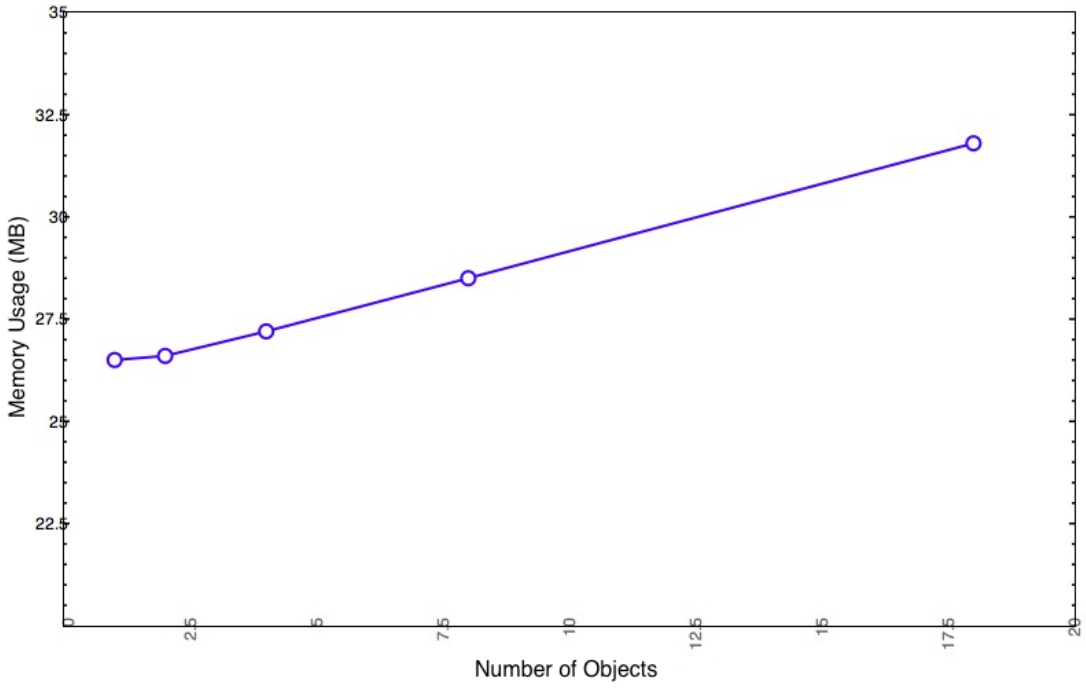


Figure 5.3: Graph of our GPU PBCB memory usage for varying number of geometric objects in a scene.

5.3.4 Scalability

Scalability is the ability of our algorithm to scale to varying geometric complexity and image size. Because our algorithm supports only three types of geometry (i.e. triangle, sphere, and box), geometric complexity is a result of the number of surfels rasterized. To study this, we compare the render times for our Cornell Box with varying numbers of surfels generated. The results are collected in Table 5.5.

Table 5.5 demonstrates that our render times are not affected by the number of surfels rasterized. This is due to the massive parallelism provided to us through the GPU. It can rasterize millions of triangles simultaneously, in parallel. For example, the ATI Radeon HD 5870 in our traditional architecture system can

Number of Surfels	Render Time	Mem. Usage	Surf. Gen. Time
9000 (500/obj.)	199 sec	31.8 MB	16 sec
18000 (1000/obj.)	198 sec	37.56 MB	115 sec
27000 (1500/obj.)	184 sec	43.03 MB	380 sec
36000 (2000/obj.)	189 sec	48.99 MB	890 sec
45000 (2500/obj.)	192 sec	54.44 MB	1746 sec
54000 (3000/obj.)	202 sec	59.90 MB	2986 sec

Table 5.5: Comparison of GPU PBCB render times for varying numbers of surfels rasterized.

process 850 million polygons per second [3]. However, our memory usage and surfel generation time are not constant.

Our memory usage scales linearly with the number of surfels required. This is expected and natural given that we are simply generating and storing additional surfels. This can be seen in Figure 5.4.

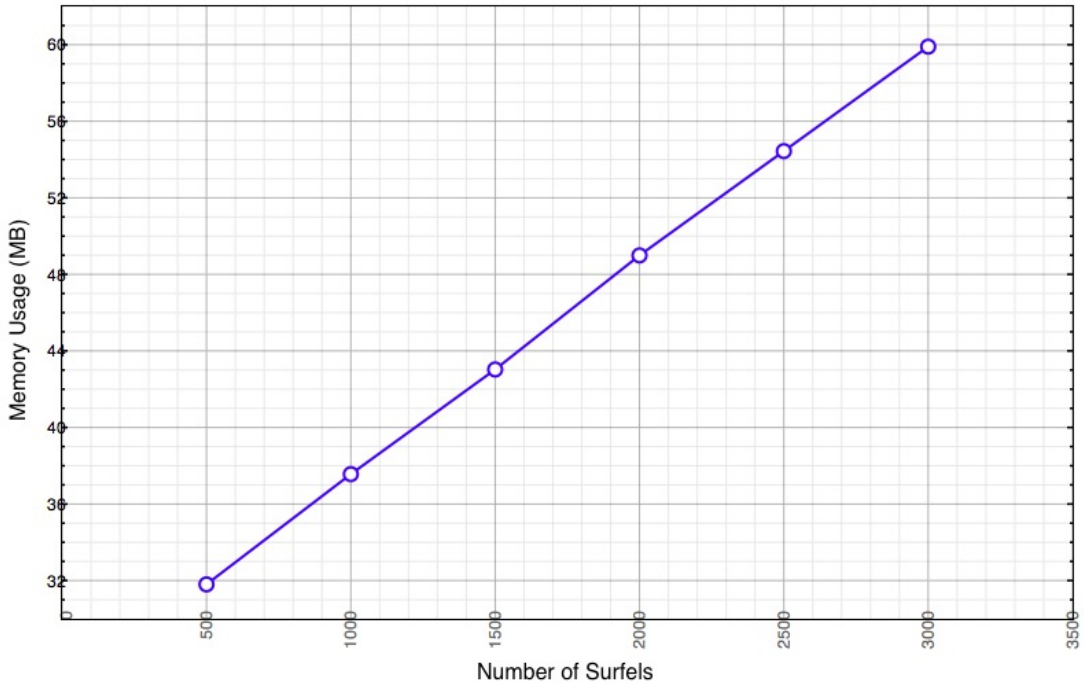


Figure 5.4: Graph of our GPU PBCB memory usage for varying number of surfels generated per object in our Cornell Box scene.

Our surfel generation time is more interesting. Here we have an exponential growth curve for our render times based on how many surfels we generate per object. This is visualized in Figure 5.5. The reason for this is that our surfel generation algorithm uses an $O(n^2)$ algorithm in order to determine the two closest points (see Section 4.2).

Image Size	Render Time	Mem. Usage
100x100	8 sec	19.02 MB
200x200	32 sec	20.60 MB
400x400	129 sec	27.02 MB
800x800	506 sec	52.67 MB

Table 5.6: Comparison of GPU PBCB render times for varying image dimensions.

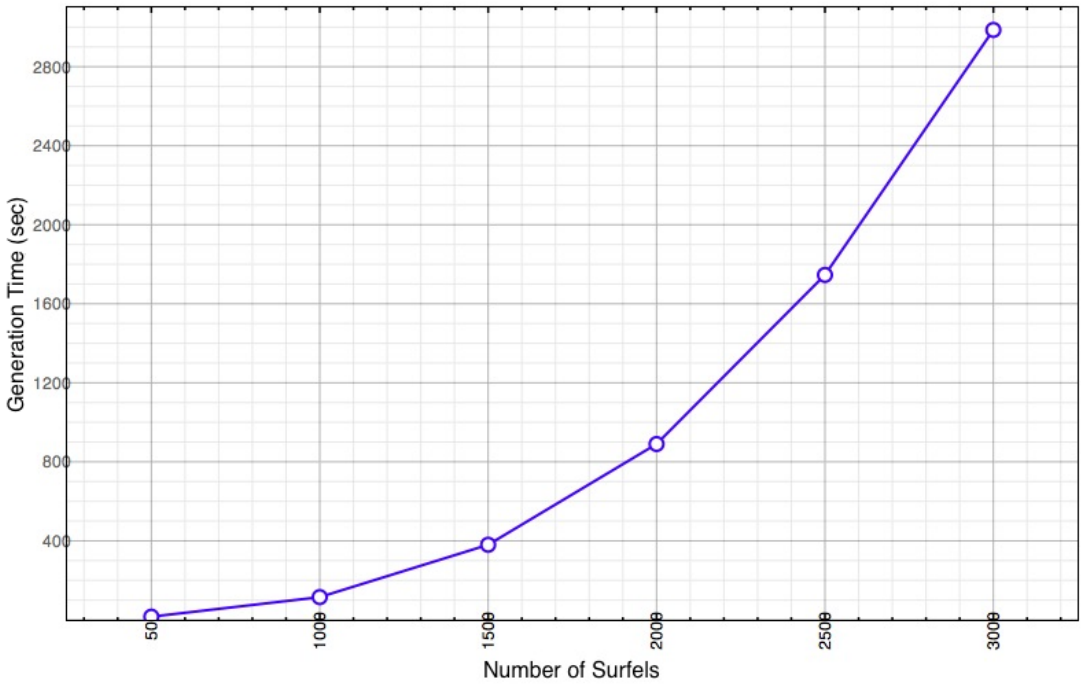


Figure 5.5: Graph of our surfel generation times for varying number of surfels generated per object in our Cornell Box scene.

The other form of scalability we are interested in is how our algorithm scales to rendered image dimensions. The goal is linear scaling in each dimension,

meaning that doubling the width and height should quadruple the render time. The results are collected in Table 5.6, and clearly show that we do indeed scale linearly in both run time and memory usage.

5.4 Additional Scenes

In this section we present two additional scenes rendered using our GPU PBCB algorithm.

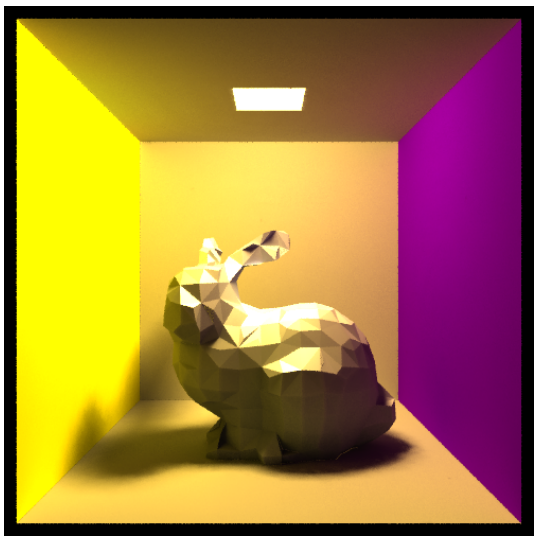


Figure 5.6: 500 vertex Stanford bunny [34] in our Cornell Box.

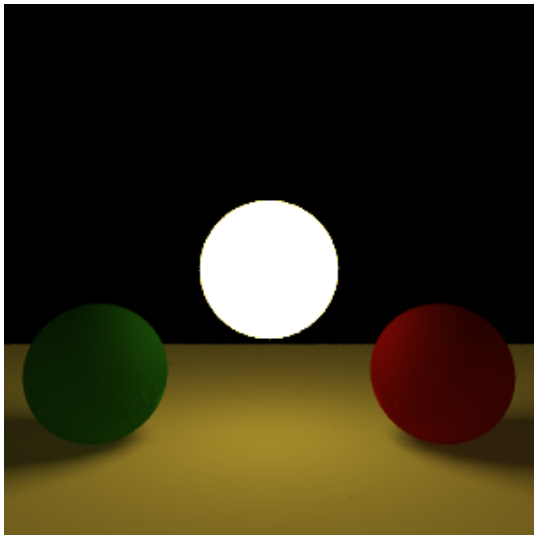


Figure 5.7: A scene with 3 spheres: one green, one red, and one emissive.

5.5 Conclusions

In this thesis we have presented a GPU-accelerated implementation of the Point-Based Approximate Color Bleeding algorithm [8]. We have demonstrated a 41.65x speedup compared to 256 sample Monte Carlo ray-tracing, as well as a 3.12x speedup over PBCB using a software-based rasterizer.

We have also gathered data that compares traditional architecture systems with heterogeneous architecture systems that place the GPU and CPU on the same die and share memory. We found that for our usage scenario, the heterogeneous architecture achieves better performance despite comprising weaker specifications. However, this issue is not fully explored as our algorithm naively implements the cube-face rasterization. We believe it is possible to further explore rasterization batching and CPU-GPU parallelization in order to achieve even further performance gains, in which case the traditional may overtake the heterogeneous architecture.

Lastly, we contribute a novel surfel generation algorithm that supports arbitrarily transformed geometry. This is important because proper coverage is a necessity to capture the proper shading in order to calculate indirect illumination as accurately as possible. And although our generation algorithm shows exponential growth as the number of surfels per object increases, we are pleased with the practical results in coverage we have obtained on transformed geometry.

The core of ray-tracing has always been about producing beautifully rendered, photorealistic images. Point-Based Color Bleeding was developed to speedup the rendering process, while not sacrificing visual quality. Our goal has been to further push the boundaries of performance by accelerating PBCB with GPU hardware. Overall, we believe the work we have done demonstrates that GPU-

acceleration of the Point-Based Color Bleeding algorithm is a viable means to pursue even faster render times. However, because we do not have an exact implementation of Christensen's PBCB algorithm [8], we believe that further work is required to truly access the comparison between his and our own.

5.6 Future Work

5.6.1 Persistent Surfel Storage

Currently, our algorithm stores the surfels in RAM, not a persistent file on the hard drive. This requires our renderer to re-generate the surfels for each render. One of the main benefits of PBCB in production is that it is possible for a scene to have surfels generated, persistently stored in a file, and loaded at render time to be reused. This amortizes the cost of the surfel generation process across all renders for the same scene.

In our renderer, the most effective way to implement persistent surfel storage would be to write the VBO array memory to a file as binary data, as opposed to storing it as text mesh-file format. In this way, a simple memory map operation would map the data directly into the VBO structure without any text parsing and processing whatsoever.

For our Cornell box scene, the surfel generation takes 16 seconds per render (see Table 5.5), which could be avoided with persistent surfel storage. And although the surfel generation is not included in our render run time results, the feature would increase our overall render throughput.

5.6.2 Dynamic Surfel Surface Area Computation

Having a dynamic density for surfels would help to homogenize the surfel size throughout the scene. Our algorithm naively creates a user-provided number of surfels per geometric primitive: for triangles and spheres, exactly the specified number of surfels are generated, and for boxes, each face generates the specified number (i.e. specified value multiplied by six per box). This results in variable surfel sizes across the same geometric primitive at different scales. For example, the smaller triangles that compose the ceiling of our Cornell box, and the larger triangles that compose the walls, both generate 500 surfels. Because the primitives have different surface areas, the surfel density is variable, resulting in small surfels for the small triangles, and larger surfels for the large triangles.

Our proposed solution to this problem is to have a user-provided surfel density in the form of minimum distance. The current algorithm would be modified to continue decimating the random points until the specified minimum distance is met. Another way in which to achieve the same result, in perhaps a more intuitive manner, would be to have a user-provided surfel surface area. We would then solve for the minimum distance that would provide such a surface area, and use that, per the previously described algorithm modification.

Furthermore, it would behoove us to analyze the results of varying the surfel surface area on final render quality and speed, as well as surfel memory requirements. The tradeoffs to consider are that larger surface areas would result in fewer surfels, thus requiring less memory and reducing render times by requiring fewer triangle rasterizations. But smaller surfels would increase the sampling density of the scene's radiance information, resulting in more accurate indirect illumination values, and would decrease the surfel generation run time by requiring

fewer passes for the decimation step within our algorithm.

5.6.3 Rasterization Batching

On more traditional CPU-GPU architectures, where the CPU and GPU must communicate via the PCI bus, care must be taken to avoid synchronous communication over the slow PCI bus [28]. Due to the latency of CPU-to-GPU communication, it is in our best interest to batch as much of this communication as possible. In our algorithm, we raster each cube-face serially. This means that each 8x8 cube-face texture is rasterized, then copied from GPU to CPU memory, and processed. Therefore, each cube requires 5 data transfers (recall that the bottom of the cube is ignored). For heterogeneous system architectures, like our test system, this transfer is instant as CPU and GPU share memory. However, this data transfer is the single most time-consuming task in our algorithm on more traditional CPU-GPU architectures. The communication from GPU to CPU can be reduced however.

We can accomplish fewer transfers by packing multiple cube-face textures into one single texture per cube. In this way, we amortize the cost of one GPU-to-CPU transfer over 5 textures. The algorithm for this technique would require one additional render pass, in which the 5 cube-face textures are texture-mapped to 5 screen-aligned quadrilaterals. This creates a texture atlas per cube that requires only one GPU-to-CPU transfer and can be indexed appropriately to extract the data for each cube-face.

This idea could be taken further: to batch the entire set of cubes, or some subset, as available memory and texture size dictates. Potentially, one thread could perform the standard direct illumination calculations via ray-tracing, while

another thread rasterizes the surfel cloud onto cube-face textures, but stores them into one texture atlas for the entire scene. This is a very appealing idea to us because it would achieve great parallelism, as the CPU and GPU would be simultaneously leveraged to perform rendering tasks.

5.6.4 Parallelization

We believe the benefits of parallelization are apparent and will not discuss them further here. Suffice it to say that our algorithm can easily benefit from a model that divides primary pixels between multiple threads of control. In fact, this is precisely how our Monte Carlo ray-tracer accomplishes its parallelization.

However, our code relies on the GLUT library [23] to create and manipulate the OpenGL context. GLUT does not currently support multithreaded applications. For this reason, although our implementation supports multithreaded rendering for Monte Carlo ray-tracing, it does not for our GPU Point-Based Color Bleeding algorithm. We have acquired our results in both cases with one thread only.

Any solution to this problem will involve porting the OpenGL code that leverages GLUT to another multithreading-friendly library. The Simple and Fast Multimedia Library is one such library that is freely available [27]. Preferably, the library would support one single context that is shared between all threads of control and serializes their access, but it could also be accomplished while using GLUT by wrapping the functionality and diligent use of IPC and thread synchronization. By sharing the context, the surfel data within the VBO memory does not need to be duplicated.

Another type of parallelization that many applications utilizing the GPU

leverage is CPU-GPU parallelization. This is where concurrent work is performed on both hardware devices. That is to say: the CPU does not block on a GPU draw call. This can be accomplished via the process described in Section 5.6.3. Here, we can completely divide the rendering process into threads performing direct illumination via standard ray-tracing and threads performing indirect illumination via our GPU PBCB algorithm. Because our two types of illumination have no interdependencies, two final images can be rendered, one with the direct illumination values, and the other with the indirect illumination values, and combined after both rendering passes are complete.

5.6.5 Spatial Data Structures

Spatial data structures are meant to answer spatial relationship queries. That is to say: they can inform us about what objects are nearest to any point in space. They can be incorporated into our algorithm in two distinct ways. First, our surfel generation algorithm requires us to solve for the two closest points out of the set we generate on the surface of our source geometry. We believe an octree would address our needs [2]. Secondly, we could use a spatial data structure to store our surfels.

The uniformity of surfels makes them a natural fit for spatial data structures. Because surfels are all relatively small compared to the source geometry, and uniform in size, it allows us to more easily subdivide then into groups or bins. This is important to spatial data structures like the bounding volume hierarchy, or BVH [24].

In our test scenes, we never produced enough surfels to overwhelm the GPU's rasterization pipeline. However, there could be scenarios where there are too

many surfels for the GPU to handle in one pass. In these cases, our algorithm would suffer slowdown from having to rasterize multiple times per cube-face texture.

To solve this, we could use a spatial data structure to contain our surfels and provide us a subset of surfels relevant to the current render. Using the hierarchical view frustum culling algorithm [2] we could cull any surfels that are not the in the view frustum of a given camera before rasterization.

5.6.6 Surfel Level of Detail

One of the features in Christensen's PBCB algorithm presented in [8] is the three levels of detail used for the cube-face rasterization step. We did not implement this feature, and one of the benefits is that the number of surfels rasterized is reduced via the octree and coalesced surfels. However, there are also the additional rays to trace as well. It is unclear how this strategy will affect the run times, although we assume for the better. Because of this we believe that a true replica of Christensen's PBCB algorithm needs to be implemented, to further access the viability of GPU-acceleration in PBCB.

Bibliography

[1] Imagemagick compare. http://www.imagemagick.org/script/compare.php.

[2] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[3] AMD. Ati radeon hd 5870 specification document. AMD.com, 2008.

[4] AMD. Amd fusion apu era begins. AMD.com, 2011.

[5] Apple. 2011 13" macbook air specification document. Apple.com, 2011.

[6] P. Bourke. Distributing points on a sphere. http://local.wasp.uwa.edu.au/ pbourke/geometry/spherepoints/.

[7] P. Bourke. Sphere generation. http://local.wasp.uwa.edu.au/ pbourke/miscellaneous/sphere_cylinder/.

[8] P. H. Christensen. Point-based approximate color bleeding. Technical Memo #08-01, Pixar Animation Studios, 2008.

[9] P. H. Christensen. Point-based approximate color bleeding slide presentation. Pixar.com, 2008.

[10] D. Cline, S. Jeschke, K. White, A. Razdan, and P. Wonka. Dart throwing on surfaces. *Computer Graphics Forum*, 28(4):1217–1226, 2009.

[11] M. F. Cohen and D. P. Greenberg. The hemi-cube: a radiosity solution for complex environments. *SIGGRAPH Comput. Graph.*, 19(3):31–40, July 1985.

[12] F. C. Crow. The aliasing problem in computer-generated shaded images. *Commun. ACM*, 20(11):799–805, Nov. 1977.

[13] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM.

[14] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA, 1984. ACM.

[15] Intel. Intel i7-920 specification document. Intel.com, 2008.

[16] Intel. Intel i7-2677m specification document. Intel.com, 2011.

[17] Intel. 3rd generation intel core processor family quad core launch product information. Intel.com, 2012.

[18] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[19] J. Jimenez, D. Gutierrez, J. Yang, A. Reshetov, P. Demoreuille, T. Berghoff, C. Perthuis, H. Yu, M. McGuire, T. Lottes, H. Malan, E. Persson, D. Andreev, and T. Sousa. Filtering approaches for real-time anti-aliasing. In

ACM SIGGRAPH 2011 Courses, SIGGRAPH '11, pages 6:1–6:329, New York, NY, USA, 2011. ACM.

[20] D. Lischinski, F. Tampieri, and D. P. Greenberg. Combining hierarchical radiosity and discontinuity meshing. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 199–208, New York, NY, USA, 1993. ACM.

[21] N. Nikishin. Writing a pathtracer. http://www-personal.umich.edu/ nikitan/articles/pathtracing/.

[22] P. of Vision Raytracer Pty. Ltd. Pov-ray. povray.org, 2008.

[23] OpenGL.org. The opengl utility tookit, 2000.

[24] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann. Elsevier Science & Technology, 2010.

[25] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[26] L. Piegl. On nurbs: a survey. *Computer Graphics and Applications, IEEE*, 11(1):55 –71, jan. 1991.

[27] SFML. The simple and fast multimedia library. sfml-dev.org, 2012.

[28] L. Sherwin. Pci-sig delivers pci express 2.0 specification. PCIsig.com, 2007.

[29] J. Shin. Memusg. https://gist.github.com/526585.

[30] P. Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[31] P. Shirley. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[32] P. Shirley, C. Wang, and K. Zimmerman. Monte carlo techniques for direct lighting calculations. *ACM Trans. Graph.*, 15(1):1–36, Jan. 1996.

[33] F. X. Sillion, J. R. Arvo, S. H. Westin, and D. P. Greenberg. A global illumination solution for general reflectance distributions. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '91, pages 187–196, New York, NY, USA, 1991. ACM.

[34] G. Turk and M. Levoy. Stanford bunny. Stanford.edu, 1994.

[35] J. M. Van Verth and L. M. Bishop. *Essential Mathematics for Games and Interactive Applications*. Morgan Kaufmann, Burlington, MA, USA, 2008.

[36] E. W. Weisstein. Sphere point picking. WolframAlpha.com.

[37] WhiteTimberwolf. Octree. Wikipedia.com, 2010.