Ambient Occlusion - Using Ray-tracing and Texture Blending

Ingvi Rafn Hafthorsson
s041923

Kongens Lyngby 2007

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Ambient occlusion is the concept of shadows that accumulates at surfaces where the surface is partially hidden from the environment. The more the surface is hidden, the more ambient occlusion we have. The result is a subtle but realistic shadow effect on objects.

Ambient occlusion is implemented. To achieve this, existing methods are evaluated and utilized. Ray-tracing is used for casting rays from surfaces. The amount of rays that intersect the surrounding environment is used to find ambient values. The more rays that hit, the more shadow we get at the surface we are working on.

We use textures for storing and displaying the ambient values. Overlapping textures are implemented to eliminate visible seams at texture borders. A blending between the textures is introduced. The blending factor is the normal vector at the surface. We have three textures at the surface that each contain ambient values. To eliminate the possibility of having visible borders and seams between textures we suggest that the contribution of each texture will be values from each normal vector. The normal vector is normalized, and then we know that its values squared will sum up to 1. This is according to the well known Pythagoras theorem. We then consider each of these values to be a percentage and we know that they sum up to be 100%. This allows for us to control the contribution of each ambient texture, assigning one texture color with one normal vector value. The result of this is a smooth blending of ambient values over the entire surface of curved objects.

Preface

This thesis has been prepared at the Section of Computer Graphics, Department of Mathematical Modelling, IMM, at the Technical University of Denmark, DTU, in partial fulfillment of the requirements for the degree Master of Science in Engineering, M.Sc.Eng. The extent of the thesis is equivalent to 30 ETCS credits.

The thesis covers illumination of graphical models. In particular a shadow effect, called ambient occlusion. The reader is expected to have fundamental knowledge of computer graphics, illumination models and shadows.

Lyngby, May 2007

Ingvi Rafn Hafthorsson

Acknowledgements

First I want to thank my supervisors Niels Jørgen Christensen and Bent Dalgaard Larsen for suggesting me to go in this direction. Bent I also thank for always be willing to point me in the right direction when I got lost and always be willing to evaluate my progress, sometimes at unreasonable hours.

I thank my family for having faith in me. My mother for her biased comments (of course biased) and my sister for making the work in the last days possible.

I would like to thank my friend Hilmar Ingi Runarsson for valuable comments and constructive critisims in the last days of work.

Last I thank my girlfriend Ragnheidur for her patience and understanding in the final and most intense periods of the project work. Especially for always be willing to listen to my progress and thoughts no matter how much technical details I was talking about.

Contents

Abstract **i**

Preface **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 General Thoughts 1

1.2 Shadow Effects 3

1.3 Global Illumination 4

1.4 Ambient Occlusion 5

1.5 Contributions 6

1.6 Thesis Overview 7

2 Motivation and Goal **9**

2.1 Motivation 9

2.2 Goal 10

3 Ambient Occlusion in Practice 11

3.1 What is it 11

3.2 When to use it 12

3.3 How is it implemented 13

4 Previous Work 15

4.1 Ambient Light Illumination Model 15

4.2 The Model Refined 17

4.3 Advanced Ambient Occlusion 18

5 Occlusion Solution 21

5.1 General Approach 21

5.2 Using Vertices 22

5.3 Using Textures 25

5.4 Blending Textures 27

5.5 Combining Textures 28

6 Design 29

6.1 Import/Export 29

6.2 Data Representation 30

6.3 Algorithms 30

6.4 Objects 32

6.5 Final Structure 33

7 Implementation 35

7.1 Data Structure 35

7.2 Finding Adjacent Triangles 41

7.3 Clustering Algorithm 41

7.4 Finding Ambient Occlusion 44

7.5 Texture Blending 50

7.6 Texture Packing Algorithm 50

7.7 User Input 52

8 Testing 55

9 Results 59

10 Discussion 63

10.1 Summary 63

10.2 Contributions 64

11 Future Work 67

12 Conclusion 71

A Tools 75

A.1 COLLADA 75

A.2 COLLADA DOM 76

A.3 Softimage XSI 76

A.4 OpenGL and Cg 77

B Screenshots **79**

List of Figures

1.1 The importance of shadows. 2

1.2 Ambient occlusion in a living room. 3

1.3 Hard shadows and soft shadows. 4

1.4 Ambient occlusion in a molecule model. 6

3.1 Self-occlusion and contact shadow. 12

3.2 Worn gargoyle. 13

3.3 Ambient occlusion rays. 14

4.1 Variables in the Ambient Light Illumination Model 16

5.1 Ambient occlusion found using the vertices. 23

5.2 Ambient occlusion using textures. 25

6.1 UML class diagram. 34

7.1 Cluster and the comparing plane. 42
7.2 Cluster mapped to 2D. 43
7.3 Regular clusters and overlapping clusters. 44
7.4 Cluster patches. 45
7.5 Different number of rays. 46
7.6 Distance effect for rays. 47
7.7 Cosine distribution for rays. 47
7.8 Distance attenuation. 49
7.9 Texture blending using normal vectors. 51
7.10 Packed texture. 52
7.11 The object in the packed texture. 53

8.1 Scene 1 - Cornell Box. 56
8.2 Scene 2 - Comparison. 57
8.3 Scene 3 - Texture Blending. 58
8.4 Scene 4 - Utah Teapot. 58

9.1 Clustering problem. 60
9.2 The teapot problem. 61

10.1 Ambient occlusion comparison. 64

12.1 Cornell box scene. 72

B.1 Scene 1. 80

B.2 Scene 1. 81
B.3 Scene 1. 82
B.4 Scene 1. 83
B.5 Scene 2. 84
B.6 Scene 2. 85
B.7 Scene 2. 86
B.8 Scene 2. 87
B.9 Scene 2. 88
B.10 Scene 2. 89
B.11 Scene 3. 90
B.12 Scene 3. 90
B.13 Scene 3. 91
B.14 Scene 3. 91
B.15 Scene 4. 92
B.16 Scene 4. 92
B.17 Scene 4. 93

Chapter 1

Introduction

1.1 General Thoughts

There are many reasons why we want to model the world around us. There can be educational purposes, recreational or simply curiosity. By creating models we present the possibility of exploring objects that would be beyond our reach in real life. For example we can model molecules and simulate their behavior, and thereby explore something that would be hard to do otherwise. There is also the possibility of modeling a fictional world that has only the restraints of the imagination of its creator.

If we want to simulate the real world we have to consider physics and try to incorporate them in our model. This can be the physics of how light transports and reflects or how objects interact with each other. It could also be a global effect like the earths gravity pull. The possibilities are endless. It would be impossible to simulate exactly the real life physics in to a virtual world, the computer power needed for that would be enormous. Instead it is common to simulate physics by "cheating" and trying to consider only things that affect the viewer and not consider anything that the viewer can't see anyway. Another way of trying to simulate the real world is by simplifying the physics and thereby simulate something that looks realistic to a viewer but does in fact not obey the rules of physics.

Shadows are something that are everywhere around us, they are so common that we usually don't think about them, they simply are there. If we draw a scene that has some lights in it but we don't draw the shadows that would be cast by the light, the observer immediately identifies that there is something wrong. The image would look unrealistic and it would be hard to identify the objects in the model, their appearance and placement. This can bee seen on figure 1.1 where a man is hanging in a rope above a surface.

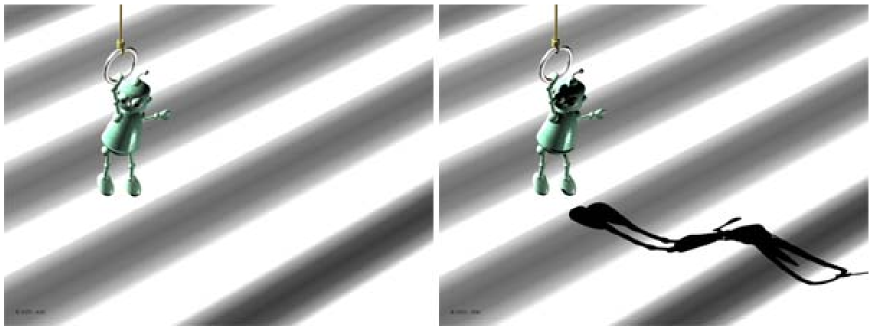

Figure 1.1: The importance of shadows. On the left it is hard to identify the location of the man and what the surface looks like. On the right we see that the man is hanging slightly above the surface and the surface is rippled.
(Image source: `http://artis.inrialpes.fr/Publications/2003/HLHS03a/`)

A special property of the things around us is the fact that they cast shadows on themselves and on objects close to them. If you are in an area with no special light sources, this effect can be seen. Figure 1.2 illustrates this, where we have a computer generated image of a living room. Notice the accumulated shadows in the corners and under or around objects. In computer graphics this effect is called ambient occlusion and this effect is the main concept of the paper. The name ambient occlusion refers to the ambient light that was first presented in the Phong illumination model[23] and occlusion which is the fact that objects can occlude or be occluded by other objects. The ambient term introduced by Phong is a constant illumination value that is applied to all areas in a scene. When the ambient value is used, it can make images look dull and that is the reason why we have ambient occlusion. Its purpose is to generate ambient values for areas in a scene based on how much they are shadowed by the environment.

Ambient occlusion can be simulated by considering the surrounding environment at each point in a model and thereby we are simulating real-life characteristics.

Ambient occlusion is a kind of global illumination and also a soft shadow effect. Therefore some discussion on these topics is needed.

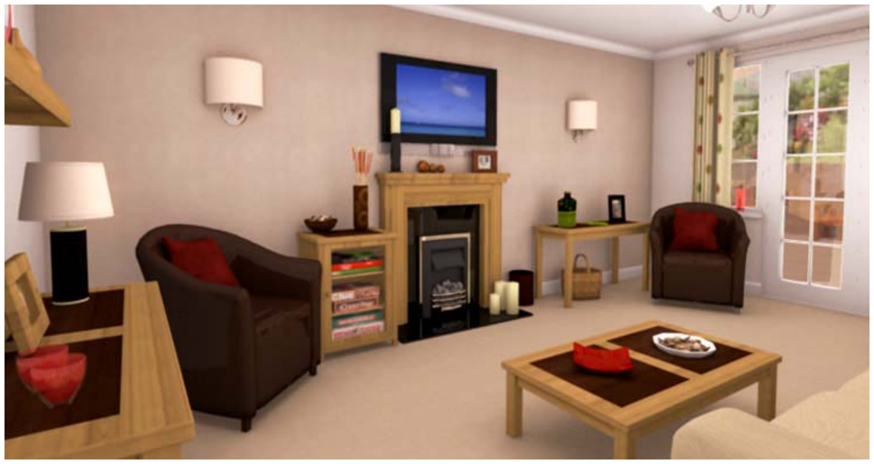

Figure 1.2: Computer generated image of a living room. The only illumination applied to this scene is ambient occlusion. The scene looks realistic even though it has no light sources.
(Image source: http://www.icreate3d.com/services/lounge-visualisation-large.jpg)

1.2 Shadow Effects

Shadows are an important aspect of graphical scenes. They help us visualize the geometry of objects, their position and size. There are two kinds of shadows, which are hard shadows and soft shadows. Hard shadows appear when there is a single point light source and they can be thought of as having two states. Either a point is in shadow or it is not. This can give interesting results but isn't a very realistic approach. Soft shadows, on the other hand, are created when light comes from an area or multiple light sources. Then points can be in full shadow, when not seeing the light source, or they can be partially shadowed when seeing a part of the light source. This creates a soft shadow effect and it is this that we are used to from real life. Figure 1.3 illustrates the difference between hard shadows and soft shadows.

Soft shadows are especially interesting since they add a realistic view of a scene. Hasenfratz et al.[16] offer a detailed description of shadow effects and real-time soft shadow algorithms. A more general survey of shadow algorithms is presented by Woo et al.[27]. Here many types of algorithms are examined and discussed which aids users in taking an informed decision that suits for a given task.

Two popular real-time shadowing algorithms are Shadow Maps introduced by

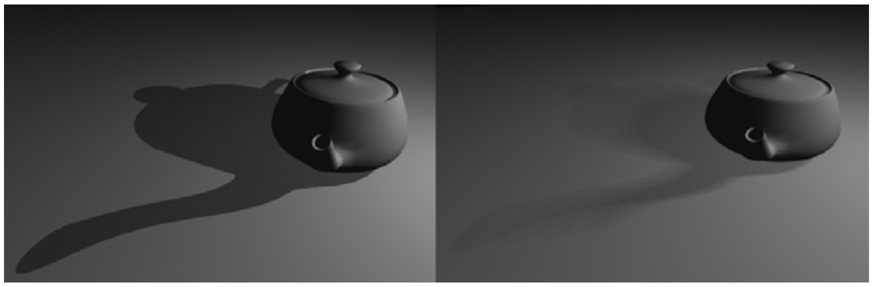

Figure 1.3: On the left we see hard shadows with one light source. On the right we see soft shadows with multiple light sources. (Image created with Softimage|XSI®)

Lance Williams in 1978 [26] and Shadow Volumes introduced by Frank Crow in 1977 [12]. Shadow mapping can be very fast but can give unrealistic results, while shadow volumes give more accurate results but can be slower than shadow mapping. These two methods have been combined by Chan et al.[8] where the benefits of both are used such that shadows maps are used where accuracy is not important and shadow volumes where it is important. This is done by identifying the pixels that will have a more visual effect on the viewer than others.

As we have seen, ambient occlusion is the accumulation of shadows at areas that are blocked by the environment. Therefore we can say that ambient occlusion is a soft shadow effect.

1.3 Global Illumination

Global Illumination models illuminate a scene, by calculating how much light or shadow should be at any given point. They are called global illumination algorithms because they do not only consider the light coming directly from light sources, but also any light that is reflected from other objects in a scene. The models can vary in complexity, going from photorealistic images to a more dynamic approach, which is more suited for where ever human interactions are required. Examples of global illumination algorithms are Ray-tracing[25], Radiosity[15] and Photon Mapping[18] which are all widely used.

Ray-tracing shoots rays from the viewer through each pixel that should be rendered. Each ray will then possibly hit some objects, and if it does the color value of the pixel will be updated. The ray can then be reflected from the object and to other objects, thus contributing to the color of the pixel from all the objects it has bounced off.

Radiosity is based on splitting the scene into patches and then a form factor is found for each pair of patches, indicating how much the patches are visible to one another. The form factors are then used in rendering equations that lead to how much each patch will be lit and then we have the whole scene illuminated.

In Photon Mapping, photons are sent out into the scene from a light source. When a photon intersects the scene, the point of intersection is stored along with the photons directions and energy. This information is stored in a photon map. The photon can then be reflected back into the scene. This is usually a preprocess step and then at rendering time, the photon map can be used to modify the illumination at each point in the scene when using for example ray-tracing.

We can think of ambient occlusion as a simple kind of global illumination algorithm, since it considers the surrounding environment but does not consider any light sources. Remember that typically, global illumination models consider all light sources and also light bouncing from other surfaces. Ambient occlusion is a relatively new method and has been gaining a lot of favor in the gaming and movie industry and is now being used extensively.

1.4 Ambient Occlusion

It is best to describe what ambient occlusion is by imagining a real-life circumstances. A good example is the shadows that appear in corners of a room. It is a shadow that objects cast on itself or on objects that are close to them, and this effect is the main concept in the report. Figure 1.4 shows a complex computer generated molecule with ambient occlusion shadows as the only illumination applied to it. Notice that the depth of the image is clear, we instantly identify the structure of the object.

Details about ambient occlusion can be found in chapter 3 where general thoughts about why and when to use it and how it is implemented are discussed. The origins of ambient occlusion is discussed in chapter 4 along with a discussion on how it has evolved and some advanced ambient occlusion implementations. This finally leads to a discussion of the solution for ambient occlusion presented in this paper which can be found in chapter 5.

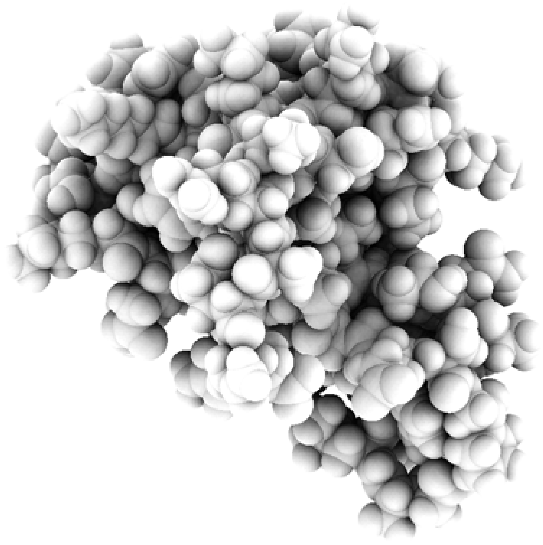

Figure 1.4: Ambient occlusion in a large molecule model. (Image source: http://qutemol.sourceforge.net/sidetoside/)

1.5 Contributions

Ambient occlusion is evaluated, what it is and how is it generally implemented. Existing ambient occlusion implementations are evaluated which leads to the approach introduced in this paper.

First ambient occlusion is found for each vertex in an object and the values associated with each vertex so they can be displayed when the object is rendered.

This idea is expanded such that textures are applied on an object. The polygons of the object are clustered together and a texture is applied to each cluster. Ambient values are now found for each part of the texture. The texture stores the ambient values and at render time, each texture is displayed on the object and we get an overall ambient occlusion.

Next step is to make the textures overlap each other. This is done by having the polygon clusters overlap, meaning that one polygon can belong to more than one cluster. Now the textures are overlapping and we are therefore finding ambient values more than once for some locations on an object.

This leads to us going to blend between the ambient values in an effort to get a smooth looking ambient occlusion. The blending will be done by using the values of the normal vectors as to how much each ambient value will contribute to the final color for each texture. Overlapping textures and blending between them using the normal vectors has not been implemented before, to my knowledge. Details about how this is done is discussed in details in chapter 7 - Implementation.

The main contribution is to create textures that contain ambient values, make them overlap each other and blend between them using the normal vectors at each point as the blending factor.

We have many textures for complex objects and therefore we will create a texture atlas from all the cluster textures, to lower texture memory needed. A texture atlas is one texture that contains many small independent textures.

1.6 Thesis Overview

In chapter 2 there is a discussion about why we would want to implement ambient occlusion, along with the goal that we want to achieve.

Chapters 3, 4 and 5 cover details about ambient occlusion in general, the predecessor of ambient occlusion, existing implementations and the proposed solution presented in this paper.

Chapters 6 and 7 cover the design and implementation details.

In chapters 8 and 9 the testing of the algorithm is discussed which is followed by results discussion.

The general idea of ambient occlusion and the path that was taken in this report is discussed in chapter 10.

In chapter 11 there is a talk about extensions and improvements of the implementation.

Finally in chapter 12 we conclude the thesis.

Chapter 2

Motivation and Goal

2.1 Motivation

Generating visually pleasing graphical images can be a difficult task. We need to consider many factors to gain the result that is needed, often using a complex global illumination model to achieve this. This can be a time consuming task.

Objects and scenes need to look realistic, at least that much it will let the observer feel like it possesses real-life characteristics. This can be achieved in many ways e.g. by passing objects through an illumination model algorithm which calculates light and shadows for any given point, taking into consideration, existing lights and other things that affect the scene.

When complex objects are in equally distributed light, such as regular daylight, they will cast shadows on parts of themselves. Some parts will be less visible to the surrounding environment and will therefore not get as much illumination as others, thus being in more shadow. As mentioned earlier this effect is called ambient occlusion, and can be seen in figures 1.2 and 1.4.

If we have a static object, an object with no moving internal parts, then it is well desirable to think of these shadows as constant. Meaning that no matter the surrounding objects or lights, these shadows will always be the same. Of course the surrounding light will have an effect, but these shadows are still there.

The motivation would be to create a simple to use algorithm that finds ambient occlusion in objects and stores it in a convenient way. Then the ambient occlusion values can be accessed fast and be used again and again. This is thought of as a preprocessing step, meaning that the algorithm should be used on objects, the output stored and used later for rendering. Possibly in real-time rendering.

2.2 Goal

The main objective will be to generate a natural looking illumination. Mainly the shadow effect, called ambient occlusion, which are the shadows that accumulate on locations on objects that are occluded by the surrounding geometry. There will be a discussion about how this has been implemented before which will lead to the method introduced in this report.

Chapter 3

Ambient Occlusion in Practice

In order to implement ambient occlusion, we first need to discuss what it is, in what circumstances we benefit from using it, and how it is generally implemented.

3.1 What is it

One special property of the things in the environment around us is the fact that they cast shadows on themselves or other things close to them. This property is best described by imagining the shadows that accumulates in corners of rooms or the shadow on the ground beneath an object such as a car. When objects cast shadows on themselves it is called self-occlusion but when casting shadows on the surrounding environment it is called contact shadows. Contact shadows are a positive side-effect of ambient occlusion, since generally it is designed to handle only self-occlusion. Self-occlusion and contact shadows are illustrated in figure 3.1.

Ambient occlusion is the shadows that accumulates on places of objects, which are not fully visible to the environment. Figures 1.2 and 1.4 in chapter 1 both catch the visual effects of ambient occlusion.

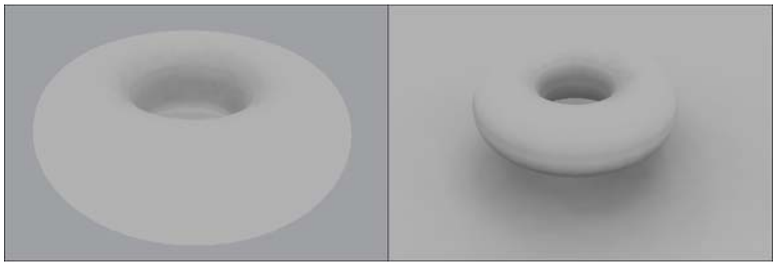

Figure 3.1: On the left we see self-occlusion where a torus occludes its inside. On the right we see contact shadow. The torus is casting shadow on the plane beneath.

3.2 When to use it

The main reason for using ambient occlusion is to achieve visually pleasing soft shadows, which make objects look real, without the effort of a more complex global illumination model. Since ambient occlusion does not consider any light sources but still can generate realistic images, it can be used early in development process to aid in visualizing a scene. Also developers can use less lights if ambient occlusion has been applied which would save time in the development process. It can be a tedious and time consuming task to place lights in good locations for getting realistically lit scenes.

Ambient occlusion is view-independent, meaning that calculations are made on all parts of an object and then they can be used even though the object is moved around and rotated. In other words we only have to calculate the occlusion values once for each object and then use them again and again, since the values will not change even though some global lighting effects change. This fact also allows the ambient occlusion values to be shared amongst many instances of the same object. It is popular to create texture maps that holds the ambient occlusion values. The texture maps can then be shared amongst multiple instances of an object.

Contact shadows are a positive side effect of ambient occlusion. If we have a static scene with many objects and it is known that some of the objects will never move, we can apply ambient occlusion on that objects together. This would give us shadows between objects that are close to one another. This can for example be applied to a static scene in a video game. Then ambient occlusion is applied to the whole scene and we get pleasing soft shadows where objects in the scene are close to one another. Right side of figure 3.1 illustrates contact shadow.

One property of ambient occlusion is that it can be used to simulate effects, like rust or dirt that would accumulate on an object. We tweak some settings in the algorithm such that we could shoot few random rays and perhaps apply a color to our shadow such that it will look like dirt that accumulates in a corner of a room. Figure 3.2 shows a gargoyle that looks worn and weathered after ambient occlusion has been applied to it.

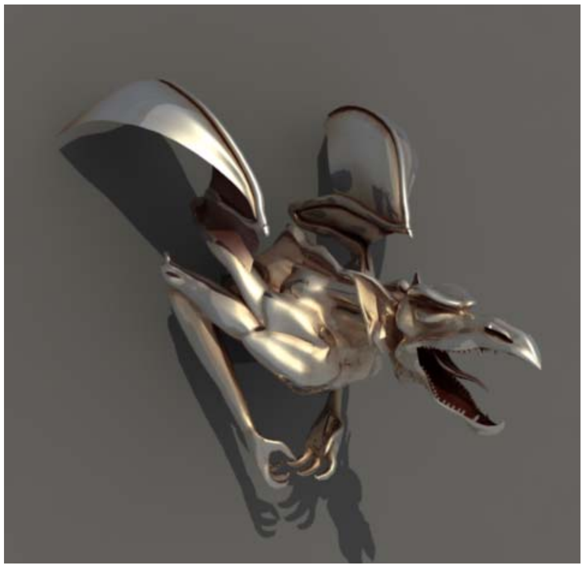

Figure 3.2: Ambient occlusion has been applied to the gargoyle model to get a worn effect.
(Image source: http://vray.info/features/vray1.5_preview/gargoyle_worn.png)

In general it can be a good choice to apply ambient occlusion to objects and scenes. The effect of it can greatly enhance images without to much effort, especially given the fact that no light sources are needed and that it is view-independent.

3.3 How is it implemented

The basic approach for calculating the ambient occlusion value at each point is with the help of ray-tracing. Rays are traced inside a hemisphere around each points normal vector and the amount of occlusion will be a value depending on how many of the rays hit other surfaces in the scene. Figure 3.3 illustrates this.

These values are pre-computed and stored for each point for later reference. Here we have the possibility of choosing how many rays are cast for each point, the more we use the better looking ambient occlusion we would get. Also distance can be used, such that if a ray hits but it is far away then it would not count as much compared to if it were closer. Last we could find the angle that is between the normal vector and a ray, and the wider it is the less that ambient occlusion value should count.

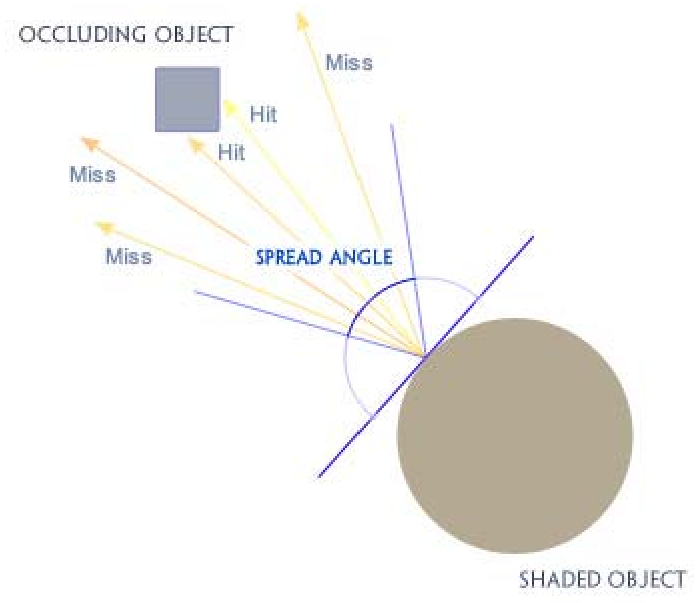

Figure 3.3: Rays are shot out from a point and a ratio is found indicating how many rays hit the scene. The ratio represents the ambient occlusion value for a given point. (Image source: http://www.christopher-thomas.net)

Chapter 4

Previous Work

This chapter covers the predecessor of ambient occlusion, going from the first model based on obscurances The model is refined and leads to the popular ambient occlusion that is now widely used in the gaming and movie industries. Last there is a discussion about advanced implementations.

4.1 Ambient Light Illumination Model

The predecessor of the ambient occlusion used in this paper is the Ambient Light Illumination Model introduced by Zhukov et al.[28]. The purpose of the model is to account for the ambient light, presented in the Phong reflection model[23], in a more accurate way.

The classic ambient term[1] introduced by Phong, illuminates all areas of a scene, whether it would actually have some “daylight” reaching it or not. The Phong reflection model is a local illumination model and does not count for second-order reflection in contrast with Ray-tracing[25] or Radiosity[15]. The classic ambient term has been extended by Castro F. et al.[6], where the polygons in a

[1]See Advanced Animation and Rendering Techniques[24], page 42, for details of the Phong reflection model.

scene are classified into a small number of classes with respect to their normal vectors. Each class gets a different ambient value and then polygons will get the ambient value from the class that they belong to. The method introduced offers a considerably better looking images with a relatively small increase in computation time compared to the Phong reflection model.

The idea of the Ambient Light Illumination Model lies in computing the obscurance of a given point. Obscurance is a geometric property that indicates how much a point in a scene is open. The model is view independent and is based on subdividing the environment into patches similar to radiosity. Obscurance for a given patch is then the part of the hemisphere that is obscured by the neighboring patches. This gives us visually pleasing soft shadows in corners of objects or where objects are close to one another. A big advantage of the model is that scenes look realistic without any light sources at all.

The definitions of the model are as follows: P is a surface point in the scene, and ω is a direction in the normal hemisphere Ω with center P, aligned with the surface normal at P and lying in the outer part of the surface. This is described on figure 4.1.

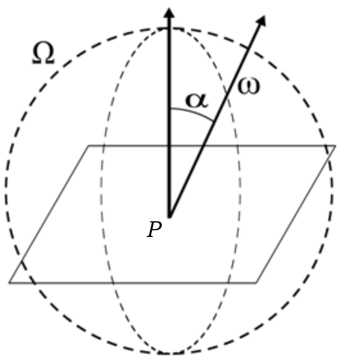

Figure 4.1: The variables introduced in the Ambient Light Illumination Model.

A function $L(P, \omega)$ is defined as:

$$L(P, \omega) = \begin{cases} \text{distance between } P \text{ and the first intersection point of the ray } P\omega \text{ with the scene} \\ +\infty \text{ if the ray } P\omega \text{ does not intersect the scene.} \end{cases} \tag{4.1}$$

Obscurance at point P is then defined as follows:

$$W(P) = \frac{1}{\pi} \int_{\omega \in \Omega} \rho(L(P, \omega)) \cos \alpha d\omega \tag{4.2}$$

Where:

- $\rho(L(P,\omega))$ is an empirical mapping function that maps the distance $L(P,\omega)$ to the first obscuring patch in a given direction to the energy coming from this direction to patch P. The function takes values between 0 and 1.
- α is the angle between the direction ω and the normal at point P.

For any surface point P, $W(P)$ will always take values between 0 and 1. Obscurance value 1 means that the patch is fully open, thus it had no intersection on the visible hemisphere and 0 means fully closed.

4.2 The Model Refined

The Ambient Light Illumination Model has been refined and simplified over the years by the gaming and movie industries and is now commonly called Ambient Occlusion.

In the ambient light illumination model, obscurance is defined as the percentage of ambient light that should reach each point P. Recent implementations[4, 9, 19, 20] reverse the meaning of this and define ambient occlusion to be the percentage of ambient light that is blocked by the surrounding environment of point P.

Ambient occlusion is then defined as:

$$A(P) = \frac{1}{\pi}\int_{\omega\in\Omega} V(P,\omega)\cos\alpha d\omega \tag{4.3}$$

Where $V(P,\omega)$ is the visibility function that has value 0 when no geometry is visible in direction ω and 1 otherwise. Note that this is opposite of the obscurance formula. The biggest difference is that the distance mapping function is not used in particular. We only get the value 0 or 1 from $V(P,\omega)$ for any ω.

There is in fact no particular difference between the words obscurance and occlusion. Objects can be obscured from light, thus being in shadow. Objects can be occluded by other objects and then being in shadow. The ambient light illumination model only talks about obscurances and never occlusion. Somewhere along the way the word occlusion gained popularity and ambient occlusion became well known.

There are many recent implementations that use either the ambient light illumination model or the simplified ambient occlusion. Many times there are some enhancements introduced, where often the goal is a real-time ambient occlusion solution.

4.3 Advanced Ambient Occlusion

In [19] the suggested solution is to approximate the occluder by a spherical cap when finding the ambient occlusion on the receiving object. A field is pre-computed around each object which represents the occlusion caused by that object on the surrounding environment. Then at run-time, the average direction of occlusion [2], along with the distance, is retrieved and evaluated to find ambient occlusion on the receiving object.

Similarly in [21] the average occluded direction is used. Here a simple method for storing ambient occlusion is presented, which is easy to implement and uses little hardware resources. A grid is constructed around each object. Then for each grid element, ambient occlusion values that the object would cast in the specific location, can be pre-calculated and stored for later reference. The benefits are faster run-time computations and shorter precomputation times which makes it suitable for real-time rendering.

In chapter 14 from NVIDIA's GPU Gems[4] a dynamic approach for finding ambient occlusion is suggested. Each vertex in an object is converted to a surface element, which means that a disk is created at each vertex. A disk is defined by its position, normal and the area it covers. Then when finding ambient occlusion, an accessibility value is found at each element based on angles and distances between elements.

The Ambient Light Illumination Model is taken to another level in [22]. Here an important feature in Radiosity[15] is added to the model, which is color bleeding. A technique is presented which combines color bleeding with obscurances with no added computational cost. An important feature is that depth peeling[13] is used, which extracts layers from the scene and for each pair of consecutive layers, the obscurance is computed between them. This allows for real-time updates of moving objects, using depth peeling and ray-casting.

The method introduced in [17] simulates a global illumination solution by using the ambient light illumination model. It estimates ambient light more accurately than the Phong reflection model, without the expense of Radiosity[15]. The

[2] The average direction of occlusion is sometimes called the bent normal.

illumination computations are stored in obscurance map textures, which are used together with the base textures in the scene. By storing the occlusion values in textures, fine shading details and faster rendering can be achieved. This model generates patches, similar to radiosity, by first assigning polygons to clusters according to a certain criteria and then the clusters are subdivided into patches. Then, similar to methods described earlier, the distance and direction is used to find the incoming ambient light at each point, using the previously generated patches.

Industrial Light and Magic have developed a lighting technique which includes what they call Reflection Occlusion and Ambient Environments[20]. Both techniques use a ray-traced occlusion pass that is independent of the final lighting. The latter, Ambient Environment, consists of two things which are Ambient Environment Lights and Ambient Occlusion. The purpose of ambient environments is to eliminate the need of using a lot of fill lights. Ambient occlusion is an important element in the creation of realistic ambient environment. There is an ambient occlusion pass and the results are baked into an ambient occlusion map for later reference.

CHAPTER 5

Occlusion Solution

As has been discussed, the goal is to create a natural looking overall illumination effect, ambient occlusion to be precise. Following is the flow of how the goal is achieved.

5.1 General Approach

We will calculate ambient occlusion with the use if ray-tracing or specifically, ray casting. This means that for a given point on a surface, rays will be cast in random directions relative to that points normal vector. We keep track of how many rays intersect the scene and find the ratio with the total number of rays that were shot. This would give us a good approximation of how much each point is obscured from the rest of the scene. This can be seen on figure 3.3 on page 14. By doing it like this we only need two know two things for any given point of a surface, which is the location of the point and the normal vector of the point. Details of how this is implemented is discussed in section 7.4.

5.1.1 Alternatives

We could use the extended ambient term[6] for finding ambient values. This is not an ambient occlusion approach but still a possibility for obtaining decent ambient values on an object. When using the extended ambient term, the triangles in the mesh would be classified into a small number of classes according to their normal vectors. Each class will have a different ambient value that is associated with the triangles in the class. A triangle will then get the ambient value that his class has and the result will be a better result than only using one constant ambient value for the whole scene like when using the ambient term in the Phong reflection model[23]. This is a simple approach and is just a small enhancement from the constant ambient value in the Phong model. We want to get more detailed ambient values.

We have the possibility to go all the way and apply e.g. radiosity[15] to our object. Then we would get a very realistic illumination including the ambient occlusion effect. Radiosity is a computationally expensive algorithm and is therefore avoided here. We are aiming at a simple ambient occlusion solution but not an overall global illumination that considers light sources and reflections.

We will use the general approach which is ray casting. Now we need to decide how to apply ray casting on an object for calculating ambient values.

5.2 Using Vertices

We state that we want to find ambient occlusion for a mesh. A mesh is a way to describe how a model looks like. It contains at least some vertices and normals along with information about how the vertices are structured so that they can form the object. Now we need to identify what approach we can use to find the ambient occlusion that we want. We start by considering using the vertices directly, since then we have the values needed, which are the vertex locations and the normal vector for each vertex. We traverse the vertices in the object and find how much each vertex is obscured from the rest of the scene. Rays are cast out from each vertex and we find a ratio between how many rays hit the scene and the total number of rays, which will be our ambient value. Each ambient value is then associated with the corresponding vertex and the object can be shaded with ambient occlusion.

By using the vertices we introduce a problem. Imagine a complex object that has some parts that are highly tesselated for details but also has areas that are defined with very few vertices. In this case we have high calculation time on some parts and very little calculation on other parts. Ambient occlusion would in many cases be very detailed where it is not necessary and not detailed enough where it should in fact be more detailed. In other words, we restrict us to much when using the vertices as points for finding the ambient occlusion, since they are defined in a way we do not know about in forehand and have little control over. This is best described in figure 5.1. There it can be seen that the sphere is defined with many vertices but the floor beneath has only vertices in the corners. This looks unrealistic since the floor should have some shadows cast on it by the sphere. This leads to the sphere getting decent ambient occlusion but the rest does not.

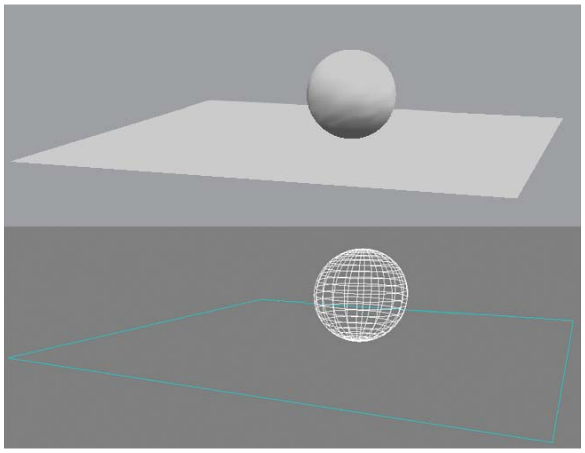

Figure 5.1: Here the ambient occlusion has been found for each vertex. The sphere has many vertices and therefore the shadows look fine. The ground beneath has only vertices defined in the corners and therefore does not get any shadows. This makes the image look unrealistic.

Possible solutions:

- We could have the restriction that the imported model should be tesselated evenly, meaning that there should be similar distance between every vertex in the model. Then applying ambient occlusion on vertices should look good. Modeling tools, for example Softimage|XSI®, have the possibility of subdividing polygons and edges which allows the modeler to create an evenly tesselated object.
- We could apply our own polygon subdivision algorithm on the object. The algorithm would be designed to add vertices and edges such that it evens out the distance between vertices.

- Another possibility is that we apply a texture manually to the model. Then we calculate ambient values for relevant parts of the texture by casting rays and store the values in the texture for display. This would give us evenly distributed ambient occlusion on an object no matter the underlying triangle structure.

- It would be possible to create a solid 3D texture to store the ambient values. Then for points at the surface of the object, ambient values will be found and stored in the solid texture and displayed.

- One possibility could be to change the topology of the object by for example splitting it up in to individual pieces. Then we apply separate texture on each piece that ambient values are found for.

- We could use multiple textures. Then we cluster triangles together and each cluster will have a local texture mapped to it. This sound similar to splitting up the object but is in fact a little bit different since here we are not changing the structure of the model.

It is not desirable to have the restriction that the model should be evenly tesselated, since then the model would possibly be defined with more vertices than would be needed. The number of vertices greatly affects rendering time and the fewer they are the faster the image will be rendered. Similarly, applying our own polygon subdivision algorithm will cause the same problem.

Applying a texture manually and finding ambient values for the applied texture would be a suitable solution. The downside is that we are restricting the modeler to do more work than he would like. It is a good practice not to put to much restrictions on the user, but keep implementations as simple and automatic as possible.

Applying a 3D texture to the entire object is very inefficient and therefore not a desirable option.

Last we have the possibility of applying multiple textures on an object. One way would be to split the model into parts and treat each part independently and apply a texture on each part. Other way is to keep the object intact but still have multiple textures that are each applied on different parts of the object. The latter is more appealing since then we keep our model intact.

5.3 Using Textures

We will use multiple local textures which we assign to polygon clusters. We then need to cluster the polygons together and apply separate local textures to each cluster. This leads to us getting continuous texture mapping for each cluster. This approach is similar as before but eliminates using the vertices for finding and storing the ambient occlusion. The idea is based on an idea presented in [17] where the polygons are clustered together. We now find ambient occlusion for each texel in a texture. A texel is one part of a texture. This will lead to us finding ambient values evenly over the whole object, no matter how the underlying polygon structure is. Finally we assign texture coordinates to the vertices in the clusters.

By using multiple textures to store and display the ambient values, we introduce a new problem. We will have textures joining at cluster borders making the texture seams visible in some cases. This can give unpleasing results as can be seen on figure 5.2 where the texture seams can be seen. This problem needs to be addressed.

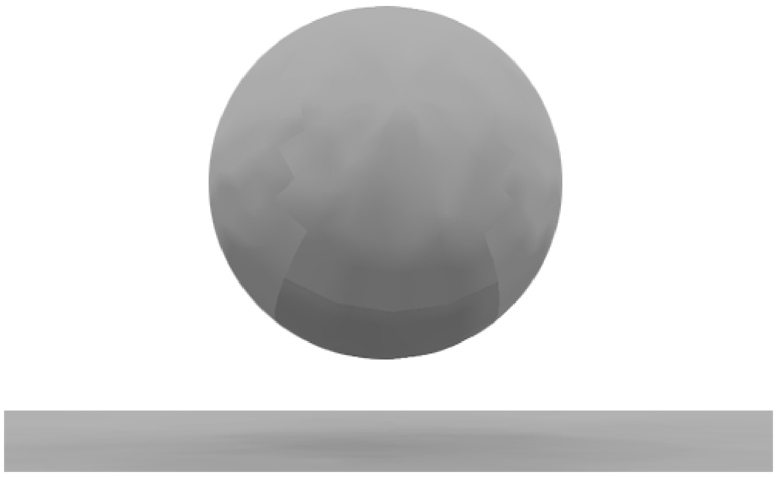
Figure 5.2: Here the ambient occlusion has been found for polygon clusters and stored in textures. The texture seams can be seen.

Possible solutions:

- We could evaluate the borders of each texture and find where a border is connected to another texture border. Then we could share the borders between two textures or blend between the ambient values at the borders, where the textures are adjacent to one another. Similar approach is suggested in [17].

- As before we could create a 3D texture. For points at the surface of the object, ambient values will be found and stored in the solid texture and displayed. There should not be any visible seams since we are working on one continuous texture in 3D and therefore the object would get a smooth overall ambient occlusion.
- Like before we could apply a texture manually on the object and find ambient values for it.
- Instead of applying a texture manually we could use another approach which is called pelting[5]. Pelting is the process of finding an optimal texture mapping over a subdivision surface[7]. The result from pelting is a continuous texture over most of the object but there will still be places where a cut is made where seams can be seen.
- We could let the textures overlap. Then we are finding ambient values more than once on some parts of an object which would lead to us wanting to blend between the values.

The problem with sharing or blending between texture borders is that we still have multiple textures that are adjacent to one another. Since the textures are not continuous, and hardware is designed to work in continues texture space, the seams could still be visible.

As before we conclude that using 3D textures is inefficient and do not consider that anymore.

By applying texture manually we still have the problem of visible texture seams. On many objects, we can't create a texture where all points have unique texture coordinates and then we can't have continuous texture over the whole object. This results in us getting places where there will be visible seams. For example there is no way to assign continuous texture coordinates on a sphere so that every point is assigned a unique pair of texture coordinates.

If we would use pelting we will almost have a continuous texture space over the whole object. A temporary cut is made in the object and there texture seams can be visible when the texture is applied. In [5], a scheme is introduced that blends smoothly over the cut, between different texture mappings on the subdivision surface. The final result is a seamless texture on an object.

The pelting approach would therefore solve our problem of having visible texture seams. The idea of making multiple textures overlap and blend between them would also work and since we already have multiple textures we will continue in that direction. Therefore our solution would be to introduce the overlapping of textures that then needs to be blended.

5.4 Blending Textures

The suggested solution for the texture seam problem is to make the textures overlap. This will lead to places on objects where the ambient occlusion values will be found more than just once. We then blend between these values to get smooth ambient values where the textures are overlapping.

We now need to evaluate how the blending should occur:

- One way to blend would be to look at the textures color values and average them such that we display the average of the values.
- We introduce using the normal vectors at each point on objects to blend between different textures. Then each value of the normal vector will control how much each of the textures that need blending, will contribute to the final color.

By taking the average of the texture color values we will have each texture contribute the same amount. This can lead to us having some textures more visible than others since then the jump between textures where they start blending, could be significant and therefore be visible.

If we would use pelting[5] to create a texture that contains the ambient values, then we could possibly skip to have to blend at all. Pelting works such that if we have an object we choose a cut place and there the object will be cut temporarily in an effort to flatten out the model and apply texture coordinates to the vertices. Then we could choose the cut place to be a location on the object that we would identify as not getting any ambient values at all. This means that the vertices around the cut should all be totally open to the environment. Then we would get a continuous texture mapping except where the cut is, but there the seam should not be visible since there are no ambient values there.

We will introduce using the normal vectors as the blending factor. We use the normal vector at each point to blend between different texture values to get a smooth transition on the surface. When we are evaluating the blending between three textures we will look at the normal vector for the point. The normal vector has three values which are the x, y and z coordinates. The normal vector needs to be normalized and then we can take advantage of the property of normalized vectors that the sum of their values squared is equal to 1 (Pythagoras Theorem). We use one normal value as the blending factor for one texture and then sum that values up to get the final ambient value at each point. This is described better in chapter 7.5 and illustrated visually on figure 7.9.

5.5 Combining Textures

One problem that arises with using many textures is that texture memory needed can be very high. If we have a complex object then we can have many clusters, and each cluster having its own texture. We are therefore creating many textures for complicated objects.

Since this is not a part of the main goal that we are concentrating on, we will create a simple texture packing algorithm. We stack each texture in a texture atlas that is large enough to contain the textures. Efficiency will be minimal, meaning that there can be large part of the texture that are not used. This should be optimized and is discussed in chapter 11 - Future Work.

Now we have discussed the approach that we take in implementing ambient occlusion, how we calculate, store and display the relevant data. next step is to design and implement the solution.

Chapter 6

Design

6.1 Import/Export

There are many ways for exchanging digital assets. Usually developers have their own format which means that exchanging the assets can be difficult when they need to be used by other applications than from the developer that created it. COLLADA[3] is an effort to eliminate this problem, by providing a schema that allows applications to freely exchange digital assets without loss of information. COLLADA stands for COLLAborative Design Activity. Here we will discuss the available data representation in COLLADA along with what we choose for this implementation. More details about COLLADA and some history can be found in appendix A.

The geometry data is imported from a COLLADA file. The name of the file is defined at runtime and the scene can then be imported and used in the ambient occlusion calculations. When the calculations are done, the new data will be exported in a new COLLADA file that the user has defined at runtime.

6.2 Data Representation

Geometry in COLLADA can be defined in many ways and can therefore be fairly complex. In general there are many forms of geometric descriptions such as B-Splines, Meshes, Bezier and Nurbs to name some. Current version of COLLADA[1] only supports splines and meshes. Here we will concentrate on using meshes for describing our geometry as that is a simple and common way to do it. Each mesh can contain one or more of each of the following elements: lines, linestrips, polygons, polylists, triangles, trifans and tristrips. To simplify our implementation we will concentrate on using triangle elements. With that assumption we restrict our COLLADA schema to have a geometry mesh, represented with simple triangles. Further we restrict us to have one object in one schema, meaning that we can only have one mesh that is defined with one triangle element. Discussion on how to expand this can be found in chapter 11.

6.3 Algorithms

There are a number of algorithms that need to identified and implemented and when combined the result will be the ambient occlusion solution.

The most obvious algorithm that needs to be implemented is the one that finds the ambient occlusion values. The algorithm will work in such a way that for a given point on an object, rays will be shot out inside a cone around the points normal vector. The ambient value that the rays find will be a value between zero and one. One meaning that the point is fully occluded by the environment and zero meaning that the point is totally open such that there is nothing occluding the point. On figure 3.3 on page 14, five rays are shot and two of them hit the surrounding environment. Then the ambient value for that point would be $\frac{2}{5}$. We will introduce two factors that will modify the ambient value further. They are distance and an angle factor. The longer the ray has traveled, the less the ambient value will be since a ray that hits the scene that has traveled a long way would not have much fact in real life. The angle factor is an angle between each ray and the normal vector of the point. The wider the angle is, the less ambient value the point will get, since the point that the ray hit is not right above the point. Details of how the algorithm is implemented can be found in section 7.4.

We need to create clusters. Each cluster will contain a number of triangles. The clusters will be able to overlap each other meaning that one triangle can belong

[1] Version 1.4.1

to one or more clusters. When creating a cluster we start by finding a triangle that has not been assigned to a cluster already. We evaluate the plane that the starting triangle lies in. We then use that plane as a comparing plane for the remaining triangles that will be added in the cluster. Clusters also have to cover a continuous space, meaning that a triangle can only be added to the cluster if it is adjacent to some other triangle in the cluster. Details of the clustering algorithm is in section 7.3.

For the clustering algorithm to work, we need to find what triangles are adjacent to each other. This means that each triangle will know what other triangles are adjacent to him. This is done by looking at all triangles and if two triangles have two of the same vertices then they are adjacent to one another. This can be a time consuming task for a large mesh. Details of how this is implemented can be found in section 7.2.

When we have created the clusters and found the ambient occlusion values we will have many textures. We will create a texture atlas which is a texture that contains all the other textures. This will be done by copying each textures values to the texture atlas. The textures size will be based on the size of all the cluster textures so that they will fit in one texture. The texture coordinates for each triangles vertex will be updated so that we have correct mapping to this newly created texture atlas. Details of how this is implemented can be found in section 7.6.

When the texture atlas has been created we need to export it as an image so that the texture can be used later. The texture values are exported in an uncompressed bitmap image file.

After we have applied the identified algorithms, we have a texture image and new texture coordinates for each vertex in the mesh. This information is exported to a COLLADA file.

The algorithms that have been identified are:

- Finding Ambient Occlusion
- Clustering Algorithm
- Finding Adjacent Triangles
- Texture Packing Algorithm
- Exporting Texture Image
- Exporting new COLLADA data

6.4 Objects

The design is object oriented. Therefore the objects needed for the implementation need to be identified.

First we will need triangle objects that store the data that defines one triangle in three dimensions. A triangle will contain three vertices and normal vectors for each vertex. That information is enough to define a triangle but we need something more. It is possible to obtain the center of the triangle which is found using the vertices. Since a triangle always lies in a plane, we will be able to access the normal vector of the triangles plane. Each triangle will have a unique integer ID and it will also contain the IDs of the cluster that it belongs to. Each vertex in a triangle can have three texture coordinates associated with them. Triangles need to know which triangles are adjacent to them and therefore they will have a list of adjacent triangles. Finally there are two variables that are used when triangle clusters are created. These are a variable indicating if the triangle has been assigned to a cluster or not, and a value indicating the state of the triangle.

We have patches that similar to the triangles, will have their center point and normal accessible. Patches will contain triangles and the triangles will be used to define each patches center and normal. If the patch contains only one triangle then we simply use thats triangle center and normal for the patch. If there are many triangles we average the centers and the normals over all the triangles in the patch. The special case of a patch containing no triangles needs to be considered. Each patch can store the ambient occlusion value that is associated with it. Finally a patch will need to know if it is actually used or not.

We then have clusters that consist of triangles and patches. Each cluster creates a set of patches. Every triangle in the cluster will then be assigned to a patch in that cluster. There are no triangles without a patch, but we can have a patch with no triangles. This can happen in two circumstances. Either the patch is not used at all, this can e.g. happen if the patch is around the edge of the cluster (See figure 7.4). This can also happen if we are so unfortunate to have no triangle mapped to the patch. Then we need to find the center and normal of the patch in another way. This is discussed in section 7.1.3.

Data importer will import the data from a COLLADA file and manipulate in a way such that it will be convenient to work with the data. The importer will locate the triangle mesh in the document and load the relevant data.

Controller will handle user input along with assigning the data to relevant locations using the algorithms that are implemented.

The objects that have been identified and will be implemented are:

- Triangles
- Patches
- Clusters
- Data Container
- Controller

6.5 Final Structure

The structure and relations between objects can bee seen in the UML diagram on figure 6.1.

The algorithms mentioned earlier will all be located in the *myMain* class except the ambient occlusion algorithm, which will be located in a *myCluster* object.

myMain is the controller. He starts by instantiating a *myDAEdata* object with the COLLADA file as input. *myDAEdata* loads the data into a database with use of a helper class called *myMesh*. What *myMesh* does is that he loads a COLLADA mesh element and extracts information from it that can then be retrieved from *myMesh*. The information needed from the COLLADA input file are the vertices, normals and faces of the triangle mesh. After the file has been loaded and the relevant data extracted from it the controller will start creating *myTriangle* objects and from the triangles he creates *myCluster* objects. Then each cluster will create a number of *myPatch* objects. Now all the objects have been created. Implementation details about the objects are discussed in chapter 7.

There are two other helper classes in the diagram which are *myRandomRays* and *myQueue*. The first class will create a certain number of random rays that are used for finding ambient occlusion. The queue class is used by the controller when the clusters are created. The random ray generator is discussed in section 7.4.1 and the queue class in section 7.3.1.

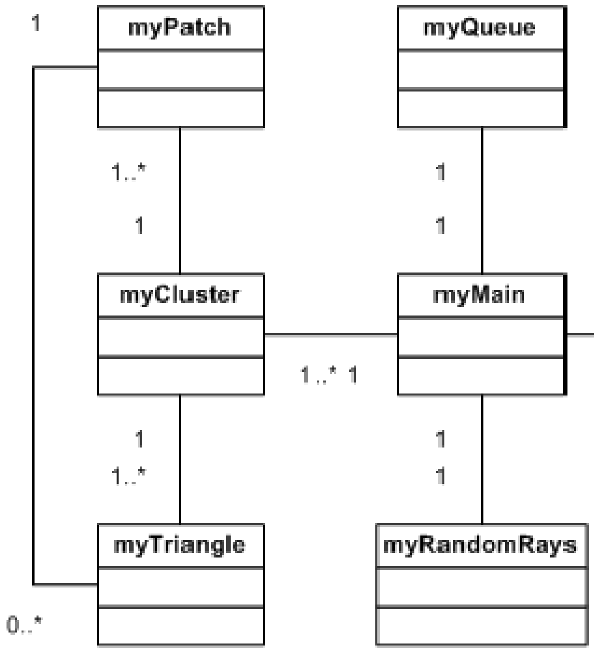

Figure 6.1: UML Diagram showing the objects in the solution and the relationship between them.

Chapter 7

Implementation

We need to implement the objects and algorithm that have been identified in the design chapter. First we discuss the flow of the data relative to the objects. Then we go into details about each object that is implemented. Following that is a dedicated section for each of the algorithms that have been identified.

7.1 Data Structure

We have interactions and connections between objects which can be seen on the UML diagram presented in section 6.5. The basic flow of the program relative to the data is as follows. After importing a model we use it to create triangle objects. The model has to be defined with simple triangles. Softimage|XSI® was used when creating models for rendering and testing. Softimage has the possibility of exporting scenes in a COLLADA document and it offers the possibility of converting all polygons to triangles. Softimage is discussed in appendix A. Each of the imported triangles will be associated with one or more clusters. Each cluster will then contain a set of triangles along with a set of patches. Each patch will be assigned a number of triangles belonging to that cluster, so that each patch will have zero or more triangles. Then ambient occlusion values are found for each patch in a cluster and the values stored as textures. After this is done we create one texture from the cluster textures, export the texture image and write the new COLLADA data back to a file. We will now discuss each object that makes up our data structure, the important variables and methods.

7.1.1 Triangle

A triangle can be thought of as the most primitive object in the data structure. Each triangle has a unique integer ID. To define a triangle we need to set its three vertices and the normal vector for each vertex.

When the vertices have been set we find the center of the triangle by averaging over the three vertices. Similarly the triangles planes normal vector is found by using the vertices.

Each triangle will belong to one, two or three clusters. It should not happen that triangle is assigned to more than three clusters. This could happen if the comparing angle when creating clusters is to low. Each triangle has a unique ID and all the IDs of the clusters that the triangle belongs to can be accessed to know what clusters it belongs to. Also the number of clusters that one triangle belongs to can be accessed.

Each triangles vertex will have texture coordinates assigned to it. One vertex will always have three texture coordinates assigned to it, irrelevant of how many clusters it belongs to. The reason for this is to simplify the implementation. This allows us to add three texture coordinates to each vertex and create a texture that contains all the ambient values. In most cases a triangle will belong to one cluster. So that when a texture coordinate is added for the first time to triangles vertices, we add those coordinates to all three texture coordinates. This will lead to us blending between the same values of a texture if the triangle only belongs to one cluster in the end. When and if the second and third texture coordinates are added, we add that coordinate to the relevant texture coordinate variables in the triangle. The three texture coordinates are stored in three variables that can be accessed globally.

There are two variables that belongs to triangles that are used when we are creating clusters from the triangles. One is a boolean variable indicating if the triangle has been assigned to a cluster or not. The other is an integer variable that can have three states that are used in the Breath-first search algorithm. The states are white, gray or black and are defined as integer values. All triangles start with the default value of white, meaning that the triangle has not been evaluated in the search algorithm. Then when the algorithm is working the state can go to grey and black. This is discussed in detail in section 7.3.1. There are two other variables that are a part of the search algorithm that are not used here but still implemented. They are the ID of the predecessor of each triangle and the distance in the search three that this triangle has with the triangle we began with.

Each triangle needs to know what other triangles are lying adjacent to him. Therefore each triangle contains a list of IDs of the triangles that are adjacent to him. This adjacency list is created by the algorithm described in section 7.2. The adjacency list is then used in the clustering algorithm described in section 7.3

7.1.2 Patch

Patch objects represent location on a surface that we want to find ambient occlusion for. Patches are created for each cluster and we then find the ambient values for each patch. We do this instead of using the vertices as was discussed in chapter 5. Each cluster creates an array of patches of size $n * m$ that will represent ambient values for that cluster. The size of n and m are chosen to be the width and height of the cluster when it is mapped to 2D and multiplied with a value that can be defined at runtime. This allows users to control how many patches will be created for each cluster, and therefore control the details of how the overall ambient occlusion will be.

Each patch will have a number of triangle objects associated with it. The triangles are added to a patch from the outside. There are no restrictions of the number of triangles that can belong to one patch. We then use the triangles to define the center and normal of each patch, by taking the average of the centers and normals of the triangles.

The patch has a boolean variable indicating if it is used or not. By default it is assumed that patch is used when it is created. After triangles have been added to a patch we can retrieve the center and normal of the patch. In some cases, patches will not have any triangles associated with it. This special case needs to be treated in the cluster that creates the patch, since the patch has now way of defining its center and normal vector. How this is done is discussed in next section, section 7.1.3. When the center and normal for a patch is found from the outside the values can be set for the patch.

There are two reasons for a patch not having triangles:

- No triangle was mapped to the patch. This can happen when the patch resolution is set higher than the number of triangles in the cluster.

- The patch is not used at all. This can happen in many cases since a cluster is usually not exactly formed as a $n * m$ square (See figure 7.4).

In the first case we find the center and normal of the patch in another way, since the patch is used but has no triangles. In the latter case the patch is not used and we set a variable in the patch indicating that the patch is not used.

Finally each patch will store the ambient occlusion value that is associated with it.

7.1.3 Cluster

Each cluster has a unique ID so it can be accessed. A cluster consists of one or more triangles and is defined by them, meaning that the triangles control in a certain sense how the cluster behaves. It is therefore possible to add triangles to each cluster.

Each cluster is created with triangles, and all the triangles are aligned inside a certain angle with one of the major axis planes. Therefore we can set the plane that each cluster was created with which is used by the cluster. Details about how the clusters are created is discussed in section 7.3.

When a cluster has been created with the triangles, we need to set some variables so that the cluster will behave as we want it to. These are

- The number of random rays to shoot out for each ambient occlusion calculation.
- The distance the rays can travel.
- The angle of the cone around the normal that the rays lie in.
- The texture size factor. This value is multiplied with the width and the height of the cluster to get the texture size.

Because we are finding ambient occlusion values for each cluster we will have access to the global object, the BSP tree that represents the whole object. This is used when finding ambient occlusion for the cluster.

The object we are working on is a three dimensional object. However each cluster is thought of as being in two dimensions. We think of it as lying in the plane that was used when the cluster was created. See figure 7.2 where a cluster is mapped to 2D. We want to create a texture for this 2-dimensional cluster. This means that we will always be using only two of the three coordinates that each triangle in the cluster is defined with. First we set variables that allow us to access the correct two of the three x, y and z variables. We then need to find the highest and lowest of each of the values. When we know the minimum and maximum values we withdraw one from the other and then we have the clusters texture dimension. That values are then multiplied with the texture size factor mentioned above to get the final dimension of the texture in a cluster. If the clusters width is n and height is m and the multiplier is t. Then the clusters width will be $(n{*}t)$ and height will be $(m{*}t)$. These values will usually have some decimal points so therefore they are converted to integer values by dropping the decimal points. These values are now this clusters texture size.

We now create a 2-dimensional array of patches where its dimensions will be the texture dimension found above. After the patches are created we start adding the clusters triangles to them. We do this with the help of a mapping function which input is coordinates in 2D. The unimportant 3D coordinate is dropped before calling the mapping function. The function then returns a coordinate to the texture for the cluster so that the triangle will be associated with the patch that it lies in.

When the patches have been created and all the triangles are associated with a patch, we can get to the most important part. This is calculating the ambient occlusion values for each patch. We loop through each patch and call a function that finds the ambient occlusion. The input is a point and a normal and it returns a value indicating the ambient occlusion value for the given patch.

The special case of a patch not having any triangles can cause trouble. This can happen if the object has few triangles and the patch resolution is high. Then the problem is that we don't have the patch center and normal defined. What we do then is that we find the center in another way using a function in cluster objects designed for finding patch center. This function will look at the location of the patch in the $n * m$ array of patches. It then finds the four corners of the patch and takes the average of them. Then we have the exact center for the patch.

When we have the center we create a ray and let its starting location be above the cluster we are working on. This can be done since we know what plane the cluster lies in. We use the center of the patch, found earlier, move up from the cluster along the planes normal vector. Then we shoot the ray in the direction of the cluster and it should hit the cluster exactly in the center we found earlier.

If it hits then we have the normal of the point we hit. We then set this as the patch center and normal for the ambient occlusion calculation. If the ray does not hit the cluster, then it is because the patch that we are looking at is not used at all. This can happen often in complex objects and happens frequently. Then we set the patch as not being used and he will not be treated anymore.

When we have found ambient occlusion for each patch in a cluster and stored it in an array of patches, the array can be accessed from outside. We also can access the height and width of the texture array.

7.1.4 Data Container

Here we read in a COLLADA file and put the data in a convenient data structure that allows us to manipulate it in a desired way. The container has a load and save functions that will load and save COLLADA files.

We load the imported mesh in a Triangle Mesh that is convenient to use. It also creates a Binary-Search Partition(BSP) tree out of the mesh. The BSP-tree is used when finding ambient occlusion. It allows for us to conveniently check if a ray hits the object.

7.1.5 Controller

The controller reads input from user and assigns it to relevant variables. This is for example the name of the input file, the name of the output file that we will save to, the number of rays to use for ambient calculations. and more. This is discussed in section 7.7.

When the input has been read, we start by loading the data in the data container. From that we start creating our Triangle objects and following that we create our Cluster objects. Each cluster creates Patch objects and then finds ambient values and we access it in the controller to create a texture atlas. This texture atlas is a texture that contains the ambient occlusion values for all the clusters. The controller then renders the object using OpenGl and Cg.

The controller also exports a bitmap image of the texture and writes new COLLADA data to a file.

7.2 Finding Adjacent Triangles

Each triangle needs to know which other triangles are adjacent to him, which is used when creating triangle clusters. This simple algorithm will loop through all triangles and compare it to all the other triangles. If they share two vertices then we have two adjacent triangles and they can be added on each others adjacent triangles list. Actually though this algorithm is simple to implement it can be very time consuming. If we have N triangles, then for each vertex we would look at $N - 1$ triangles, which makes it an algorithm of type $O(n^2)$.

The calculation is lowered slightly by using the fact that when working with triangles, each triangle will have at most three adjacent triangles, and exactly three when working on a closed mesh. This fact allows us to stop looking at triangles when we have already found all the triangles adjacent to the one we are looking at at each time. With this the calculation time is lowered slightly, since we only consider triangles that we have not found three adjacent triangles for.

7.3 Clustering Algorithm

Each cluster will be assigned a unique ID which will go from 0 to $n - 1$ where n is the number of clusters found. The clustering algorithm loops through all triangles and registers if they have been processed or not. When all triangles are processed the algorithm can successfully quit. We now have signed all the triangles to one or more clusters. The clustering algorithm uses an abbreviation of the Breadth-first search(BFS) algorithm found in the book Introduction to Algorithms[11]. The attributes needed when applying it is that each triangle has an attribute called color which can have the values white, grey or black. The algorithm is discussed later in the section.

The approach for creating clusters is that we start with a triangle that has not been assigned to a cluster. We evaluate the triangle in such a way that we compare the normal of the triangle, with the normal of one of the major axis planes. The major axis planes are three, the XY-plane, the XZ-plane and the YZ-plane. We find what plane, our starting triangle is closest to by comparing the triangles normal with the axis planes normal. The axis plane that has the smallest angle will be used as comparison for the rest of the triangles that we want to put in the current cluster.

When assigning a triangle to a cluster we do it only if the following criteria is met:

- The triangle is adjacent to some other triangle in the cluster.
- The triangles plane lies in a certain angle with the comparing axis plane.

When following this criteria we will have a cluster that contains triangles that are all adjacent to one another and are close to lying in the comparing plane. This is best illustrated in figure 7.1. The default angle used is 60° meaning that each triangle in a cluster will have an angle less than than 60° degrees with the comparing plane. The comparing angle can be changed at runtime.

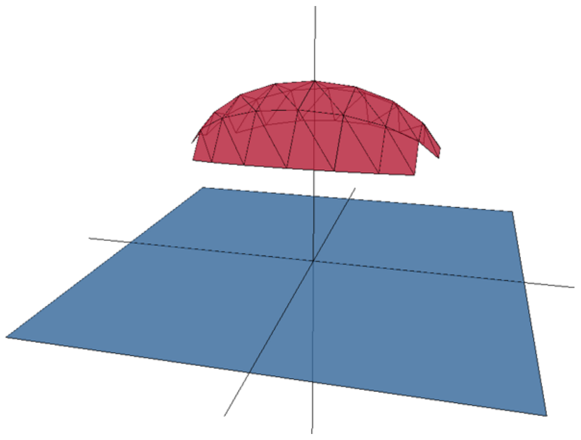

Figure 7.1: Here we see a cluster and the plane that it was compared with.

The advantage of doing this is that each cluster will now have a plane that it is aligned with. The cluster can in a certain sense be mapped straight onto the plane by simply dropping one of the relevant 3D coordinate. See figure 7.2 where one coordinate has been dropped. This helps in the next step.

We now consider how the clustering is achieved with the help of the Breath-first search algorithm.

7.3.1 Breath-first Search Algorithm

The Breath-first Search(BFS) algorithm uses a class that represents a First-in first-out(FIFO) queue system. FIFO means that the element that has been the

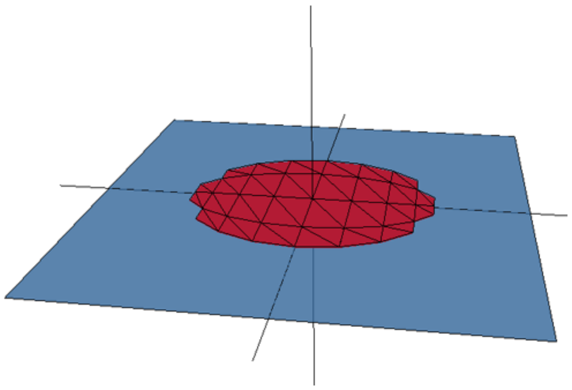

Figure 7.2: One coordinate has been dropped and the cluster is now mapped to 2D.

longest in the queue will get out first when the queue is dequeued. The class has an array of integers which represent the items in the queue. The public functions available are to add an item in the queue, get an item from the queue, get the number of elements in the queue and a boolean variable that indicates if the queue is empty or not. The queue is used in the breadth-first search algorithm when clusters are created.

The algorithm starts with a random triangle that has not been assigned to a cluster. This triangle is put in the queue. Then we loop until the queue gets empty. When the queue is empty we have successfully found all triangles that we want. We now look at an element in the queue, which in the first iteration is the first triangle. We get all triangles adjacent to this triangle and loop through them. If one of the adjacent triangles color is white we set its color as grey and put in in the queue. By this we will end up with all the triangles in the mesh. What is done here is that the algorithm is modified such that when we look at the adjacent triangles we only consider it if its angle is less than the comparing axis planes normal vector, mentioned earlier. With this we assure that the triangles are all adjacent and lie in this plane as can be seen on figure 7.1 where one cluster is shown, that was found using the BFS algorithm.

When one iteration of the algorithm is done we will have one cluster. Then we look at the remaining triangles that have not been assigned to a cluster and choose one randomly. We now loop through the algorithm again, adding all triangles that fulfill the criteria as before. Note that we look at all the triangles each time, not only the ones that have been assigned to a cluster already. The result of this will be the overlapping clusters that we want. The most important thing is that when we choose the starting triangle, then we have to choose one that has not been assigned to a cluster already. If that restriction were not, then we could get the same clusters over and over again.

The result when all triangles are processed will be overlapping clusters. See figure 7.3 where overlapping clusters are shown.

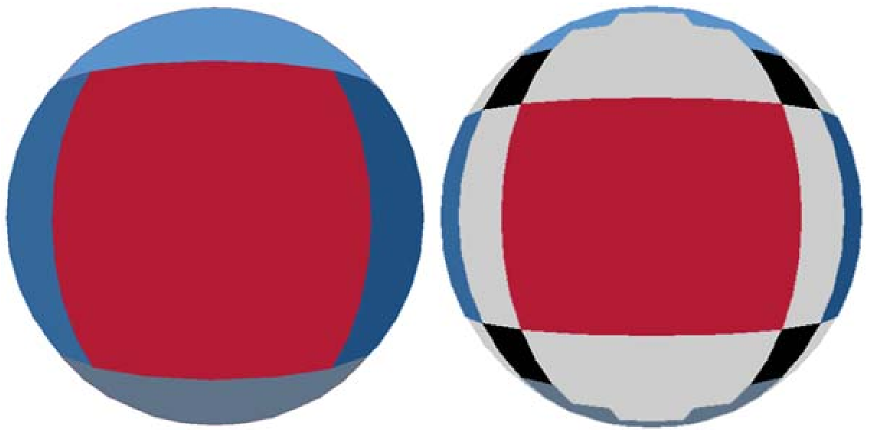

Figure 7.3: On the left side we see clusters on a sphere. On the right side the clusters are overlapping. The grey parts are where there are two clusters overlapping and the black parts are where there are three clusters overlapping.

7.4 Finding Ambient Occlusion

Now that we have created clusters of triangles it is time to calculate ambient occlusion values and associate them with the triangles. We will split each cluster into parts which will be on the form $n * m$. These parts are the patches and the ambient occlusion values will be found for each patch. In order to find the ambient values we need to know the center of each patch and its normal vector. To do this we will use the triangles that are assigned to the cluster. Each triangle will be mapped to one patch. To do this we simply drop one of the triangles center coordinate like mentioned earlier. Note that we need to drop the correct coordinate which is found based on the plane that the cluster lies in. This results in each triangle belonging to one patch and one patch containing one or more triangles. There is the special case of a patch not containing any triangles, which is treated in a special way. This has been discussed in section 7.1.3. Since usually a cluster will not be exactly mapped to a square we will have some patches around borders that are not used. This can be seen on figure 7.4 where the patches around the borders are not used. Now that we have all the triangles associated with a patch it is simple to find the center and normal for each patch. We average each triangles center and normal and then we have the information needed.

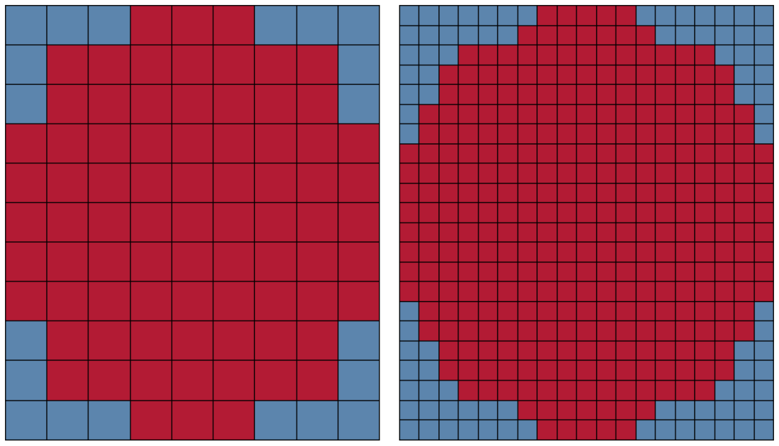
Figure 7.4: Example of cluster patches. Each square is one patch. The blue patches are not used. The number of patches will vary depending on the texture size factor. On the left side the factor is 0.5 and on the right side it is 1.0.

Now rays are cast out from each patch and the ambient values found and stored in the patches. We also create a float array of size $n * m$ and store each ambient value there.

7.4.1 Random Rays

How should the rays be chosen? There is the possibility to define a certain number of rays that are evenly distributed around a normal of a point. This would give us very even ambient occlusion but it lacks the option of letting the user define the number of rays at runtime. If we allow for us to define the number at runtime, it is possible to get a much better result for a large number of rays or even some other result if we choose to have very few rays. The latter does not sound good but can in fact be used for some cases. It is possible that we want to create noise in the image, which would result in a nice looking effect like for example, rust or dirt that should look random and noisy but not smooth. See figure 7.5 where using different number of rays is illustrated. We allow for any number of rays and we choose them randomly. A random ray generator is implemented.

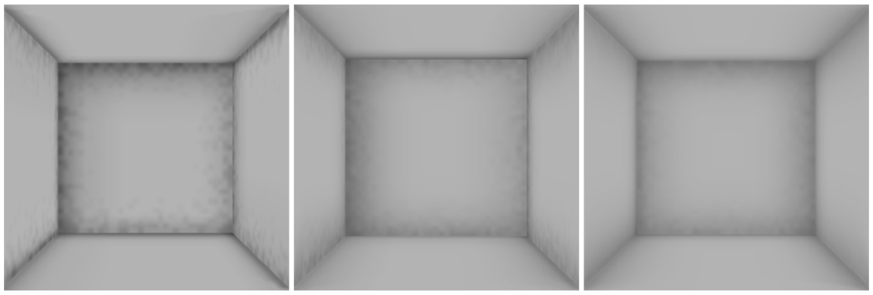

Figure 7.5: The number of rays used for each scene were 8 on the left side, 32 in the middle and 64 on the right side.

In the simplest case, each ray will either hit the scene or not, and therefore it should either have the value of zero or one. One meaning that it has hit and zero that it has not hit. This would not give us a realistic result, given the fact that a ray that has traveled a long distance or has large angle with the normal, would in real-life count very little or nothing. To make this fact count, we introduce two factors that will effect the ambient occlusion value for each ray:

- The distance that a ray has traveled.
- The angle between the ray and the normal.

7.4.2 Distance Factor

The distance factor can be defined which would be the maximum distance a ray can travel if it hits the rest of the scene. If it has traveled longer than the maximum distance allowed, it will not have any effect. If it has traveled shorter than the maximum distance then it will have a decreasing effect the longer it has traveled. This is described on the graph on Figure 7.6. The ambient occlusion value is 0 for a distance farther than maximum distance and a value between 0 and 1 for a distance between 0 and maximum distance.

7.4.3 Angle Factor

We have the possibility of restricting the angle that rays are shot in inside the hemisphere Ω. The default value is $90°$ which means that rays are shot above the plane of the patch.

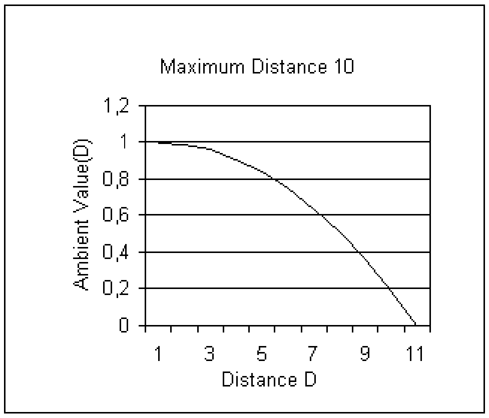

Figure 7.6: Here we can see how the ambient occlusion value decreases with distance.

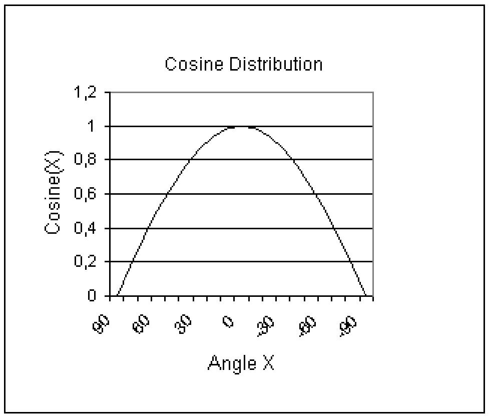

Figure 7.7: Cosine distribution for angles going from 90 degrees to -90 degrees.

We introduce a restriction on the angle that each ray has with the normal vector of each patch. This angle is known as the cone angle, since rays will be shot out inside a cone with this angle around the normal of each patch.

If we imagine two simple objects that are close to one another. If we then want to find the ambient occlusion value for a point on one of the objects, and the other object is located right above the point. Then the ambient occlusion value for a ray shot out from the point along the normal will hit the other object and that value should be high as would happen in real life. The angle between the ray and the normal of the point is low and taking the cosine of that angle would give us a value very close to 1 thus indicating that the point is highly occluded by the other object. If the other object is on the other hand located much to the side of the point, then the angle is high and the cosine value of the angle gives us a value close to 0, indicating that the point is not much occluded. This makes sense in real life situations. The cosine distribution for angles can be seen in figure 7.7 Cosine Distribution.

7.4.4 Combining it all

If we think in the terms introduced in sections 4.1 and 4.2 in relation with the original obscurance model and ambient occlusion as it is generally implemented, we get ambient occlusion as before to be:

$$A(P) = \frac{1}{\pi} \int_{\omega \in \Omega} V(P, \omega) \cos \alpha d\omega \tag{7.1}$$

Where:

- P is a surface point in the scene.
- Ω is the normal hemisphere with center P.
- ω is a direction in the normal hemisphere Ω with center P.
- α is the angle between the ray $P\omega$ and the normal at P

One difference here from the ambient occlusion formula presented in 4.2 is that the function $V(P, \omega)$ is defined as taking values between 0 and 1 based on the distance at the intersection. That is similar as the original ambient model does it. The recent approaches do not particularly introduce distance, but rather a ray just hits or not. Usually the maximum distance is defined such that a hit is not recorded if the ray has gone farther than the maximum distance. We let distance have a decreasing effect the longer a ray has gone if it intersects the scene. The function then takes values between 0 and 1 as can be seen on the graph on figure 7.6. The effect of letting distance have decreasing effect on the ambient values is illustrated on figure 7.8. The effect can be clearly seen in closed scenes when using a high value for maximum distance. This is a design decision and varies from implementation to implementation.

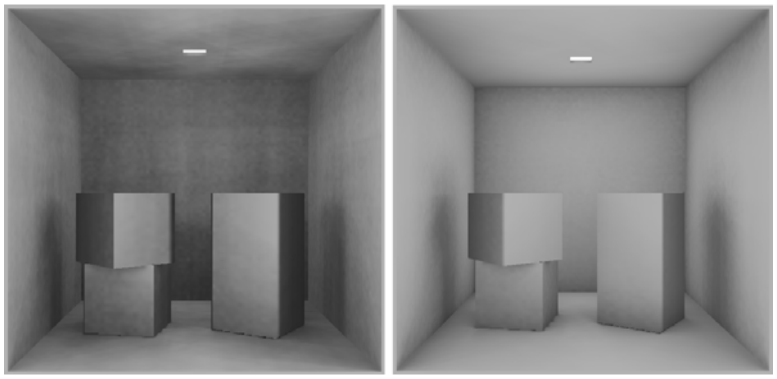

Figure 7.8: Distance attenuation. On the left side, rays can travel within the maximum distance and have an effect on the ambient value if they hit the scene. On the right side, the same applies except distance has decreasing effect the longer a ray has traveled within maximum distance. The results of this are clearly visible.

Ambient values are thought of as being percentage values and therefore we have the normalization factor $\frac{1}{\pi}$ so we get values between 0 and 1 from the formula $A(P)$.

Finally we define specific number of rays and then we change the integration to be a summation over a number of samples N:

$$A(P) = \sum_{i=1}^{N} V(P, \omega_i) \cos \alpha_i \tag{7.2}$$

Where:

- N is the number of rays shot out.
- α_i is the angle between the normal at P and ω_i.

Formula 7.2 is applied with N number of rays to all patches and the resulting ambient values stored.

7.5 Texture Blending

We need to blend between different texture values to get a smooth transition between textures. One suggestion was to take the three ambient values that each triangle will have and simply take the average over them and display that as the result. This can lead to bad transition between textures. Instead we will use the normal vectors at each point as the blending factor. This means that when we are evaluating the blending between three textures we will look at the normal vector for the point. The normal vector has three values which are the x, y and z coordinates. When the normal vector is normalized its values squared will sum up to be 1. This is convenient since then we can use the values as the blending factor. Then when moving over a surface, we will get a smooth transition since the normal values are changing slowly from triangle to triangle. This is best illustrated visually and can be seen on figure 7.9. The figure illustrates two textures mapped to a circle in 2D. This can be applied to 3D where we have three textures and three coordinates. On the figure the texture contributions are controlled by the normal vectors.

To achieve the blending, a Cg[14] fragment program was implemented. Cg is discussed in appendix A on page 77. The input to the program is a color value, three texture color values, and a normal vector. The output is a new color value that is found using the three texture colors and the normal vector at the point. The normal vectors need to be normalized and then each value is squared and the values summed up. The sum of the values will the be 1, which makes it convenient to use them as percentage value that each texture color should contribute. Each texture color is multiplied with a normal vector value and the results are summed up. The sum is multiplied with the incoming color and that is our output color. Now we have added the ambient contributions from three textures and the blending will be smooth.

7.6 Texture Packing Algorithm

The ambient occlusion values in each cluster will be stored in one texture. That texture will be of size $n * n$ where n can be 32, 64, 128 etc. The size of n will be based on the area that the ambient values in each cluster will cover. The areas from each cluster are summed up and then n is chosen to be of size big enough

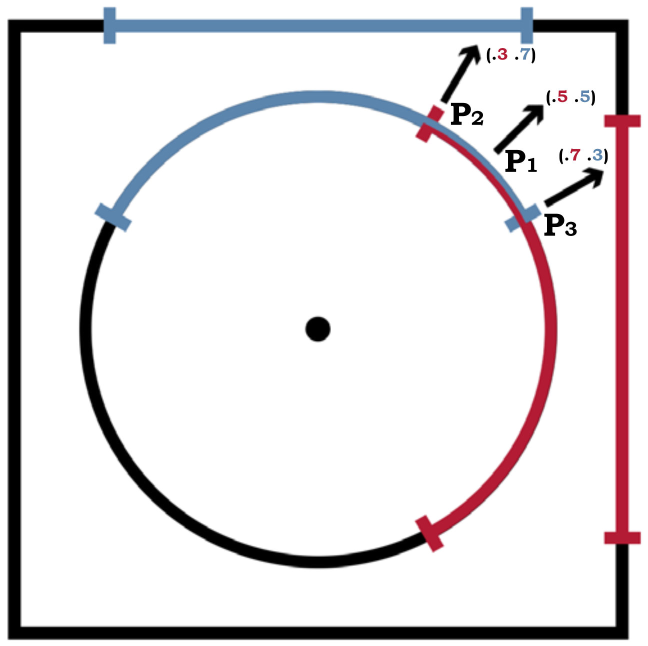

Figure 7.9: Here we have two textures, blue and red, mapped to a circle. Contribution of each texture is controlled by the values of the normal vectors at each point.

so that the texture can contain all the cluster ambient occlusion values. Then when packing, the texture with the largest y value is chosen first and put that ambient array in first. The algorithm tries to use space efficiently but could be made much more efficient. In figure 7.10 we see an example texture that was created with the algorithm. Figure 7.11 shows the object that the texture was created for.

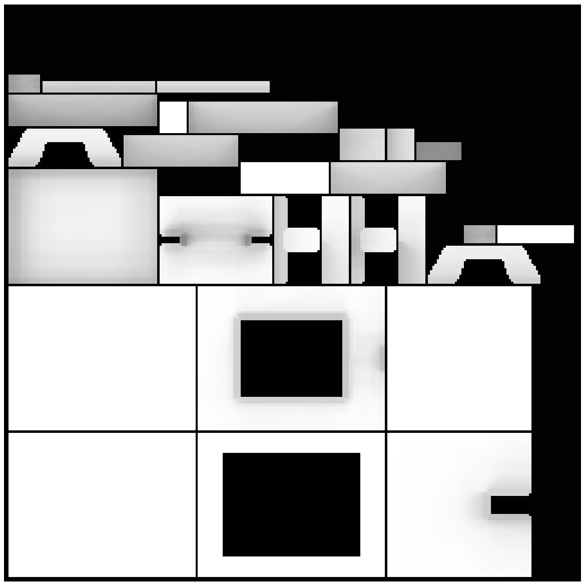

Figure 7.10: Here is a texture that was created with the texture packing algorithm.

In an effort to use texture space efficently, user can set a variable at runtime, that controls how big the texture should be. If the output shows that the packed texture could be in a smaller texture, we can set this variable lower and then the texture should be smaller if we run the whole ambient occlusion algorithm again.

7.7 User Input

Some variables need to be defined at runtime. It is not necessary but can greatly affect the ambient occlusion for a given scene. If they are not defined by the user, default values will be used.

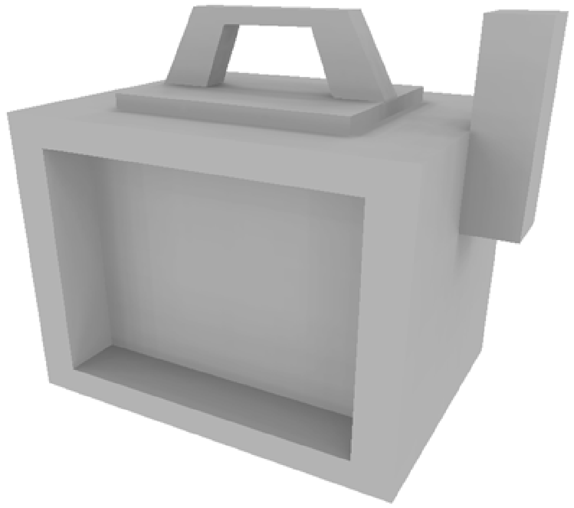

Figure 7.11: Here is the object that the packed texture on figure 7.10 was created from.

The name of the input file is the only thing that has to be set at runtime, other variables have default values that are used if they are not defined at runtime. This includes the name of the output COLLADA file and the exported texture. It is preferred that these values are set at runtime though.

The default number of random rays that are used for each ambient occlusion value is 64. This usually gives a very pleasing ambient occlusion with a small calculation time. This value can be changed to any positive value and can be experimented with. In practice, the more rays used should give better looking images, but when reaching some high number of rays the difference is so little that it is impossible to see the difference but we still have much higher computation time. The effect of choosing different number of rays can be seen on figure 7.5.

The maximum distance that a ray can travel and still have an affect on the ambient occlusion can be defined. Default value is 10. If the scene uses a very small coordinates for the x, y and z values of vertices, this number is probably to high and it should be set lower. It is best to experiment with the number and find a value that gives a pleasing result.

We can set the maximum cone angle around a normal that rays will be cast in. This value defaults to 90°, meaning that the rays are all in the hemisphere above the patch. This can be changed to be lower than 90° and changing the value can give results that are desired for a given scene.

The size of the textures that each cluster will create can be set at runtime. The texture size will be a factor of the width and height of each cluster multiplied with this value. The default value is 3. This can be set much higher if we have a small object that is highly tesselated. This variable can greatly affect the outcome of the ambient occlusion in a scene.

The default compare angle that is used when clusters are created is 60°. This value can be changed to any value between 50° and 70° degrees. It is sometimes necessary to change the default value as we can create clusters that are not correct. This problem is discussed in chapters 8 and 11.

Chapter 8

Testing

Different scenes were created that tried to catch the aspect of the ambient occlusion program created here. Four scenes were rendered that tried to cover the things that an implementation like this should be able to handle.

- **Scene 1** is a simple Cornell box. The purpose of the scene is to test the overall visual effect of our ambient occlusion implementation using a scene that is well known. Number of triangles is 84.
- **Scene 2** is a sphere hovering over a plane. The purpose of this scene is to test the effect of using different parameters. We change the number of rays, the cone angle and the distance. Number of triangles is 3.902.
- **Scene 3** is an object that tries to catch the effect of overlapping textures and the blending between them. It has curved surfaces and therefore overlapping of textures. Number of triangles is 34.548.
- **Scene 4** is a Utah teapot. Number of triangles is 24.138.

If we talk generally about the scenes. Scene 1 is very simple and therefore has low calculation time since it only has 84 triangles. The scene catches the overall effect of the ambient occlusion solution. Figure 8.1 is a rendered image of the scene.

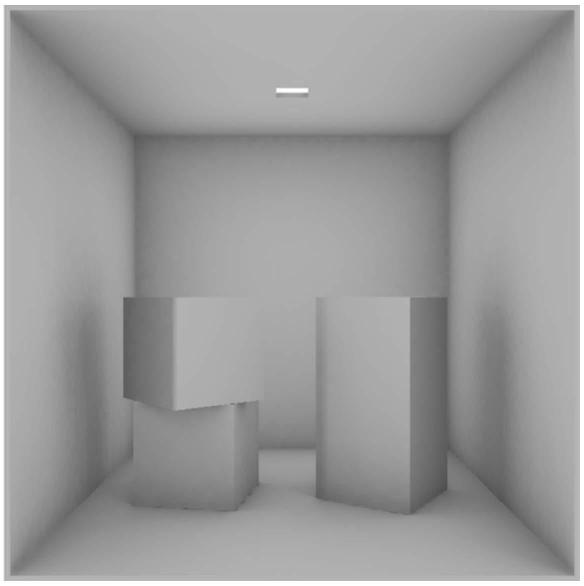

Figure 8.1: Scene 1 - Cornell Box.

In scene 2 we do a comparison using different number of rays, maximum distance and cone angle. Tweaking these variables can give very different results and can be chosen to suit for a given task. Figure 8.2 shows the results.

Scene 3 is a high resolution object that was created specifically for the purpose of testing the blending between overlapping textures. Figure 8.3 shows a rendered image of the object. Overlapping occurs on all the parts that are standing out. The object can be seen from different angles and with closeups in appendix B.

Scene 4 is the famous Utah teapot and can be seen on figure 8.4. The purpose of this scene is to show how the algorithm works on a typical real object. The teapot was chosen since it is an object that is well known in the graphics industry.

More images of the scenes can be found in appendix B. There the scenes are shown from different angles, more closeups and higher resolutions.

The results from testing these scenes is discussed in the next chapter.

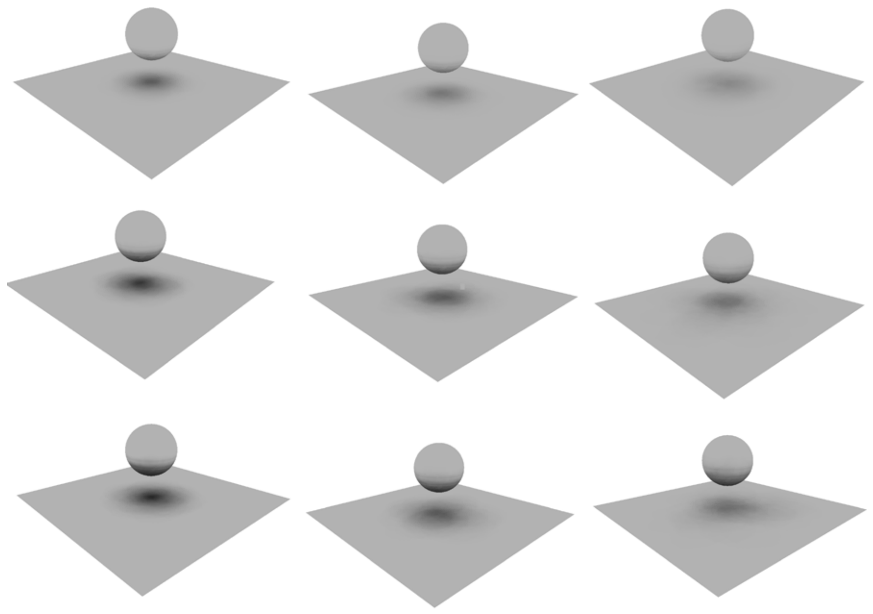

Figure 8.2: Scene 2 - Comparison - The top row has maximum distance of 5, middle row 10 and bottom row 15. The left column has a cone angle of 30°, the middle column 40° and the right column 50°. Each column can be seen in higher resolution in appendix B.

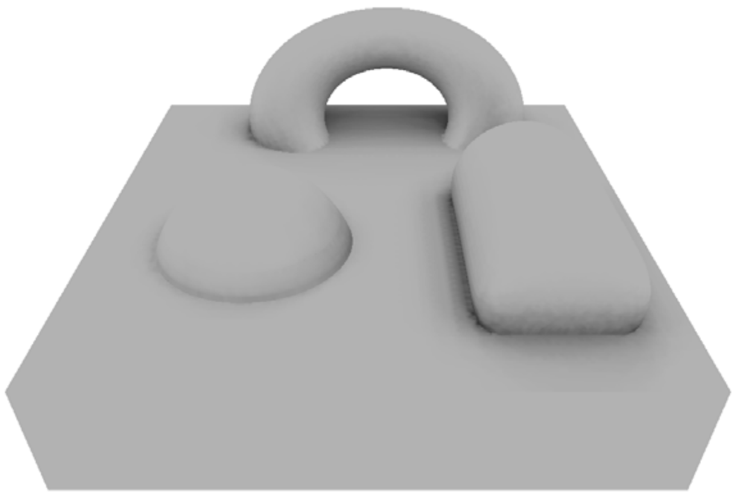
Figure 8.3: Scene 3 - Texture blending.

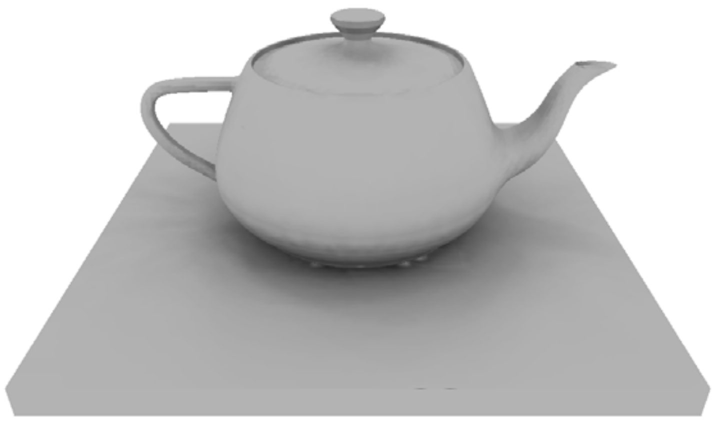
Figure 8.4: Scene 4 - Utah teapot.

Chapter 9

Results

There were some problems identified when testing the scenes presented in the previous chapter. First of all the algorithm is very slow for complex objects. It greatly depends on the scenes and the variables that can be defined, how fast it will be. If we set the texture size to be high and choose to have many rays for each patch, then the algorithm will take long. When the scene contains many triangles, and we choose the variables to be of reasonable values, the algorithm that finds adjacent triangles will be by far the most time consuming part. This should be treated in some way and discussion about that can be found in chapter 11.

The images rendered came out good in most cases. There was one particular problem that was identified and caused a lot of thinking and testing. When objects are structured in a certain way, there is the possibility that clusters will be created incorrectly. Then we will get triangles assigned to the same cluster, which will be mapped to the same patch, but do in fact not belong together. In other words, we will have triangles belonging to the same patch in a cluster but these triangles can be located on totally different parts of the object. Only thing that the triangles have in common is that they are mapped to the same cluster and same patch in the cluster. Figure 9.1 illustrates the problem. There we have two triangles from different parts of an object, but because of how the object is structured, they are mapped to the same patch. What happens then is that when finding ambient value for a patch, its center and normal vector will

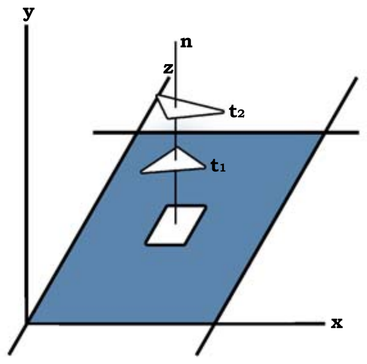

Figure 9.1: Triangles with same patch.

be completely wrong. That is because we take the average of the triangles that belong to the patch and the triangles should not all belong to that patch. This problem was identified when the Utah teapot was rendered and is now known as the teapot problem. In this particular case it works to set the the comparing angle, when clusters are created, to be less than 60°. An angle of 57° worked well which caused the clusters to be smaller and the problem does not present itself. This works for this case but can easily be created again if an object is structured in a certain way. Figure 9.2 shows the teapot rendered with different angles.

Testing shows that memory needed to run the program can be high for large objects. This was lowered slightly when the code was optimized, but can still be improved more. This is mainly when C++ pointer are not released when they are not in use anymore. The biggest memory usage is when we find ambient occlusion for each patch, when that is done and we have created one texture that stores the ambient values, the patch clusters are dropped, and memory is released.

In general the results indicate that the goal has been achieved. Ambient occlusion works as intended for the test scenes, and especially the most important part, texture blending is working.

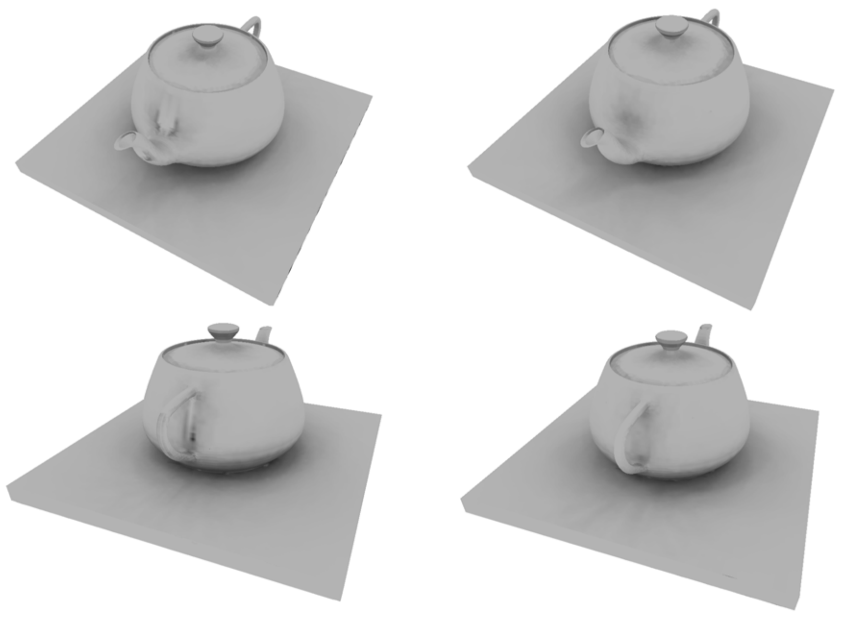

Figure 9.2: The teapot problem. On the left the clusters were created with a compare angle of 60°. Then the problem presents itself. On the right the compare angle is changed to 57° and then we have correct clustering.

CHAPTER 10

Discussion

10.1 Summary

The purpose of using ambient occlusion is that we want a more realistic ambient value than is presented in the Phong reflection model. There a constant ambient value is added to an entire scene. The result of this can be a dull looking image since the surrounding environment of a given point on a surface is not considered. Instead we want to consider each point and add a different ambient value to each one, based on the surrounding environment.

Ambient occlusion adds a subtle but realistic effect to objects and scenes without the need of using a more complex global illumination method. We don't need to consider any light sources when finding ambient occlusion. This fact can make development process much easier because developers can apply ambient occlusion and then be able to use less light sources. Also the complication of placements of lights will be less because we generally would need fewer light sources when ambient occlusion has been applied.

Ambient occlusion can be used to simulate other things than just shadows. If the algorithm is designed in such a way, we can tweak parameters to get interesting results. This can be for example to simulate dirt or rust that accumulates on objects or to get a weathered and worn effect. Imagine a statue that has been out in all kinds of weather for many years.

It can be a good choice to apply ambient occlusion to scenes. The result can greatly effect the outcome in rendered images without to much effort. Two important features are that no light sources are needed and the results is view-independent, meaning that we only need to find ambient values once for a static scene.

10.2 Contributions

In this paper we have evaluated ambient occlusion and created an algorithm that simulates the shadows that accumulate on parts on objects that are not fully visible to the environment. We find ambient occlusion by casting rays from points on a surface and the ratio of how many rays intersect the scene is stored. This value indicates what the ambient value at a given surface point should be.

Textures are used for storing and displaying the ambient values on surfaces. We create multiple local textures which contain the values. A problem was identified with this approach which is that texture seams are visible at borders where textures are adjacent to one another. The problem of having visible texture seams is well known. 3D Studio Max offers the possibility of applying ambient occlusion to objects and moreover it offers to render the ambient values to a texture. Figure 10.1 shows an ambient occlusion texture rendered for a teapot using 3D Studio Max. The seams are clearly visible on the teapots base and stout.

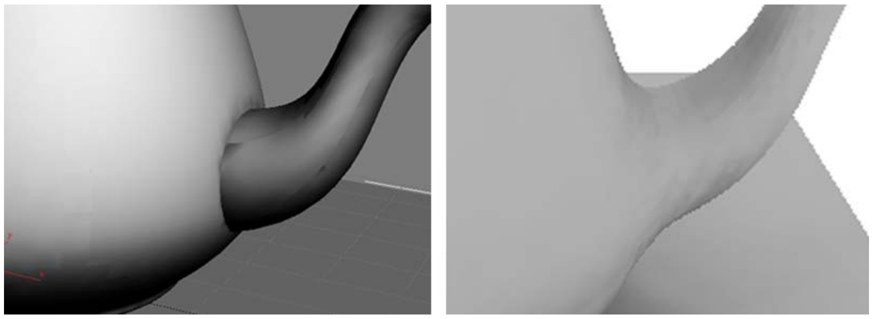
Figure 10.1: On the left we have ambient occlusion rendered to texture with 3D Studio Max. Seams are visible. On the right is a teapot rendered to texture using the texture blending implemented here.

To overcome the problem of having visible seams we suggested making the textures overlap and then we find the ambient values as before. This results in us getting multiple ambient values on a curved surface which leads to us blending between these values. We blend using the normal vector at the surface as the contributing factor. Each of the three normal vector values is used to decide the color of each of the three textures that are overlapping. This gives us smooth blending and less visible texture borders over curved surfaces.

Chapter 11

Future Work

Following is a discussion about what can be extended or fixed.

The imported geometry data should be represented with simple triangles. And furthermore it should be defined in one polygon mesh, meaning we can only have one instance of a mesh declared in the imported COLLADA file. These are design decisions. Other types of geometry data could be e.g triangle strips, triangle fans and polygons. There are two possible ways to handle other possibilities. Either the implementation should be expanded in such a way that it will handle these kinds which would not be to much of an effort. Other way would be to create or use a so called conditioner which would convert the kinds mentioned earlier in to the preferred type which would be triangles. Often graphic cards are optimized in such a way that they are optimized to work fast for triangle strips. Then it would be preferred to convert all data to triangle strips and then work on that. This is not a goal for this kind of work so simple triangles were chosen to work with.

The way adjacent triangles are found is very time consuming. If we have a very complex scene as input, that has many triangles, then this algorithm can be the most time consuming part of the overall calculations. Usually there is the need to tweak all the variables for a given scene so that it looks good, and when we are experimenting with them, we are always changing the variables and then running the ambient occlusion on it again. Then this time consuming algorithm that finds adjacent triangles can be a broblem. One possibility would be to save the adjacent triangles list in a text file or database and then load it at runtime. Then we only have to find the adjacency list for the first time for a given scene. Other possibility would be to allow us to render the scene with certain parameters, and then change them while rendering to see different results. Each triangle knows what triangles are adjacent to him and if we do it like this, we are working on the same triangles, only changing other parameter.

The physical memory needed for calculating ambient values in complex scenes can be high. This is because in some cases, I have neglected to properly get rid of variables that are not needed anymore, C++ pointers and such. This was not a major concern for the implementation, but is still something that needs to be optimized.

Next step would be to apply the ambient occlusion to an object and store the information such that it can be used later in a final scene. It would be possible to create some simple animation with the ambient occlusion applied to the animated object. Then the algorithm would be applied to each frame in the animation and stored and then used when the scene is rendered.

The texture packing algorithm is somewhat inefficient. The goal there is to make sure that the local cluster textures fit in one large texture. The texture sizes are evaluated such that the largest cluster texture is put in first and so on. The packing could be made more efficient by using some well known texture packing algorithm.

Texture is exported as an uncompressed bitmap image which is an algorithm I wrote myself. Since bitmaps are uncompressed, at least in this case, the size of the texture image can be large. This could be expanded so that the textures are exported in a different format that supports compression.

There are usually many parts of an object that have no shadows from ambient occlusion. These parts could be evaluated in such a way that they would have much smaller part of the ambient texture applied to it, to lower the memory usage. Same could be done with parts that are highly in shadow or parts where there are the same shadow values over a large part. We could lower the texture space needed for that as well.

It can be a difficult task to find the correct parameters to use for a given scene. The parameters that can have a significant effect on the outcome are the number of rays to cast, the maximum distance, the cone angle and the texture size in clusters. It would be a good expansion to allow for the user to see visually if he makes some changes in the parameters. A graphical user interface(GUI) could be designed which would load an object with default parameters. Then it would be possible to easily change parameters and the results are shown as soon as they are ready. This can also have good impact on the part that finds adjacent triangles since then it would be done when the scene is loaded and stored.

The teapot problem identified in the previous chapter needs to be considered. One possibility would be to look at the location of a triangle when he is being assigned to a cluster. We would map each triangle to 2D when we are clustering by dropping the relevant 3D coordinate. Then compare the triangle that we are considering now, with all the triangles that have been assigned to the cluster. If some other triangle is located close to this triangle, we compare their values. If the 3D coordinate that was dropped when the triangles were mapped to 2D is significantly different, we have a triangle that should not belong to this cluster.

CHAPTER 12

Conclusion

In this paper we have implemented an algorithm that adds ambient occlusion on objects. We find the ambient occlusion with help of ray-tracing by shooting out rays from the surface of an object. The number of rays that hit the surrounding environment indicates how much shadow that surface point is in.

Figure 12.1 shows a rendered image using the algorithm implemented here. The scene is a Cornell box. The results from the implementation can also be seen on the figures in chapter 8 - Testing and in appendix B - Screenshots.

There are many possible options at each step when implementing something like has been done here. Some of the options have been evaluated and discussed, their pros and cons. Many approaches do it similarly as has been done here. That is to trace rays for finding ambient values and use textures to store and display the values. The implementation details vary from designs. The biggest contribution made in this process has been the overlapping textures and how their values are blended in an effort to have seamless textures on surfaces.

We create clusters that consists of all the triangles in the mesh we are working on. The clusters overlap each other which leads to us finding ambient values more than once for some points on an object. This is done so that we can have a smooth transition between clusters and the overall ambient occlusion will look realistic.

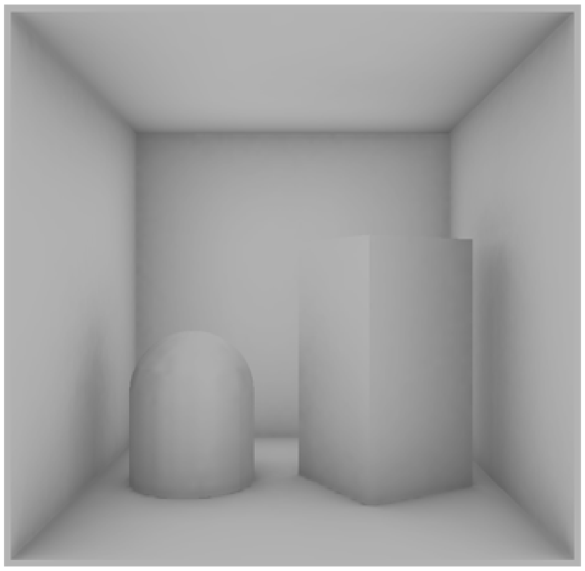

Figure 12.1: Cornell box scene rendered with the ambient occlusion algorithm implemented in this paper.

We apply the ambient values to textures and blend between them using the normal vectors at each point. This gives us a smooth transition between the textures. We create one texture that stores all the ambient values, meaning that we are blending between different locations on the same texture.

There are many aspects of computer graphics that have been looked at. Shadow algorithms have been discussed, such as shadow mapping and shadow volumes. Illumination models such as the Phong reflection model which is a local illumination model. More importantly, global illumination models have been discussed, such as ray-tracing, radiosity and photon mapping.

The predecessor of ambient occlusion, the ambient light illumination model, was analyzed. The model leads to the more general approach of ambient occlusion which is becoming very popular in the gaming and movie industries. Recent implementation have been discussed which all had a big influence on the solution presented here.

Textures and texture mapping is used extensively here along with texture blending and finally rendering of scenes.

COLLADA was used for importing digital assets. It has been a great experience learning about COLLADA, what it stands for and how it is used. COLLADA is an effort in having a common digital asset exchange schema that different applications can use. It is being adopted by many companies and for example COLLADA is used in the well known Playstation 3 console from Sony.

The overall process of this thesis has been very informative and most of all fun. There has been a lot of learning and sometimes wrong assumptions made. When looking back at some stages in the process and looking at the assumptions made at some points, it was clearly easy to go in wrong directions.

When thinking about the options available and the way it was done here, I find that to be a very interesting idea. The idea of overlapping textures and blending between them using the normal vectors is appealing and can create a realistic smooth looking images.

We conclude that the goal has been achieved. We have created an algorithm that creates multiple overlapping textures on an object and finds ambient values for the textures. When the object is rendered, the blending occurs on overlapping textures. The images rendered with this algorithm look smooth and usually there are no abrupt shading changes visible.

Appendix A

Tools

There are a lot of tools needed to get this kind of project to work. The flow of the work relative to the tools used is as follows.

Data is represented in COLLADA[3], and imported into the COLLADA Document Object Model(DOM)[10]. The COLLADA DOM is an application programming interface(API) that provides a C++ object orientation of a COLLADA document. It follows that we will be using C++ for programming since the COLLADA DOM was chosen.

To help visualize the implementation, OpenGL[1] was used to a great extent. Also Cg[14] has been used to create a fragment program that allow us to use hardware to calculate values.

Softimage|XSI® was used for creating objects and scenes for testing.

A.1 COLLADA

COLLADA stands for COLLAborative Design Activity which defines an XML-based schema to enable 3-D authoring applications to freely exchange digital assets without loss of information[3]. It was created through the collaboration of many companies and was adopted as an official industry standard by the Khronos group in January 2006. COLLADA is supported by many tools and is already used by thousands of developers. COLLADA is very young. After SIGGRAPH '03, Sony Computer Entertainment, established a working group, which did a thorough analysis of all existing digital asset formats. Many other companies in the gaming industry became involved and at SIGGRAPH '04 the first public presentation of COLLADA 1.0 was made. Since then it has gone through revisions and current version is 1.4.1. COLLADA has many goals including to be used as a common data exchange, be adopted by many digital-content users and provide an easy integration mechanism which enables all data to be available through COLLADA. It also has the goal of liberating digital assets from binary formats which is why it was chosen to be an XML-based schema. Arnaud and Barnes have written a book[2] which covers all aspects of COLLADA.

A.2 COLLADA DOM

To be able to interact with the data in a COLLADA file, the COLLADA Document Object Model[10] will be used. The DOM is a framework to be used for the development of COLLADA applications. It provides a C++ programming interface to load, query and translate instance data. The DOM loads the data into a runtime database which mirrors those defined in the COLLADA schema.

A.3 Softimage XSI

Softimage|XSI® is a 3D animation and modeling software. Many models used in this report were created with Softimage. Softimage has the possibility of exporting scenes as a COLLADA schema. It has the option of converting all geometry to triangles before exporting, which is convenient for the implementation. When a scene is created in Softimage that we want to find ambient occlusion for using the algorithm presented here, we first need to convert all meshes into one polygon mesh object. Then that one object should be exported as that are the requirements in the implementation.

A.4 OpenGL and Cg

The implementation requires constant visual representation to see what is going on and OpenGl[1] is great for that since it allows for a fast visual representation of the data whenever needed. OpenGL stands for Open Graphics Library and is a software interface to graphics hardware. It consists of procedures and functions that allow a programmer to specify objects and operations for producing graphical images. Cg[14] stands for C for graphics. The Cg language allows for control of objects, their shape, appearance and motion which will be rendered using graphics hardware. This means that Cg allows us to program the graphics hardware directly without going into the hardware assembly language. In this particular case Cg is used to blend between different locations on a texture. A Cg fragment program is created which input is a pixel color, three texture colors, possibly from different places on one texture, and the normal vector at the given point. Each texture color is multiplied with one of the normal vector values and the colors are then added together. The output is then the input color multiplied with the texture colors.

Appendix B

Screenshots

- **Scene 1** is a Cornell box. Here we see it with different parameters.
- **Scene 2** is a sphere hovering over a plane. This scene was created mainly to test the different number of rays and cone angle for the rays.
- **Scene 3** is an object with smooth surfaces. The purpose is to look at how the texture blending is working. There are some closeups for identifying if we can see texture seems.
- **Scene 4** is a Utah teapot.

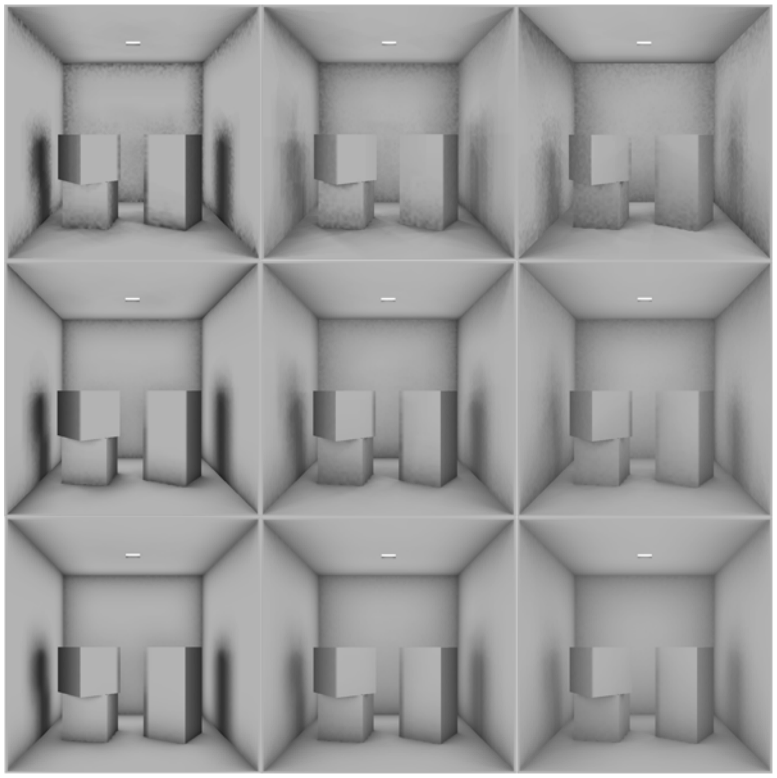

Figure B.1: Scene 1 - Comparison - The top row has used 16 rays, middle 64 rays and bottom 256 rays. The cone angles are 30° in the left column, 50° in the middle column and 70° in the right column.

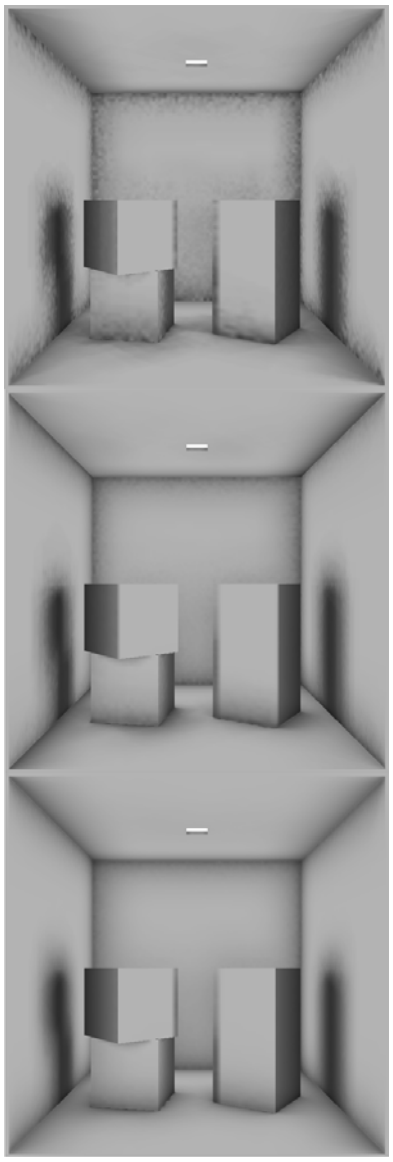

Figure B.2: Scene 1 - Comparison - The cone angle is 20°. The top has 16 rays, middle 64 rays and bottom 256 rays.

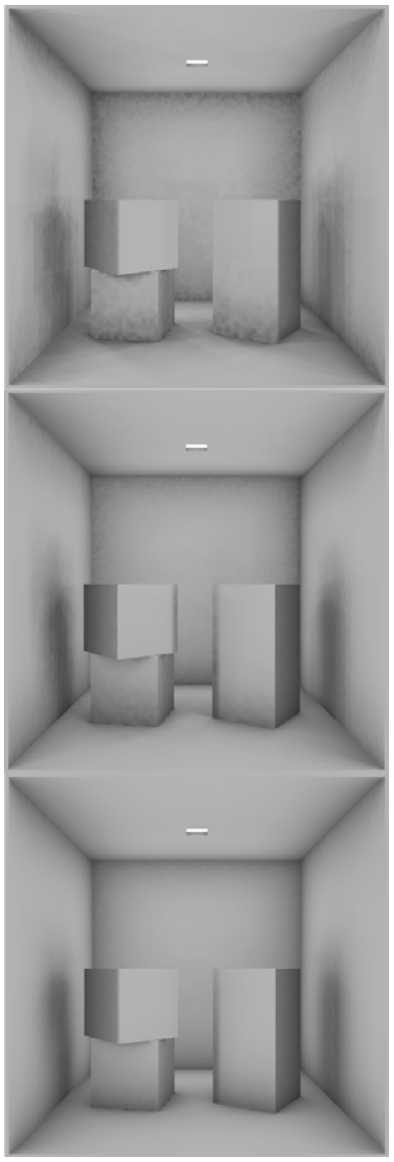

Figure B.3: Scene 1 - Comparison - The cone angle is 50°. The top has 16 rays, middle 64 rays and bottom 256 rays.

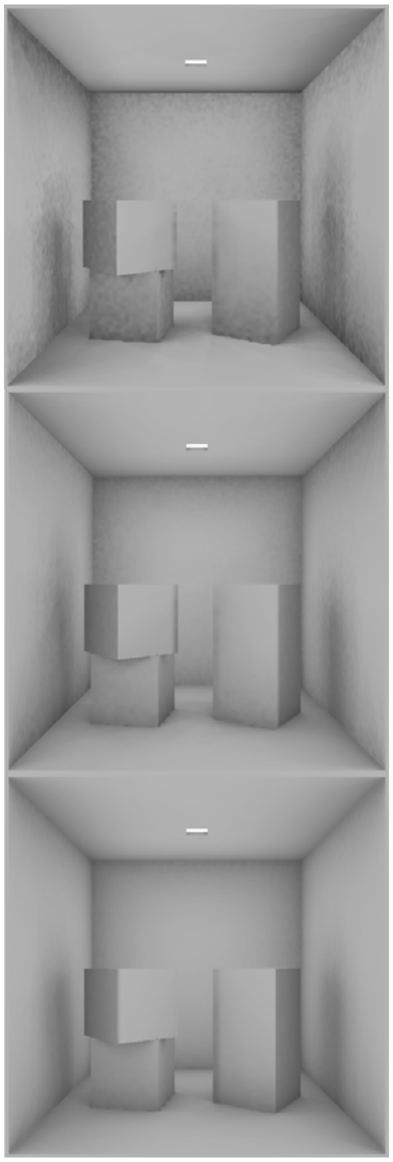

Figure B.4: Scene 1 - Comparison - The cone angle is 70°. The top has 16 rays, middle 64 rays and bottom 256 rays.

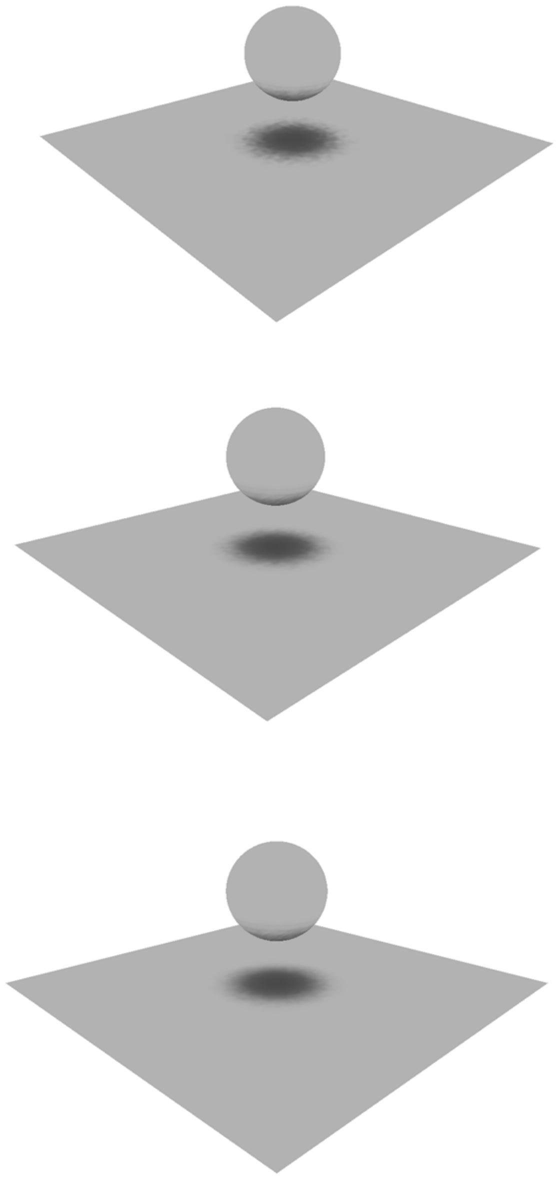

Figure B.5: Scene 2 - Comparison - The cone angle is 20°. The top has 16 rays, middle 64 rays and bottom 128 rays.

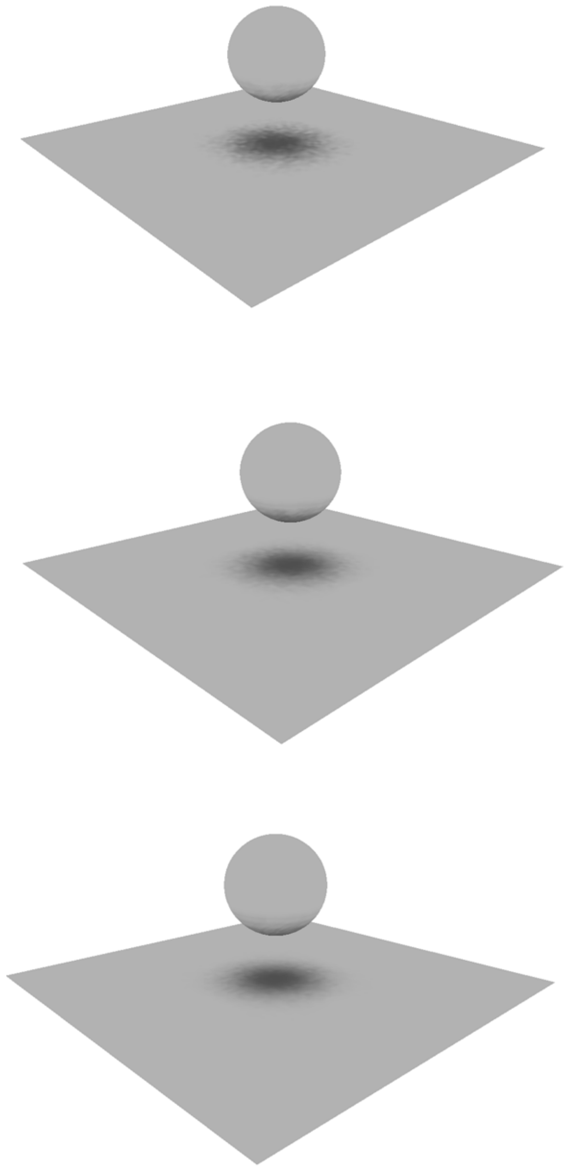

Figure B.6: Scene 2 - Comparison - The cone angle is 35°. The top has 16 rays, middle 64 rays and bottom 128 rays.

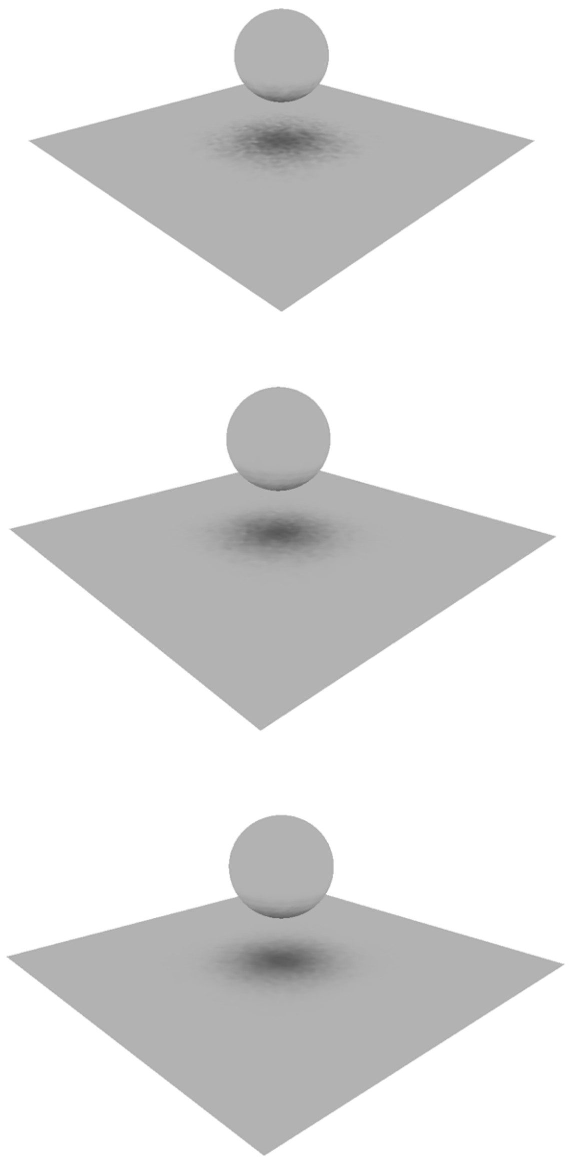

Figure B.7: Scene 2 - Comparison - The cone angle is 50°. The top has 16 rays, middle 64 rays and bottom 128 rays.

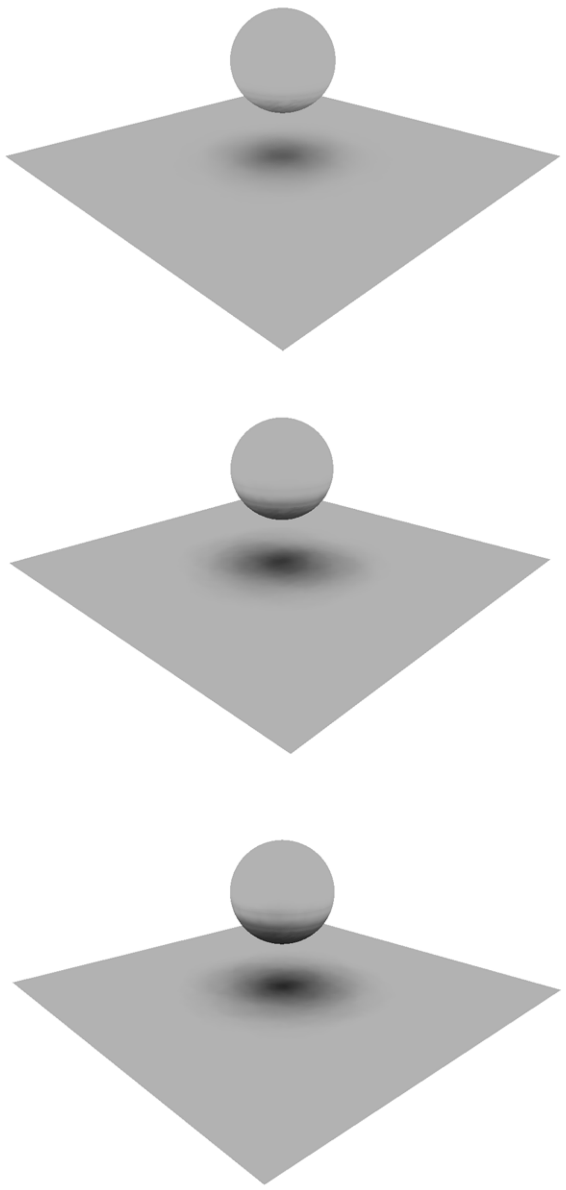

Figure B.8: Scene 2 - Comparison - The cone angle is 30°. The top has maximum distance of 5, middle has 10 and bottom 15.

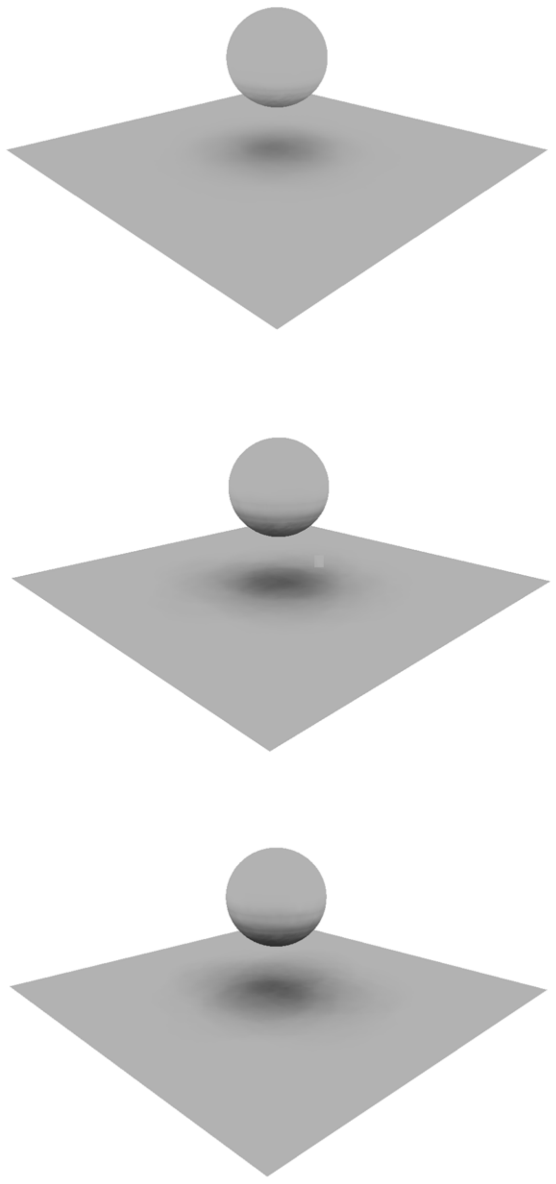

Figure B.9: Scene 2 - Comparison - The cone angle is 40°. The top has maximum distance of 5, middle has 10 and bottom 15.

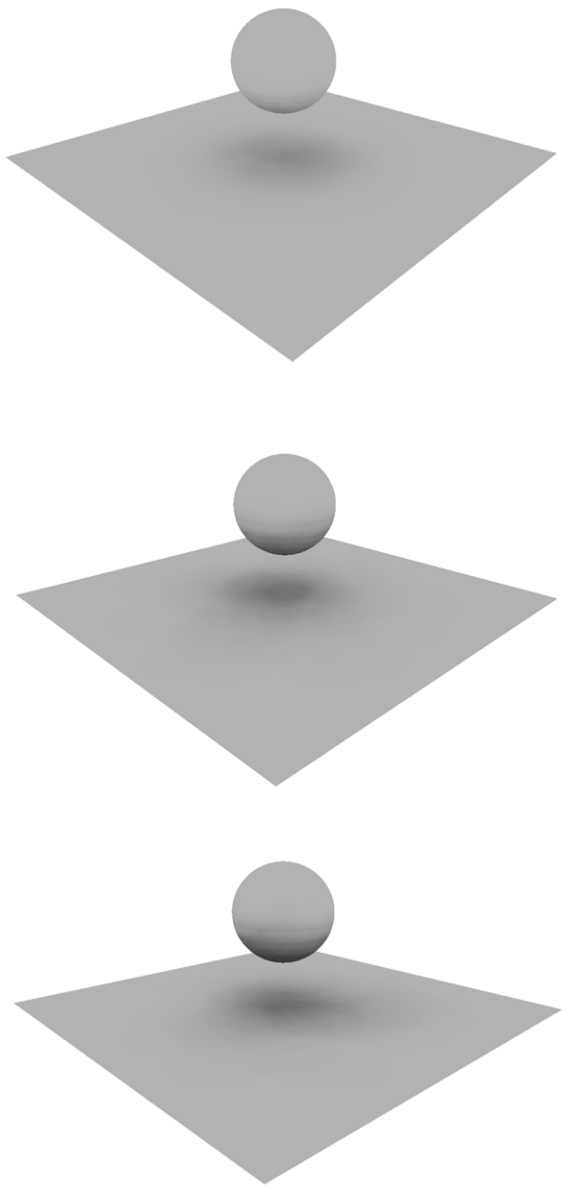

Figure B.10: Scene 2 - Comparison - The cone angle is 50°. The top has maximum distance of 5, middle has 10 and bottom 15.

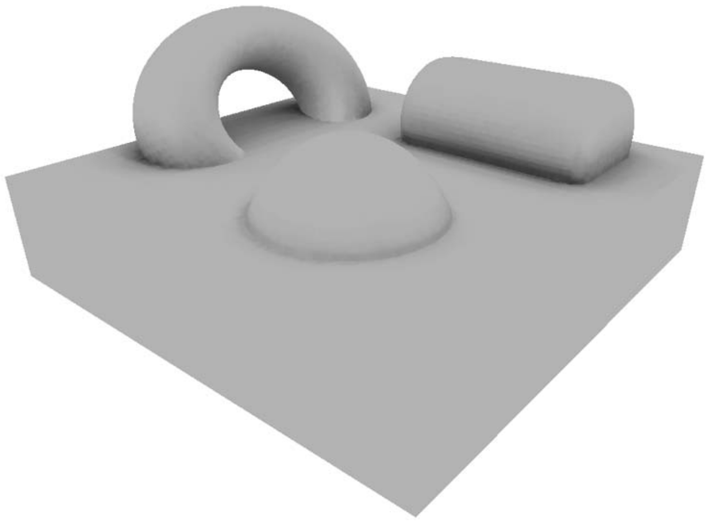

Figure B.11: Scene 3.

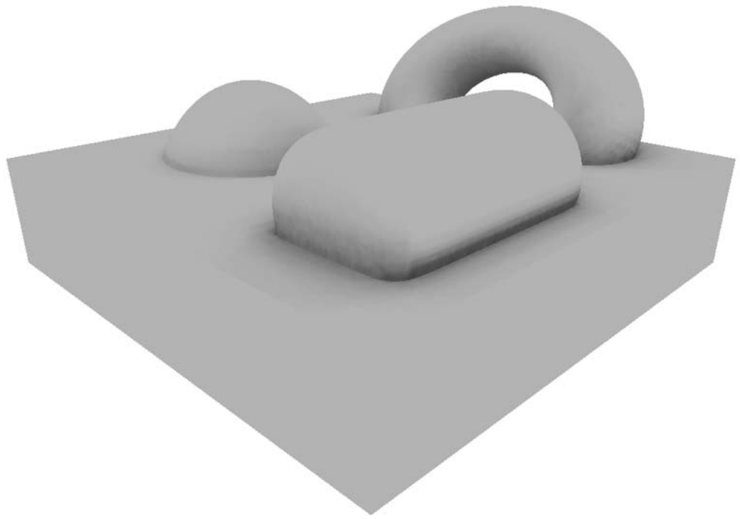

Figure B.12: Scene 3.

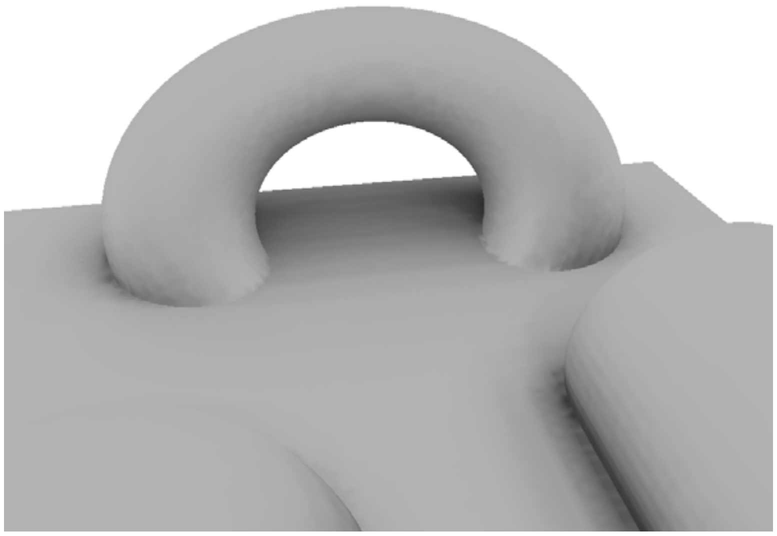

Figure B.13: Scene 3 - Here we zoom in to try to identify texture seems..

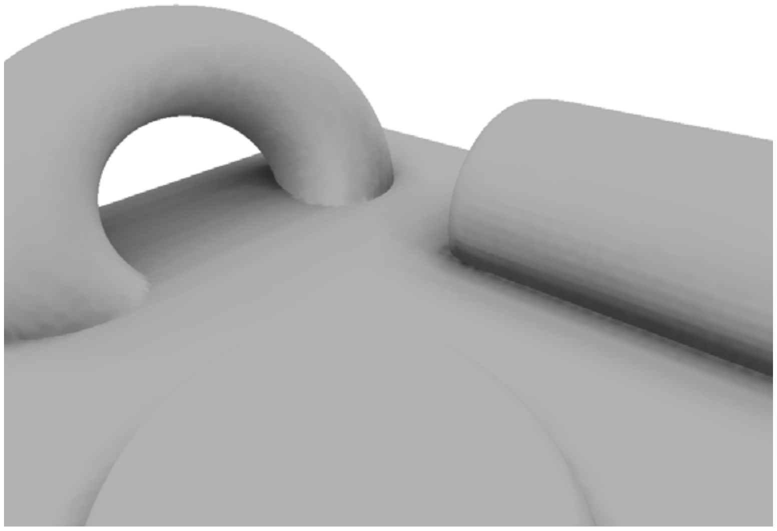

Figure B.14: Scene 3 - Here we zoom in to try to identify texture seems.

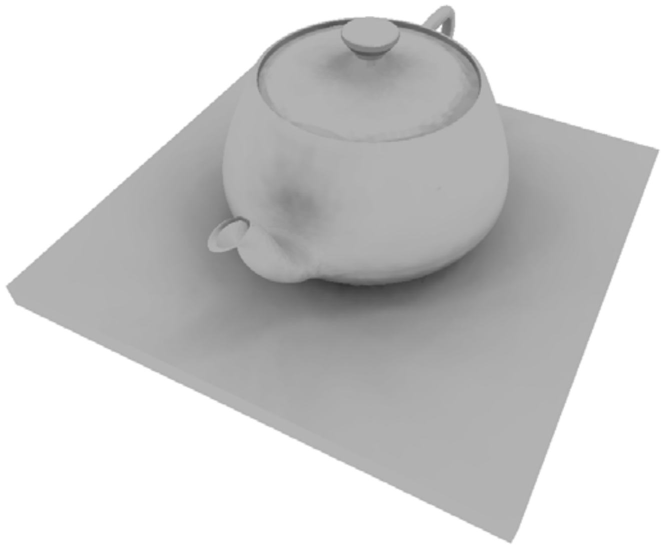

Figure B.15: Scene 4.

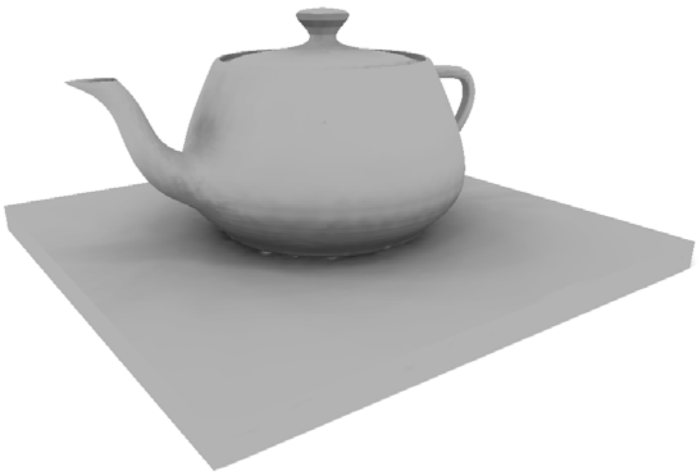

Figure B.16: Scene 4.

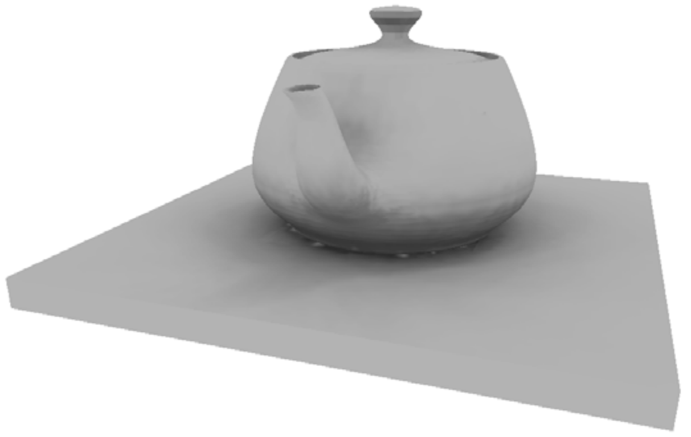

Figure B.17: Scene 4.

Bibliography

[1] Akeley K., Segal M., *The OpenGL Graphics System: A Specification*, Version 2.1, July 30, 2006.

[2] Arnaud R., Barnes M.C., *COLLADA Sailing the Gulf of 3D Digital Content Creation*, A K Peters, Ltd. 2006.

[3] Barnes M. Ed., *COLLADA - Digital Asset Schema Release 1.4.1 Specification*, Sony Computer Entertainment Inc., June, 2006.

[4] Bunnell M., *Dynamic Ambient Occlusion and Indirect Lighting*, GPU Gems 2, Chapter 14, pp. 223-233, NVIDIA Corporation, 2005.

[5] Borshukov G., Piponi D., *Seamless Texture Mapping of Subdivision Surfaces by Model Pelting and Texture Blending*, MVFX, a division of Manex Entertainment.

[6] Castro F., Neumann L. and Sbert M., "Extended Ambient Term," in *Journal of Graphics Tools*, vol. 5, no. 4, pp. 1-7, November, 2000.

[7] Catmull E., *A Subdivision Algorithm for the Computer Display of Curved Surfaces*, PhD thesis, University of Utah, December, 1974.

[8] Chan E., Durand F., "An Efficient Hybrid Shadow Rendering Algorithm," in *Proceedings of the Eurographics Symposium on Rendering*, Eurographics Association, pp. 185-195, 2004.

[9] Christensen P. H., "Global Illumination and all that," in *Siggraph 2003 course 9: Renderman, Theory and Practice*, Batall D. Ed., ACM Siggraph, pp. 31-72, 2003.

[10] *COLLADA DOM 1.4.0 - Programming Guide*, Sony Computer Entertainment Inc., 2006.

[11] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Introduction to Algorithms*, Second Edition, McGraw-Hill Book Company, 2001.

[12] Crow F. C., "Shadow Algorithms for Computer Graphics," in SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques, pp. 242-248, 1977.

[13] Everitt C., "Interactive Order-Independent Transparency," Technical Report, NVIDIA Corporation, 2001.

[14] Fernando R., Kilgard M. J., *The Cg Tutorial, The Definitive Guide to Programmable Real-Time Graphics*, NVidia Corporation, Addison-Wesley.

[15] Goral, C. M., Torrance, K. E., Greenberg, D. P. and Battaile, B., "Modeling the Interaction of Light Between Diffuse Sources," in Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '84. ACM Press, New York, NY, pp. 213-222, 1984.

[16] Hasenfratz J. M., Lapierre M., Holzschuch N. and Sillion E., "A Survey of Real-time Soft Shadows Algorithms," in *Computer Graphics Forum*, vol. 22, no. 4 pp: 753-774, 2003.

[17] Iones A., Krupkin A., Sbert M. and Zhukov A., "Fast Realistic Lighting for Video Games," in *IEEE Computer Graphics and Applications*, vol. 23, no. 3, pp. 54-64, May 2003.

[18] Jensen H. W., "Global Illumination Using Photon Maps," in *Rendering Techniques '96*, Pueyo X. and Schröder P. Eds., Springer-Verlag, pp. 21-30, 1996.

[19] Kontkanen J. and Laine S., "Ambient Occlusion Fields," in *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*, ACM Press, pp. 41-48, 2005.

[20] Landis H., "Production-Ready Global Illumination," in *SIGGRAPH 2002*, course notes #16 (Renderman in Production), pp. 87-102, ACM, July 2002.

[21] Malmer M., Malmer F., Assarsson U. and Holzschuch N., "Fast Precomputed Ambient Occlusion for Proximity Shadows," in *Journal of Graphics Tools*, 2006.

[22] Méndez-Feliu Á., Sbert M., Catá J., Sunyer N. and Funtané S., "Real-time Obscurances with Color Bleeding," in *Proceedings of the 19th spring conference on Computer graphics*, Szirmay-Kalos L., Ed. SCCG '03. ACM Press, New York, NY, pp. 171-176, 2003.

[23] Phong B. T., "Illumination for Computer Generated Pictures," in *Communications of the ACM*, vol. 18, no. 6, pp. 311-317, June 1975.

[24] Watt A., Watt M., *Advanced Animation and Rendering Techniques. Theory and Practice*, Addison-Wesley, 1992.

[25] Whitted T., "An Improved Illumination Model for Shaded Display," in *Communications of the ACM*, vol. 23, no. 6, pp.343-349, June 1980.

[26] Williams L., "Casting Curved Shadows on Curved Surfaces," Computer Graphics Lab, New York Institute of Technology, Old Westbury, New York, 1978.

[27] Woo A., Poulin P., Fournier A., "A Survey of Shadow Algorithms," in *IEEE Computer Graphics and Applications*, vol. 10, no. 6, pp. 13-32, November 1990.

[28] Zhukov S., Iones A., Kronin G., "An Ambient Light Illumination Model," in *Rendering Techniques '98*, Drettakis G. and Max N., Eds., pp. 45-56, 1998.