HIGH QUALITY RENDERING FOR POINT SAMPLED GEOMETRY

FENG LIU

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 2006

Your file Votre référence
ISBN: 978-0-494-20777-2
Our file Notre référence
ISBN: 978-0-494-20777-2

Abstract

High Quality Rendering for Point Sampled Geometry

Feng Liu

In recent years, point sampled geometry models have received growing attention in computer graphics. Compared with polygon based models, the quality of rendered images for point sampled geometry is much lower, so far. This is primarily due to the discrete form of representing a 3D surface just by sample points. Clearly more effort is needed in rendering techniques for point sampled models. In this thesis, we describe the results of our efforts towards producing high quality images for point based geometry. We adapt a number of advanced mesh rendering techniques for discrete surfaces represented as point clouds. These techniques include self shadowing effects using ambient occlusion, diffuse lighting estimation using spherical harmonic representations of irradiance environment maps and specular lighting estimation by casting a reflected ray at each point into a pre-blurred version of the environment map. Ambient occlusion computations are improved by using octree hierarchy based on feature analysis of the point cloud data. Other lighting calculations are performed directly on the Graphics Processing Unit (GPU) using vertex and fragment shaders. We improve the overall rendering quality and speed by combining pre-computed ambient occlusion results with environment lighting and procedural textures directly on the GPU. The adapted techniques have been implemented as a plug-in for the Pointshop3D public domain software system.

Acknowledgments

First of all, I would like to thank my supervisors, Dr.Mudur and Dr.Fevens, for steering me into this exciting topic, and giving advice and directions to achieve these great rendering effects. Special thanks to Sushil Bhakar for helping me with each step of my research work.

I'd also like to thank my parents for their support and encouragement in many things over the years, and forgiving me staying at home for only one and a half months during the last 6 years.

We thank Paul Debevec for providing light probes. (Available from http://www.debevec.org). We gratefully acknowledge for 3D models Stanford 3D Scanning Repository, Mannequin model by Hugues Hoppe and Gnome and Igea model from Pointshop3D site. We also thank the Pointshop3D community for providing such an excellent tool for experimentation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Real time high quality rendering

Rendering is the process of producing the pixel values in an image of an object or scene from a higher-level description of its components, usually geometric. Typically one renders objects composed of polygons or point clouds. One of the important goals of rendering is to create high quality images which are indistinguishable from photographs, a goal referred to as photorealism. Another important goal is interactivity for visualization, simulation and other real time applications, for which the typical frame rate should be 20 frames per second or higher. These two goals have historically been at odds with each other. Photorealism employing methods like ray tracing and radiosity has resulted in beautiful pictures, but at the cost of slow algorithms taking hours to days. Advances in graphics hardware have enabled increasing interactivity, from the earliest SGI machines to today's powerful desktop GPU (Graphics Processing Unit). However, the traditional focus in hardware improvements has been on rendering more polygons rather than production of high quality images. In the last few years, as a result of improved programmable graphics hardware along with new efficient image synthesis algorithms, there has been an amazing convergence of these ideas, enabling us to conceive of real-time photorealistic rendering as an achievable goal in the near future.

1.2 Point sampled geometry

Our particular interest is in real time photorealistic rendering of models represented by point sampled geometry. For point sampled geometry models, the shape of objects is represented discretely purely in the form of sampled surface points. Typically these sampled surface points can be directly obtained using 3D scanners. Point sampled geometry has received growing attention in computer graphics in recent years. There are two main reasons for this new interest: on the one hand, there is a drastic increase in the polygonal complexity of computer graphics models as the models become larger and more detailed. The overhead of managing, processing and manipulating very large polygonal meshes has led many researchers to question the future utility of polygons as the fundamental graphics primitive. On the other hand, modern 3D digital photography and 3D scanning systems facilitate the ready acquisition of point samples on 3D surfaces of complex, real-world objects. These techniques too generate huge volumes of 3D point data and have thus created the need for advanced technique for point sampled geometry processing and rendering.

1.3 Issues to be addressed in high quality rendering of point sampled geometry

The main issues different from mesh based models in rendering of point based models are due to the following: Firstly, point sampled geometry models have only discrete points. Absence of any kind of explicit topological continuity information implies the need for inferring this continuity through neighboring points. The standard solution is to associate with each point a surface region of influence. Hence, we represent each point by a surface element, a disk shaped element, called surfel. For rendering such models, we also need to give each point other properties. Typically, each surfel has the following information associated with it: position, normal, color, radius, and when needed, texture coordinates. Secondly, due to the absence of other types of information, say, curvature etc., the number of point samples needed to represent complex geometry is very large. Hence high quality rendering in real time requires specialized techniques to correctly address the discontinuous nature of representation and efficiently handle the large volume of point data.

1.4 Objective of our research

So far, in point based graphics, researchers have only used simple illumination models and techniques (point light source at infinity or directional light source, and usually the most common, Phong lighting calculations). People perceive materials more easily under natural illumination (represented by environment light map than the above simple illumination as clearly demonstrated in the example which shows the quality of the two rendered images in Figure 1 and Figure 2. While these images are renderings based on based on polygonal models of the object, the primary objective in our work is to adapt this and other real-time techniques to efficiently rendering point sampled geometry. Specifically, we would like to address natural illumination, global illumination like effects and texturing of point sampled geometry models. A secondary objective is to implement these adapted techniques and incorporate them into the Pointshop3D public domain point sampled geometry software.

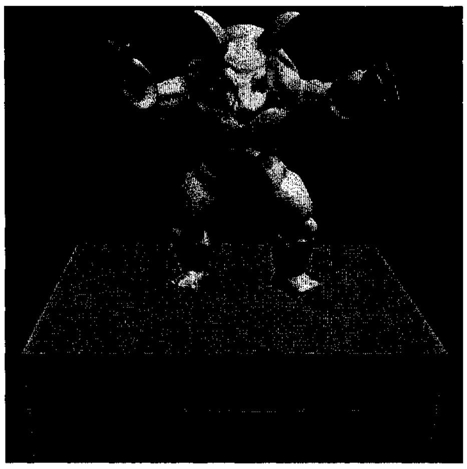

Figure 1: Directional light source (from [1])

1.5 Our Rendering Techniques for Point Sampled Geometry

Currently there have been impressive developments in the photorealistic rendering of polygon based meshes. Therefore an important direction in rendering point sampled geometry would be to demonstrate that popular techniques for rendering of high quality images of mesh based geometry can also be applied to point sampled geometric representations with similar performance. In particular, we modify mesh rendering techniques of ambient occlusion, environment lighting and procedural

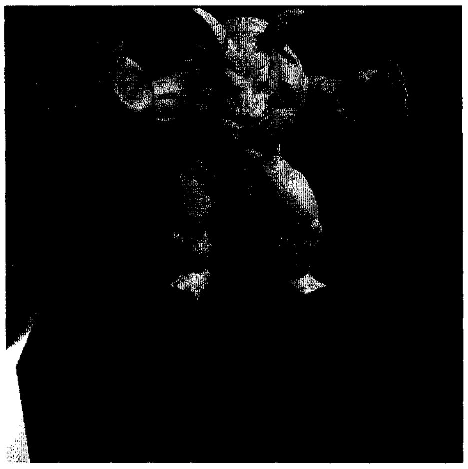

Figure 2: Natural Illumination (from [1])

textures for high quality surface rendering of point cloud data. A brief overview of these techniques follows.

1.5.1 Environment Lighting

With environment lighting, the object is illuminated by distant surrounding lights, represented by an environment map, which encodes the amount of light coming from each directions around the object. We calculate the light reflected off the surface of the object maintaining both diffuse(or matte) and specular(or shining) reflectance behaviors. For estimating the diffuse component we use spherical harmonic representations of irradiance environment maps. Specular behavior is captured by reflecting a ray into a blurred version of the environment map. For increased efficiency, these calculations can be performed directly on the Graphics Processing Unit (GPU) using vertex and fragment shaders. A shader is a program that is run by the GPU used in 3D computer graphics to determine the final surface properties of an object or image. Vertex shader is applied for each vertex and run on a programmable vertex processor (part of the GPU), while fragment shader is applied for each pixel, running on a pixel processor (also part of the GPU), which usually features much more processing power than its vertex-oriented counterpart.

1.5.2 Ambient Occlusion

Ambient occlusion is a shading method used in 3D computer graphics which helps add realism to local reflection models by taking into account attenuation of light due to occlusion. Unlike local methods like Phong shading, ambient occlusion is a global method, meaning the illumination at each point is a function of other geometry in the scene.

In this method, a self shadowing factor is calculated for each vertex in a preprocessing stage using ambient occlusion technique. After the calculation, we can get an ambient occlusion value for each point on the surface (`ShadowValue`) that has to be rendered. We multiply each point's original color value by (1 − `ShadowValue`), and use that as the new color of that point. This preprocessed model is then rendered to achieve more photorealistic results.

1.5.3 Procedural Texture

A procedural texture is a computer generated 2D/3D image created programmatically with the intention to create a realistic representation of natural elements such as wood, marble, granite, metal, stone, and others. The well established technique for procedural textures in 2D and 3D uses the Perlin Noise formulation. By changing the parameter values in this formulation, one can produce different procedural texture patterns. These can be programmed within the GPU.

1.5.4 Contributions of this Research

There are three main contributions from this work: Firstly, a number of techniques developed for high quality rendering of mesh based models have been efficiently adapted for point based models as well. These include natural lighting through the use of environment maps and procedural textures using Perlin Noise formulation, both in combination with global illumination like effects using ambient occlusion. Secondly, we have evolved efficient methods for ambient occlusion using the hierarchy of an Octree structure and feature based analysis of point sets and incorporated this within the surfel definition. Further, as many of these techniques as feasible have been implemented within the GPU. Lastly, we have incorporated our point based renderer as a plug-in for the Pointshop3D public domain software system.

1.6 Organization

The organization of the remainder of this thesis is as follows.

Previous and related work is discussed in Chapter 2.1. Adaptation of the three techniques for point sampled geometry are described individually in subsequent chapters. Environment Lighting is described in Chapter 3; Ambient Occlusion is detailed in Chapter 4; Procedural Texture is provided in Chapter 5. Chapter 6 contains our conclusions and potential for future work. The Appendix provides brief details of the PointShop3D public domain software and the process adopted to incorporate our renderer as a plug-in component.

Chapter 2

Related Work

2.1 Recap

Point sampled geometry consists of data only for discrete points, and there is no connectivity information between them to define the surface topology. The standard solution is to associate with each point a surface region of influence. That means we give each point some properties, such as position, normal, color, radius, and sometimes texture coordinate. Point sampled surface representations have been proposed as an alternative to the popular triangle mesh representations. Advances in 3D scanner technology have considerably simplified their acquisition. Also with no requirements of connectivity information, their representation is much simpler.

In this chapter, we give an overview of earlier work related to our research on the rendering of point sampled geometry models.

2.2 Previous High Quality Rendering Techniques for Point Sampled Geometry

High quality rendering of point sampled geometry has been the concern of a large number of researchers. In one of the early papers [2], Zwicker et al. describe a renderer using elliptic splats for points represented as surfels. As already mentioned earlier, a surfel is a surface element represented by a point on the surface, a normal to that surface, an extent for the point given as a radius and other material properties such as color, reflectance, texture coordinates, etc. Zwicker et al.'s work also incorporates texture filtering. The filter formulation is screen space based and needs software implementation. Screen space can be thought of as a grid of rows and columns, with each unit in the grid representing the smallest addressable unit in a virtual buffer. This quantity is called a pixel, and each pixel is identified by its unique (x,y) integer coordinates. In a subsequent paper [3], they provided an object space formulation of the EWA (elliptic weighted average) filter and described a hardware implementation using textured polygons. Object space is also called model space in some 3D references; it is the coordinate system in which a specific 3D object is defined. Usually, but not always, each object will have its own distinct object space with the origin at the object's center. Botsch et al. have described a number of works in which they have tried to improve the performance and rendering capability of the EWA renderer. Their improvements include the use of point sprites instead of polygons [4]; a point sprite is a screen-aligned textured quad placed by rendering a single point. This resulted in reduced number of vertices being sent to hardware. Other improvements include correction for perspective [5], per pixel shading [6] and deferred splatting [7].

2.3 Ray Tracing for Point Cloud

Ray tracing is the most used method for rendering objects/scenes with global illumination effects, particularly with specular reflection effects. The standard ray tracing method works by tracing in reverse, a path that could have been taken by a ray of light which would intersect the imaginary camera lens. As the scene is traversed by following in reverse the paths of a very large number of such rays, visual information is built up, on the appearance of the scene as viewed from the point of view of the camera under the specified lighting conditions. The ray's reflection, refraction, or absorption are calculated when it intersects objects and media in the scene.

A number of papers have reported efforts in applying ray tracing techniques to point based models to simulate global illumination effects. The main problem arises due to the fact that both points and rays are singular geometric entities having no spatial extent. The different methods vary in the way in which this problem is solved and also correspondingly in the acceleration data structures that are employed. Schaufler and Jensen [8] proposed to intersect a ray with a point sampled surface by creating a cylinder around the ray. Adamson and Alexa [9] proposed a number of ray-surface algorithms based on their moving least squares (MLS) implicit surface definition. They initially create a sphere hierarchy and intersect the rays with this hierarchy to find an approximate intersection. Finally they carry out the ray intersection with the MLS surface inside the sphere. In [10] they further improved the efficiency of intersection calculations. Wald and Seidel [11] use a combination of different techniques including an SIMD (Single Instruction, Multiple Data) accelerated intersection code, together with a highly optimized specially built kd-tree data structure. They report 7 - 30 frames per second for a 512×512 image of sufficiently large point based model. In [12] Adams et al. describe techniques for ray tracing of deformable point sampled surfaces. Once again the emphasis is on clever update of the hierarchical data structure for accommodating the deformation in each frame.

2.4 Radiosity for Point Sampled Models

The radiosity method is another powerful and popular graphics method for achieving global illumination effects, particularly for environments with diffuse surfaces. It simulates many reflections of light around a scene, generally resulting in softer, more natural shadows. Compared to ray tracing, the strongest point of radiosity is that the light distribution computation result is view-independent.

Dobashi et al. [13] describe a very straight forward extension of the standard radiosity technique. For this they consider each surfel as a finite element and calculate inter-reflections among the surfels.

They too assume that a point model is represented by a set of surfels. Each surfel has its position and properties of the original surface around the point such as a reflectance and a normal vector. Firstly they compute the area for each surfel. Because the surfels overlap, they use a method to take into account the overlap for computing the area of each surfel. The computed area is called effective area. Then the inter-reflection is computed using a method which uses points instead of patches. To make the intensity distribution accurate, they add new points adaptively before calculating the inter-reflection.

2.5 PointShop3D

To test our adaptation of various techniques for photorealistic rendering of point sampled geometry, we have incorporated our implementation in Pointshop3D [21]. Pointshop3D is an open source software system for interactive editing of point-based surfaces, which was developed at the Computer Graphics Lab at ETH Zurich.

Pointshop3D supports a great variety of different interactive techniques to alter shape and appearance of 3D point-based models, including cleaning, texturing, sculpting, carving, filtering and re-sampling.

In recent years, Pointshop3D was enhanced to support Boolean operations and free form deformation for shape editing. In addition, tools for cleaning and reconstruction of scanned data were introduced.

Pointshop3D supports only one kind of input data format known as the SFL file format. The SFL file is a binary format featuring an extensible set of surfel attributes, data compression, upward and downward compatibility, and transparent conversion of surfel attributes, coordinate systems, and color space.

Since Pointshop3D is an open source software system, it has numerous plugins. One is called EWAGLRenderer. It is a fast and high quality hardware renderer which can handle elliptical surfels. In our work, we have substituted this hardware renderer with our new renderer incorporating the different adapted rendering techniques.

2.6 BRDF

The BRDF is the "Bidirectional Reflectance Distribution Function". It gives the reflectance of a target as a function of illumination geometry and viewing geometry. The BRDF depends on wavelength and is determined by the structural and optical properties of the surface, such as shadow-casting, multiple scattering, mutual shadowing, transmission, reflection, absorption and emission by surface elements, facet orientation distribution and facet density.

Chris Wynn [16] says, that a BRDF can be thought of as a weighting function that describes how light is reflected when it makes contact with a surface. In general, it is a function of wavelength as well as two directions - an incoming light direction w_i, and an outgoing viewer direction w_o. Since these two directions are unit vectors, it's convenient to write these vectors as $w_i = (\theta_i, \phi_i)$ and $w_o = (\theta_o, \phi_o)$. In this notation (known as spherical coordinates), the angle represents the angular rotation of a vector from being aligned with the up vector and the angle ϕ represents the angular rotation of a vector in the plane perpendicular to the up vector. Figure 3 shows intuitively how these two angles together define a direction on a hemisphere with radius = 1.

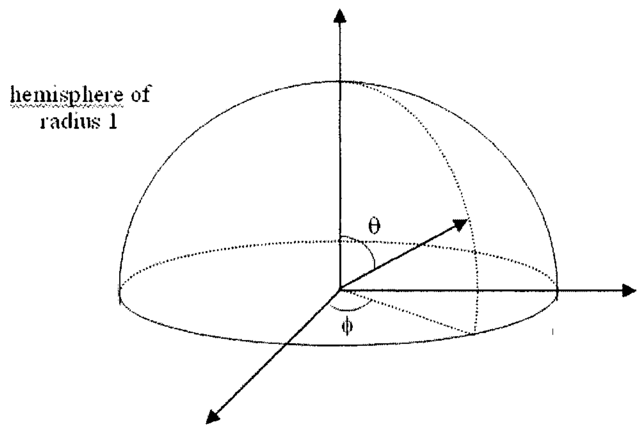

Figure 3: Relationship between a direction vector and associated angles θ and ϕ, from [16]

2.7 Efficient Representation for Irradiance Environment Maps

2.7.1 Precalculate part

Lighting in most real scenes is complex, coming from a variety of lighting sources including area lights and large continuous lighting distributions like skylights. But current graphics hardware only supports point or directional light sources.

In their paper, Ramamoorthi et al. use the term irradiance environment map for a diffuse reflection map indexed by the surface normal, since each pixel simply stores the irradiance for a particular orientation of the surface. Their main contribution is the rapid computation of an analytic approximation to the irradiance environment map. For rendering, they demonstrate a procedural algorithm that runs in real time using hardware implementation. Since irradiance varies slowly with orientation, it need only be computed per vertex and interpolated across triangles.

Given an environment map, they firstly calculate the 9 lighting coefficients, L_{lm} for $l \leq 2$, by integrating against the spherical harmonic basis functions. Each color channel has to be treated separately.

$$L_{lm} = \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} L(\theta, \phi) Y_{lm}(\theta, \phi) \sin\theta d\theta d\phi$$

For Y_{lm}:

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

To do the integration, all the pixels in the environment map are summed, weighted by the basis functions Y_{lm}.

2.7.2 Rendering Part

In [14], Peter-Pike and J. Sloan implemented the rendering part for Ramamoorthi's paper [1] using a better method.

The basis spherical harmonic functions are:

$$\begin{aligned}
y_0^0 &= \frac{1}{2\sqrt{\pi}} \\
(y_1^1; y_1^{-1}; y_1^0) &= \frac{\sqrt{3}}{2\sqrt{\pi}}(-x; -y; z) \\
(y_2^{-2}; y_2^1; y_2^{-1}) &= \frac{\sqrt{15}}{2\sqrt{\pi}}(xy; -xz; -yz) \\
y_2^0 &= \frac{\sqrt{5}}{4\sqrt{\pi}}(3z^2 - 1)
\end{aligned}$$

$$y_2^2 = \frac{15}{4\sqrt{\pi}}(x^2 - y^2) \tag{2}$$

To illustrate things easier, we define the following variables:

$$n_0 = \frac{1}{2\sqrt{\pi}}; n_1 = \frac{\sqrt{3}}{2\sqrt{\pi}}; n_2 = \frac{\sqrt{15}}{2\sqrt{\pi}}; n_3 = \frac{\sqrt{5}}{4\sqrt{\pi}}; n_4 = \frac{\sqrt{15}}{4\sqrt{\pi}}; \quad h_1 = \frac{2}{3}; h_2 = \frac{1}{4}; \tag{3}$$

where h_i are the convolution coefficients divided by π (irradiance is turned into exit radiance), and the n_i are the normalization coefficients of the basis functions.

Then for each vertex, the diffuse color contributed by the environment map lighting is calculated as follows:

$$\begin{aligned}
E_{00} &= L_{00}n_0 \\
E_{1-1} &= L_{1-1}[n_1h_1(-normal.y)] \\
E_{10} &= L_{10}[n_1h_1(normal.z)] \\
E_{11} &= L_{11}[n_1h_1(-normal.x)] \\
E_{2-2} &= L_{2-2}[n_2h_2(normal.x \cdot normal.y)] \\
E_{2-1} &= L_{2-1}[n_2h_2(-normal.y \cdot normal.z)] \\
E_{20} &= L_{20}[n_3h_2(3(normal.z)^2 - 1)] \\
E_{21} &= L_{21}[n_2h_2(-normal.x \cdot normal.z)] \\
E_{22} &= L_{22}[n_4h_2[(normal.x)^2 - (normal.y)^2]] \qquad (4)
\end{aligned}$$

Then for each vertex p:

$$\texttt{Diffuse}(p) = \Sigma E_{lm}(p) \tag{5}$$

Sloan further used a clever programming trick in order to use fewer GPU registers. We too have adopted this in our implementation. For more detail about the GPU implementation, please refer to Section 3.1.

2.8 Dynamic Ambient Occlusion

Global Illumination adds realism, so we have also considered incorporating some global illumination like effects into our rendering. In Bunnell's paper [15], he describes a new technique for computing diffuse light transfer and shows how it can be used to compute global illumination for animated scenes.

He treats polygon meshes as a set of surface elements that can emit, transmit, reflect light, and shadow each other. Because this technique works without calculating the visibility of one element to another, it is efficient enough for it to perform in real time. Instead of calculating the visibility between every two surface elements, the dynamic scene is rendered in two passes.

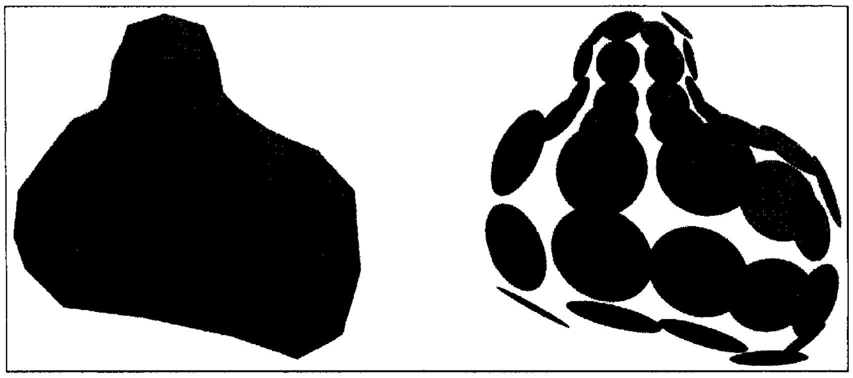

Figure 4: Change a Polygon Mesh into a Set of Surface Elements, from [15]

The first step of this technique is to change one polygon mesh into a set of surface elements (see Figure 5). A surface element is defined as an oriented disk with a position, normal, and area. Each vertex of the polygon mesh is changed to a surface element; for this the vertex's position and normal are used as the element's position and normal respectively. For the area part, this method calculates the area at a vertex as one-third of the sum area of the triangles that share the vertex (or one-forth of the sum area of the quads).

Ambient occlusion is a technique that is used to add self shadowing effects to diffuse objects lit with environment lighting; it provides soft shadows by darkening surfaces that are partially visible to the environment.

To calculate the accessibility value at each element, one just has to use 1 minus the amount by which all the other elements shadow the element.

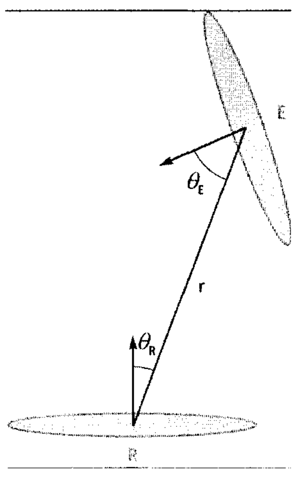

Figure 5: The Relationship between Receiver and Emitter elements, from [15]

In Figure 5, E is emitter, R is receiver, θ_E is the angle between the emitter's normal and the vector from emitter to receiver, θ_R is the corresponding angle for the receiver element, and A is the area of emitter.

R's shadow amount which is cast by E is:

$$\frac{A\cos\theta_E\cos\theta_R}{\pi r^2 + A}$$

Because the visibility between two elements is ignored, there will be a problem called double shadowing.

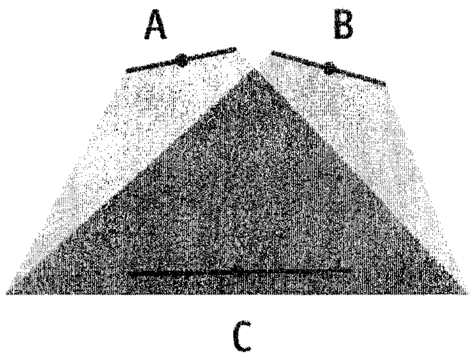

Figure 6: Double Shadowing 1, from [15]

In Figure 6, both A and B are visible to C, so C is shadowed by both A and B.

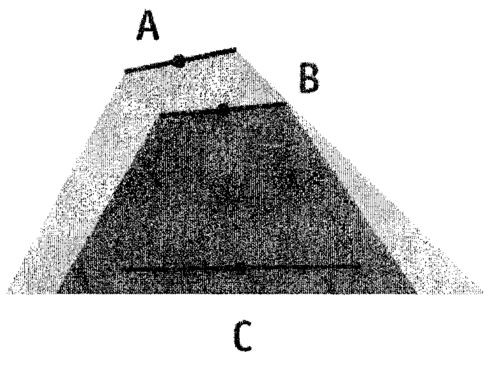

Figure 7: Double Shadowing 2, from [15]

In Figure 7, C is not visible to A. When calculating shadow for C, if we sum up occlusion values caused by A and B, C will be shadowed too much.

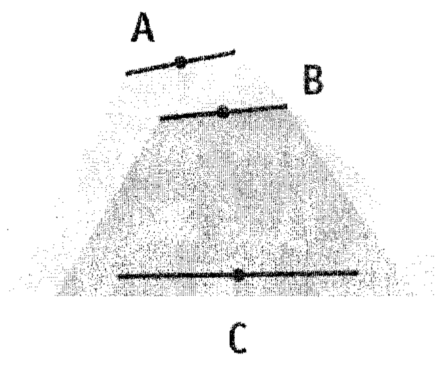

Figure 8: Double Shadowing 3, from [15]

In Figure 8, we may consider that A shadows correctly, but B shadows too much. We therefore need to devise a method to lighten B's shadow. This is done by rendering in two passes. In the second pass, we already know how much B is in shadow in the previous pass. At this point, we multiply B's form factor by its shadow in the previous pass.

The first image is rendered only by one pass. Because of double shadowing, it shadows too much, especially in the teeth area. The second image and third image are rendered by 2 and 3 passes respectively. We can see that there is no big difference between them. This means rendering in two passes is enough to eliminate double shadowing. The last image is rendered by ray-tracing. Again, there is not much difference between the second and forth image. From this example, one can see that ambient occlusion can achieve effects similar to global illumination; more important, it

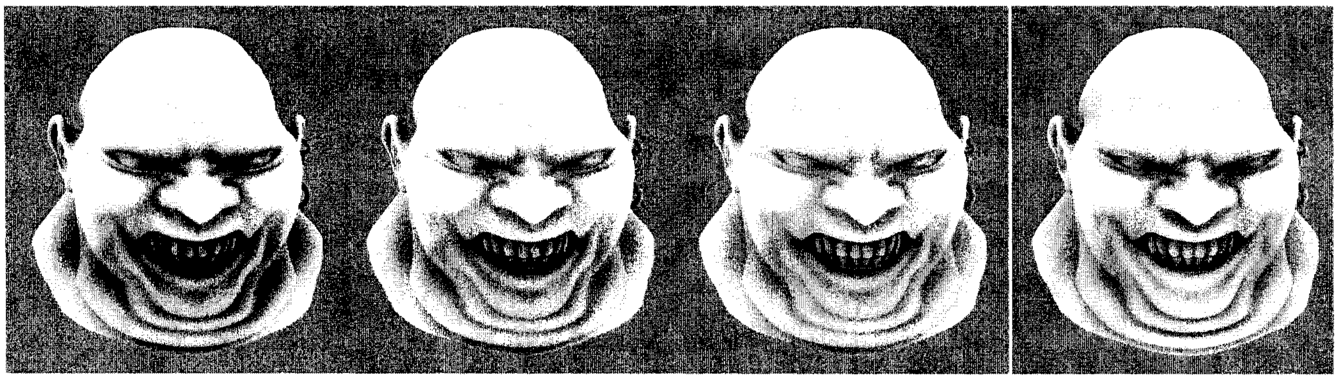

Figure 9: Compare Rendering Effects Between Different Passes and Ray-tracing, from [15]

can be implemented in real time.

2.9 Jim Blinn Model for Specular Reflection

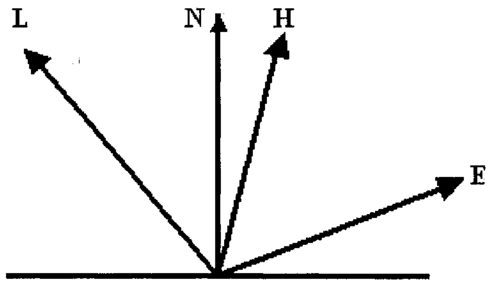

Figure 10: Reflection Model

In Figure 10, L is light direction, E is eye direction, N is normal, H is the half angle vector of L and E.

The surface is assumed to be composed of a collection of mirror like micro facets that are oriented in random directions on the surface. The specular component of the reflected light is assumed to come from reflection from the facets oriented in the direction of H. Then the specular reflection is a combination of four factors:

$$S = DGF/\,(N \bullet E)$$

where:

D is the distribution function of the directions of the micro facets on the surface;

G is the amount by which the facets shadow and mask each other;

F is the Fresnel reflection law.

The distribution Function (D) is the evaluation of the distribution of the number of facets pointing in the direction of H. Blinn uses the formula given below to calculate D; this formula is based on modeling the micro facets as ellipsoids of revolution.

$$D_3 = [\frac{c_3^2}{\cos^2\alpha(c_3^2 - 1) + 1}]$$

where $\alpha = \cos^{-1}(N \bullet H)$, c_3 is the eccentricity of the ellipsoids, 0 for very shinny surface, 1 for very diffuse surface.

The Fresnel reflection (F) gives the fraction of light incident on a facet that is reflected rather than absorbed. It is a function of the angle of incidence of the light ($\varphi = \cos^{-1}(L \bullet H) = \cos^{-1}(E \bullet H)$) and the index of refraction of the substance.

This is the equation for Fresnel:

$$F = \frac{1}{2}[\frac{\sin^2(\phi - \theta)}{\sin^2(\phi + \theta)} + \frac{\tan^2(\phi - \theta)}{\tan^2(\phi + \theta)}]$$

This formula can be simplified into the one below:

$$F = \frac{(g - c)^2}{(g + c)^2}[1 + \frac{(c(g + c) - 1)^2}{(c(g - c) + 1)^2}]$$

where $c = (E \bullet H)$, $g = \sqrt{n^2 + c^2 - 1}$.

Note: this expression must be divided by 2.0 to give correct result.

2.10 Perlin Noise

Many people have used random number generators in their programs to create unpredictability, make the motion and behavior of objects appear more natural, or generate textures. Random number generators certainly have their uses, but at times their output can be too harsh to appear natural.

Ken Perlin introduced the notion of procedural texture mapping for realistic representation of objects found in nature for wood, marble, etc. He achieved this by using noise functions, now known as Perlin Noise functions. Simply speaking, Perlin Noise function adds up noise functions at a range of different scales. These functions capture the random behavior of natural objects [17]. Researchers have also investigated the possibility of modeling based on procedurally generated geometry [18]. Stamminger et al. [19] show how to sample procedural scenes and then to model and render them using point based modeling and rendering.

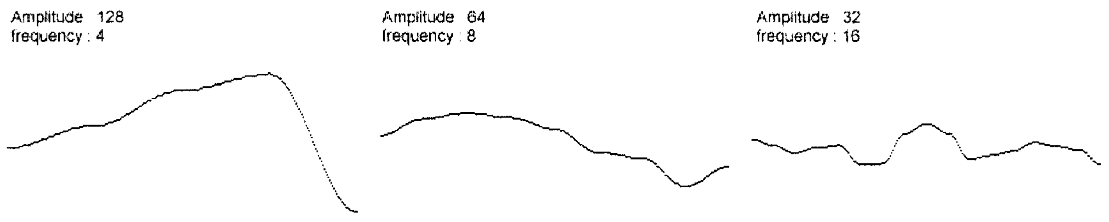

Figure 11: Noise functions 1

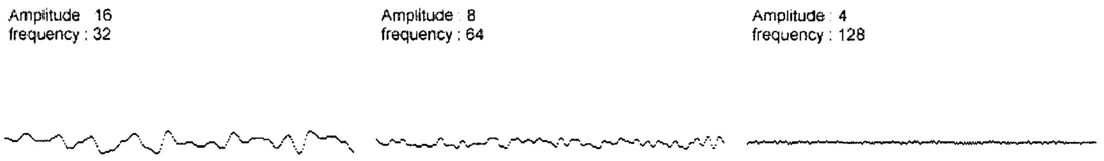

Figure 12: Noise functions 2

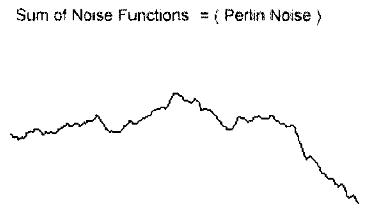

Figure 13: Perlin Noise composed by previous noise functions

Chapter 3

Environment Lighting

Splatting is a standard technique which associates at rendering time a spatial extent of influence for a point in order to ensure hole-free rendering. Splat based rendering gives reasonably acceptable results in terms of rendering the discrete representation as a continuous looking surface. However even till today, most of the renderers for point sampled geometry employ a very simple local illumination model. Typically, this includes one or more distant point light sources and possible variation in the diffuse, specular and shininess coefficients of the standard Phong illumination model. This makes it very difficult to create high quality images with point sampled surfaces. Also, because of the discontinuous nature of the points and the splats, the rendering results are even poorer under magnification. This lack of effort in high quality rendering could be due both to the infant nature of this field and also to the lack of direct hardware support for rendering the spatial extent of discrete 3D points making it inefficient to produce high quality images. Clearly more complex lighting models have to be considered.

In this chapter we consider the rendering of point sampled surfaces with both diffuse and specular material properties under distant illumination, as specified using an environment map. We have combined two of the earlier works for continuous surfaces: (1) spherical harmonic representations of irradiance environment maps for illumination of diffuse surfaces with light falling on the objects from all the directions [1], and (2) glossy reflection for illumination of surfaces with specular behavior [20]. We have adapted these formulations for point based surface representations and for computational efficiency programmed it on the GPU using vertex and fragment shaders. This hardware accelerated implementation has been incorporated into a public domain point based renderer, namely PointShop3D [21], enabling us to efficiently produce high quality rendered images of point sampled geometry.

The rest of this chapter is organized as follows: Section 3.1 introduces the techniques which we used for diffuse component; Section 3.2 illustrates how to calculate the specular component; Section 3.3 shows the rendering result with both diffuse and specular components.

3.1 Irradiance Environment Map for Diffuse Lighting:

There are three benefits of using this technique of spherical harmonic based irradiance environment map introduced in [1]:

- people perceive materials more easily under irradiance map than geometry lighting (point light and directional light).
- The cost of evaluating the light condition is independent of the number of light sources. Because the lighting environment is represented as a whole, and the integral is pre-computed.
- Since the first 9 coefficients approximate the integrated result over the hemisphere very well, it is computationally efficient even for real time rendering of point based models.

For the benefit of the readers, we repeat parts of the formulation given in [1] below as we use the same for point based lighting as well. For a complete detailed derivation, the reader is referred to the original paper [1].

After ignoring shadows and near-field illumination, the irradiance E is a function of surface normal n only and is given by an integral over the upper hemisphere represented using spherical harmonics:

$$E(\theta, \phi) = \sum_{l,m} A_l L_{lm} Y_{lm}(\theta, \phi) \tag{6}$$

where Y_{lm} denote spherical harmonic functions, L_{lm} denote the spherical harmonic coefficients of the incident light in their expansion, and A denotes the dot product of the normal and the direction vector for which analytical formula have been provided [22].

We can calculate L_{lm} in a preprocessing operation:

$$L_{lm} = \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} L(\theta, \phi) Y_{lm}(\theta, \phi) \sin\theta d\theta d\phi \tag{7}$$

Just 9 coefficients ($l \leq 2$) are sufficient for a close approximation. The average error is less than 1%. For an environment map ($EMap$), we precalculate the 9 spherical harmonic coefficients:

$$L_{lm}(0 \leq l \leq 2, -1 \leq m \leq 1)$$

These calculations are done separately for each of RGB channels and stored in an array SH, which is then passed on to the vertex shader. Here is the pseudo code to get SH:

```
$for(i = 0; i < \texttt{EMapHeight}; ++i)$
    $for(j = 0; j < \texttt{EMapWidth}; ++j)$
        $for(col = 0; col < 3; ++col)$
        {
            $\texttt{index} = l * (l + 1) + m + 1;$
            $SH[col][\texttt{index}] += (-1)^{\texttt{index}} * \texttt{EMap}[i, j] * Y_{lm} * d\omega;$
        }
```

The computation of the diffuse lighting component for each point is carried out in the vertex shader. For this, we need to carry out the following calculations for each point, say p and its given normal: $N(x, y, z)$.

Please refer to Equation 2 and Equation 3 for Spherical harmonic basis functions and associated variables, and refer to Equation 4 for each frequency's diffuse color contribution. These equations are all given in Section 2.7.

Summing all the frequencies together:

$$\texttt{Diffuse}(p) = \sum E_{lm}(p)$$

As mentioned earlier, in our actual vertex shader implementation, we have reused a clever trick introduced in [14], which uses less number of GPU registers. In what follows, R_{lm} , G_{lm} , B_{lm} are

Table 1: Diffuse Lighting GPU program parameters

C/R	cAr	cAg	cAb	cBr	cBg	cBb	cC
x	$-c_1 R_1^1$	$-c_1 G_1^1$	$-c_1 B_1^1$	$c_2 R_2^{-2}$	$c_2 G_2^{-2}$	$c_2 B_2^{-2}$	$c_4 R_2^2$
y	$-c_1 R_1^{-1}$	$-c_1 G_1^{-1}$	$-c_1 B_1^{-1}$	$-c_2 R_2^{-1}$	$-c_2 G_2^{-1}$	$-c_2 B_2^{-1}$	$c_4 G_2^2$
z	$c_1 R_1^0$	$c_1 G_1^0$	$c_1 B_1^0$	$3c_3 R_2^0$	$3c_3 G_2^0$	$3c_3 B_2^0$	$c_4 B_2^2$
w	$c_0 R_0^0 - c_3 R_2^0$	$c_0 G_0^0 - c_3 G_2^0$	$c_0 B_0^0 - c_3 B_2^0$	$-c_2 R_2^1$	$-c_2 G_2^1$	$-c_2 B_2^1$	any

just L_{lm} in RGB channel respectively.

Before we look at the actual vertex shader pseudo code, we need to define the following:

$$c_0 = n_0; c_1 = h_1 n_1; c_2 = h_2 n_2; c_3 = h_2 n_4;$$

Table 1 shows the formulation of the different parameters which are maintained in the GPU registers.

Now we can implement Equation 4 and Equation 5 within the vertex shader as shown below:

```
$float3\ x, y, z;$
$x_1.r = dot(cAr, \texttt{vNormal});$
$x_1.g = dot(cAg, \texttt{vNormal});$
$x_1.b = dot(cAb, \texttt{vNormal});$
$float4\ vB = \texttt{vNormal}.xyzz * \texttt{vNormal}.yzzx;$
$x_2.r = dot(cBr, vB);$
$x_2.g = dot(cBg, vB);$
$x_2.b = dot(cBb, vB);$
$float\ vC = \texttt{vNormal}.x * \texttt{vNormal}.x - \texttt{vNormal}.y * \texttt{vNormal}.y;$
$x_3 = cC.rgb * vC;$
$float3\ \texttt{vDiffuse};$
$\texttt{vDiffuse} = x_1 + x_2 + x_3;$
```

Below we show a few images of point sampled objects rendered with diffuse lighting only.

Figure 14 is an environment map called rnl_probe.hdr, whose format is mirrored ball and dimension is 900 $\times$ 900; Figure 15 is the igea model, which has 134,345 points, illuminated by the environment map, only with diffuse lighting.

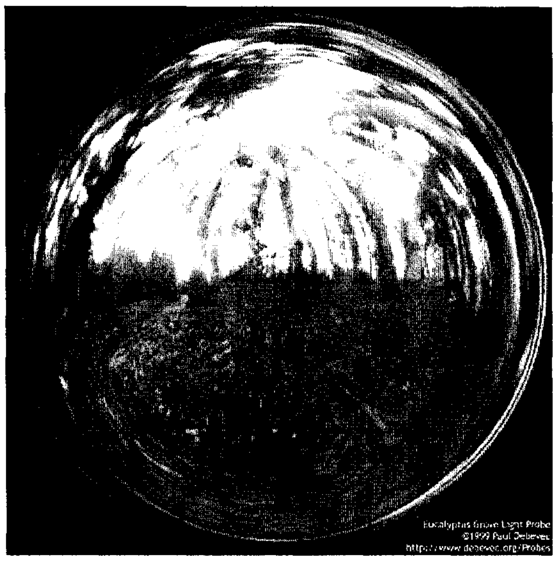

Figure 14: Environment map from rnl_probe.hdr

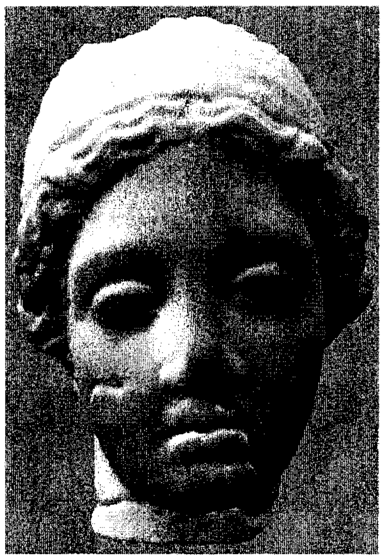

Figure 15: Model igea illuminated by rnl_probe.hdr, only with diffuse component

Figure 16 is an environment map called uffizi_probe.hdr, whose dimension is 1500×1500; Figure 17 is the igea model illuminated by the environment map, also only with diffuse lighting.

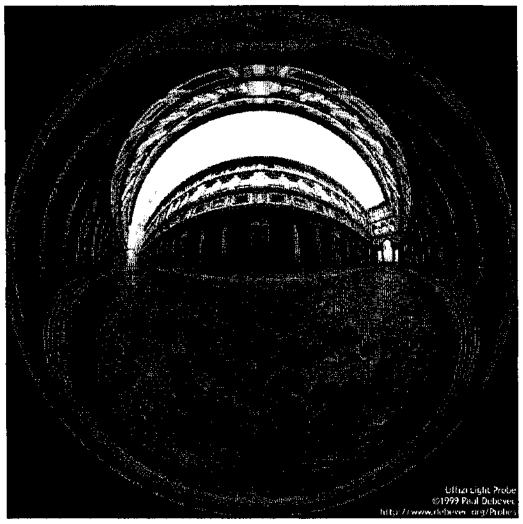

Figure 16: Environment map from uffizi_probe.hdr

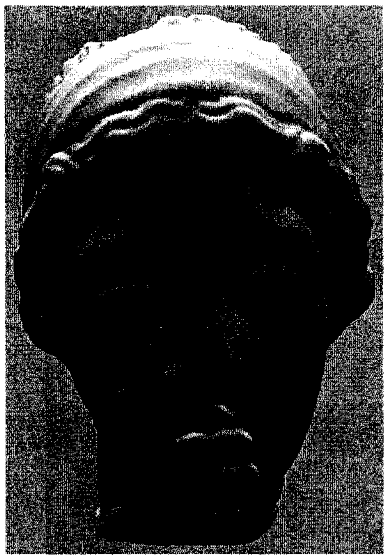

Figure 17: Model igea illuminated by uffizi_probe.hdr, only with diffuse component

3.2 Environment Map based Specular Lighting:

We also use the same environment map to simulate specular lighting effects. This part of the calculation is also done in GPU. Here, we get a vector from the eye position to each point. From this vector, we compute a reflected vector. This represents a perfect mirror reflection.

In order to simulate glossy reflection, rather than perfect mirror reflection, we also create another random vector within a cone centered around the mirror reflection vector. Based on this reflected vector, using the environment map as a texture, we index into the texture map and get a color value `vSpecular` as the incident light in that direction.

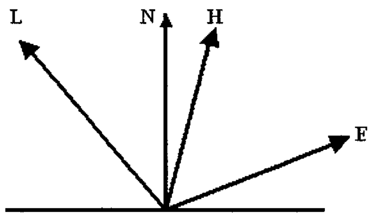

Figure 18: Specular Reflection

where $H = (L + E)/2$.

Then we calculate a Fresnel term F to determine how much light is reflected, assuming that the rest is absorbed by the object (see Figure 18):

$$F = \frac{(g-c)^2}{(g+c)^2}[1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}]$$

where $c = (E \bullet H)$, $g = \sqrt{n^2 + c^2 - 1}$, n is the index of refraction. Note: this expression must be divided by 2.0 to give correct result.

Lastly we compute a value for the distribution term D given by:

$$D = [\frac{c^2}{\cos^2\alpha(c^2-1)+1}]$$

where $\alpha = \cos^{-1}(N \bullet H)$, c is the eccentricity of the ellipsoids, 0 for very shiny surface, 1 for highly diffuse surface.

We are now ready to compose diffuse and specular lighting components together. K_d and K_s are the weights for diffuse and specular components respectively. For example, using $K_d = 0.8, K_s = 0.2$:

$$\texttt{Final\_Color} = K_d * \texttt{vDiffuse} + K_s * \texttt{vSpecular} * F * D; \tag{8}$$

The GPU implementation of the specular lighting component is distributed between the vertex shader and the fragment shader programs as follows. First in vertex shader, we connect camera position with each vertex to form a view direction. From the view direction and the vertex's normal, we can get a mirror reflection direction and perturb it randomly as described above to obtain the reflected vector. Using this reflected vector, we get a corresponding 2D texture coordinate.

Below we show the pseudo code to get the 2D texture coordinate based on a ray into the mirrored environment map:

```
float2  GetTexcoordFromRay(float3  vReflect)
{
    float2  texCoord;
    float  $u, v$;
    float  $\theta, \phi$;
    $\theta = \arccos(\texttt{vReflect}.z)$;
    float  tan_phi;
    if($|\texttt{vReflect}.x| < 0.000001$)
        $\texttt{tan\_phi} = 0$
    else
        $\texttt{tan\_phi} = \texttt{vReflect}.y \,/\, \texttt{vReflect}.x$;

    if($\sin\theta \geq 0$)
        $\phi = \texttt{atan2}(\texttt{vReflect}.y, \texttt{vReflect}.x)$;
    else
        $\phi = \texttt{atan2}(-\texttt{vReflect}.y, -\texttt{vReflect}.x)$;

    if($\texttt{vReflect}.x \geq 0$)
        $u = \theta \,/\, (\pi * \sqrt{1 + \texttt{tan\_phi}^2})$
    else
        $u = -\theta \,/\, (\pi * \sqrt{1 + \texttt{tan\_phi}^2})$

    $v = \texttt{tan\_phi} * u$

    $\texttt{texCoord}.x = (u + 1) * 0.5$;
    $\texttt{texCoord}.y = (1 - v) * 0.5$;

    return  texCoord;
}
```

We then output this 2D coordinate to the fragment shader of the same rendering pass. Secondly, in vertex shader, we also calculate the fresnel term and distribution term based on the three vectors (view direction, reflected vector, normal). We save these F and D scalar values and output them to fragment shader too.

In fragment shader, based on the 2D texture coordinate, we do texture mapping to get a color, then multiply this color by F and D in order to get the final value for the specular lighting component. This is then composed with the diffuse component to get the final color for the point using Equation 8.

3.3 Rendering Result with both Diffuse and Specular components

As described earlier, PointShop3D [21] is a public domain system for interactive shape and appearance editing of 3D point sampled geometry. PointShop3D encourages developers to create their own plug-in components. We have implemented our environment lighting computations described above using vertex and fragment shaders for efficiency and replaced PointShop3D's EWAGLRenderer, a hardware renderer which uses the standard Phong lighting model in its shader, with our own renderer.

To further decrease the mirror effect of the specular part, we don't use the original environment map for texture mapping. Instead, we use Gaussian blur, which is typically used to reduce image noise and reduce detail levels, to blur the environment map. After blurring, the 9 spherical harmonic coefficients, L_{lm}, almost remain the same values; that means the diffuse component is not affected by the blurring.

Figure 19 is a blurred environment map from rnl_probe.hdr. Figure 20 to Figure 23 show the model gnome, which has 54,772 points, illuminated by the blurred environment map, with increasing glossy surfaces.

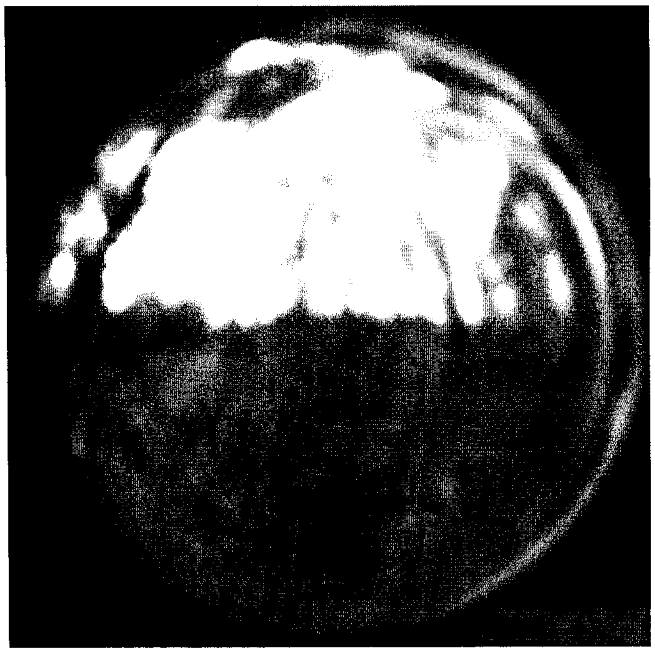

Figure 19: Blurred environment map from rnl_probe.hdr

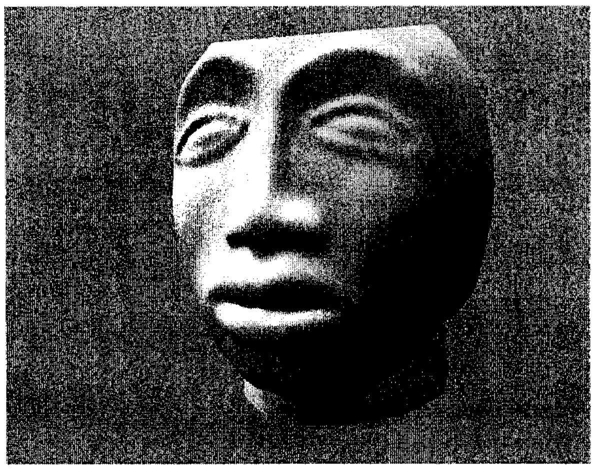

Figure 20: Model gnome illuminated by the blurred environment map, only with diffuse component

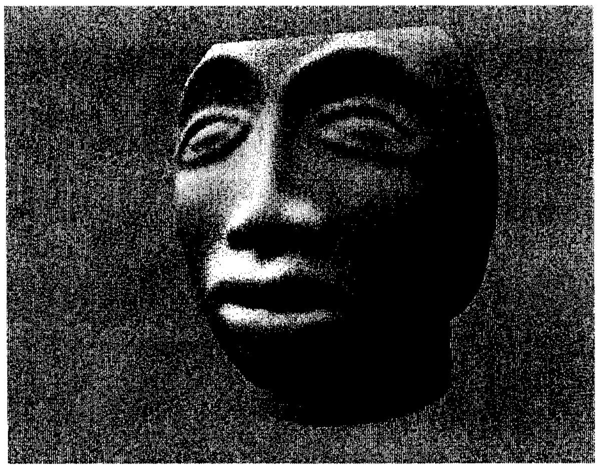

Figure 21: Model gnome illuminated by the blurred environment map, little glossy surface

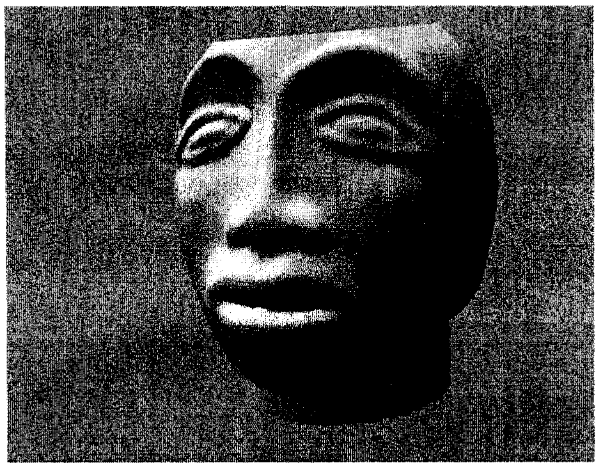

Figure 22: Model gnome illuminated by the blurred environment map, medium glossy surface

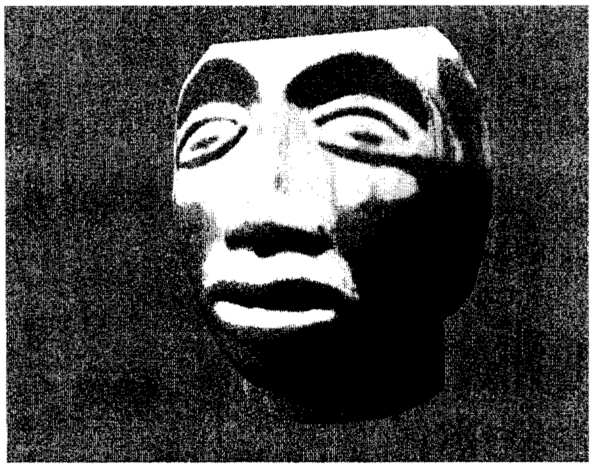

Figure 23: Model gnome illuminated by the blurred environment map, high glossy surface

For a point based model Armadillo, which has 345,944 points, our renderer can achieve a frame rate of 37 frames per second, while the frame rate of PointShop3D's EWAGLRenderer is 50 frames per second. Our renderer is slightly slower, but the rendering results are much more photorealistic. Figure 24 shows the EWAGLRenderer's result, Figure 25 shows our renderer's result.

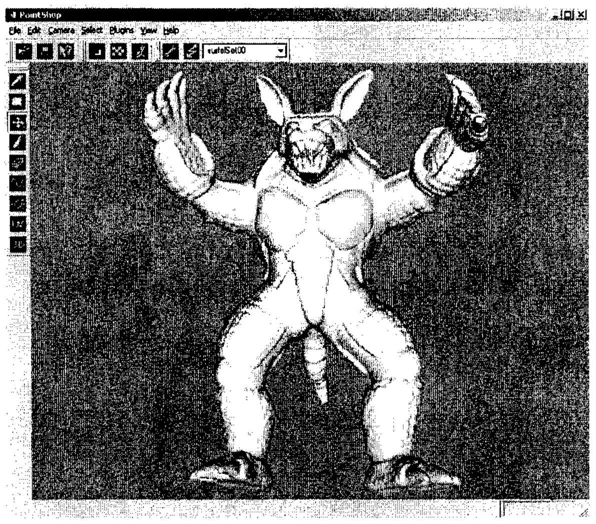

Figure 24: Model Armadillo rendered by Pointshop3D's EWAGLRenderer

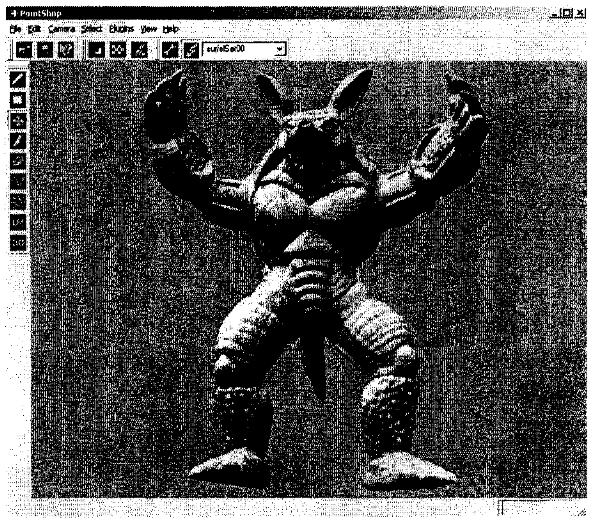

Figure 25: Model Armadillo rendered by our Environment Lighting renderer

Chapter 4

Ambient Occlusion

Local illumination models like the Phong model do not take into account secondary light effects, like parts of an object's surface shadowed from the light by other parts, usually nearby parts.

The ambient occlusion technique [23] [24] tries to attenuate light based on shadowing factors computed for parts of the object. It basically adds shadows to diffuse objects. Ambient occlusion is a crude approximation of the full rendering equation [25]. It takes into account inter-object visibility only. Surfels are shadowed based on whether they are partially occluded to the environment. For this we calculate the accessibility value, which is the percentage of the hemisphere above each surface point not occluded by geometry [24]. This is done in two passes as described below.

4.1 First Pass

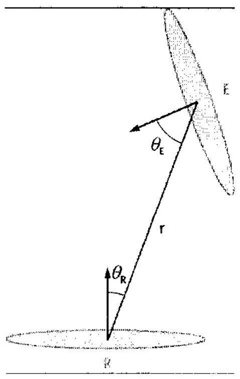

Figure 26: The Relationship between Receiver and Emitter

We refer to the surfel that is shadowed as the receiver R and to the surfel that casts the shadow as the emitter E. Amount of shadow that is transferred to R from E is given by an approximate form factor,

$$\frac{A\cos\theta_E\cos\theta_R}{\pi r^2 + A} \tag{9}$$

To calculate how much in shadow is receiver R, we add the form factors treating all other surfels as emitters. We also clamp the value to be less than 1.

The initial shadow value from first pass is

$$T_i = max(1, \sum_{j\neq i}\frac{A_j\cos\theta_e\cos\theta_r}{\pi r^2 + A_j}) \tag{10}$$

We create an octree hierarchy of surfels to efficiently do this calculation. To compute the ambient occlusion (shadow) factor for each surfel, we consider only those surfels which lie within a cone with geometry defined as follows, also shown in Figure 27:

- The cone axis is along the normal associated with the surfel
- The cone apex is at the 3D point associated with the receiver
- The cone angle has been empirically chosen to be 150 degrees. Surface parts that are nearly co-planar contribute very little to this occlusion. Smaller cone angle results in rejecting too many surfels. On the other hand larger cone angle makes less efficient culling.

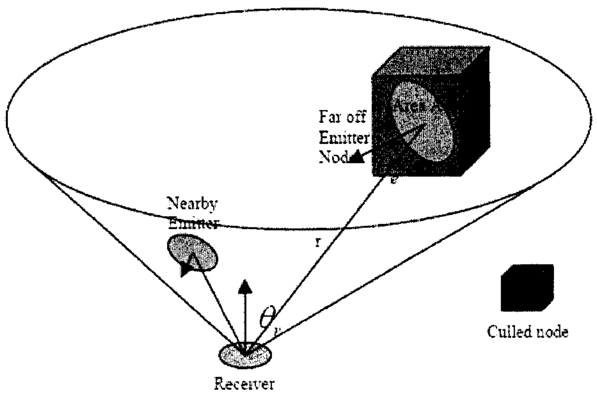

Figure 27: Illustration for Ambient Occlusion Computation

All surfels lying outside this cone are not included in the ambient occlusion computation. Fast culling of surfels is made possible by traversing the octree hierarchically and retaining only those octree nodes lying inside the cone.

To further speed up the calculations, distant and nearly flat nodes of octree are treated as single surfels. To determine whether the node is flat or not, we do a feature analysis based on Eigenvalue computations in that node. For further details about this method of feature analysis, reader is referred to [26] [27]. For non-flat nodes we continue traversing its child nodes until the sub node includes 10 or fewer surfels.

We have therefore two cases in our form factor computation:

- Form factor for a distant flat node: All the surfels in a given node are merged to form a surfel at the center of the node. Area of this new surfel is equal to sum of areas of all surfels in the node. Since the node is locally planer, we use the normal of any surfel as normal of this new surfel. This way we approximate the computation of form factors for surfels in a flat node.
- Form factor for nearby or non-flat node: Form factors are computed for every surfel in node and added up to give the form factor for that node.

4.2 Second Pass

Surfels tend to be too dark after the first pass. This is due to ignoring the fact that an emitter which is itself in shadow should not fully add up to the shadow factor (refer to Figure 6 to Figure 8). For more detail, refer to [15]. We can correct this by using the shadow values obtained in first pass in a second pass as follows:

Final shadow factor:

$$S_i = max(1, \sum_{j \neq i} (1 - T_j) \frac{A_j \cos\theta_e \cos\theta_r}{\pi r^2 + A_j}) \tag{11}$$

The use of hierarchical structure for occlusion computation is not new. A GPU based hierarchical ambient occlusion computation technique has already been presented in [28]. However we are not aware of any other work that makes beneficial use of feature analysis as we have done.

Bunnell [15] provides a method which can compute ambient occlusion of a thousand vertices model in real time. Bunnell encoded the vertex information into textures to perform the calculation on GPU. Although this method can achieve very high performance, it is limited to models with a few thousand vertices. High density point models can have hundred thousand points or more. Therefore, given the present capacity of programmable GPUs, we have limited our preprocessing to a software implementation rather than GPU implementation. We use the octree combined with eigenvalue analysis to improve the performance. Using this method we can preprocess ambient occlusion calculations for a model having about 100,000 points in approximately 24 minutes.

For each point of the point based model, we get one ambient occlusion value. We multiply the original color at the point with its ambient occlusion value to get the new color for that point. Now the model has self-shadow information without increasing the model's storage size.

4.3 Rendering Results using Ambient Occlusion

In this section, we present some examples showing the self-shadowing effects provided by Ambient Occlusion Calculation.

Figure 28 shows the original mannequin model, which has 87,183 points.

Figure 29 shows the modified mannequin model preprocessed by one pass ambient occlusion calculation. It is too dark in some areas because of double shadowing (refer to Section 2.8).

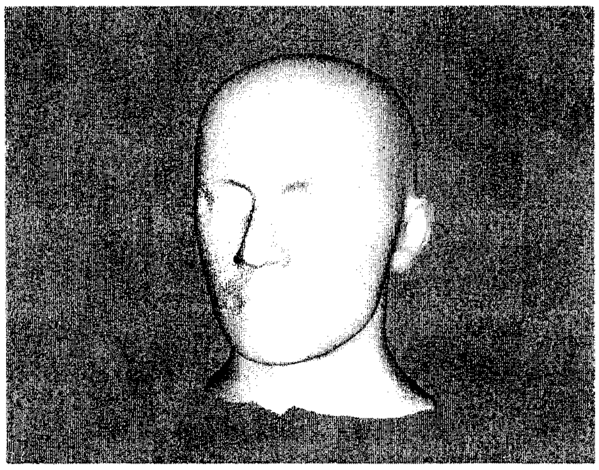
Figure 28: Original model of mannequin, without self-shadowing

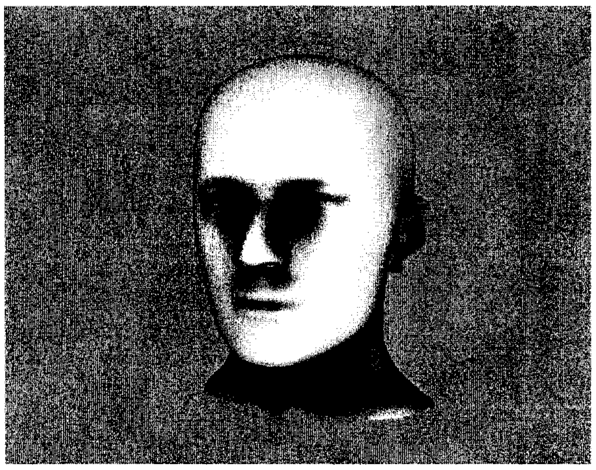
Figure 29: One pass ambient occlusion precalculation

Figure 30 shows the modified mannequin model preprocessed by two pass brute force ambient occlusion calculation.

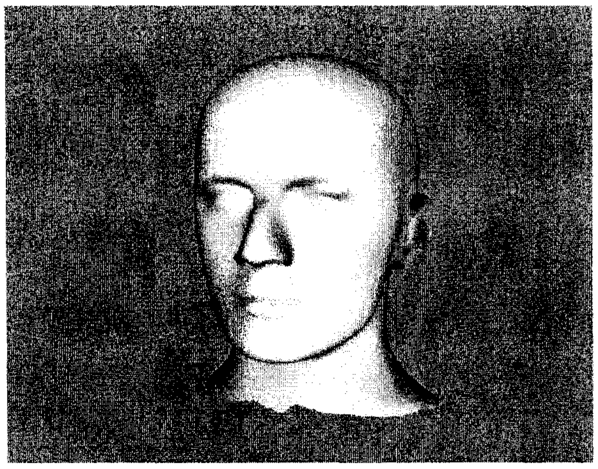

Figure 30: Two pass brute force ambient occlusion precalculation

Figure 31 shows the modified mannequin model preprocessed by two pass ambient occlusion calculation with our acceleration algorithm. Comparing 30 with 31, we can see that the rendered results are almost the same. It takes 28,793.5 seconds to brute force calculate the model mannequin. On the other hand using our acceleration algorithm, it only takes 1,477.14 seconds for the same model, almost 20 times faster.

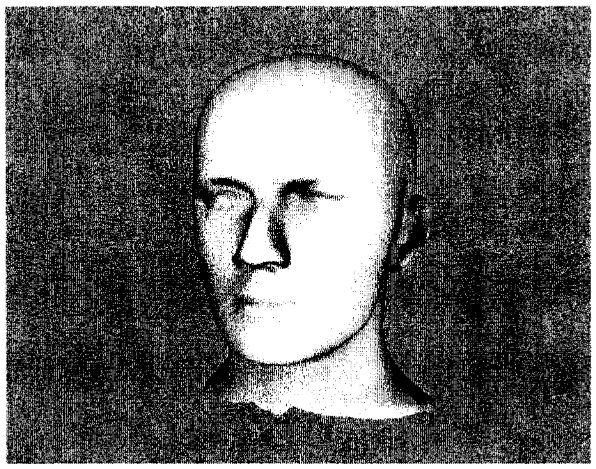

Figure 31: Two pass ambient occlusion precalculation optimized by octree and eigenvalue evaluation

4.4 Rendering Results Combining Ambient Occlusion with Environment Lighting

We precalculate Ambient Occlusion value for the point based model mannequin to give self-shadow effect, then render it using our Environment Lighting technique from Chapter 3; by this we can achieve even better rendering quality.

Figure 32 and Figure 33 are the rendering results of the mannequin model illuminated by the environment map rnl_probe.hdr (refer to Figure 14). Figure 34 and Figure 35 are the rendering results of the mannequin model illuminated by the environment map uffizi_probe.hdr (refer to Figure 16).

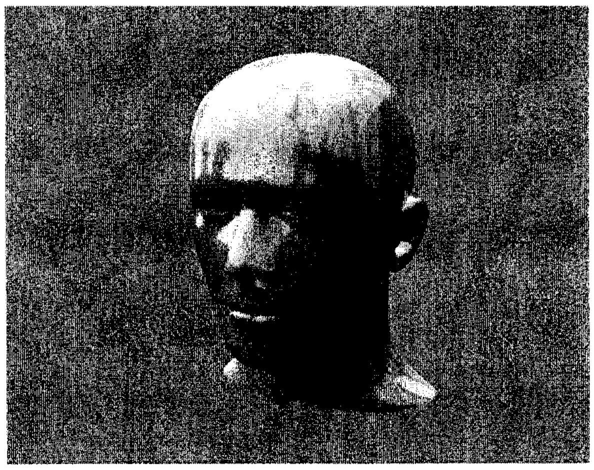
Figure 32: Original model of mannequin illuminated by rnl_probe.hdr

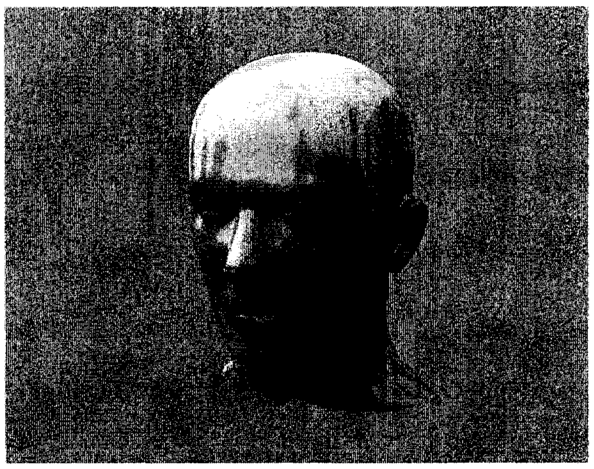
Figure 33: Self-shadowing model of mannequin illuminated by rnl_probe.hdr

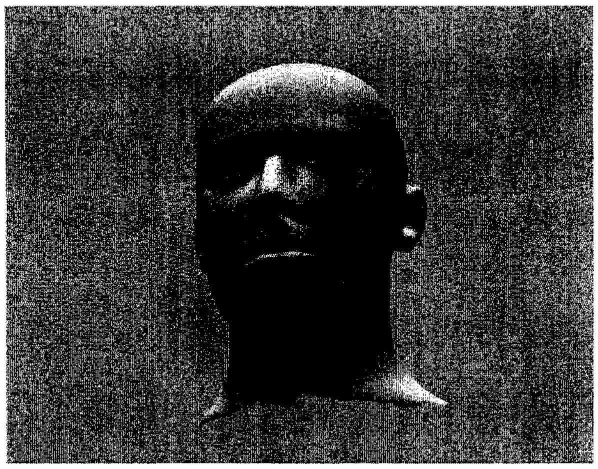

Figure 34: Original model of mannequin illuminated by uffizi_probe.hdr

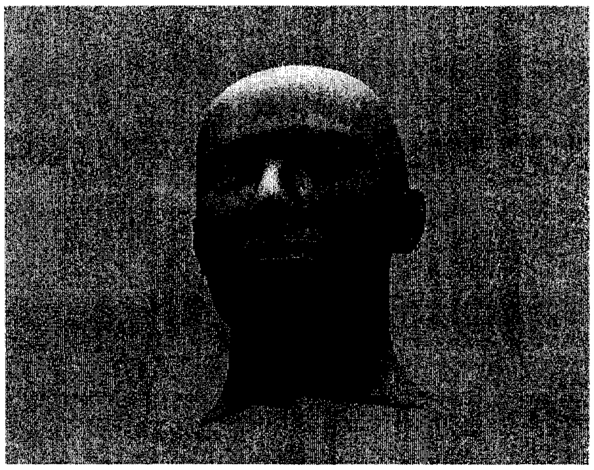

Figure 35: Self-shadowing model of mannequin illuminated by uffizi_probe.hdr

For the four figures, we use $K_d = 0.7, K_s = 0.3$ for Equation 8. Compare Figure 32 with Figure 33, or compare Figure 34 with Figure 35, and one can see that the modified model gives much more realistic rendering results, especially in the nose, ear, and lip regions.

Chapter 5

Procedural Texture

Texture mapping is a difficult problem for point sampled geometry, so we also tried to implement procedural texturing for point cloud surface data.

5.1 Implementation of Procedural Texture

When loading a point based model we use standard Perlin Noise function to generate a 3D noise texture.

You can set different frequencies and amplitudes to generate noise textures with different appearances. Usually a noise texture looks like this:

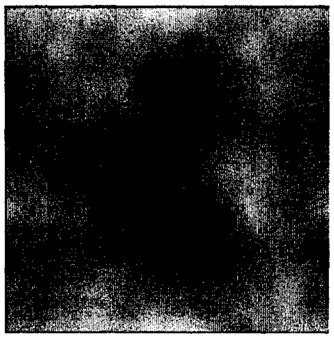

Figure 36: Noise texture

We use vertex shader and fragment shader to render the procedural texture. We pass 6 parameters to fragment shader; then carry out the texture mapping in fragment shader. The 6 parameters:

- volume texture (`volumeTexture`): when loading the point based model, compute a 3D noise texture using perlin noise; directly passed to fragment shader. (This is the only parameter passed directly to fragment shader, others are passed through vertex shader to fragment shader)
- light vector (`lightVec`)
- view vector (`viewVec`)
- normal (`normal`)
- original color of the point (`vColor`)
- texture coordinates (`texCoord`): when loading the model, compute the bounding box of the point based mesh, then scale the mesh's position to the range of $[0, 1]$, save it as texture coordinates for texture mapping

This is the CG sample code for one of our procedural texture patterns:

```
float3  SimpleMarble(float3  vColor,
                     float3  texCoord,
                     float3  normal,
                     float3  lightVec,
                     float3  viewVec,
                     sampler3D  volumeTexture) {
    float3  baseColor = float3(1.0 * vColor.r, 0.843 * vColor.g, 0.624 * vColor.b);
    float3  noisy = tex3D(volumeTexture, texCoord).rgb;
    float  marble = (0.2 + 5 * abs(noisy.x + noisy.y + noisy.z - 0.5));
    // Phong
    float  diffuse = 0.5 * dot(lightVec, normal) + 0.5;
    float  specular = pow(saturate(dot(reflect(-viewVec, normal), lightVec)), 64);
    // We assume dark parts of the marble reflects light better
    float  K_s = saturate(1.1 - 1.3 * marble);
    return  diffuse * marble * baseColor + K_s * specular * float3(0.5, 0.5, 0.5);
}
```

5.2 Procedural Texture Rendering Results

Below are some rendered images with procedural texture for point based models:

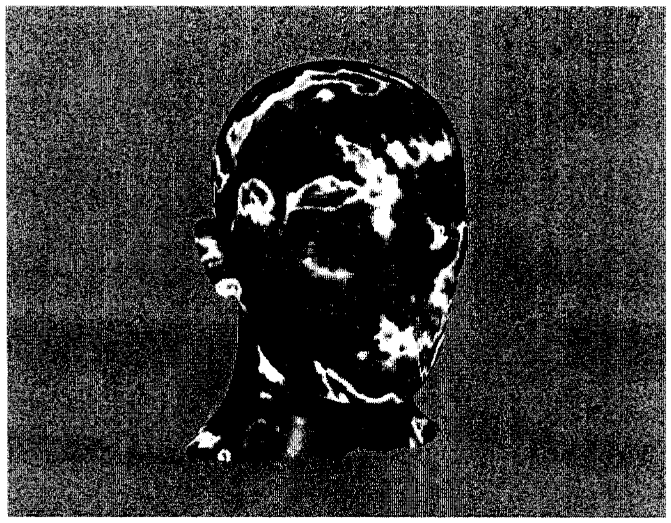

Figure 37: Veined Marble pattern for model mannequin

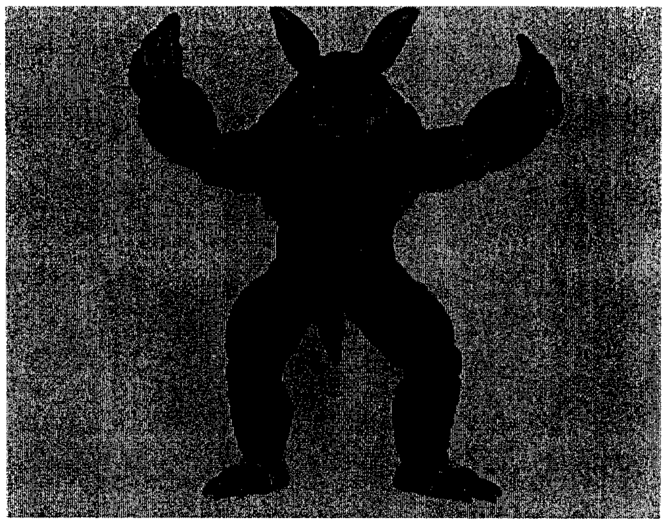

Figure 38: Plaid Fabric pattern for model Armadillo

5.3 Procedural Texture Combined with Ambient Occlusion

If we use ambient occlusion to add self-shadowing effect to the model first, then render with a procedural texture, we can get much better rendering results. Comparing Figure 39 with Figure 40 or Figure 41 with Figure 42, one can see that self-shadowing effect gives more detail of the model shape.

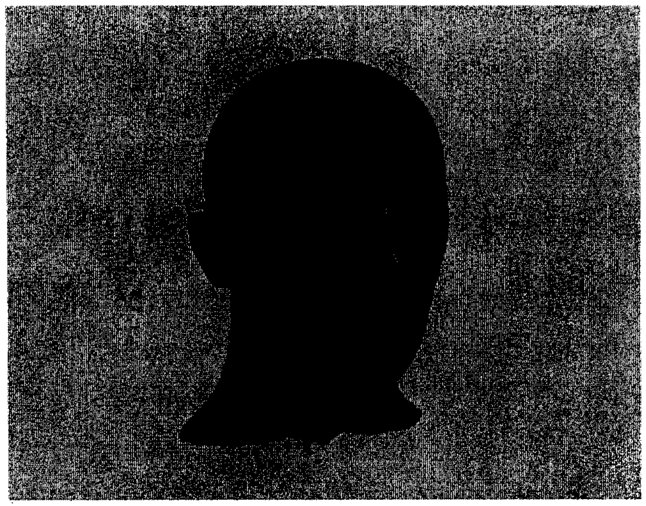

Figure 39: Plaid Fabric pattern for original model mannequin, without self-shadow effect

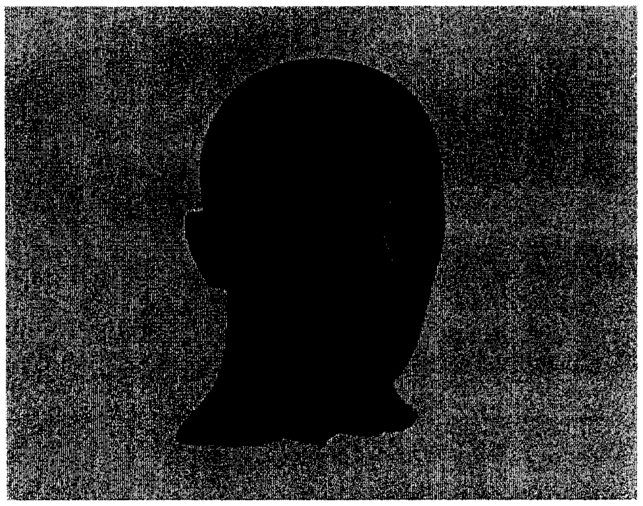

Figure 40: Plaid Fabric pattern for modified model mannequin, with self-shadow effect produced by ambient occlusion calculation

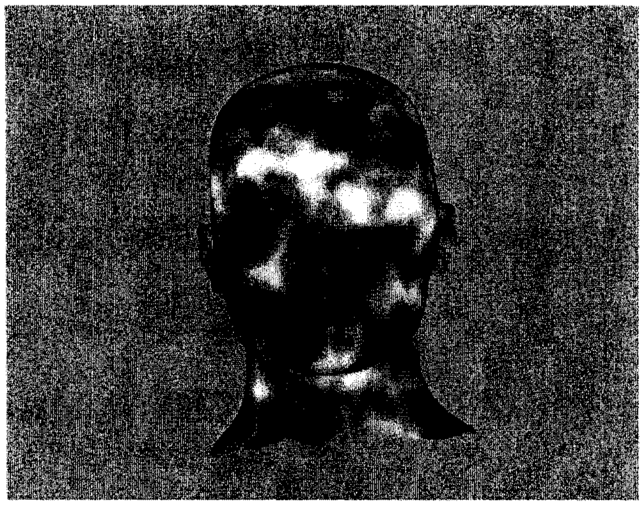

Figure 41: Simple Marble pattern for original model mannequin, without self-shadow effect

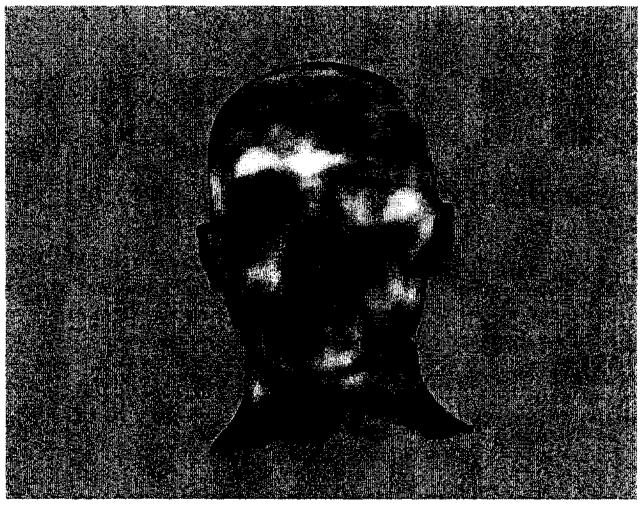

Figure 42: Simple Marble pattern for modified model mannequin, with self-shadow effect produced by ambient occlusion calculation

Chapter 6

Conclusion

6.1 Significant Contributions

In this thesis we have demonstrated that high quality mesh rendering techniques can be adapted for point based models yielding equally good quality images.

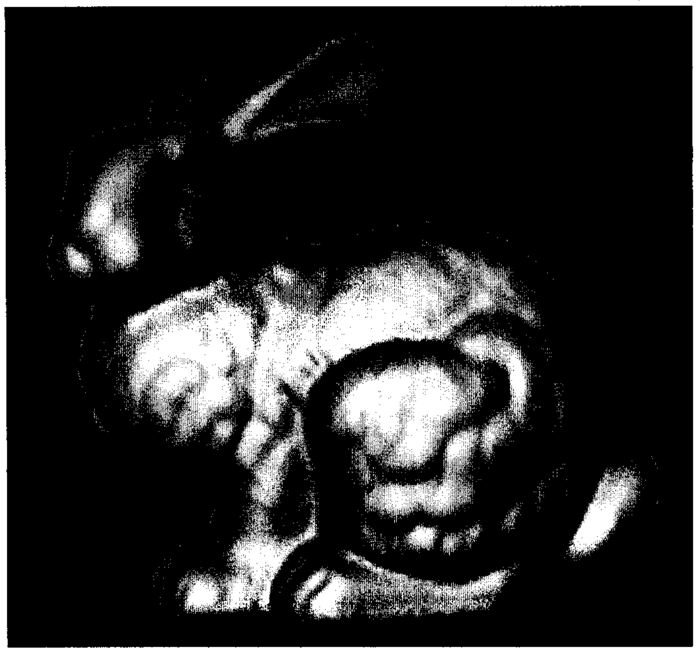

Figure 43: Triangle mesh model bunny by real time BRDF technique

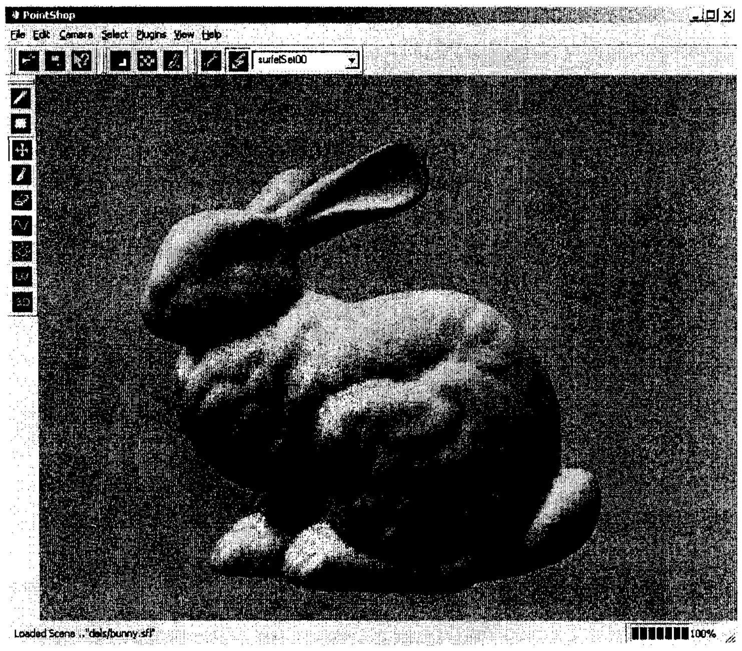
PointShop
File Edit Camera Select Plugins View Help
surfelSel00
Loaded Scene ..."dels/bunny.sfl"
100%

Figure 44: Point based model bunny using our environment lighting technique

We have adapted ambient occlusion techniques to enable self shadowing effects. An efficient method has been devised for ambient occlusion computation using a hierarchic structure and a flatness measure based on feature analysis using eigenvalue computations on the point cloud data. We also combine ambient occlusion with environment map based lighting supporting diffuse and specular behavior. For further efficiency, our renderer performs environment lighting and procedural texture mapping using vertex and fragment shaders. These shaders get ambient occlusion results directly embedded in color channel, thus avoiding any data overhead due to ambient occlusion information.

6.2 Future Work

Although we use octree hierarchy combined with eigenvalue evaluation to accelerate our ambient occlusion calculation, the preprocess time is still too much, usually about 30 minutes for big models. We think the best way to increase the efficiency considerably could be to make use of current powerful graphics card support. Next step we will try to do the ambient occlusion for big point based models in GPU.

To achieve more global illumination effects for point sampled geometry, such as soft shadows, inter-reflection, subsurface scattering, etc, we can try to use Spherical Harmonic Lighting [29] and Precomputed Radiance Transfer [30] on point sampled geometry.

Chapter 7

Appendix- GPU Based Implementation

Our implementation is incorporated into the open source software called Pointshop3D, which is available on http://graphics.ethz.ch/pointshop3d/. In their SIGGRAPH 2002 paper [21], M. Zwicker et al. describe this software (Pointshop3D) to interactively render point sampled geometry. Pointshop3D encourages developers to create their own plug-in components. EWAGLRenderer is a fast and high quality hardware renderer. It uses the standard Phong lighting model. We have replaced the EWAGLRenderer renderer with our own renderer, which combines our environment lighting and procedural texture techniques. Below is the pseudo code for EWAGLRenderer's Phong lighting shader:

```
float3  lightVec = normalize(position.$xyz$);
float3  eyeVec = float3(0.0, 0.0, 1.0);
float3  halfVec = normalize(lightVec + eyeVec);
float  diffuse = $dot$(normalVec, lightVec);
float  specular = $dot$(normalVec, halfVec);
float4  lighting = $lit$(diffuse, specular, 1024);
res.$rgb$   = lighting.$y$ * color.$rgb$ + lighting.$z$ * float3(1.0, 1.0, 1.0);
res.$a$ = 1.0;
```

Figure 45 is the rendered result obtained using EWAGLRenderer's Phong lighting model:

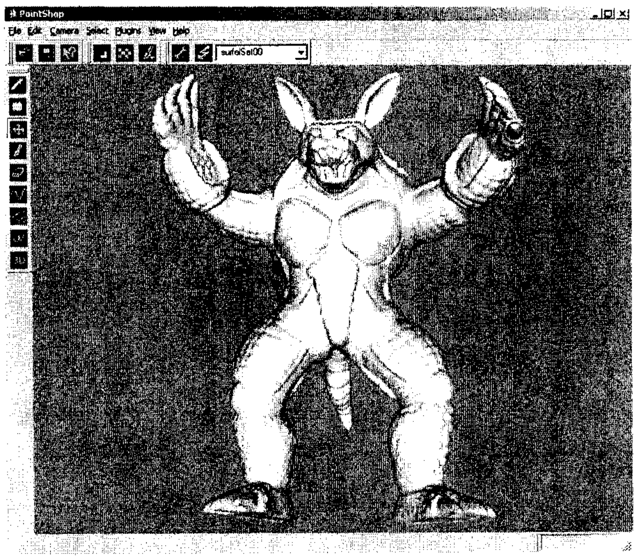

Figure 45: Model Armadillo rendered by PointShop3D's EWAGLRenderer

For environment lighting technique, we precompute the 9 spherical harmonic coefficients to represent the environment map in the initialization time. Then we use vertex shader and fragment shader to calculate the diffuse and specular components for each point in real time.

For diffuse component part (refer to Section 3.1), we use the precalculated 9 spherical harmonic coefficients to get the diffuse color for each point in vertex shader. For specular component (ref to Section 3.2), we calculate the Fresnel (F) and distribution (D) value for each vertex, also compute a reflected ray, index into the environment map to get a specular color value (`vSpecular`) . The final equation is:

$$\texttt{Final\_Color} = K_d * \texttt{vDiffuse} + K_s * \texttt{vSpecular} * F * D; \tag{12}$$

where K_d and K_s are the weights for diffuse and specular components respectively.

Figure 46 is the rendering result of our environment lighting technique.

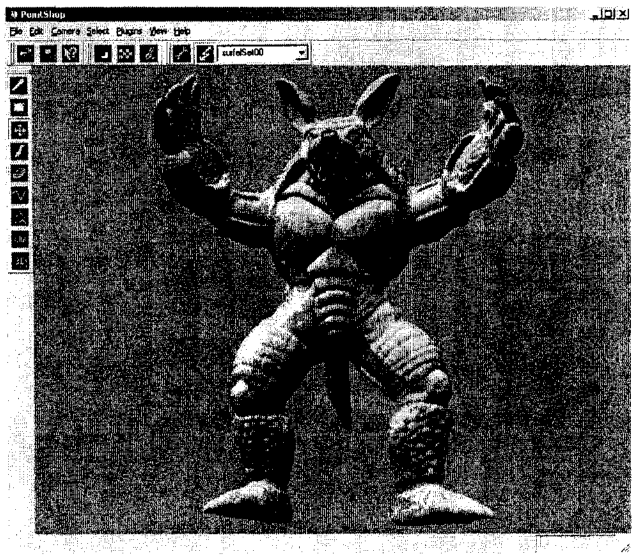

Figure 46: Model Armadillo rendered by our Environment Lighting renderer

For procedural texture technique, we use standard perlin noise method to generate a 3D volume texture when initializing. Then we do the procedural texture rendering for point sampled geometry also using vertex shader and fragment shader. Here is the pseudo code for the simple marble pattern:

```
float3  SimpleMarble(float3  vColor,
                     float3  texCoord,
                     float3  normal,
                     float3  lightVec,
                     float3  viewVec,
                     sampler3D  volumeTexture) {
    float3  baseColor = float3(1.0 * vColor.r, 0.843 * vColor.g, 0.624 * vColor.b);
    float3  noisy = tex3D(volumeTexture, texCoord).rgb;
    float  marble = (0.2 + 5 * abs(noisy.x + noisy.y + noisy.z - 0.5));
    // Phong
    float  diffuse = 0.5 * dot(lightVec, normal) + 0.5;
    float  specular = pow(saturate(dot(reflect(-viewVec, normal), lightVec)), 64);
    // We assume dark parts of the marble reflects light better
    float  $K_s$ = saturate(1.1 - 1.3 * marble);
    return  diffuse * marble * baseColor + $K_s$ * specular * float3(0.5, 0.5, 0.5);
}
```

Figure 47 is the armadillo model in simple marble pattern:

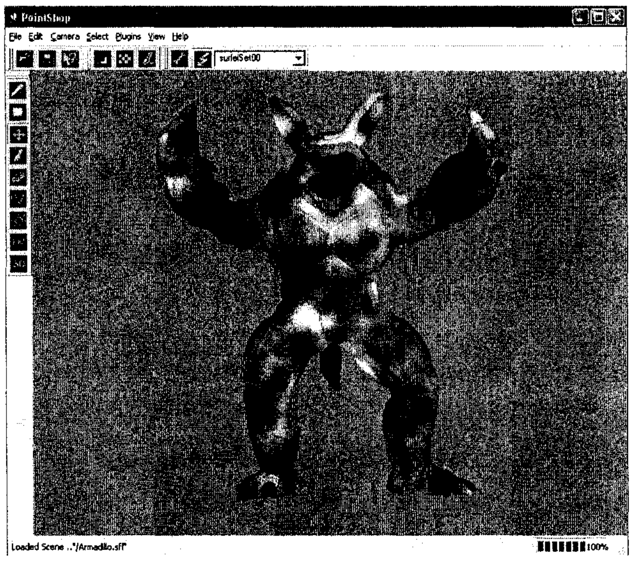
PointShop
File Edit Camera Select Plugins View Help
surfelSet00
Loaded Scene ..*/Armadillo.sfl
100%

Figure 47: Simple Marble pattern for Model Armadillo

Bibliography

[1] R. Ramamoothi and P. Hanrahan, "An efficient representation for irradiance environment maps," in *SIGGRAPH 2001*, 2001.

[2] M. Zwicker, H. Pfister, J. van Baar, , and M. Gross, "Surface splatting," in *SIGGRAPH 2001*, 2001.

[3] L. Ren, H. Pfister, and M. Zwicker, "Object space ewa surface splatting:a hardware accelerated approach to high quality point rendering," in *EUROGRAPHICS 2002*, 2002.

[4] M. Botsch and L. Kobbelt, "High-quality point-based rendering on modern gpus," in *Pacific Graphics 2003*, 2003.

[5] M. Zwicker, J. Rasanen, M. Botsch, C. Dachsbacher, and M. Pauly, "Perspective accurate splatting," in *Graphics Interface 2004*, 2004.

[6] M. Botsch, M. Spernat, and L. Kobbelt, "Phong splatting," in *Eurographics Symposium on Point-Based Graphics 2004*, 2004.

[7] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt, "High-quality surface splatting on today's gpus," in *Eurographics Symposium on Point-Based Graphics 2005*, 2005, pp. 17–24.

[8] G. Schaufler and H. Jensen, "Ray tracing point sampled geometry," in *Rendering Techniques 2000*. Springer-Verlag, 2000, pp. 319–328.

[9] A. Adamson and M. Alexa, "Ray tracing point set surfaces," in *Proceedings of Shape Modeling International 2003*, 2003, pp. 272–279.

[10] A. Adamson, M. Alexa, and A. Nealen, "Adaptive sampling of intersectable models exploiting image and object-space coherence," in *SI3D 2005*, 2005, pp. 171–178.

[11] I. Wald and H. Seidel, "Interactive ray tracing of point based models," in *Proceedings of 2005 symposium on Point Based Graphics*, 2005.

[12] B. Adams, R. Keiser, M. Pauly, L. Guibas, M. Gross, and P. Dutre, "Efficient raytracing of deforming point-sampled surfaces," in *Proceedings of the 2005 Eurographics conference*, 2005.

[13] Y. Dobashi, T. Yamamoto, and T. Nishita, "Radiosity for point-sampled geometry," in *12th Pacific Conference on Computer Graphics and Applications 2004*, 2004.

[14] Peter-Pike and J. Sloan, "Efficient evaluation of irradiance environment maps," in *ShaderX 2: Shader Programming Tips and Tricks with DirectX 9*, 2003, pp. 226–231.

[15] M. Bunnell, "Dynamic ambient occlusion and indirect lighting," in *GPU Gems 2*, 2005.

[16] C. Wynn, "Real-time brdf-based lighting using cube-maps," NVIDIA Corporation, Tech. Rep., 2000.

[17] K. Perlin, "An image synthesizer," in *ACM SIGGRAPH 85*, 1985.

[18] D. Ebert, F. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling: A Procedural Approach.* Morgan Kaufmann Publishers, 1998.

[19] M. Stamminger and G. Drettakis, "Interactive sampling and rendering for complex and procedural geometry," in *Rendering Techniques 2001*, 2001.

[20] W. Heidrich and H. Seidel, "View-independent environment maps," in *Graphics Hardware '98*, 1998.

[21] M. Zwicker, M. Pauly, O. Knoll, and M. Gross, "Pointshop 3d: An interactive system for point-based surface editing," in *SIGGRAPH 2002*, 2002.

[22] R. Ramamoorthi and P. Hanrahan, "On the relationship between radiance and irradiance: Determining the illumination from images of a convex lambertian object," in *Journal of the Optical Society of America A*, 2001.

[23] S. Zhukov, A. Iones, and G. Kronin, "An ambient light illumination model," in *Rendering Techniques 1998*, 1998.

[24] H. Landis, "Production-ready global illumination," in *SIGGRAPH 2002 course notes*, 2002, pp. 87–102.

[25] J. Kajiya, "The rendering equation," in *SIGGRAPH 1986*, 1986.

[26] S. Bhakar, L. Luo, and S. Mudur, "View dependent stochastic sampling for efficient rendering of point sampled surfaces," in *Journal of WSCG 2004*, 2004.

[27] S. Gumhold, X. Wang, and R. McLeod, "Feature extraction from point clouds," in *Proceedings of the 10th International Meshing Roundtable*, 2001, pp. 293–305.

[28] P. Matt and S. Green., "Ambient occlusion," in *GPU Gems*, R. Fernando, Ed. Addison-Wesley, 2004, p. 279C292.

[29] R. Green, "Spherical harmonic lighting: The gritty details," in *Game Developers Conference 2003*, 2003.

[30] P.-P. Sloan, J. Kautz, and J. Snyder, "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments," in *SIGGRAPH 2002*, 2002.