Master's Thesis

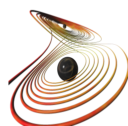
Cube Map Ambient Occlusion

Jürgen Oswald

Institute of Computer Graphics and Knowledge Visualisation, Graz University of Technology

www.cgv.tugraz.at

Supervisor: Dr.-Ing. Sven Havemann



Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am .... (Unterschrift)

Zusammenfassung

Diese Arbeit beschäftigt sich mit Methoden zur Berechnung ansprechender, der Realität nachempfundener Beleuchtung von digitalen 3D Modellen und Szenen. Die verwendete Technik, *ambient occlusion*, ist eine Annäherung an existierende, physikalisch korrekte Beleuchtungsmodelle. Der große Vorteil besteht in der einfacheren, und dadurch deutlich schnelleren Berechnung. Im Zuge dieser Arbeit wird ein bestehendes Verfahren erweitert mit dem Ziel eine höhere Qualität und effizientere Berechnung des Ergebnisses zu erzielen. Die Lösung wird zur Laufzeit für einen Teilbereich der Szene berechnet und für zukünftige Verwendung in speziellen Texturen gespeichert. Durch den Zugriff auf bereits gespeicherte Teillösungen reduziert sich der weitere Berechnungsaufwand.

Die Arbeit führt zu erst in die Theorie und mathematische Grundlage realistischer Beleuchtung ein. Im Speziellen wird die sogenannte *rendering equation* erläutert welche die Basis nahezu aller realistischen Beleuchtungsmodelle darstellt. Darauf aufbauend wird das in dieser Arbeit verwendete Beleuchtungsmodell behandelt. Es stellt eine grobe Vereinfachung dar, welche dennoch eine erstaunlich gute Annäherung bietet. Danach folgt der genaue Ablauf der benötigten Berechnungsschritte zur Beleuchtung einer konkreten 3D Szene mit Hilfe dieses Verfahrens. Weiters wird eine Möglichkeit gezeigt die berechnete Teillösung zur Laufzeit zu speichern und darauf zur schnellen Darstellung zuzugreifen. Dazu wird das Ergebnis in virtuellen Projektoren gespeichert welche es erlauben die enthaltene Information auf die Szene zu projizieren und somit darzustellen. Durch die diversen Annäherungen und Vereinfachungen können sich Fehler in der Berechnung der Beleuchtung ergeben. Es wird deshalb weiters ein Verfahren behandelt welches typische Beleuchtungsartefakte und Berechnungsfehler effizient reduziert.

Die vorgestellten Algorithmen und Methoden werden dazu verwendet eine software library zur Beleuchtungsberechnung zu erstellen. Das Ergebnis der Arbeit ist eine library die einfach in bestehende Anwendungen zu integrieren ist, und eine höhere Qualität in der Darstellung digitaler 3D Modelle ermöglicht.

Abstract

This work is concerned with the computation of appealing illumination for digital 3D scenes and models. The technique presented in this thesis is called *ambient occlusion*. It is an approximation to existing and physically correct illumination models. The simple and fast computation along with the quality of the output is what makes this technique popular in the computer graphics community. This work is focusing on extending an existing method for computing ambient occlusion. The goal is to achieve a higher quality of the outcome and a more efficient computation. The solution is computed at runtime for parts of a scene and stored in dedicated textures for subsequent access. Making use of the previously stored partial solutions allows to reduce the remaining workload for displaying the illumination.

The work starts by giving an introduction to the theory and the mathematical foundations of realistic lighting simulation. Emphasis is put on the *rendering equation* which is the basis of nearly all works dealing with realistic lighting. Subsequently the ambient occlusion illumination model is introduced. It is a coarse approximation to the models of realistic lighting but the outcome is of surprisingly high visual quality. The introduction is followed by the exact procedure that is necessary to illuminate a 3D scene using this technique. After this a method is presented that shows how to store the computed illumination and how to retrieve the data for fast reconstruction of the illumination without having to recompute it. This is done by storing the data in virtual projectors that allow to project the lighting back on the surfaces of a scene. During the course of this computation approximations are used to speed up the process. This leads to a number of possible errors and artefacts in the computed solution. To work around this problem the thesis also discusses a method to reduce typical lighting artefacts efficiently.

The methods and algorithms described in this thesis are used to create a free to use software library for lighting simulation. The outcome of this work is a library that is easy to integrate into existing software projects and allows for higher quality renderings of digital 3D models.

Preface

I would like to thank my supervisor Sven Havemann as well as Prof. Fellner for giving me the opportunity to work on a master's thesis in the exciting field of real-time global illumination. Thanks also to my parents and my friends for accompanying me all the way from the start to the finish of my study.

Special thanks to Thomas Richter-Trummer for a lot of worthwhile technical discussions.

Contents

Zusammenfassung **v**

Abstract **vii**

Preface **ix**

1 Introduction **1**
1.1 Let there be light . . . 1
1.2 Global Illumination . . . 1
1.3 Goal . . . 2
1.4 Outline . . . 3

2 Related Work **5**
2.1 Outside-In . . . 7
2.2 Inside-Out . . . 9
2.3 Object-Based . . . 11
2.4 Screen Space-Based . . . 12
2.5 Unclassified . . . 14
2.6 Cube map approach . . . 14

3 Theory **17**
3.1 Foundations of Theory . . . 17
3.1.1 The Rendering Equation . . . 17
3.1.2 Ambient Occlusion . . . 18
3.1.3 Cube Mapping . . . 19
3.1.4 Projective Texturing . . . 20
3.1.5 Bilateral Filtering . . . 20
3.2 Cube Map Ambient Occlusion . . . 22

4 Implementation **23**
4.1 Software Architecture . . . 23
4.2 Cube Map Rendering . . . 23
4.2.1 Placement of cube maps . . . 23
4.2.2 Render to Texture . . . 24
4.2.3 GLSL shader programs . . . 27
4.2.4 Cube map setup . . . 28
4.2.5 Computing fragment world positions . . . 29
4.3 Ambient Occlusion Computation . . . 31
4.3.1 Horizon-split ambient occlusion . . . 31
4.3.2 Sampling step size . . . 33
4.4 Projective Texturing . . . 35
4.5 Ambient occlusion blending and post processing . . . 38
4.6 Automated cube map placement . . . 41
4.7 Library Interface . . . 42
4.7.1 Defining the scene . . . 42
4.7.2 Wrapping the render calls . . . 43

5 Discussion **45**
5.1 Benchmarks 45
5.1.1 Ambient occlusion computation 45
5.1.2 Projective texturing 46
5.1.3 Cube map memory consumption 48
5.1.3.1 Temporary data 48
5.1.3.2 Ambient occlusion data 48
5.2 Discussion: Strengths and Weaknesses 49
5.2.1 Out-Takes 52
5.3 Conclusion and Future Work 53
5.3.1 Automated cube map placement 55
5.4 Interesting bugs 55

Bibliography **57**

List of Figures **58**

Chapter 1

Introduction

1.1 Let there be light

Lighting is one of the key elements in the synthesis of photo realistic computer generated images. Not only does it influence the look and feel, it also helps us understand the structure and the shape of objects contained. Without proper illumination the results can look dull, unrealistic and even unpleasant to the human eye. This is why it is important to put effort into the research and development of high quality lighting solutions. The findings of this field of research are used for example by movie productions, architectural visualizations or video games.

The mathematical foundation of realistic lighting simulation is the *rendering equation* introduced in [Kaj86] and shown here in (1.1). Trying to compute realistic lighting is trying to solve this equation.

$$I(x,x') = g(x,x') \left[\varepsilon(x,x') + \int_S \rho(x,x',x'') I(x',x'')\, dx'' \right] \tag{1.1}$$

Calculating the exact solution of the equation is a complex computational problem. Solving it analytically is undoable in many cases and solving it numerically can be an extremely time consuming-task. There are two terms in particular that make this equation hard to solve. First one needs to process all geometric information of a scene for every point to be illuminated. This is what is denoted by the $g(x,x')$-term. Depending on the complexity of the scene this can be a difficult and time-consuming task. What is worse is that the equation has a recursive term - $I(x',x'')$ - which in this case means that the computation has to be repeated for an unknown amount of iterations to get to the exact solution. This is why research is done to find good approximations to this model. The goal is to keep the error as low as possible while trying to reduce the computational complexity. A detailed description of the other terms in the rendering equation is given later on in chapter 3.1.1.

The benefits of this research range from lower production times and cost to the ability of using more sophisticated methods in real-time image generation. By now it has become absolutely feasible to create images that cannot be distinguished from real photos anymore. The computation times are low enough for photo realistic lighting techniques to be used in movie productions and off-line rendering of still images. Using professional modeling tools like Autodesk's *3DStudio Max* or *Maya* home users are able to compute images with life-like lighting conditions. What is left however is to make these methods work fast enough for use in real-time rendering applications.

1.2 Global Illumination

The incorporation of *global illumination* phenomena may be one of the most important cornerstones of photo-realistic image synthesis. Global illumination mainly denotes the interaction of light with the surroundings when it bounces from surface to surface. In the rendering equation this is taken into account by the recursive term. Crude approximations completely discard this interaction. Two important software libraries for real-time rendering, *OpenGL* and *DirectX*, use this approximation to compute illumination quickly. They both make use of the *Phong illumination model* that is explained in detail in [AMHH02]. One of the traits of this model is that the lighting of a point on a surface is based only on illumination coming directly from a light source. In reality there may also be light bouncing off a wall first and then hitting the surface point on an indirect path from the source. It turns out that these phenomena are extremely important and must not be

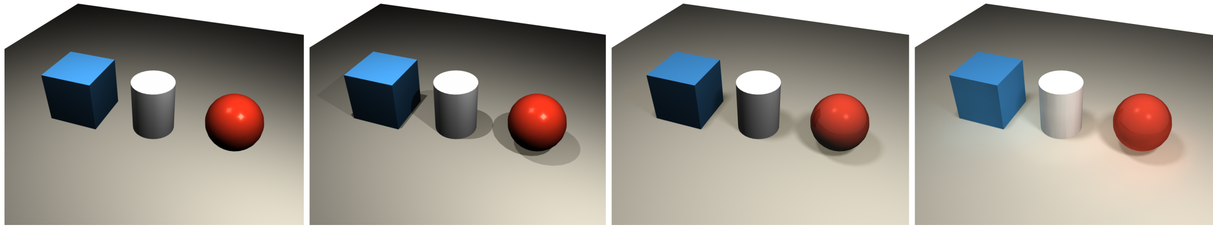


Figure 1.1: Example of various different illumination models ranging from simple to complex. The scene was set up and lighted using Maya. First image: Plain OpenGL direct illumination. Second: Hard shadows added. Notice how the inclusion of shadow adds to the feel that the objects are actually placed on the floor and not floating. Third: Soft shadows. Fourth: Global illumination. Notice the subtle blue and red shimmer on the left and right side of the cylinder and on the floor. In these examples a material with Phong reflection behaviour is assigned to all objects. This limits the amount of realism that can be achieved. Complex real world surface structures like brushed metal or human skin for example cannot be modeled accurately using this simple type of material.

neglected when trying to simulate realistic lighting. Not taking it into account can result in dull and non-realistic looking renderings. Another problem of this approximation is the complete lack of a geometry term. The Phong illumination model simply is not capable of computing shadows as caused by objects blocking the path of the light. It is lacking some of the most crucial and natural phenomena of real-life lighting. Figure 1.1 shows several differing illumination models ranging from coarse approximations, like the Phong model, to highly sophisticated global illumination rendering techniques.

Another important property that affects the outcome of a lighting simulation is the surface material assigned to the objects of a scene. The material interacts with the light and determines how light is propagated. The most common properties for a material are its diffuse, its specular and its ambient component. The diffuse component is used to describe how matte a surface is while the specular component describes the shininess in a simple way. The ambient component is used for a crude approximation of indirect illumination. The Phong illumination model makes use of these parameters. Again this is a very crude approximation to the type of materials that exist in the real world. Typically it is possible to identify this very simple material in computer generated images as it has a distinctive look that makes it resemble plastic in real life. For more elaborate materials like the human skin, or brushed metal more complex surface materials are needed. This work however is mainly concerned with the diffuse propagation of light which the simple Phong material is suitable for.

A typical side effect of global illumination is that it makes renderings turn brighter. This is because the propagation of light does not just stop after the first hit, but instead keeps on bouncing and transmitting energy to nearby surfaces. Even though this effect is very subtle and hard to notice in real life it adds a lot to the realism of computer generated images. What is also important to know is that the properties of a light beam may change after it bounces off a surface. Assume for example a bright white light is shining at a red wall. After bouncing off this wall the light will lose most of its spectrum. The beam of light that keeps propagating after the bounce will consist mainly of red light. This red beam will eventually cause a subtle red shimmer on objects that are close to the red wall. This effect is called *color bleeding* and is another important trait of global illumination. All in all global illumination techniques add to the realism of a computer rendering and are therefore a desirable feature to have in a rendering application.

1.3 Goal

The aim of this work is to create a quality lighting solution that is close to global illumination and fast enough to be used in real-time applications. The latter constraint poses a big challenge. For a framerate of 24 frames per second for example this means the entire computation for drawing an image needs to be finished within roughly 40 milliseconds. This can only be achieved by using a range of clever approximations to the rendering equation. Another challenge is to create a software library that is stand-alone and easy to integrate into existing software projects while still being efficient. High performance lighting simulation is usually coupled tightly to the internals of a software project to allow for quick computation and thus a high framerate. This work has to deal with enabling an efficient computation without knowing the internal representations used in the host application. Existing work usually assumes that the algorithms are implemented directly at the core of a rendering application which is not the case here. Another important point for the ease of use of the library is the amount of parameters that need to be tweaked by the user. The lower the number the more easy it will

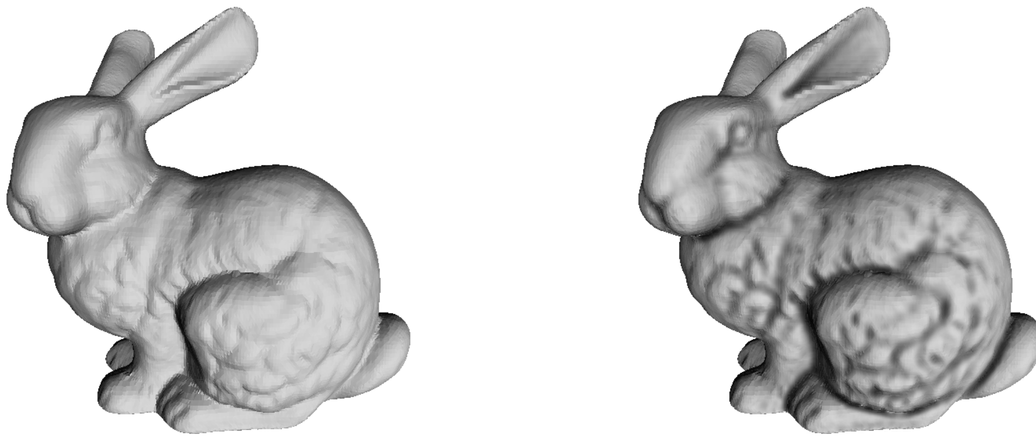


Figure 1.2: Both images are captured from a real time rendering application. Left: Rendering of the Stanford bunny with regular OpenGL lighting. Right: OpenGL lighting with ambient occlusion superimposed. The ambient occlusion adds natural looking soft shadows that accentuate the shape of a scene. In this example the strength of the shadow is exaggerated for a better visualization of the difference.

be to integrate the library into existing projects. Finally the goal is also to keep the amount of precomputations, that are necessary, as low as possible

To meet these goals a technique called *ambient occlusion* was chosen. It is an approximation to global illumination that produces soft shadows and accounts for the propagation of light from neighbouring surfaces. Figure 1.2 shows what a rendering making use of ambient occlusion may look like. Thanks to some clever simplifications the resulting workload is reduced drastically. The resulting outputs on the other hand look surprisingly similar to complex and expensive global illumination simulations. Yet users do not need to define a single light source to achieve this illumination, making it very simple to use.

Aside from the requirements concerning the illumination model there are also requirements to the software architecture of the resulting library. First, the interface to the library is supposed to follow the *bridge pattern*. This allows to separate the interface from the implementation. Given the need for updates or bugfixes, users may replace the library with newer versions without having to make changes to their code.

Next, for an efficient implementation the library needs to make heavy use of two important features from OpenGL. The ability to render to textures as well as the ability to use shader programs. The low level calls to OpenGL are hidden by two lightweight classes with the goal of portability in mind. The amount of software dependencies is kept as low as possible. Both classes come with their own types where necessary which makes the use of the features independent from the chosen graphics library. In the theoretical case of a severe change in OpenGL or in case the implementation is changed to DirectX internally the interface of the classes ideally remains unchanged.

The computation is controlled by an abstract backbone which allows to inherit from. While common functionality is already implemented there, the virtual functions allow to refine or implement more specialized strategies for the ambient occlusion algorithm.

1.4 Outline

This document starts out by giving a brief overview of the core idea behind ambient occlusion. After that a range of existing work on this topic is discussed and a classification is made to help sorting the related work by their behaviour. A short paragraph explains which techniques sound promising and are therefore used and adapted in this work. Consecutively the important mathematical foundations of light simulation as well as the original theory behind ambient occlusion is discussed. What follows is a detailed description of how to implement the chosen algorithms and how they all work together to form a software library. Part of this work also consists of developing a software library that enables ambient occlusion for regular rendering applications. This is why the software interface is also discussed in a separate section. Next the quality as well as the computation times of the implementation are analysed using three different scenes with different geometric properties. Finally the strengths and weaknesses are discussed followed by an outlook on things to improve in the future.

Chapter 2

Related Work

Ambient occlusion was developed as a cross between the unrealistic constant ambient light approximation of the Phong illumination model and the accurate but slow *radiosity* method [GTGB84]. Radiosity attempts to simulate light or heat transfer assuming for the scene to consist of perfectly diffuse surfaces. The idea for ambient occlusion is based on the following observation that is mentioned in [ZIK98]. Imagine a car standing on the ground on a cloudy day. There will most likely be shadow on the area below the car. Figure 2.2 shows a picture of this. The interesting thing is that this shadow can be computed simply by looking at how much the ground is obscured by the car. In most cases it is not necessary to take the light sources into account and still get natural looking results. So this illumination model is based solely on the geometry of a scene, not on the placement and properties of the light sources. This is what makes it easy to set up and also more easy to compute than radiosity for example. A complete method on how to put this model into practical use is described in [SZ98]. Note that this article uses the term *obscurances* to describe the ambient occlusion model. The article mainly revolves around the use of 2D textures to store precomputed lighting information. When used in this way the 2D textures are typically called *light maps*. The lighting information is retrieved in real time by applying the light maps on the surface of a polygonal scene. The actual ambient occlusion computation is mentioned only briefly in this article but it is one of the first publications to mention it as a mean for generating realistic shadow. This is what makes it important.

A very important characteristic of ambient occlusion is that it simulates diffuse light transport only. This means that light is assumed to reflect in all directions with equal probability when hitting a surface. Many real world objects behave in a way similar to this. Indoor walls of a room painted white are an example for this. However, it is not the only possible behaviour. Mirrors or glossy surfaces for example reflect light in a very specific direction. Materials like these cannot be modeled accurately with ambient occlusion. Instead there are other methods like *precomputed radiance transfer (PRT)* [SKS02] that allow to render non-diffuse materials in real time. The goal of PRT is to allow for complex materials like brushed metal or weaved objects to be rendered efficiently at a high visual quality. Sophisticated natural phenomena like soft shadows, caustics, inter-reflections between surfaces and *subsurface scattering* can be modelled using PRT. Caustics appear when light is focused by optics and concentrates on a small surface area. Subsurface scattering denotes the simulation of light as it propagates below the surface of a material, like the human skin for example. Figure 2.1 shows the occurence of caustics and subsurface scattering in real life. While soft shadows and inter-reflections can be modelled using certain ambient occlusion adaptions, subsurface scattering and caustics cannot be modelled accurately. PRT allows for more realism in this regard. The downside of PRT on the other hand is the lengthy precomputing step that is needed for it to work efficiently in real time.

The obscurance method mentioned before resembles *accessibility shading* [Mil94] which uses the accessibility of a certain point on a surface for rendering purposes such as the simulation of aging of a material. The motivation behind this comes from the real world phenomenon of dust setting down in the creases of objects. Typically creases are harder to clean than open and flat surfaces which is why they become darker over time due to the countless environmental influences. The article suggests two measures for the accessibility of a surface point. The first one is to find the biggest radius of the sphere touching the surface point and not intersecting any other geometry. The second measure can be thought of as making use of a spherical cleaning device to clean the surface. The harder the device is able to reach a point the less accessible it is. This can be expressed more formally by measuring the minimal distance of a surface point to the center of the cleaning device. For open surfaces this value will be equal to the radius of the device. For creases the distance will increase.

A couple of years ago the topic has regained significant interest in the scientific community. Within a short period of time a broad number of articles dealing with this problem have been published. Most of them use the term *ambient occlusion* to describe this particular illumination model. The vast majority of work is focusing on using the solutions on polygonal models.

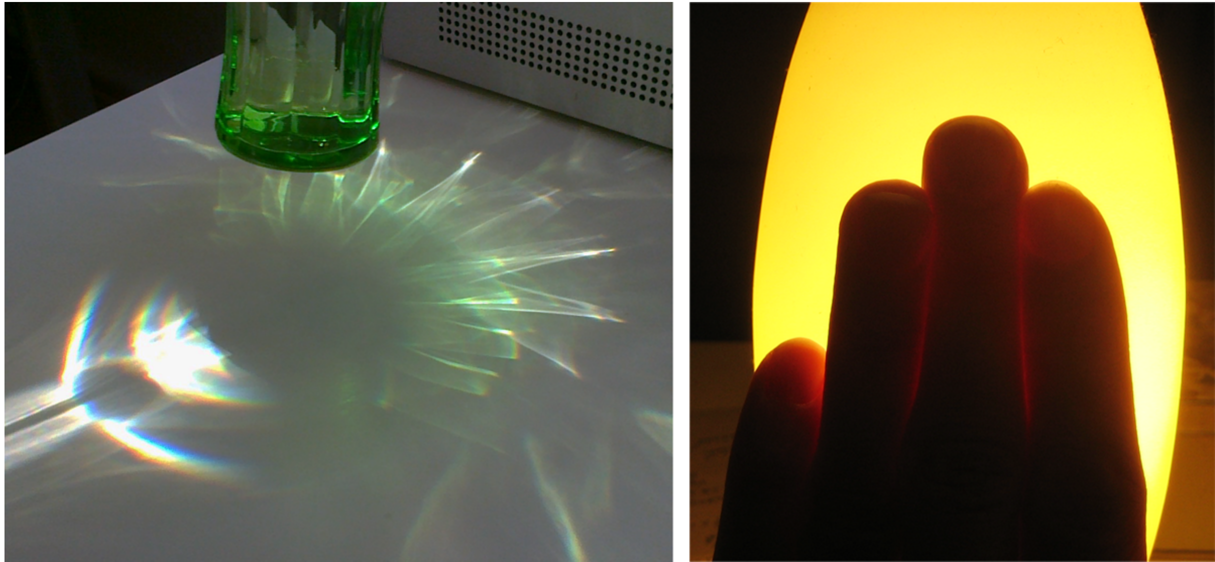


Figure 2.1: Left: Caustics in real life. Incident light is focused by a glass of water. Patterns of bright white appear where the light is concentrated on the surface. Right: Subsurface scattering. A certain amount of light penetrates the skin, propagates within the flesh, and exits at another point. This is what is causing a subtle glow of the hand in this image.

Figure 2.2: Left: The essence of ambient occlusion lighting as found in real life. A noticable dark shadow below the car that fades out softly. Right: The illumination is caused by a big cloud in front of the sun on a sunny day. As can be seen there are no hard shadows but instead there is equal diffuse lighting coming from the sky.

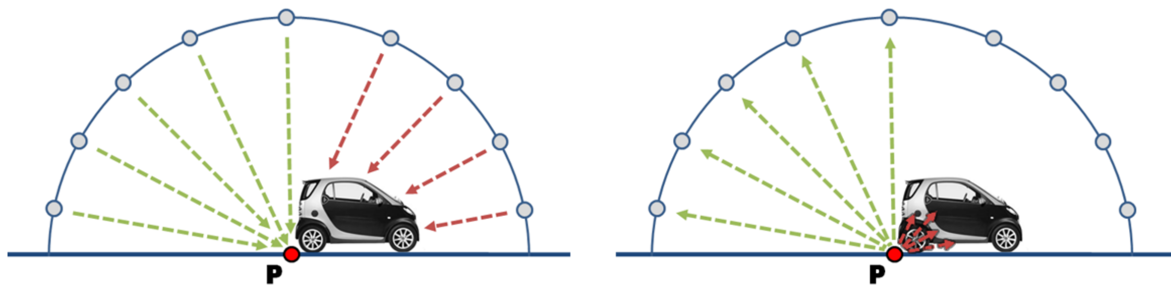


Figure 2.3: Left: A sketch of the outside-in approach for the evaluation of ambient occlusion. Right: Inside-out evaluation. Both methods yield the same result. 5 out the 9 samples yield a clear line of sight between the surface point P and the sky.

For a start one can look at the state of the art report on ambient occlusion [Kne07]. It gives a good overview on a range of differing implementations. Another nice overview on the history and development of ambient occlusion is given in [MFS09]. Based on their behaviour the different types of algorithms can be classified in the following way.

- Outside-In
- Inside-Out
- Object-Based
- Screen Space-Based
- Unclassified

This listing is followed by a quick overview on how some of the proven methods are used in this work to form a new approach that can be classified as screen space-based.

2.1 Outside-In

All approaches for computing ambient occlusion have on thing in common. They all try to find out how obscured a point on the surface is by the surrounding geometry. The outside-in approach tries to resolve this as follows. Imagine sitting on a cloud in the sky looking at a specific point on the ground. The main question that needs to be resolved is whether or not the point on the ground can be seen. If the point can be seen, then for this cloud it is not obscured. Repeating this step for all clouds in the sky one can say if the point is obscured rather heavily or if it is rather openly accessible. The left image in figure 2.3 shows a sketch of this type of evaluation. Again looking at the car example. Points on the ground below the car are not likely to be seen by many of the clouds. These points should thus be rendered darker than points beside the car.

The first way to implement this technique is by making use of *shadow mapping*, a technique that is explained in detail in [HLHS03]. A shadow map is basically a texture that contains a depth map of the scene as seen from the position of the light source. Each pixel in the texture contains the depth to the closest visible point of the surface. Given this information it is possible to find out if an arbitrary point on the surface is lit by the light source. If it is not lit another object is casting a shadow on the point. The following explains how this technique can be used to compute ambient occlusion. For each cloud in the sky a light source is set up pointing at the center of the scene. Using the shadow maps it is possible to find out the number of the light sources that can see a specific point of the surface. The number corresponds to the ambient occlusion value. This approach is explained in [MK05]. Typically shadow mapping is used to generate hard or soft shadows caused by a limited number of distinct light sources. The evaluation is done per pixel for the current view of the scene. The interesting thing about this approach is how a method for computing hard shadows is used to generate soft shadows. This is achieved by using a high enough number of light sources. Eventually the big amount of hard shadows that are cast start to blend into a single, smooth shadow area. Figure 2.4 shows this. In this particular implementation the light source positions are generated from an environment map which encodes the environmental lighting. The lights are placed at a certain distance from the centerpoint of the scene. The distribution is not even but instead more lights are placed in the direction of the brighter parts of the environment map. What is crucial for this technique is to be able to handle a large number of light sources. If the number is too low, single distinct shadows will be noticeable which

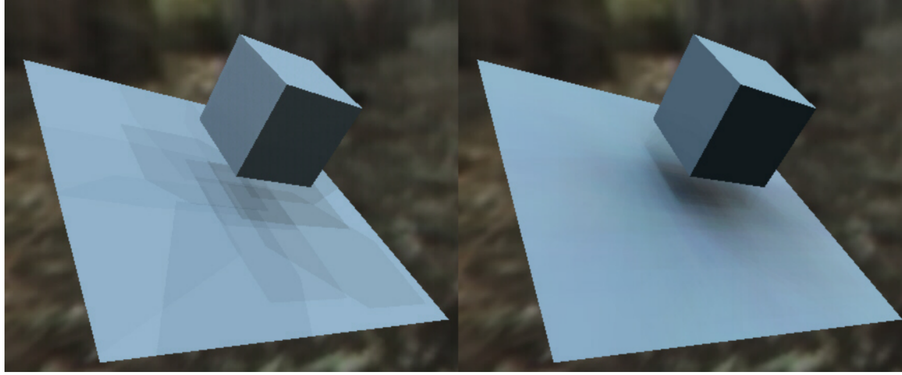


Figure 2.4: Left: Shadow mapping causes hard shadows. In case the number of light sources is too low the distinct shadows are distinguishable to the human eye. Right: Using a big enough number of light sources the individual shadows start blending into a single, smooth shadow area that is similar to ambient occlusion. The image is originally published in [MK05].

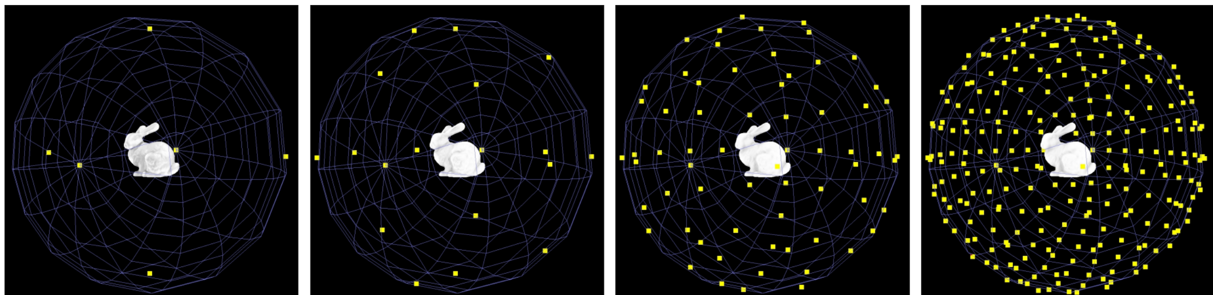


Figure 2.5: A scheme used for generating distributions of sampling points on a sphere around an object. The scheme is progressively refining the sampling while keeping the points of previous distributions undisplaced. The image is originally published in [SSZK04].

completely ruins the perception of the effect. The paper gives hints on how to optimize the memory usage as well as on how to balance the work-load between vertex and fragment shaders to achieve optimal frame-rates.

What is worth mentioning is that due to the non-even distribution of the lights the technique in general does not implement true ambient occlusion. In order to do so an even distribution would have to be chosen. However, this behaviour may be forced by using a single intensity environment map as input. Regardless of that the shadows resulting from this technique look really nice and resemble ambient occlusion a lot. A downside of this method is that the solution is view dependent which means that it needs to be recomputed whenever the camera moves.

Another way to implement ambient occlusion using light sources placed on a sphere around the scene is described in [SSZK04]. Again a principle similar to shadow mapping is used. In this case the evaluation is not carried out per pixel but instead lighting is calculated per vertex. The first step is to generate shadow maps for each of the light sources placed. Then all vertices of the scene are rendered as points. This allows to evaluate whether or not a point is visible to a light source. Repeating this for every point and every shadow map it is possible to find out exactly the number of light sources visible to each vertex of the scene. Computation time gradually increases with the amount of lights. This is why the article also focuses on how to generate increasing amounts of well distributed points on a sphere. The intent behind this is to be able to start out computing a solution with a low amount of light sources and refine it later on. As a constraint already computed positions are to be included in the next highest level of detail. This way the results of previous computations can be reused. The scheme described starts out by placing the lights at the 6 vertices of a unit octahedron. In order to increase the level of detail points are added at the mids of the edges and projected to the unit sphere. This allows for a nice way to be able to chose between quality and computation time at runtime. Figure 2.5 shows an example of this subdivision scheme.

Once an initial solution is computed, vertex colouring is used to apply the lighting information on a scene. By doing this the solution becomes view independent and can be precomputed for static scenes. The downside is that the quality becomes dependent on the amount of vertices and their placement. So this solution may not be able to recreate the lighting properly for surfaces with low amounts of vertices. This is an unpleasent property. Planar surfaces like tables, floors, or

walls of buildings are often made of big polygons with small amounts of vertices that are placed at the outer edges. In order to take advantage of this lighting method one may have to add vertices which increases the work-load when rendering these surfaces. Also it is necessary to either use an automation tool for adding the vertices or to spend the time adding them manually.

2.2 Inside-Out

This approach is simply an inversion of the outside-in approach. The right image in figure 2.3 shows a sketch of this principle. Think of standing on the ground and counting the amount of clouds in the sky that can be seen. When lying on the ground below the car, chances are that not a single cloud can be seen. When standing beside the car a lot more clouds are likely to be seen. The smaller the amount of clouds that can be seen, the darker the point is to be shaded.

Various different techniques can be used for solving this problem. A straight forward way, for example, is to use *raytracing* to find the amount of clouds that can be seen. Raytracing is a method for generating images by following rays that are emitted from the eye of the viewer. A ray can be thought of as a single beam of light that is shot into the world. Every time the beam hits an object the interaction with the material of the object is evaluated and a colour is computed from this interaction. This is essentially the inversion of the real world process of a lightbeam bouncing around and eventually reaching the eye of a viewer. Raytracing is often used to create images that are used as ground truth for benchmarks. It typically allows to create very high quality solutions at the cost of lengthy computation times. One of the main challenges of raytracing is to find intersections of rays with a scene quickly. There is an active field of research on how to reduce the computation times. In order to find out more about this topic one may have a look at [PH04] for a start. The book is about generating lifelike renderings using physical phenomena and making use of raytracing. It covers basically everything one needs to know to get started but also goes beyond that. Unfortunately there are hardly any articles on how to employ and optimize this technique solely for the purpose of computing ambient occlusion. However, it is rather straight forward to come up with a basic method. Essentially one goes about generating a number of rays that evenly sample the hemisphere above a surface point. For this point the ambient occlusion value is proportional to the amount of rays not hitting any other part of the scene. These rays correspond to the directions that are not occluded and where clouds can be seen.

Another way is to use the form factor computation known from radiosity. Form factors basically express the amount of energy transferred between two parts of a scene. To compute this energy transfer, geometric properties of the two parts are used. These properties consist of the area of both parts, their distance as well as their relative orientation. More energy is transferred in case the 2 parts are facing each other. [IKSZ03] suggests that this principle can be reused for the computation of ambient occlusion. Surface points that are surrounded by big parts of the scene that are very close are likely to be more obscured. For such a point the amount of sky to be seen is rather low which is why it should be shaded darker. The intent of this article is to derive a framework that allows to compute and use the solution in video games. The actual computation may take some time which is why it is done in a preprocessing step. The results of this step are stored in textures and used for rendering the illumination in real time for static scenes.

NVIDIA, one of the leading manufacturers of consumer grade graphics hardware, has described in a paper how to compute ambient occlusion per vertex in real-time for dynamic scenes and objects [Bun05]. This is done by approximating the scene geometry with circular discs as shown in figure 2.6. The discs can be used to find out how much a surface point is obscured by all other discs in the scene using again the form factor computation. In this article a disc is generated for each vertex of a polygonal model. This is not ideal because it makes the solution depend partly on the tesselation of the model and not only on its shape. Also the fact that the solution is computed per vertex has the same negative properties mentioned before. Still this method can be a good choice for a number of applications. Especially in the field of computer games where 3D models are typically hand crafted by skilled artists the problems can be overcome by keeping them in consideration when creating the models. The actual computation involves computing the form factor for every disc with all other discs which yields a runtime of $O(n^2)$. In order to bring down the computational complexity the article mentions a hierarchical optimization. When computing the ambient occlusion for a surface point, discs that are far away are grouped together and replaced by a bigger disc with averaged properties. This employs a level of detail hierarchy where far away geometry is approximated by successively coarser models. Figure 2.7 gives an idea on how this approximation is working. At the end of this article there is also a short section about how to add one pass of indirect lighting. Adding this method allows to simulate colour bleeding which is a valuable feature.

There is a follow up paper that explains how to adjust this algorithm to work at per pixel precision [JH07]. Simply increasing the vertex resolution of the scene or moving the computation to a fragment shader happens to introduce two notable rendering artefacts. These artefacts remained unnoticed in the initial solution because of the lower vertex resolution that would blur them out. The first artifact is caused by the hierarchical disc grouping. It is a fact that using a coarser approximation for a number of discs results in a differing value of the computation because the approximation is not perfect. This means that there are visible seams whenever a different level of detail is chosen for neighbouring surface points. This happens because of a discrete criterion that switches between the various levels of detail depending

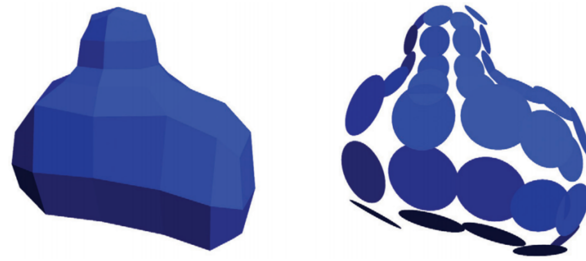


Figure 2.6: NVIDIA uses discs to approximate polygonal objects. This allows for an easier computation of ambient occlusion. The image is originally published in [Bun05].

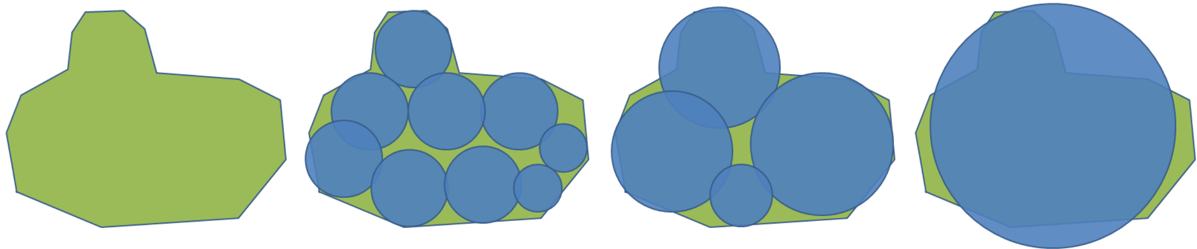


Figure 2.7: Hierarchical disc grouping. Varying the size of the discs an object may be approximated more or less accurate. From left to right in this image a more coarse approximation is achieved by using less and less discs of bigger size. In case the illumination has to be evaluated for points that are far away from this object a more coarse approximation is used. This way less computations have to be carried out which results in a faster evaluation.

on the distance of a surface point to the discs. The problem is mitigated by interpolating the two levels of detail along the borderline where the criterion switches. The second type of artefacts is caused by a numerical problem. Using the form factor method to compute ambient occlusion includes a term to divide by the squared distance of a surface point to a disc. When evaluating surface points that are at the exact same position or very close to a disc the occlusion value goes towards very high values. This means that the disc causes an extremely powerful shadow on this surface point and all neighbouring points close enough. This results in lots of unwanted dark spots. The solution to this problem consists firstly of moving the discs to the centers of the polygons instead of placing them at the vertices to approximate the scene geometry. Secondly, for the finest level of detail actual polygons are used instead of the discs. This way the dark spots near the disc center points are effectively removed.

There is another short paper that makes use of the disc and form factor based computation [Was05]. What it contributes is the clever use of a spatial subdivision structure to bring down the computational complexity. The optimization is based on two observations. First, close geometry has a bigger influence on the ambient occlusion than far away geometry. Following this observation it is reasonable to take into account only geometry that is closer than a certain user defined distance threshold. The second observation is that ambient occlusion by definition only takes into account geometry located in the upper hemisphere above a surface point. In practice this means for this algorithm that it may take into account only discs that are close enough to a surface point and above that point. In order to retrieve this subset if discs quickly the paper suggests putting them into an *octree* data structure. Octrees belong to the set of spatial data structures and are often used in computer graphics to retrieve information of locality between points in space efficiently. Using them in this case allows to quickly discard large amounts of discs in case they are too far away or in case they are not in the upper hemisphere. This in turn reduces the amount of computations needed to find the ambient occlusion for a surface point.

Finally there is a paper on a seminar project of mine that builds on the previous 3 presented techniques [Osw08]. It uses the same principles to compute the lighting but stores the resulting illumination in light maps. This allows to decouple the complexity and detail of the computation from the vertex resolution of the scene. It also allows to precompute and store the solution for static scenes. One of the challenges tackled in this work is the automated generation of texture coordinates for use with the light maps and *texture atlases*. Texture atlases denote textures, usually of high resolution, that contain big numbers of small textures used on individual parts of a scene. Putting these individual textures into a single texture of bigger scale allows for more efficient rendering. The lower the total amount of textures needed to render a scene, the

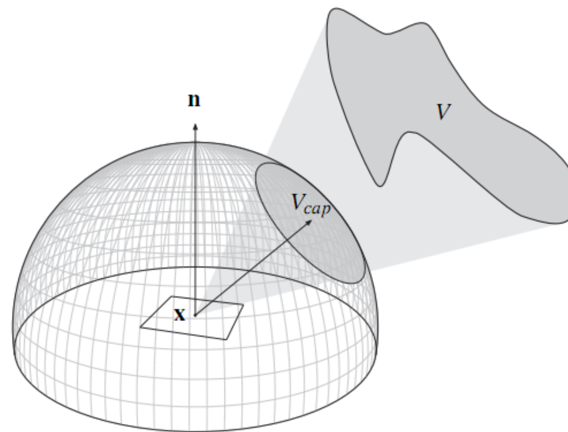


Figure 2.8: Approximation of an occluding object V by a spherical cap V_{cap}. This approximation is valid only for the surface point x which is the current point of evaluation in the ambient occlusion computation. The image is originally published in [KL05].

more efficient it is because changing the texture is a costly procedure. For static scenes this work precomputes the lighting within a reasonable amount of time and allows for a very quick rendering of the ambient occlusion illumination. The dependence on texture atlases and its mandatory use of texture coordinates however does not make it suitable for quick updates.

2.3 Object-Based

This type of methods does not resolve the ambient occlusion for the entire scene. Instead it is only concerned with the shadowing that is caused by a few manually selected objects. It is used mainly in video games to allow for key objects to cast a smooth shadow on other objects in their vicinity.

The first object based technique is called *ambient occlusion fields* [KL05]. Given a point to shade on the surface and a selected occluding object, the occluder is approximated by a spherical cap with known orientation, size, and average occlusion direction. Figure 2.8 shows what this cap may look like in an example. Using the disc properties it is possible to evaluate the shadow the occluder causes on the surface point by solving an analytic integral. For practical purposes this computation is replaced by a simple lookup table that converts the cap properties to occlusion values. What is still missing is to get to the cap properties given a point in space. First of, cap properties are computed beforehand in a preprocessing step as a field around the occluder. In order to be able to retrieve these properties at an arbitrary point within the field, the properties are encoded as radial functions. The function values are stored in a *cube map*. Cube maps can be pictured as 6 textures forming the sides of a cube. They can be used to associate the pixels of the textures with a direction in space. The direction is defined by the point of center of the cube and the position of the pixel on the imaginary cube. More information on cube maps is given in section 3.1.3. Figure 3.2 also shows what a cube map may look like. In this approach the cube map allows to associate the pixels of each of the 6 textures with one specific function that points in a particular direction. The cap parameters can be computed by solving for the direction as well as the distance. The theory is a bit complicated but the outcome is an efficient method for computing ambient occlusion shadows caused by moving objects of rigid shape. The memory consumption and the resulting framerates are suited well for the use of this method in computer games.

A slightly similar technique is described in [MMAH07]. Again a field of occlusion values around a selected object is precomputed. In this case the ambient occlusion values are directly stored in a 3 dimensional texture. What is interesting is that since the final shadow values are precomputed the properties of the receiving surface are not taken into consideration at all. Usually the normal vector of a receiving surface needs to be considered. However the authors report that leaving this part away from the computation already yields decent results. In order for this to work, the computation must be carried out on the entire sphere around a point in the occlusion field. This is a contradiction to the definition of ambient occlusion which says that only the upper hemisphere must be taken into account for the shading. In order to increase the quality the paper suggests not only storing the raw ambient occlusion values but also an average occlusion direction. This addition has a negative impact on the memory cost but improves the visual quality of the solution. For a point on a surface the occlusion is retrieved by looking up the precomputed values in the 3 dimensional texture using the coordinates of the

point in the object space of the occluder. Just as the paper above this method is well suited for use in video games when trying to compute the shadow caused by a limited amount of important objects.

2.4 Screen Space-Based

Research recently started to focus on the field of *screen space ambient occlusion* (SSAO). This set of methods typically makes use of the *depth buffer* to compute ambient occlusion on the fly for the current frame. The depth buffer contains the depth for all rendered pixels. By knowing the camera position and viewing direction it is possible to reconstruct the 3D coordinates of the parts that are currently visible.

The big advantage of this approach is that typically no precomputation is needed and the evaluation can be made fast enough to work in real time. One of the first papers to mention this technique was published by the gaming company *Crytek* [Mit07]. The article on screen space ambient occlusion is part of a paper describing the set of technologies employed in one of their game engines. Unfortunately the paper contains very little information about the details of the implementation of this technique. What is mentioned though is that for every pixel the depth values of a number of the neighbouring pixels are used in a depth comparison. The depth values are used in a certain way to find out if a pixel is in a crease or if it is on a flat and open part of the surface.

The approach is extended by a different paper to account for the influence of direct illumination [RGS09]. This evaluation is not part of traditional ambient occlusion computations. It allows for the shadows to have colours based on the parameters of the light sources. Traditional ambient occlusion shadows are grayscale only. The technique works by placing a number of sampling points around a pixel and looking if the sampling points are below or above the scene geometry. Figure 2.9 shows an example. In case the sampling points are below the surface of the scene geometry, it means that the surface is causing occlusion in this direction. In case the sampling points are above, the pixel may be illuminated by the lights pointing from this direction. In addition to this the paper also proposes a method to incorporate one bounce of indirect lighting. This is done by evaluating the geometric relation of the sampling points and applying the form factor equation to compute the energy transfer. The energy used to distribute is taken from the pixel colours after illumination. This means that the evaluation has to be done in an additional render pass which may slow down the computation. The outcome however looks appealing and adds another layer of realism. The paper also addresses a typical problem of screen space approaches. Due to its nature the depth buffer only contains geometric depth information about the parts of the scene that are currently visible. However, there may be objects hidden behind other parts that would actually have an influence on the shadowing in real life. These objects and their shadow are usually not captured by screen space methods. In order to work around this problem the paper suggests two possible solutions. First is *depth peeling*. Instead of having just one framebuffer, this techniques typically describes the availability of multiple depth buffers that correspond to different layers of depth for one specific view. Using these additional buffers it is possible to take into account back facing objects or objects that are hidden behind other parts of the scene. Another approach is to have multiple cameras that have a slightly different position and looking direction. Combining the depth buffers of all of these cameras allows to retrieve more information about the scene and about objects that may be hidden for one camera. The technique described works at interactive to real-time framerates for complex and dynamic scenes.

NVIDIA has also proposed a screen space method called *horizon-split* or *horizon-based ambient occlusion* [BSD08], [DBS08]. Both papers talk about the same method in essence. The idea is rather interesting. In order to find out the amount of occlusion it is usually necessary to integrate or sample the entire hemisphere. The papers takes advantage of the nature of the scene being represented in a depth buffer to cut down the computation costs. The depth buffer can be seen as a 2D *heightfield* defined by the visible parts of a scene. A heightfield is a two-dimensional array of numbers describing the height of all points from the ground. For every pixel on the screen there is a horizon that discriminates the part of the hemisphere that is above the heightfield and the part that is below. Points that are below will always contribute as occluding geometry while points that are above may contribute as non-occluding. The papers basically describe how to find the horizon for every pixel and how to compute ambient occlusion from this information. The results are similar to ambient occlusion and allow for a computation at real-time framerates.

As mentioned before there is a downside to simple screen space methods. Objects that are not visible do not contribute to the shadowing even though they should. Some techniques to work around this problem have been discussed already.

Another way to get rid of this problem is proposed in a paper about a technique called *hybrid ambient occlusion* [RBA09]. Instead of using the depth buffer as the scene representation a *voxel* grid is used. Voxel stands for volumetric pixel and denotes a volume element in a three-dimensional grid. In this approach a voxel grid is used as means to approximate the geometry of the scene which can be seen. An example is shown in figure 2.10. A voxel is marked as solid in case there is an object at this location in space. It is marked empty in case there is no object. The voxel grid is constructed in real time and encoded in a regular 2D texture. Each cell only needs a single bit describing whether or not the cell is empty. Using 4 channels, with 32 bit each, per texture and up to 8 output buffers it is possible to get a resolution of up to 1024^3 for the grid. The actual ambient occlusion computation is carried out in screen space for all

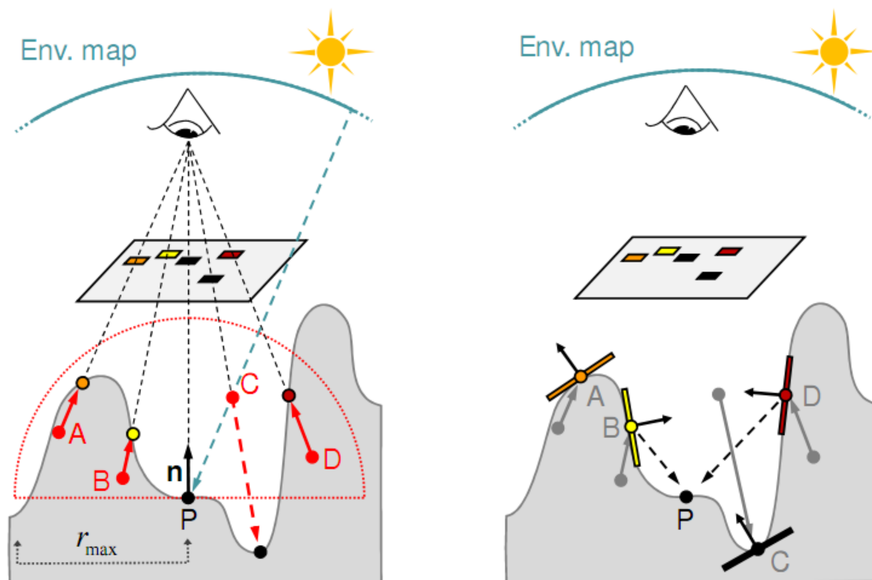


Figure 2.9: Left: Sampling points A, B, C, and D are placed randomly around the point of evaluation P. C is the only point that is above the surface of the scene geometry. This means that there is no occlusion in the direction of C from point P. Right: The occluding points A, B, and D are assigned a surface patch and energy transfer is computed between these points and point P. This is an approximation of indirect illumination where light is bouncing from these occluding points to P. The image is originally published in [RGS09].

pixels. In order to do this rays are traced in a number of directions in the upper hemisphere. The voxel grid allows to find out quickly whether or not the ray is hitting a non-empty cell. In order to cut down the number of computations the solution is computed for a down-sized framebuffer. A *bilateral filter* is used subsequently to upscale the solution to the size of the actual rendered image. In essence this filter is a geometry aware blurring filter. While plain areas are blurred more, the filter tries to avoid blurring along edges. This makes the outcoming solution look smooth while retaining detail where necessary.

One more recent paper that tries to overcome the typical screen space problems is called *ambient occlusion volumes* [McG10]. As the name suggests, this approach does have similarities to the *shadow volumes* technique. Shadow volumes are a method for computing hard shadows. The ambient occlusion volume approach is based on the assumption that all occluders have a predefined maximum radius of influence. In case an occluder is too far away from a point, it does not contribute to the shadowing of the point. Building on this, in this work a volume is constructed around each polygon of a scene. See figure 2.11 for an example. The volume is created in such a way that all points outside of the volume are too far away to be influenced by the polygon. Sending this volume as geometry to the graphics hardware allows to invoke a

Figure 2.10: Representation of the Stanford dragon model as voxel grid at different grid resolutions. Depending on the resolution the approximation becomes more accurate or less accurate. Coarser approximations allow for a quicker but less accurate lighting computation. The image is originally published in [RBA09].

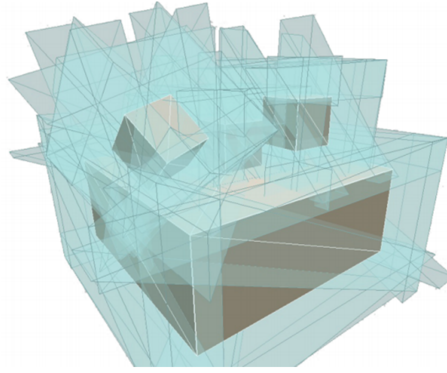


Figure 2.11: Ambient occlusion volumes. The constructed volumes are used to decide whether or not a polygon is close enough to have an influence on the shadowing of an arbitrary surface point. The image is originally published in [McG10].

fragment shader for every fragment that is within the volume and thus may be darkened by the polygon. The good thing about this idea is that it reduces the amount of computations needed per fragment. Typically there is only a small subset of fragments influenced by a polygon and this technique allows to find this set quickly. The actual computation is done by analytically resolving the occlusion caused by the polygon using the form factor method. It does introduce artefacts at points where a lot of flat objects are close to eachother. In this case the occlusion is not resolved correctly and points may be shaded darker than they should. However, the quality is still very high and comes close to raytraced solutions. The author mentions that the speed of the algorithm is fast enough for interactive applications but may not be as fast as other screen space methods.

2.5 Unclassified

There is another approach worth mentioning that does not fit into the chosen classification. In [Eva06] *signed distance fields* are used to approximate the look of ambient occlusion. A signed distance field is a function that returns the distance from an arbitrary point in space to the closest point on the surface of a model. This mans that the distance field yields a value of 0 for all points located directly on the surface. Moving away from the surface the value increases based on the chosen distance metric. The paper discusses an interesting finding. An error is made when reconstructing the distance field from a discrete set of sampling points using interpolation. This error gets higher the bigger the curvature of the surface. Instead of yielding a value of 0 on the surface the values differ depending on the curvature. Using this difference allows to shade creases and crevices darker than parts that are on the flat of a surface. What is necessary to make this technique work is to compute the distance field for a scene and store it as a set of discrete sampling points in a 3D texture. At runtime the distance field value is computed for the points on the surface by using interpolation of the nearby values in the texture. The paper shows how to compute the signed distance field directly on the graphics hardware which makes it fast enough to be used for dynamic scenes. Depending on the resolution of the 3D texture, big scenes with small details may not be captured very well by this technique.

It is also worth mentioning that ambient occlusion may also be used in non-polygonal rendering frameworks. As an example there is a paper that deals with computing this type of illumination in volume rendering applications [RMSD*08]. The presented technique requires a precomputing step and is able to apply the solution at interactive frame rates.

2.6 Cube map approach

In this work some of the proven methods are taken and combined with an idea to form a new approach to ambient occlusion. Screen space methods are currently among the fastest techniques capable of solving this problem. In addition the screen space methods typically do not need any form of precomputation which makes them suitable for use with fully dynamic scenes. While they are fast enough to be computed at real-time framerates a lot of redundant computation is carried out. This work tries to explore a way of storing previously computed results and retrieving them more quickly. For this purpose, screen space methods are used to compute ambient occlusion at discrete positions in space. The results are stored in textures which can then be used to quickly restore the results and apply them on a scene. An advantage of this is the ability to use higher quality settings which would not be fast enough to recompute every frame. In a way this

resembles the idea of spherical harmonic lighting where illumination coefficients are used for quick retrieval of lighting information. The difference is that the spherical harmonics coefficients take a lot of time to compute. In this work a framework is derived that works with uncompressed lighting information at the expense of a higher memory consumption. The resulting lighting information can be computed and retrieved quickly at run time.

Chapter 3

Theory

The goal of this work is to derive a framework that allows to compute ambient occlusion for the current view, but also stores the results in a way that makes it possible to retrieve the solution quickly in consecutive frames. The motivation of this comes from the fact that, as long as the camera does not move too much, most of the previously computed information might be transferred from one frame to the next. It would be a waste of resources to compute the same solution more than one time in case it was possible to simply retrieve the previous solution.

Screen space ambient occlusion techniques are suited for computing a solution for the current view but usually do not store the results in any way. Instead the computation is repeated for every frame. After the computation is finished the results are displayed on screen. Then they are discarded and recomputed from scratch in the next frame. In this work cube maps 3.1.3 are used to store previously computed results and apply them in consecutive frames using projective texturing 3.1.4. It is important to note that this is only possible because ambient occlusion is view independent and can therefore be computed without taking into account the position and orientation of the camera.

In real-time applications the amount of doable computations is very limited due to the time contraints. In order to achieve 24 frames per second for example the computation needs to be finished after 42 milliseconds in each frame. Due to this contraint sacrifices often have to be made in the quality of the solution to get the results fast enough. Often these sacrifices result in visible artefacts. In order to fix this it is common to blur the resulting images. This way artefacts can be concealed. As blurring has the effect of removing high frequency signals, like certain types of noise, this only works when the resulting image is supposed to be of low frequency. This is the case with ambient occlusion which is why it can be used here. This work makes use of a certain type of blurring filter, called *bilateral filter*. This kind of blurring filter has the property of preserving edges which is important in the context of this work. It is fast enough to be applied to the computed solution in a post-processing step and it enhances the quality of the outcoming image.

3.1 Foundations of Theory

3.1.1 The Rendering Equation

As mentioned before, an important base of realistic lighting simulation is the rendering equation. The equation itself has its foundation in the research of radiative heat transfer. It attempts to approximate the Maxwell equations for electromagnetics using geometrical optics and has proven to produce excellent results when used in the context of photo realistic rendering. The equation and the accompanying description of terms is taken from [Kaj86] and looks as follows.

$$I(x,x') = g(x,x') \left[\varepsilon(x,x') + \int_S \rho(x,x',x'') I(x',x'')\, dx'' \right] \tag{3.1}$$

$I(x,x')$ is the intensity of light passing from surface point x' to x. $g(x,x')$ is the so called geometry term. It becomes 0 in case there is no direct line of sight between x' and x. If there is a line of sight then the term becomes $\frac{1}{r^2}$ where r is the distance between the 2 points. $\varepsilon(x,x')$ describes the energy emitted by a surface at point x' reaching a point x. $\rho(x,x',x'')$ is the intensity of light that is originating at x" in direction of x' and scattered by x' to reach x. Finally there is the recursive term $I(x',x'')$ which accounts for light bouncing from one surface to another. The integration term is carried out on the set S that includes all parts of the surface of a scene. The equation states that the transport intensity of light from one surface

point to another is simply the sum of the emitted light and the total light intensity which is scattered toward x from all other surface points [Kaj86].

It is worth noting that there are phenomena in real life that cannot be modeled using the rendering equation. Properties like a changing phase or the wavelength of the light are not contained in this model. This means for example that interference, e.g. 2 waves of light cancelling out each other, is not possible. Also the refractive index of the media the light travels in is not taken into account. Another phenomenon left out is subsurface scattering where light penetrates the surface of an object, travels within the object and exits at another position. Extensions exist for some of these missing elements, but even without them the equation produces pleasing results for the simulation of light transfer.

Solving this equation in real-time is more or less like a holy grail in rendering. It is currently not feasible to compute all terms for all points of nontrivial scenes in real time. One way to work around this problem is to replace the exact computation with quicker approximations for some parts of the equation. Leaving away the recursive part for example lowers the computation complexity drastically. Unfortunately this change has a heavy impact on the outcome of the computation in terms of visual quality. A better trade-off is to take into account the first two or three recursion steps. At one point in the recursion the transferred energy may become small enough for the human eye to notice the difference between adding or not adding additional terms. Another possible approximation is to leave away parts of the environment that are too far away from the current point of computation. Again the intent behind this idea is that the energy transferred from these parts is too low to make a noticable difference.

A different approach to solving the rendering equation is to compute certain parts beforehand and store them in an efficient and easy to reconstruct way. For static geometry for example precomputed radiance transfer has become a popular method. By help of a clever rearrangement of the different terms it is possible to move most of the computation to a preprocessing step and use a very simple computation at runtime to construct the solution. What it does is to precompute the geometry term, multiplied with a cosine factor. The resulting data is encoded efficiently by use of basis functions like *spherical harmonics*. In the context of PRT, spherical harmonics are also used as means to describe the lighting conditions of a scene using only a small amount of coefficients. At runtime the solution of the equation can be computed as the sum of the product of two sets of coefficients. These are the PRT coefficients and coefficients derived from the light source. As mentioned before, the downside of this is that it requires a complex preprocessing step. Currently this is only a viable solution for static scenes or scenes with restrictions on the animations.

3.1.2 Ambient Occlusion

As mentioned earlier, ambient occlusion is computed from the geometric properties of a scene. The more a point is obscured by surrounding geometry, the darker it is shaded. Essentially this method is focusing only on the geometry term of equation (1.1) while assuming perfectly diffuse lighting conditions with equal amounts of light coming in from all directions. Another way to visualize this model is to imagine being placed in the point of interest looking up in the sky. The more parts of the sky can be seen the more open the point is and the less it is obscured. More formally what was just called sky is replaced by a hemisphere that is aligned with the surface normal. What needs to be done is to sample the hemisphere and try to find for every possible direction if there is an intersection with geometry or if a point has a clear sight to the sky. To solve this for a point P and a single direction ω the following function is used.

$$L(P,\omega)$$

The function returns the distance to the closest intersection of a ray emitting from point P in the direction of ω.

Another observation is that geometry which is far away does not have a big influence on the shadowing. Imagine the car from the previous examples being lifted in the air 1000 metres. The shadow below it will have completely vanished. Also the perception of the shadow will not be any different if the car is at 1000 metres or at 999. It seems that at one point if the distance is big enough the influence of the according object can be neglected. This can be expressed by adding a weighting function around L. Note that this function ρ is not related to the identically named function in the rendering equation. For consistency reasons these functions are named in the way they are named in their original publications.

$$\rho(L(P,\omega)) \tag{3.2}$$

In [IKSZ03] the following properties are derived for the function ρ. It should be a monotonically increasing function. This way a higher distance guarantees a lower influence. It should be 0 for intersection distance 0 and converge to 1 for large distances. Also its derivative should be monotonically decreasing. This way the function fades smoothly to the target value. The following function is suggested.

$$\rho(L) = 1 - e^{-\tau L} \tag{3.3}$$

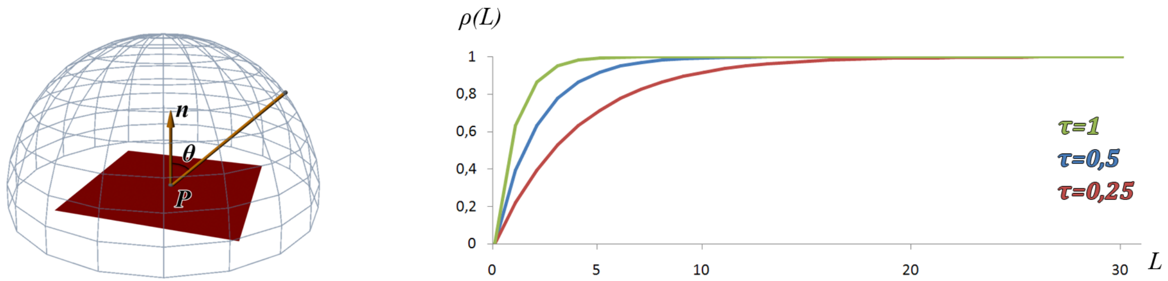


Figure 3.1: Left: Depiction of a hemisphere centered around point P aligned with normal n. θ denotes the angle between n and a ray in direction ω. Right: Illustration of the shape of the weighting function $\rho(L)$.

Figure 3.1 gives an idea what ρ looks like in general. Note that this expression can be a bit confusing because a value of 1 means that the influence of an object is to be neglected. A value of 0 means that it is to have a high influence. In practice one might use $1-\rho$ instead. Also it may be feasible to use a different kind of function but this one has proven to work.

Using the previous equations the ambient occlusion for a point P can now be expressed as follows.

$$W(P) = \frac{1}{\pi} \int_{\omega \in \Omega} \rho(L(P,\omega)) \cos\theta \, d\omega \tag{3.4}$$

Ω denotes the hemisphere aligned with the surface normal in point P. The integral is computed for all directions ω in the hemisphere. θ denotes the angle between the surface normal and the ray in direction ω. Taking the cosine of this angle ensures that more weight is given to the part of the hemisphere directly above the point P. Objects that are close to the horizon do not have as much influence on the shadow. The $\frac{1}{\pi}$-term ensures that the maximum value cannot exceed 1.

There is an alternative definition that is also used in a lot of publications. It looks as follows.

$$W(P) = \frac{1}{\pi} \int_{\omega \in \Omega} V(\omega)(n \cdot \omega) \, d\omega \tag{3.5}$$

In this equation the ρ function is basically replaced by the $V(\omega)$ term. $V(\omega)$ is the so called visibility function. It becomes 1 in case the ray emitting from point P in direction of ω is hitting any scene geometry and 0 in the other case. This is identical to the other definition with ρ being replaced by a step function that is 1 for all L >= 0 and 0 otherwise. The final difference is the notation of the cosine term that is replaced by the dot product between the surface normal n at point P and the direction of the current ray. Both notations for this term result in the same outcome.

3.1.3 Cube Mapping

Cube mapping is a form of *environment mapping*. The idea of this technique is to project the environment around a point P onto the 6 sides of a cube. This can be done straight forward by setting up a camera with a field of view of 90 degrees looking at the direction of each of the 6 principal axes $+X$, $-X$, $+Y$, $-Y$, $+Z$, and $-Z$. Figure 3.2 shows what the sides of a cube map may look like in an example. A cube map is usually represented as a set of textures containing the images of the environment.

Cube maps can be used for a number of rendering effects that need to sample the environment around a point. Reflections on specular surfaces are one of the most common examples. Instead of drawing the color of the surface material one computes the direction of the reflected ray and uses it to look into the cube map. Finally one can apply the colour of the environment on the specular surface to make it look like the environment is mirrored on it. Some more information on this topic is given in [AMHH02].

Next to cube mapping there are a few other methods to employ environment mapping. Most notably there is *sphere mapping* and *dual paraboloid mapping*. Compared to cube mapping a lot more effort is needed to generate these maps. It is necessary to warp the scene by help of shaders before rendering it into a texture. Depending on the warping scheme this may introduce unwanted artefacts when the vertex resolution is not high enough for the warping to be accurate. More information on these two methods can also be found in [AMHH02].

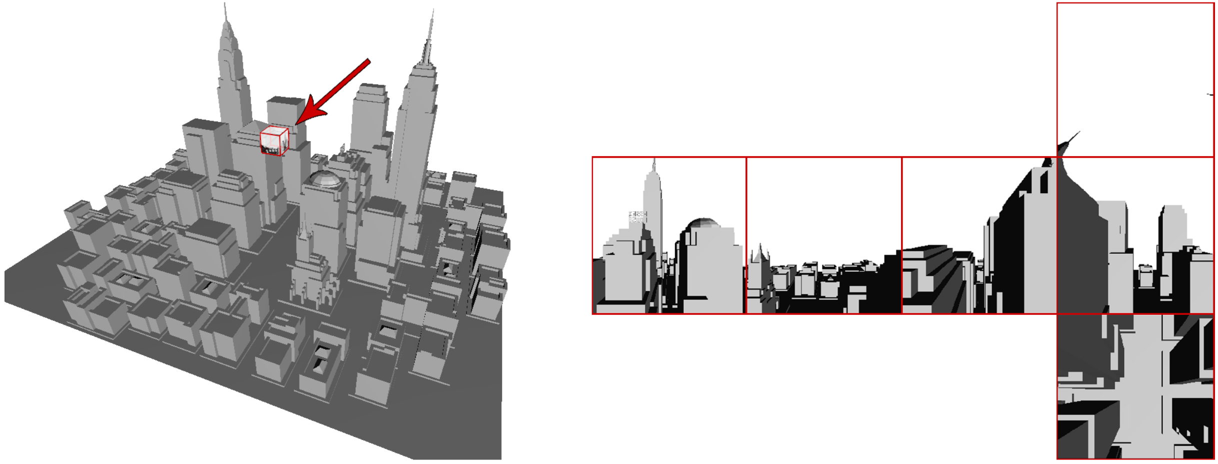


Figure 3.2: Visualization of a cube map. The cube that can be seen in the left image is just drawn for better understanding. Usually it is not shown directly in a rendering. Instead its data, which consists of the 6 textures shown on the right, is used for various special effects.

3.1.4 Projective Texturing

Projective texturing just as the name suggests allows to project the contents of a texture onto surfaces. The effect looks similar to a slide or video projector in the real world. Figure 3.3 shows an example of a projected texture in a rendered image. What this can be used for in the context of this work is to project additional light and shadow information onto a scene. For example one could capture the shadow that is cast by a pole in a texture and project it onto the floor. Similarly one can also compute ambient occlusion, render it to a texture, and then use projective texturing to apply the ambient occlusion to a part of the scene. This is exactly what is done in this work. Instead of using a single texture, a cube map is used to store the computed ambient occlusion and this map is projected onto the scene. This way the entire environment as seen at the position of the cube map appears to have ambient occlusion lighting. It is only when the camera starts translating that the effect starts to vanish. However it is possible to recompute the cube map quickly at another camera position.

3.1.5 Bilateral Filtering

Bilateral filtering is an extension to gaussian linear filtering, which essentially has the effect of removing high frequencies from signals. In case of images, the application of a gaussian filter results in blurring the image and removing details. This is done by replacing the intensity of a pixel by a weighted average of the surrounding pixels. In this work it is used to remove high frequency artefacts from the computed solution. The reason to choose a bilateral filter over other blur filters is the important property of preserving edges. Figure 3.4 shows the effect a gaussian linear filter has on an image and the effect a bilateral filter has.

A very good and detailed report about this type of filter can be found in [PKTD08]. First it contains all the necessary mathematic formulations in an easy to understand way. In addition to this it also gives some interesting fields of application as well as a number of approaches for efficient implementations. The mathematic formulation is given in the following equation.

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\|p-q\|) \cdot G_{\sigma_r}(I_p - I_q) \cdot I_q \tag{3.6}$$

with W_p being a normalization factor and G_σ being the gaussian function.

$$W_p = \sum_{q \in S} G_{\sigma_s}(\|p-q\|) \cdot G_{\sigma_r}(I_p - I_q) \tag{3.7}$$

$$G_\sigma(x) = \frac{1}{2 \cdot pi \cdot \sigma^2} \cdot e^{-\frac{x^2}{2\sigma^2}} \tag{3.8}$$

The various terms of this equation are explained in the following. First, the evaluation of the filter is performed on image I at the pixel of position p. The sum is carried out across the set S which contains all pixel positions in the

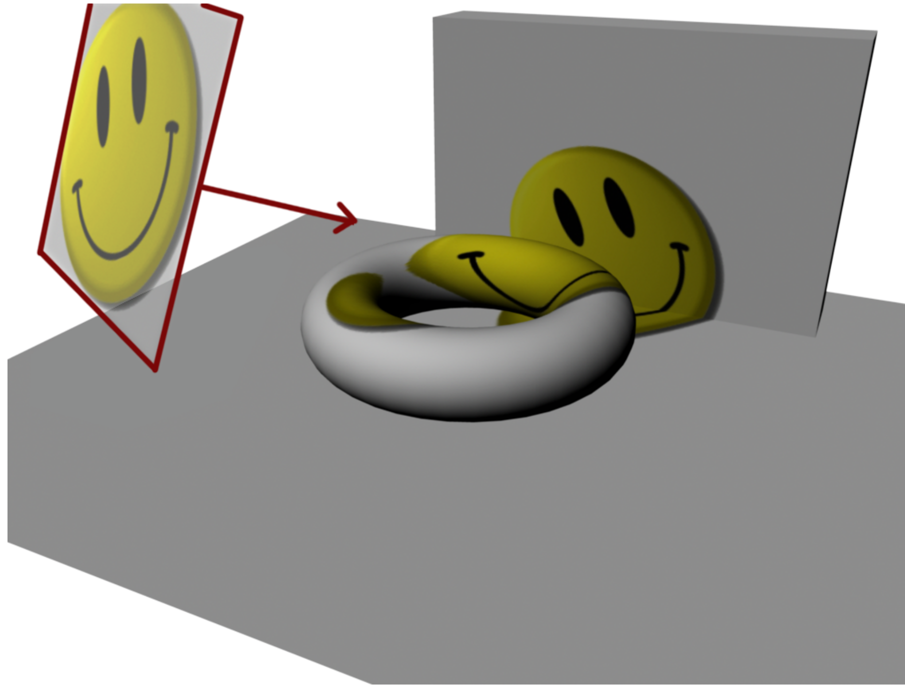


Figure 3.3: Sample image of what a projected texture can look like. The texture of a smiley face is applied on a simple scene. The behaviour is essentially like using a video projector. A difference that is worth mentioning is that the projected texture may pass through all solid objects also appearing on faces that would not be lit by a real world projector. Methods exist that allow to adapt this behaviour in case it is unwanted.

Figure 3.4: First image: Photo of a car with a bit of noise added. Second: The result of blurring the photo with a gaussian filter of size 21 x 21 and a parameter $\sigma_s = 3.0$. The noise is gone but so is most of the detail as well. Third: Result of a bilateral filter of size 21 x 21 with $\sigma_s = 3.0$ and $\sigma_r = 32.0$. The noise is reduced greatly while the edges of the details like the building or the tree leaves are preserved.

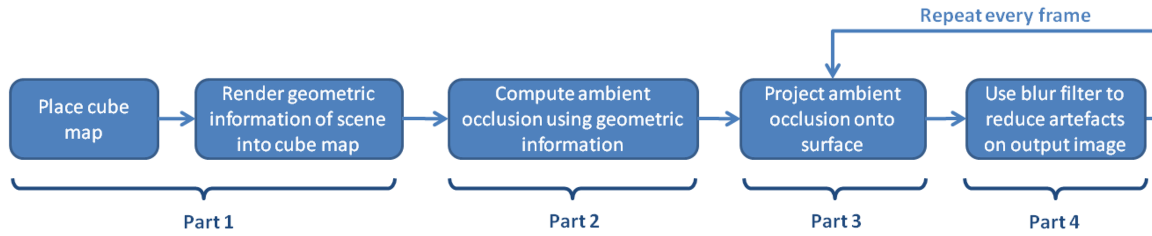


Figure 3.5: Very rough sketch of what the core algorithm described in this work looks like on a very high level.

image. G_{σ_s} is a symmetric 2-dimensional gauss function. The parameter σ_s defines the extent or range of influence of the function. Higher values lead to a more blurry result. To this point the equation corresponds almost to a regular gaussian blur filter. It is the $G_{\sigma_r}(I_p - I_q)$ term that makes it differ. What this does is make sure that a neighbouring pixel I_q only has a considerable influence on the sum in case the image intensity is similar to the intensity of the pixel I_p. Neighbouring pixels that differ too much in intensity have a lower influence than others that are of similar intensity. The intention behind this is as follows. Edges in an image are typically accompanied by a considerable difference in intensity or colour. Instead of adding the colour of one side of the edge to the colour of the other side, it is neglected or given a low influence. This way edges remain as sharp features, separating different parts of an image while at the same time blurring parts that belong together.

3.2 Cube Map Ambient Occlusion

Given these principal building blocks a framework can be derived that allows to compute ambient occlusion for selected parts of a scene, store the results and apply them in consecutive frames. Figure 3.5 shows a rough sketch of what the algorithm looks like. First a cube map is placed at an arbitrary position in the scene. Preferrably this position is selected to be close or identical with the current position of the camera. Instead of rendering colour information into the cube map, geometric information is stored in each fragment of a cube map. More precisely the world position and the surface normal of each fragment is stored. This is done by help of a shader program that encodes the information into the channels of a floating point render target.

Using this geometric information another shader is able to compute screen space ambient occlusion for all 6 sides of the cube map. Finally the contents of the cube map are projected back onto the surface of the scene and a blur filter is used to reduce artefacts in the output image. The result is a rendering that contains soft ambient occlusion shadowing that is layered on top of the existing illumination.

Chapter 4

Implementation

The following will give a detailed insight into the implementation of the cube map ambient occlusion software library. First there is an overview of the general design of the implementation. This is followed by four sections that correspond to the parts 1, 2, 3, and 4 in figure 3.5. After this there are additional sections that describe an automated cube map placement scheme and the interface to the software library resulting from this work.

4.1 Software Architecture

There are a few important classes that mainly define how the software architecture of the library looks like. Figure 4.1 shows these classes in a diagram along with their most important functions and members.

As mentioned briefly before, the interface to the library is designed using the bridge pattern. The *AmbientOcclusion* class is the only class the client is communicating to and serves as the interface. The calls to the interface are forwarded to *AbstractBackbone* which contains the main important methods and members needed for running the computation. *BackboneCubemapped* inherits from this class and implements the abstract methods using the cube map approach which is explained later in this chapter. This design allows to exchange the concrete implementation for a different strategy at runtime. Currently however there is only one implementation in use. The cube mapped implementation contains a vector of *CubeMapProjector* objects. Each time the user adds a cube map a new object is created and added to this vector. Subsequently the computation of the ambient occlusion for this cube map is started by rendering geometric information for all sides of the cube map. This involves making use of the *DeferredShading* class which runs the necessary algorithms and shader programs. Once this computation has finished for the cube map, the BackboneCubemapped class may access the resulting data and project it onto the screen. The *RenderToTexture2D* as well as the *GLSLShader* class basically encapsulate OpenGL related features for easy use by the software library. Both classes are used extensively by the backbone and deferred shading classes.

4.2 Cube Map Rendering

The goal of the first part is to capture the environment at a particular position and store geometric information about it in a cube map. In order to do this one needs to set up the 6 render targets that form the sides of the cube. In this work 6 ordinary 2D textures are used. OpenGL also allows the use of a dedicated cube map texture object. However, experiments show that the use of this does not improve rendering performance in this specific application. This is why regular 2D texture objects are used here.

4.2.1 Placement of cube maps

The implementation currently supports two ways of placing cube maps. The first one is to simply instruct the library to manually place a cube map at a position given in world space coordinates. This method does not really need any explanation.

The second way is based on an automated measuring of the percentage of the screen that is currently covered with ambient occlusion illumination. In order for this technique to work it needs to use the outcome of the previous frame as input for the next frame. A detailed description of how this works is given at the end of this chapter. It is based on the

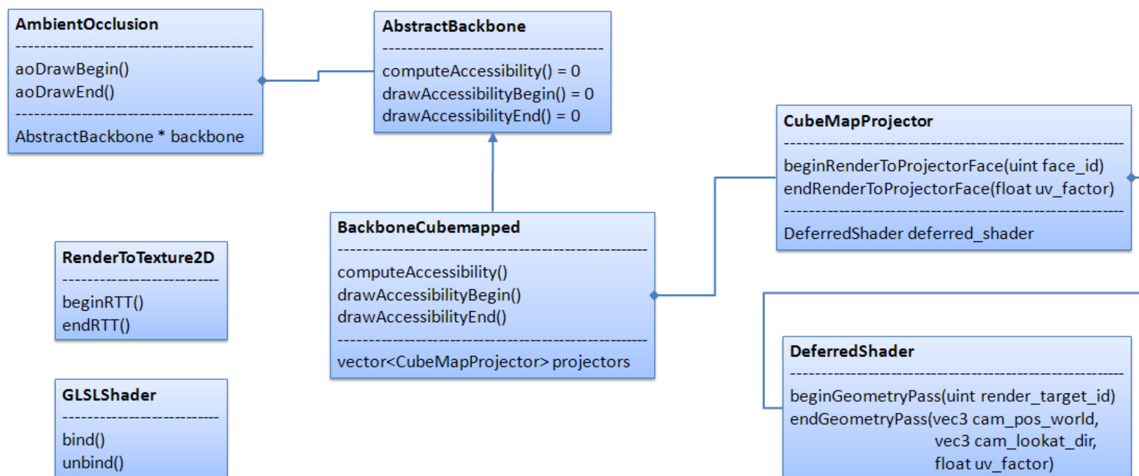


Figure 4.1: Depiction of a simplified class diagram of the most important classes in the resulting software library.

outcome of the ambient occlusion computation which is described first. The following is a quick sketch of what needs to be done.

At the beginning of the first frame there is no previous solution available. The algorithm finds that there is not a single pixel on screen that is shaded with ambient occlusion. It is decided that a new cube map is placed at the current position of the camera. In the next frame it is found that all visible pixels are shaded so no cube map needs to be placed. Eventually the user may move around the camera which causes a decrease in the percentage of the shaded pixels. Once this value is below a certain threshold another cube map is placed at the position of the camera.

This is how it works on a high level perspective. In the current implementation the pixel threshold for the automated placement is set to a value of 75%. Once the percentage of ambient occlusion shaded pixels is lower, a cube map is placed and the computation starts by rendering the environment into it. Implementation details for this technique are given later in chapter 4.6.

4.2.2 Render to Texture

The most efficient way to get the output of a render pass into a texture in OpenGL is by making use of *framebuffer objects* (FBO). Instead of rendering into the framebuffer this technique allows to bind a regular texture as target to render into. In order to make use of this feature the necessary OpenGL calls and variables are encapsulated in a separate class. Wherever it is necessary to render to a texture an instance of this class is used. The following shows the constructor of the class.

Listing 4.1: RenderToTexture2D constructor.

```
RenderToTexture2D(unsigned int num_texture_targets,
                  const unsigned int texture_resolution_x,
                  const unsigned int texture_resolution_y,
                  const EFormat texture_format,
                  const EWrapMode texture_wrap_s,
                  const EWrapMode texture_wrap_t,
                  const EFilterMode texture_min_filter,
                  const EFilterMode texture_mag_filter,
                  bool no_depth_buffer
                  );
```

It may look complex but it is really all the info that is typically needed. The names of the parameters are self explanatory and correspond largely to the properties of typical OpenGL textures. There are only two exceptions to this. Firstly it is possible to have the object contain more than one render target. This is indicated by the first parameter *num_texture_targets*. Secondly the *no_depth_buffer* parameter allows to control whether or not the render target should also contain a depth buffer. Leaving the depth buffer away allows to save memory. However it may result in incorrect renderings in case it is left away incorrectly. The rest of the parameters are duplicates to OpenGL texture properties. It is

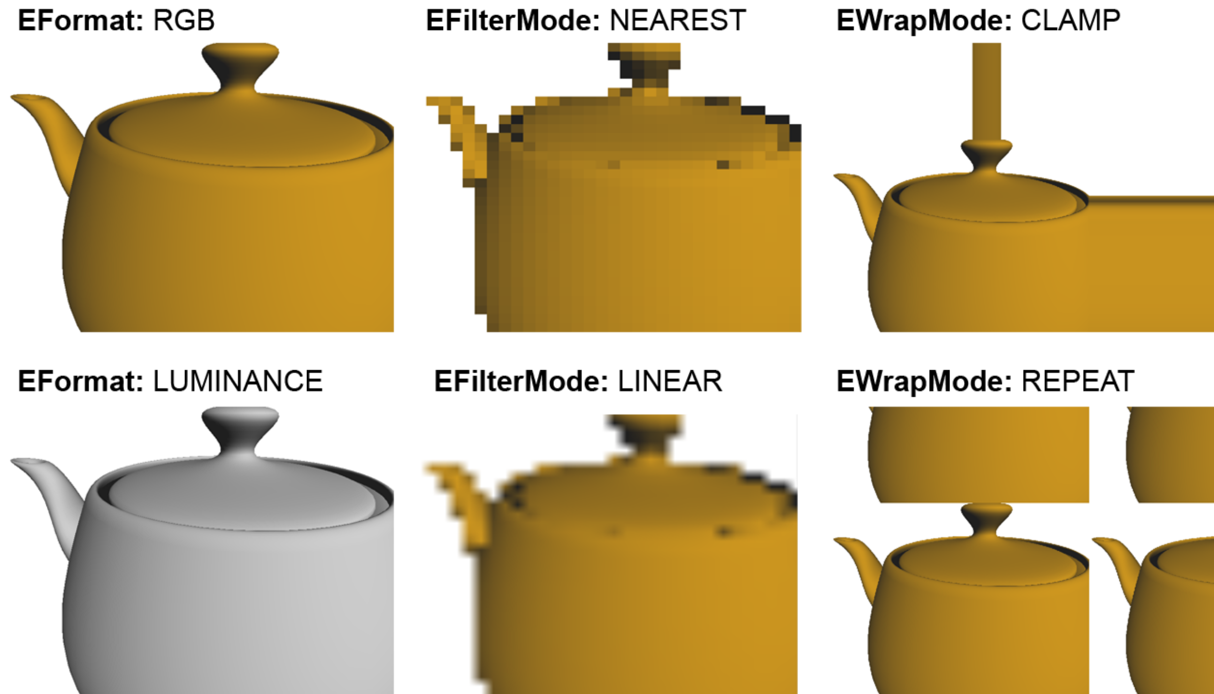


Figure 4.2: Examples of the various different settings that can be applied to RenderToTexture2D objects. There are additional formats to allow for floating point render targets as well as targets that contain alpha channels to support transparency. These formats are not shown in this image. Note that, to show the effect of filtering, the resolution of the texture was heavily decreased in the according two images in the middle.

worth noting that the format, wrap- and filter mode are declared as independent enumerations in the header of the same class. They are translated to the OpenGL types and enumerations on the implementation side of the class. This makes the interface independent from the graphics API. In case the implementation is changed to make use of a different API, users do not need to change existing code that makes use of this class. The enumerations are given and discussed in the following.

Listing 4.2: RenderToTexture2D enumerations.

```
enum EFormat
{
  RGB = 0, RGBA, LUMINANCE, LUMINANCE_ALPHA, R32F, RGB32F, RGBA32F
};
enum EWrapMode
{
  CLAMP = 0, REPEAT
};
enum EFilterMode
{
  NEAREST = 0, LINEAR
};
```

As can be seen the available texture formats range from one to four channel textures and also support floating point render targets. The names are similar to their OpenGL counterparts. The same thing applies to the texture wrapping and filter modes. These properties become important when binding the render targets as textures to read data from them. When addressing areas outside the texture, a clamped texture basically returns the colour value of the texture at its border. Repeat makes the texture tile over and over. See figure 4.2 for an example of this behaviour. Setting the filter mode to nearest basically disables any filtering of the texture values. A linear filtering takes care of returning the addressed texture value as a linear combination of the four surrounding texture elements.

There appears to be a lack of quantity of literature dealing with the subject of setting up framebuffer objects using OpenGL and C++. Good sources are rare which is why this topic is discussed in detail in this thesis. Internally, when

creating a render to texture object the following initialization happens. Note that this piece of code makes use of the *GLEW* library. GLEW stands for OpenGL extension wrangler and allows to make use of OpenGL extensions more easily. Also note that the code fragment makes use of class members that are not defined in the listings.

Listing 4.3: Creation and initialization of a framebuffer object.

```
// Create framebuffer object. Store handle in frame_buffer_object_.
glGenFramebuffersEXT(1, &frame_buffer_object_);

// Generate textures. Store their IDs in texture_ids_.
texture_ids_ = new GLuint[num_texture_targets];
glGenTextures(num_texture_targets, texture_ids_);

// Bind the FBO.
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, frame_buffer_object_);

// Initialize the textures and attach them as render targets to the FBO.
for(size_t i = 0; i < num_texture_targets; i++)
{
  glBindTexture(GL_TEXTURE_2D, texture_ids_[i]);
  glTexImage2D(GL_TEXTURE_2D, 0, texture_format_,
               texture_res_x_, texture_res_y_, 0, texture_format_,
               GL_UNSIGNED_BYTE, 0);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                  texture_min_filter_mode_);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                  texture_mag_filter_mode_);
  glTexParameteri(texture_target_type_, GL_TEXTURE_WRAP_S,
                  texture_s_wrap_mode_);
  glTexParameteri(texture_target_type_, GL_TEXTURE_WRAP_T,
                  texture_t_wrap_mode_);

  // Attach texture to framebuffer color attachment.
  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT +
                            (GLenum)i,GL_TEXTURE_2D, texture_ids_[i], 0);
}

// Initialize and attach a depth render target if requested.
if (no_depth_buffer_ == false)
{
  glGenTextures(1, &texture_depth_);
  glBindTexture(GL_TEXTURE_2D, texture_depth_);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

  glTexImage2D(texture_target_type_,0,GL_DEPTH_COMPONENT,texture_resolution_x,
               texture_resolution_y, 0,GL_DEPTH_COMPONENT,GL_UNSIGNED_BYTE,0);
  glBindTexture(texture_target_type_, 0);

  // Attach the texture to FBO depth attachment point.
  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
                            GL_TEXTURE_2D, texture_depth_, 0);
}

// Unbind FBO for now.
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
glBindTexture(GL_TEXTURE_2D, 0);
```

In order to enable and disable rendering to the framebuffer object the following lines of OpenGL code are necessary.

Once it is enabled, all render outputs get written to the attached texture.

Listing 4.4: Enabling and disabling a framebuffer object.

```
// Enable FBO.
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, frame_buffer_object_);
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT + target_id);

// Disable FBO.
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

Another useful thing is the ability to render to *multiple render targets* (MRT) in a single pass. The initialization code is already given above. In order to enable the feature the following code needs to be executed before rendering.

Listing 4.5: Enabling a framebuffer object with multiple render targets.

```
// Enable FBO with multiple render targets.
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, frame_buffer_object_);

// Store the IDs of the attachment points of all textures in an array.
GLenum * targets = new GLenum[num_render_targets];
for (size_t i = 0; i < num_render_targets; i++)
  targets[i] = GL_COLOR_ATTACHMENT0_EXT + first_target_id_offset + i;

// Pass the array of IDs to OpenGL.
glDrawBuffers(num_render_targets, targets);
delete [] targets;
targets = NULL;
```

Disabling is identical to the listing above for single render targets. This is basically all the code that is necessary to set up a framebuffer object and use it to render directly to one or more texture targets. As mentioned before, all of this functionality is contained in the RenderToTexture2D class for easy use. Here is an example of how simple it is to make use of the render to texture functionality by help of this class.

Listing 4.6: Using the RenderToTexture2D class to redirect rendering output to a texture.

```
// Create a render to texture object.
rtt::RenderToTexture2D my_render_target(1, 512, 512,
                                        rtt::RenderToTexture2D::RGB,
                                        rtt::RenderToTexture2D::REPEAT,
                                        rtt::RenderToTexture2D::REPEAT,
                                        rtt::RenderToTexture2D::NEAREST,
                                        rtt::RenderToTexture2D::NEAREST);
// Enable the FBO.
my_render_target.beginRTT();

// Render something using OpenGL.
glutSolidSphere(1.0, 3, 3);

// Disable the FBO.
my_render_target.endRTT();
```

The class allows to access the data produced by writing to the texture. Using the class interface it can be bound easily as input texture for a successive render step for example.

4.2.3 GLSL shader programs

In this work it is not the goal to render regular images to the cube map textures. Instead the textures are used to store geometric information like the world coordinates of a fragment or its surface normal. In order to be able to do so it is necessary to make use of shader programs. This work is based on using the official *OpenGL shading language* (GLSL). An alternative shading language which can also be used to achieve the same goal is Cg by NVIDIA. Again initialization and use of this functionality is encapsulated in a standalone class. Here is an example of how to make use of this class to enable a vertex and fragment shader program.

Listing 4.7: Enabling the GLSL functionality using the standalone class.

```
// Initialization .
gs::GLSLShader my_shader;
my_shader.attachAndCreateProgram("vertex_program.txt", "fragment_program.txt");

// Enable.
my_shader.bind();

// Render something using OpenGL.
glutSolidSphere(1.0, 3, 3);

// Disable.
my_shader.unbind();
```

A detailed description on how to set up GLSL is left out here as it is documented extensively in the literature. One may refer for example to the official book on this topic, *OpenGL Shading Language* [Ros06]. The GLSL wrapper class used in this thesis works together seamlessly with the render to texture class. In this way it allows to render data to a texture while having custom shader programs enabled.

4.2.4 Cube map setup

What is left to do to render a cube map at runtime is to set up the camera properly. In a simple solution it is necessary to do 6 render passes with the camera looking at all 6 cardinal directions of the cube map. The final issue that needs to be taken care of is to set the field of view of the camera to 90 degrees both in the horizontal and vertical axis. This setup allows to render the 6 sides of the cube map so they fit together seamlessly. The following code listing shows how to set up a camera looking along the positive x axis using OpenGL.

Listing 4.8: Setting up the OpenGL camera for use with the +X side of a cube map.

```
glMatrixMode( GL_PROJECTION );
glPushMatrix();
glLoadIdentity ();
gluPerspective( 90, 1.0, 0.1, 1000.0); // Fov, fov-ratio, near-plane, far-plane.
glMatrixMode( GL_MODELVIEW );
glPushMatrix();
glLoadIdentity ();
gluLookAt(cube_map_pos.x, cube_map_pos.y, cube_map_pos.z,
          cube_map_pos.x+1.0, cube_map_pos.y, cube_map_pos.z,
          0,1,0);
```

In contrast to reflection or environment mapping it is not the colour of the environment that is rendered into the cube maps in this work. Instead the cube maps are used to store geometric information that allows to compute ambient occlusion from it. In this particular case the world position of the fragments and the surface normals are written into a cube map. Additionally the linear depth of the fragments to the cube map center is stored. In order to store all of this information a total of 7 floating point numbers are needed. 3 of them to contain the world space coordinates, 3 to contain the normal vector and one to contain the depth. Unfortunately the maximum number of values to store in a texture is 4. So it is necessary to render to 2 textures simultaneously to prevent unnecessary render passes. This is where the multiple render targets technique has to be used. In addition to setting up the framebuffer object properly a shader program is also mandatory. There is no other way to write to multiple render targets than using a custom fragment shader. Using the previously discussed render to texture and GLSL classes the code to do the geometry rendering looks as follows.

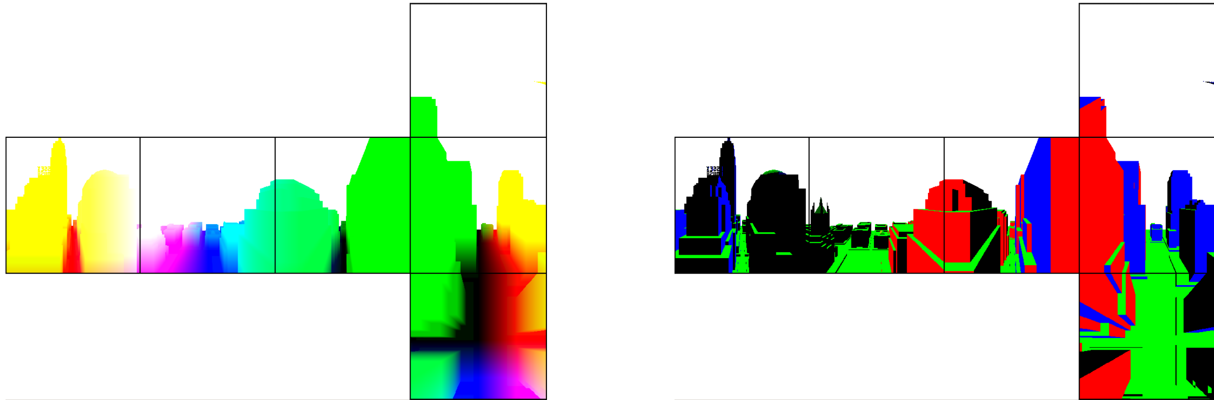


Figure 4.3: Left: Cube map containing three-dimensional world coordinates of a scene. XYZ coordinates are encoded using RGB colours. Right: Encoding of the surface normals in a cube map. A fragment depth value is also stored in the cube map but is not shown in this depiction.

Listing 4.9: Rendering geometric information of a scene into 2 render targets.

```
// Bind the framebuffer object with 2 textures starting at id 0.
mrt_render_target.beginRTMRT(0, 2);

// Bind the necessary shader.
mrt_shader.bind();

// Render something using OpenGL.
glutSolidSphere(1.0, 3, 3);

// Unbind shader and disable render target.
mrt_shader.unbind();
mrt_render_target.endRTMRT();
```

As mentioned before it is also necessary to use a fragment shader that takes care of writing to the two distinct texture targets. The following shows how to do this using the OpenGL shading language GLSL.

Listing 4.10: Fragment shader code to render to 2 separate render targets.

```
gl_FragData[0] = vec4(normal.xyz, depth));
gl_FragData[1] = vec3(world.xyz);
```

Here a three-dimensional normal vector and a floating point depth value are written into render target 0 and the three-dimensional world position is written into target 1. Figure 4.3 shows what the rendered cube maps look like. The data ends up being written to the two textures attached to the framebuffer object.

4.2.5 Computing fragment world positions

What needs to be discussed in more detail is how to get to the world space position of a fragment. First and foremost it is necessary to use a shader program to write this value. Unfortunately there is no direct access to the world position in a fragment shader. Instead it needs to be calculated from screen space coordinates which are accessible in the shader. Figure 4.4 shows the various different coordinate frames which exist in the OpenGL rendering pipeline. Using a number of matrices, objects are transformed from their local object space coordinates to the final screen space coordinates which are used to display an image on a two-dimensional screen. Given these matrices, one can undo the transformations by multiplying the screen space coordinates with the inverse projection matrix M_p and the inverse view matrix M_v. The following equation shows how to get from screen space coordinate F_s of a fragment to world space coordinate F_w.

$$F_w = M_v^{-1} \cdot M_p^{-1} \cdot F_s \tag{4.1}$$

The code listing below shows how this can be computed using a GLSL fragment shader. Note that $M_v^{-1} \cdot M_p^{-1}$ can be precomputed as it does not change for an entire side of a cube map. The data for this precomputed matrix is stored in

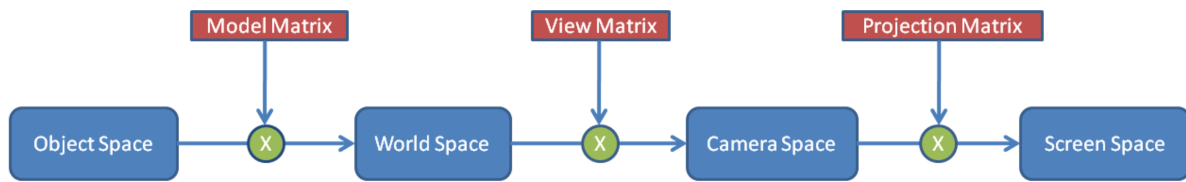


Figure 4.4: Recap of the various transformations that are passed through by a primitive when rendered in OpenGL. Input is a four-dimensional vector which describes an object in homogeneous object coordinates. Consecutively this vector is multiplied by the model-, view- and projection matrix to obtain screen space coordinates.

variable *IM* in the listing.

Listing 4.11: Fragment shader code to transform screen space coordinates to world space.

```
// Depth of the fragment in screen space.
// gl_FragCoord is a built in variable provided by GLSL.
float depth = gl_FragCoord.z;

// XY Position of the fragment in screen space. Remapped to the range [0,1].
// viewport_recip contains the screen offset from the corner (.xy) as well
// as the reciprocal resolution (.zw).
vec2 screen = (vec2(gl_FragCoord.x, gl_FragCoord.y) - viewport_recip.xy) *
                    viewport_recip.zw;

// Undoing the view and projection transformation.
// IM contains the inverted view-projection matrix.
// Before multiplication the screen space coordinates need to be
// remapped from [0, 1] to [-1, 1]. [-1, 1] is actually the native
// OpenGL screen space range for all axes. This is also the space the
// matrices operate on.
vec4 world_pos = IM * vec4(screen * 2.0 - 1.0, depth * 2.0 - 1.0, 1.0);

// De-homogenize.
world_pos.xyz /= world_pos.w;
```

Unfortunately OpenGL or GLSL do not provide the inverse matrix IM. Instead one needs to compute it manually. To do this the projection and the view matrix need to be obtained. For the projection matrix this is done using the following C++ code.

Listing 4.12: Retrieving the OpenGL projection matrix from a C++ application.

```
// Store the projection matrix in projection_matrix_data.
float projection_matrix_data[16];
glGetFloatv( GL_PROJECTION_MATRIX, projection_matrix_data );
```

The view matrix may be obtained under certain circumstances using the following code.

Listing 4.13: Retrieving the OpenGL model-view matrix from a C++ application.

```
// Store the view matrix in model_view_matrix_data.
float view_matrix_data[16];
glGetFloatv( GL_MODELVIEW_MATRIX, view_matrix_data );
```

As can be seen by the name of the enumeration used, OpenGL does not provide access to the plain view matrix. Instead it only grants access to the combined model-view matrix. Querying the view matrix this way only works in case there are no object to world space transformations in the current state of the OpenGL pipeline. In case this is not given, the matrix needs to be assembled separately by the user.

4.3 Ambient Occlusion Computation

There are a number of algorithms that are able to compute ambient occlusion from discrete world space and normal data. The main idea they have in common is to find how much area of the hemisphere is occupied by those discrete surface elements. In this work a horizon based implementation is used. This means that the geometric information of the screen fragments is interpreted as heightfield which has a horizon line that separates the empty background from the scene surface. The horizon can be used as an estimate of the ambient occlusion. It is discussed in more detail in [DBS08].

4.3.1 Horizon-split ambient occlusion

In this method the height of the horizon formed by the geometry of the scene is taken as measure for the amount of occlusion of the hemisphere at a specific point. See Figure 4.5 for various different samples of how surrounding geometry affects the horizon line. One can see easily that a high line with a steep angle means more occlusion of the hemisphere than a flat line.

The left side of figure 4.5 shows what the horizon line might look like for a single slice of a three-dimensional scene. In order to get to a true measure for the occlusion of the entire hemisphere one has to compute the horizon angle for all possible directions for a single point. The right side shows a depiction of what the hemisphere might look like when the horizon line and therefore the occlusion is computed for each direction.

Summing up all possible directions one can get a measure for the percentage of the hemisphere that is occluded by surrounding geometry. In [DBS08] it is shown that the ambient occlusion equation (3.4) can be restated to allow for a quick evaluation in a shader program. The following will briefly show how to get to this formula from the alternative original ambient occlusion definition (3.5).

First, the domain of the integral is changed to one that is similar to spherical polar coordinates. Parameter θ is already known, but z is basically the common parameter ϕ mapped from $[0, \pi / 2]$ to $[1,0]$. Furthermore, as mentioned in chapter 3.1.2 the more practical term $1-V$ is used instead of V. This way a resulting value of 1 means that the point P is to be shaded with a bright colour, while a value of 0 means that it is to be shaded with a dark colour. This makes sense in practice as the outcoming value can be used directly as colour or intensity value. Applying these changes results in the following equation.

$$W(P) = \frac{1}{\pi} \int_{\theta=0}^{2\pi} \int_{z=H(\theta)}^{1} (1-V(\omega))(n\cdot\omega)\,d\omega \tag{4.2}$$

This can be rewritten to the following equation.

$$W(P) = 1 - \frac{1}{\pi}\left(\int_{\theta=0}^{2\pi} \int_{z=0}^{H(\theta)} (n\cdot\omega)d\omega + \int_{\theta=0}^{2\pi} \int_{z=H(\theta)}^{1} V(\omega)(n\cdot\omega)d\omega\right) \tag{4.3}$$

Essentially the integral is broken up into two separate terms. The first term computes the result for all points below the horizon while the second term computes the result for all points above. Experiments in [DBS08] show that it is already a good approximation to just compute the first term. In order to cut down rendering times further the true horizon is also approximated by a piece-wise linear one. Figure 4.6 shows what this means. Using this approximation the integral can be turned into a sum and finally be computed using the following equation. The paper contains a slightly more detailed derivation that shows how to get from the first term in equation (4.3) to (4.4).

$$W(P) = \frac{1}{N_d} \sum_{i=1}^{N_d} H^2(\theta_i) \tag{4.4}$$

N_d is the amount of directions in which the height of the horizon is computed. The height H is computed by projecting the normalized vector that goes from the surface point to the horizon point onto the normal vector. Another way to get to H is to compute the sine of the horizon angle α. θ denotes the angle of the direction in which the height is computed. As mentioned before, the true horizon line is basically replaced by a piece-wise linear approximation where each piece is a completely horizontal line at height H.

The cube maps that have been rendered in the previous step contain all the necessary information needed to compute equation (4.4). What needs to be done is to find the biggest horizon angle for each direction. As mentioned before, this angle in turn allows to compute H. Figure 4.7 shows a fast and simple way to compute the horizon angle. The 3D problem can be reduced to a 2D problem once the direction θ is set. One can step along the axis x in discrete steps Δx and find the height of the point at this position. Given the size of the discrete step and the height allows to compute the horizon angle.

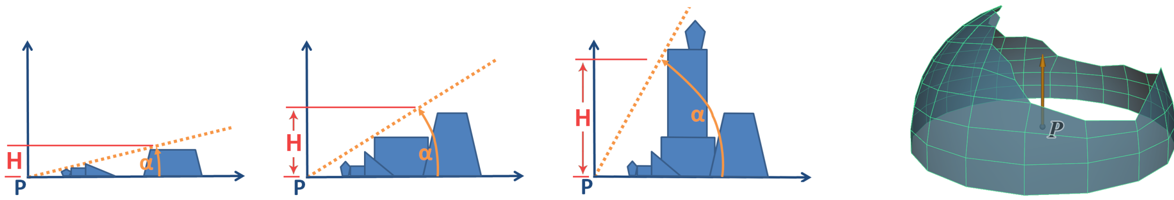


Figure 4.5: Left: 2D samples of the horizon line and horizon angle α for various different surroundings of a point P. The horizon line is drawn as dashed orange line. A steep line typically corresponds to a steep and occluded neighbourhood while a flat line indicates an open and unoccluded neighbourhood. H denotes the height of the horizon. It can be computed as $\sin(\alpha)$. Right: Depiction of what the horizon may look like for point in a 3D scene.

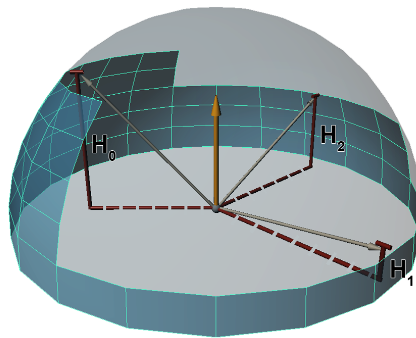


Figure 4.6: Depiction of an approximated horizon using piece-wise linear sections of constant height. In this example the horizon height H is computed for three directions θ.

In an iterative manner this process is repeated to find the biggest angle for this direction. This algorithm is also described in [BSD08].

The following code shows an implementation of this algorithm in a GLSL fragment shader.

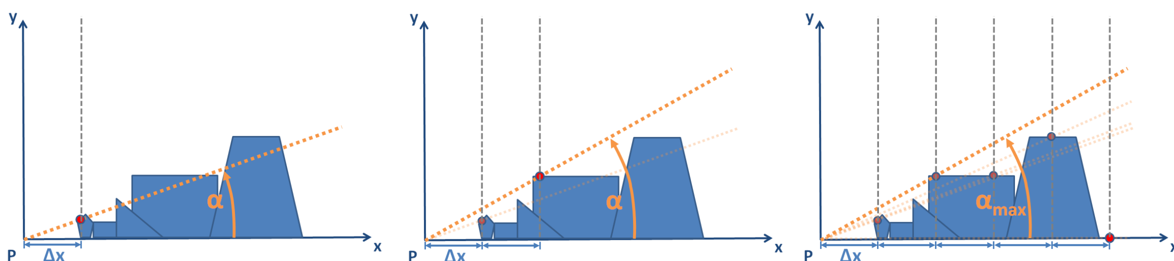


Figure 4.7: Discrete method to compute the horizon angle α. Stepping along axis x in discrete steps Δx one determines the height of the highest point at this position. This in turn allows to compute the angle α. Repeating this procedure and keeping only the highest value of α leads to the actual horizon angle α_{max}.

Listing 4.14: Computing the horizon angle for one direction using a fragment shader program.

```
//Variable to store the maximum horizon angle.
float normal_dot_horizon_steepest = 0.f;

//Loop along axis x.
const int num_steps_per_direction = 16;
for(int j = 0; j < num_steps_per_direction; j++)
{
  //Here an index is computed that allows to fetch the position and normal of
  //the scene at the sample position.
  vec2 emitter_uv_pos = receiver_uv_pos + (uv_sampling_dir * single_uv_step) +
                        (uv_sampling_dir * uv_step_size * ((float)j ));

  //Break in case the emitter is outside the valid range.
  //No useful data available there.
  if (emitter_uv_pos.x < 0  emitter_uv_pos.x > 1
   emitter_uv_pos.y < 0  emitter_uv_pos.y > 1)
    break;

  //Fetch the data of the scene at the sample position. The data was stored in
  //the textures tex0 and tex1 in the cube map rendering step.
  vec3 emitter_pos = texture2D(tex1, emitter_uv_pos).xyz; //World position.
  vec3 emitter_normal = texture2D(tex0, emitter_uv_pos).xyz;

  vec3 receiver_to_emitter = emitter_pos − receiver_pos;
  receiver_to_emitter = normalize(receiver_to_emitter);

  //Do not take into account backfacing elements.
  if (dot(−receiver_to_emitter, emitter_normal) < 0)
    continue;

  float normal_dot_horizon = dot(receiver_normal, receiver_to_emitter);
  if (normal_dot_horizon > normal_dot_horizon_steepest)
  {
    //We found a bigger horizon angle.
    normal_dot_horizon_steepest = normal_dot_horizon;
  }
}
```

The upside of this algorithm is that it is extremely easy to implement. The downside is the obvious error that is being made due to the discrete stepping. The real biggest horizon angle could be bigger than the one found. This can be seen in the second image of Figure 4.7. The horizon line should actually touch the upper left corner of the rectangle instead of passing through it a little below the top. The bigger the step size, the bigger the error to be expected. A tradeoff has to be made between accuracy and computation time. More on this topic is discussed in chapter 4.3.2.

Using this algorithm it is possible to compute the three-dimensional horizon line by repeating the algorithm for different directions. In the end the horizon line allows to compute the amount of the hemisphere that is occluded by surrounding geometry. This in turn allows to get to the ambient occlusion value. It is worth mentioning that, using this approach, the function $\rho(L)$ is replaced by a step function that is 0 for all points within the sampling distance and 1 for all points with a distance larger than that.

4.3.2 Sampling step size

A key parameter of this algorithm is the discrete step size Δx. The parameter defines how dense or how sparse the texture containing the world space fragment positions is sampled. Figure 4.8 shows an example of how the town scene may be sampled in a 3D illustration. Next to this there is a figure that shows how this sampling in 3D corresponds to the sampling of the two-dimensional fragment position map. It can be seen that the step size has an important influence on the accuracy of the computed horizon angle. Ideally one would like to take every pixel along direction θ in the fragment position map as sampling point. This is hardly doable in practice however because the amount of sampling points is rather limited to keep the computation times low. In order to take into account a wider range one needs to keep the step size at a value higher than that of a single pixel distance in most cases. The question now is how to define and how to compute the step

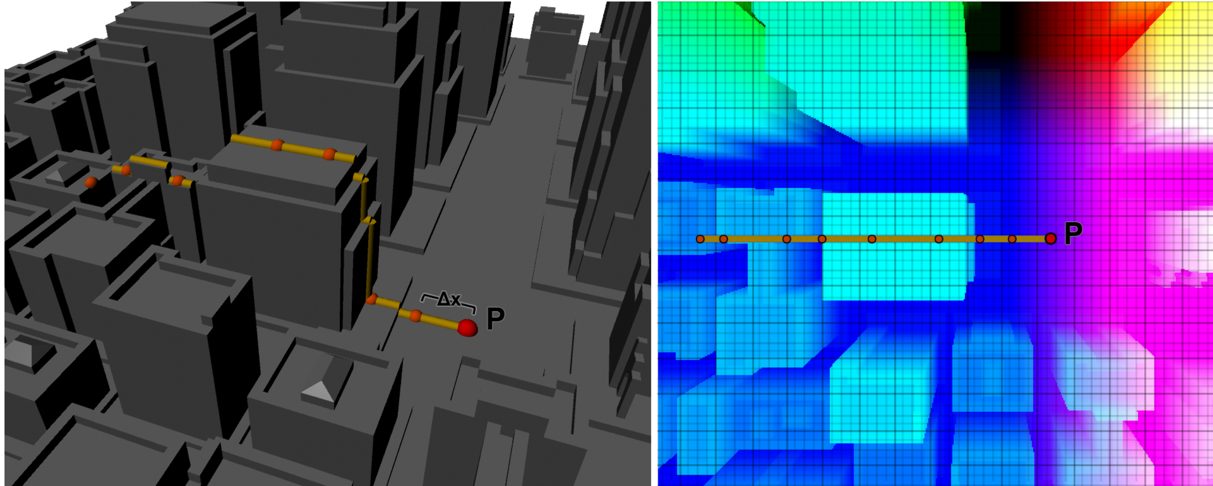


Figure 4.8: Left: 3D example of how the scene may be sampled at a point P in a specific direction. Right: Depiction of a texture containing the world position data of all fragments of a cube map face. This is where the sampling actually takes place in the implementation. See how the uniform sampling in the left image becomes a non-uniform one in the right one. This is due to the varying distance of the fragments to the position of the cube map.

size. A big problem is the fact that scenes may vary in size drastically. A detailled model of a bug may be modelled at a much smaller scale than a model of a car and yet have the same amount of detail. In addition to this users may take an arbitrary scale when modelling their scenes. So taking an absolute value of, say, one tenth of a world unit in the OpenGL coordinate frame may work for one scene, but it may be too big or too small for another one. Ideally every scene needs to have the step size adjusted manually by an expert. This is not practical though. Instead in this work the step size is derived from the diameter of the bounding box of the scene. This is a heuristic that works well enough in a number of test cases. The approach may fail when there are fine details within a large scene. In this case the fine details are likely not to be shaded correctly due to a step size that is too big.

Using this heuristic there is still some computation involved in getting from the step size as chosen in world units to the size of the step in the fragment position map. The texture coordinates needed to address this map have to have a value between 0 and 1 for both axes. The computation starts by agreeing on a step size in OpenGL world units based on the diameter of the current scene. Translating this value into texture space is a bit more elaborate. In reality two neighbouring texture elements may contain two fragments that are close together on the 3D surface but they may as well contain fragments that are far apart. Unfortunately, it is impractical to fetch every texture element and compare the world space distance. The goal is to keep the amount of texture queries as low as possible. Instead here a linearisation is used as approximation to the true surface to compute the texture space step size beforehand. Figure 4.9 shows a sketch of the geometric relation between the world space units and the texture space units. What is done is to fit a plane to the surface point P that has the surface normal n of this point as its normal vector. Now one can compute the point P' located on the plane at the distance Δx_w which is the OpenGL world unit distance as defined in the beginning. Both P and P' can now be projected to texture space by using the view and the projection matrix from the cube map side that is currently in processing. After this is done, the range of the projected points is [-1,1] so it needs to be remapped to [0,1]. This is due to the OpenGL coordinate frame which is generally [-1,1] in all directions after the projection matrix is applied. Now the distance between the points can be measured and taken as the discrete step size Δx. This needs to be done separately for each sampling direction θ.

There is an interesting fact that comes with this procedure. The sampling does not have a uniform step size when converted back to world space. When taking a look at 4.8 it can be seen that the conversion along the orange line is not linear. The parts of the roof of the building are closer to the camera than the parts around point P and thus have a different conversion. Likewise, a uniform sampling in the texture space yields a non-uniform sampling in world space. This is a downside of this implementation. It basically stems from the linearisation of the surface that is used.

Next to the ambient occlusion value some additional information is computed and stored in this step. A quality factor and a hash value of the surface normal. The quality factor basically contains information about the geometric relationship between the cube map and a surface point. This value is necessary because at one point later there may be two or more cube maps containing ambient occlusion information for the same point on a surface. Using the quality factor allows to choose the best cube map to use for this point. Ideally the cube map is chosen which contains the least amount of

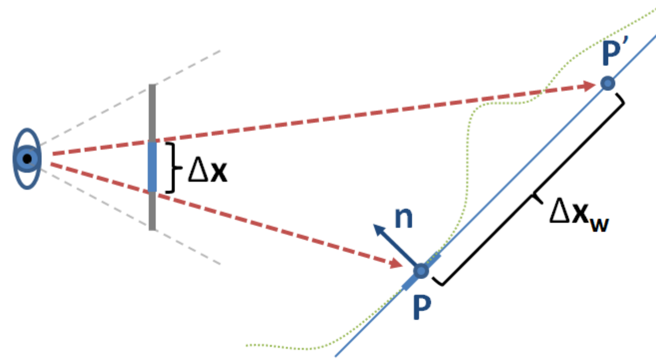


Figure 4.9: Geometric relation between the step size in OpenGL world units Δx_w and the step size in texture space units Δx. The scene is replaced by a linear approximation. A plane is fitted at point P and surface normal n and the distance is estimated using this plane. The green dotted line shows how the actual scene may look like for example.

distortions and captures the illumination of the point at the highest resolution. Figure 4.10 shows how the quality factor is computed.

The hashed surface normal is used later to identify edges of the scene in the rendered image. This information is necessary for a post processing step to remove rendering artefacts. The reason to use a hash value instead of the real normal is to save memory. It turns out to be perfectly sufficient to store the normal, which is assumed to be normalized, in the following way.

$$n_{hash} = 0.14 \cdot (n_x \cdot 0.5 + 0.5) + 0.29 \cdot (n_y \cdot 0.5 + 0.5) + 0.5 \cdot (n_z \cdot 0.5 + 0.5) \tag{4.5}$$

The result is a single number that has similar values for similarly oriented normals and differing values for differing normals.

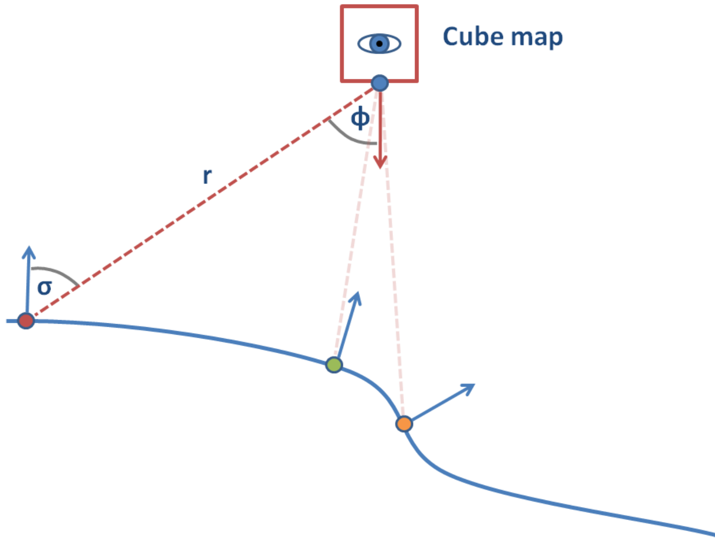
What is also worth noting is that the data for all 6 cubemap sides is rendered into a single texture, one face placed next to another. This is done in order to keep the amount of necessary texture units low in the following steps. Instead of having to bind 6 different textures to 6 texture units, only one texture has to be bound. In order to make this possible the width of the texture target is set to be 6 times the original resolution. During the rendering step the OpenGL viewport is simply moved from the left to the right with the original resolution as offset in each step. This way different parts of the texture are written to. After 6 passes the single texture contains all the data.

4.4 Projective Texturing

The outcome of the previous two steps is one or more cube maps filled with ambient occlusion data of the environment. Figure 4.11 shows a sample of the computed ambient occlusion shadow as well as the quality factor. Apart from these two values it also contains a value to store the depth from each fragment to the center of the cube map. Finally it also contains a hash value of the surface normal. The following will explain how these values are used to apply the ambient occlusion from the cube maps to the scene.

The information that is available at this point consists of the ambient occlusion information from the point of view of a cube map. As mentioned in 3.1.4 the goal is to project this information back onto the scene so it can be watched from any point of view.

The ambient occlusion data is stored in a set of textures. The task of projecting the textures onto the surface of a scene can be reduced to finding the right texture coordinates. For the purpose of this example the following description focuses on how to project only one of the six cube map sides. The side is called *projector* in this description. Figure 4.12 shows how the task of finding the right textures coordinates can be solved. First it is necessary to compute the world space position F_w for a fragment F_s on the screen. Chapter 4.2 describes how this is done by reverting the OpenGL transformations. The same principles can be used here as well. Once the world space position is known it is possible to compute the texture coordinates F_p for the fragment. This is done by applying the view and projection transformation to the world space coordinates of the fragment. More precisely the coordinate F_w is multiplied by the view matrix M_v and



$$Quality = Q_a \cdot Q_b \cdot Q_c \tag{4.6}$$

$$Q_a = 1.0 - \frac{\arctan r}{\frac{\pi}{2}} \tag{4.7}$$

$$Q_b = \cos\phi \cdot 0.5 + 0.5 \tag{4.8}$$

$$Q_c = \cos\sigma \cdot 0.5 + 0.5 \tag{4.9}$$

Figure 4.10: The quality factor consists of 3 terms. Q_a expresses the distance between the surface point and the cube map. The lower the distance the bigger the value gets. Q_b becomes big when the surface point is close to the center. Q_c becomes big when the normal of the surface point is facing the cube map. The intent behind Q_b and Q_c is that there is likely to be distortion when a surface point is off the center of the cube map and also when the surface is slanted and not facing the cube map. Those 2 terms act as a penalty for those cases. In the figure the green circle is likely to have a big quality factor. It is located at the center of the cube map and is facing it as well. The orange circle is located at the center but is not facing the cube map. The red circle is far off the center and is also not facing the cube map so it is going to have a low quality factor for this cube map. If there was another cube map directly above the red circle it would probably be preferred over the current cube map.

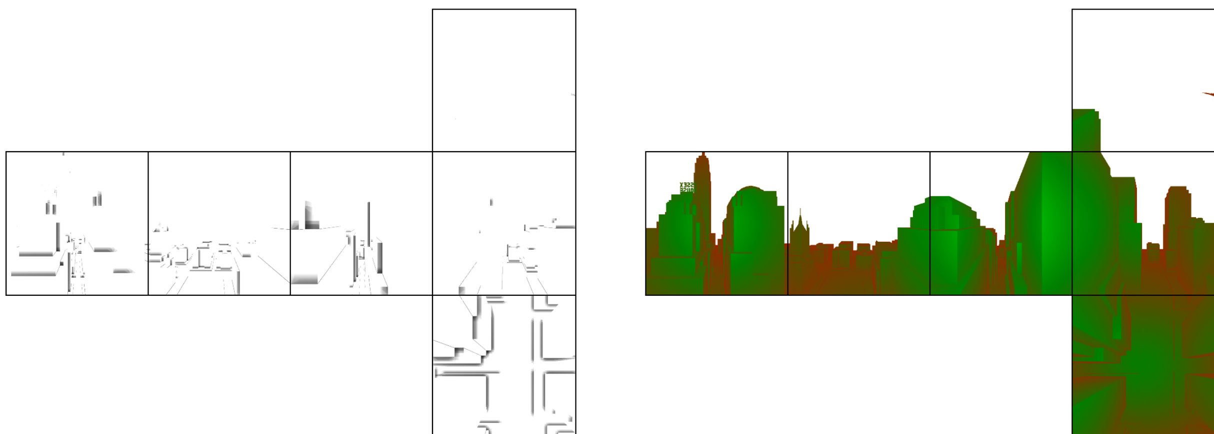


Figure 4.11: Left: Ambient occlusion data of a cube map. Right: Quality factor associated with the same cube map. In this depiction the single value quality factor is displayed as a false-colour image. Green areas stand for good quality while red areas stand for low quality.

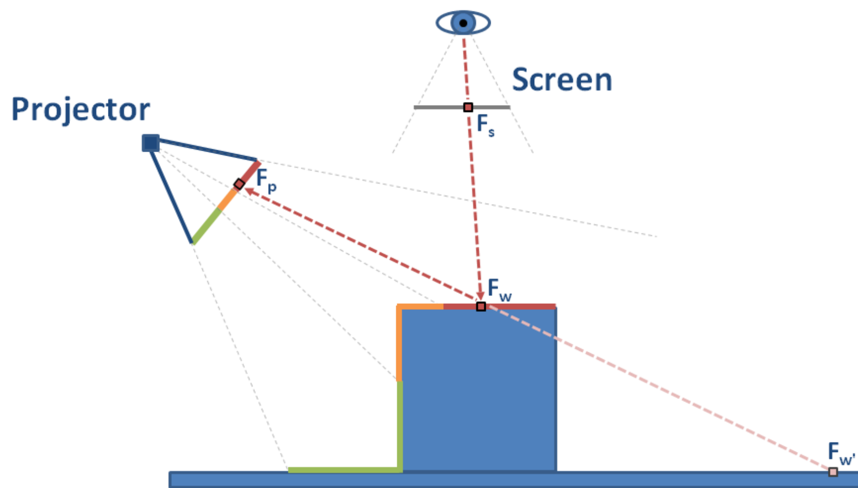


Figure 4.12: In projective texturing the goal is to find for a fragment F_s the coordinate F_p. F_p can be used as texture coordinate to find the colour of the fragment illuminated by the projector. There can be multiple parts on the surface that map to the same texture coordinates. A depth test has to be performed in order to find out the right surface point to be lit by the projector.

the projection matrix M_p of the projector. The outcome of this computation is the clip coordinate of the fragment in the projector coordinate frame. This coordinate has a range of [-1,1] in each direction. In order to use this value as texture coordinate it must be remapped linearly to the range [0,1]. Once this is done the coordinate can be used directly to access the texture stored in the projector.

$$F_p = M_p \cdot M_v \cdot F_w \tag{4.10}$$

The next thing to do is to fetch the depth value of F_p stored in the projector and compare it with the depth to the fragment F_w in world space. This is necessary because there may be parts on the surface that map to the same texture coordinate but cannot actually be seen by the projector. Figure 4.12 shows this problem. Fragment F_w' has the same texture coordinates but is hidden from the projector. The projector in turn contains only the ambient occlusion value for the correct fragment F_w.

A cube map can be seen as a combination of 6 projectors. So the above principles can also be used to project the ambient occlusion from a cube map onto the scene. What needs to be done in addition is to find out first which of the 6 cube map sides the fragment is illuminated by. Indeed every fragment is lit by only one of the 6 sides. Using the world position of the center of the cube map and the world position of the fragment one can compute a direction vector from the cube map to the fragment. This direction vector hints to the side of the cube map the fragment is associated with. One simply needs to look at the single components of the direction vector. The axis of the largest component is also the side to use. Figure 4.13 shows an example of how this works.

The above procedure describes how to project the contents of one cube map onto a scene. The aim of this work however is to have multiple cube maps projecting their computed ambient occlusion values onto the scene. In the following a rather simple and straight forward approach is proposed to achieve this goal. The downside of this approach is that it does not scale well with the number of cube maps.

The idea is to use a multipass scheme and project one cube map per render pass. In order to reuse the results from the last pass one needs to render to a ping-pong buffer. E.g. in the second pass one needs to access the result of the first pass and add in the next cube map. In this way the image is rendered into a render target which is then used as input for the next pass. In order to avoid having to recompute the fragment world positions at every pass they are computed once per frame and stored in a texture. Thanks to this the scene geometry needs to be rendered only once during the projective texturing step. Otherwise it would have to be rendered again every pass in order to produce the fragment world positions. This approach helps keeping the framerate higher for scenes with high polygon count.

The most important question that arises is how to deal with fragments that are being lit by more than one cube map. In this case the quality factor from chapter 4.3 is be used. Next to projecting the ambient occlusion onto the scene also the quality of the cube map for this fragment is stored. In the next ping-pong pass the quality from the last pass is fetched and compared with the quality of the new cube map. In case the quality is better than the previous one, the ambient

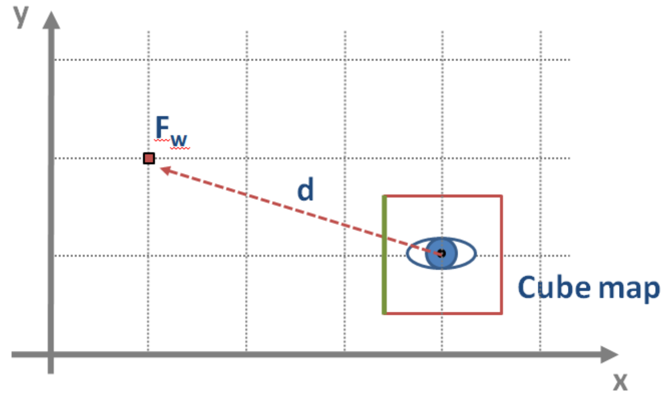


Figure 4.13: In this example the side of the cube map that is highlighted with a green colour has to be used to apply the lighting information of fragment F_w. In order to find this side the direction vector d is constructed pointing from the center of the cube map to the position of the fragment in world space. In this example the vector is $(-3,1)$. The highest component of this vector is pointing along the negative x axis. This is the side that has to be used for the projection.

occlusion information is overwritten with the new value. Also the new quality factor is stored. In this way after the last ping-pong pass every fragment contains only the contribution from the cube map that has the biggest quality associated. As mentioned, the big downside of this algorithm is that it needs multiple render passes to compute the final solution. In the current implementation every additional cube map means one additional render pass. In order to be able to limit the amount of render passes this work chooses to use only the n closest cube maps for the current camera position. This subset of cube maps can be computed easily per frame and guarantees an upper limit of render passes.

To summarize here is a short sketch of the algorithm used to project the ambient occlusion from the cube maps to the screen. In an initial pass the raw scene geometry is rendered. A shader program is activated to compute the world space coordinates of each fragment and write this value into a screen sized texture. In the next pass this texture is bound to a texture unit along with the cube map textures. A fragment shader computes the coordinates used to access the cube map data from the world position of the geometry. The ambient occlusion and quality value is fetched from the cube map and written to the first of the two ping-pong textures. This texture is now bound to a texture unit to be used as input for the following pass. Again the coordinates are computed but before writing to the second ping-pong buffer the old quality factor from the first ping-pong buffer is fetched. This value is compared to the quality of the current cube map. In case of a higher quality of the current cube map both the ambient occlusion and the quality are written to the second ping-pong buffer and used as input for the next pass. This goes on for all enabled cube maps. At the end of this process only the ambient occlusion values with the highest quality associated are found in the ping-pong buffer used last.

4.5 Ambient occlusion blending and post processing

The outcome of the previous computation steps is a screen sized texture containing the ambient occlusion information for the current screen. It is still necessary to blend this information with an already existing rendering of the scene. For this purpose the ambient occlusion as well as the render output are stored in two separate textures. Given this data a fragment shader can blend the two sources together and make the ambient occlusion appear on the rendering. [SZ98] gives a brief overview on how to blend lighting information with a rendering. A simple method that is also used in this work is to just multiply the grey scale ambient occlusion i_{ao} with the colors of the original rendering i_s. Additionally a factor α to control the contrast of the ambient occlusion is added. The following formula describes how to obtain the total illumination intensity i_{tot} using this blending process.

$$i_{tot} = i_s \cdot [(1-\alpha) + (\alpha \cdot i_{ao})] \tag{4.11}$$

What this does is remap the values of the ambient occlusion from the full range of [0, 1] to the narrowed range [1-α, 1]. The higher the contrast factor the closer the values get remapped to the full range which allows for the most detail to be displayed. A low factor maps the range to a very narrow range of values which causes a lack of detail. Figure 4.14 shows what the multiplicative blending operation looks like in an example.

There is another issue that needs to be addressed here. For the purpose of a quick ambient occlusion computation the sampling rate of the horizon line algorithm is kept low. This may produce unwanted artefacts as shown in figure 4.15 on

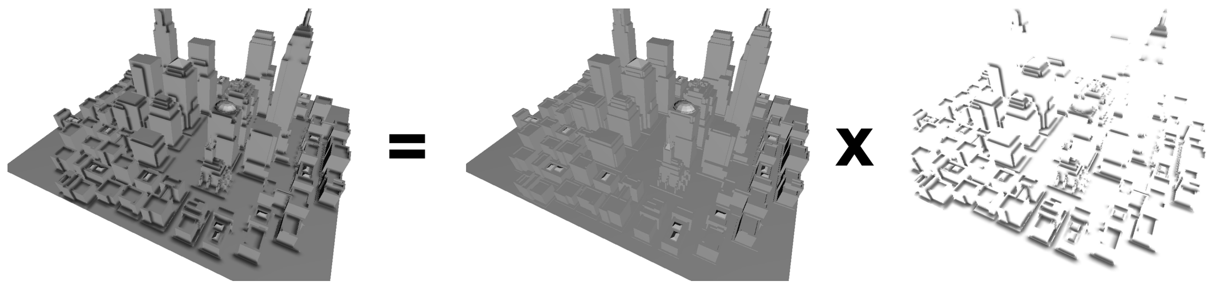


Figure 4.14: Multiplicative blending of an existing rendering with the ambient occlusion to obtain the final shaded image.

the left. Instead of turning up the sampling rate there is another way to solve this problem. Ambient occlusion shadows are typically of very low frequency. Unlike hard shadows the transitions in the shadow of ambient occlusion are very smooth. This is why it is possible to use a simple image blur filter on the ambient occlusion texture to get rid of the artefacts. The reason to do this is that it is faster to remove the artefacts this way than using a higher sampling rate and avoiding them in the first place. The most simple way to blur an image is using a lowpass filter. What this type of filter does is to compute the average fragment intensity for a given neighbourhood of a selected fragment. The result of this filter can be seen in the middle of figure 4.15. While it does reduce the impact of the artefacts significantly it has an unwanted side effect. It also blurs the shadow along edges that should remain sharp. The result is that, using this operation, the final rendering looks blurry which is something that is to be avoided. A possible solution to this problem is using an edge aware filter. A filter that has got this property and is very common in computer vision and photography applications is the bilateral filter. Typically a gaussian filter is used for blurring in bilateral filters. A gaussian filter works by computing a weighted sum of all neighbouring fragments where the weight is the value of a two-dimensional gauss function at the position of a fragment. Bilateral filters build on this but are a bit more complex. Instead of taking into account only the positions of fragments, they also include the difference in the intensity of neighbouring fragments in the computation. In case the intensity of a fragment differs too much from the intensity of the fragment that is currently evaluated a small weight is given. This helps to avoid blurring across edges of an image. The reason behind this as well as the mathematic formulation is already given in chapter 3.1.5.

In this work a slightly modified version of this filter is used. Instead of using a gaussian filter to perform the blurring a simple lowpass filter is used. Also the comparison of the fragment intensities for the edge detection is replaced by a comparison of the fragment depth and normal vector. This allows for a better detection of edges. After all the varying fragment intensities along edges are typically just a side effect of the differing illumination caused by varying geometric properties. In computer vision the scene geometry of an image may not be readily available which is why the intensities are used instead. In this work however it is possible to access these properties and it is of benefit to use them directly. The right side of figure 4.15 shows the outcome when this filter is applied. The differences may be subtle and not easy to see in this image, but they are definitely an important visual feature in the running application. For the size of the lowpass filter a 7 x 7 mask is used. In order to compare the normals of a surface the hashed normals that are stored in the ambient occlusion computation step are used. In combination with the depth this allows to find out whether or not a neighbouring fragment might belong to the same part of a surface or whether there might be an edge. Figure 4.16 shows a visualization of the edge detection algorithm.

The filtering process is carried out in another GLSL shader. The following shows a listing of the most important parts of the shader program.

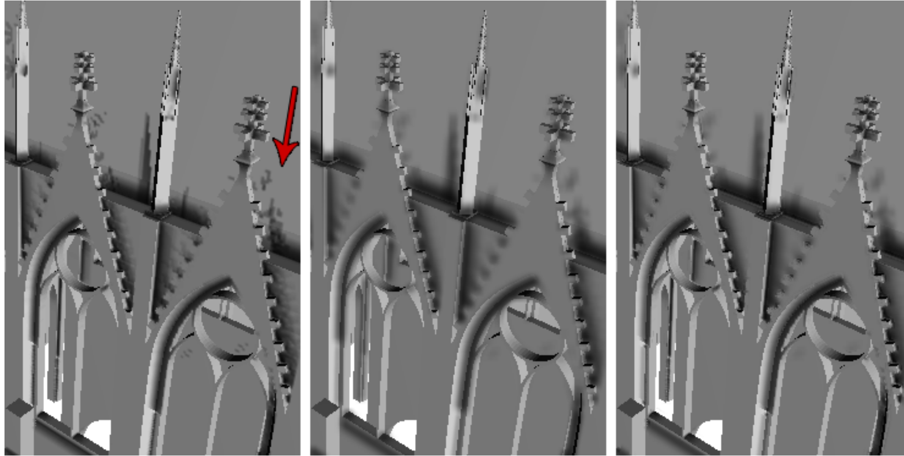


Figure 4.15: First image: Severe artefacts arising from the limited precision of the horizon line computation. Second: Blurred ambient occlusion using a lowpass filter. Third: Edge aware lowpass filter that tries to keep edges sharp while blurring the rest.

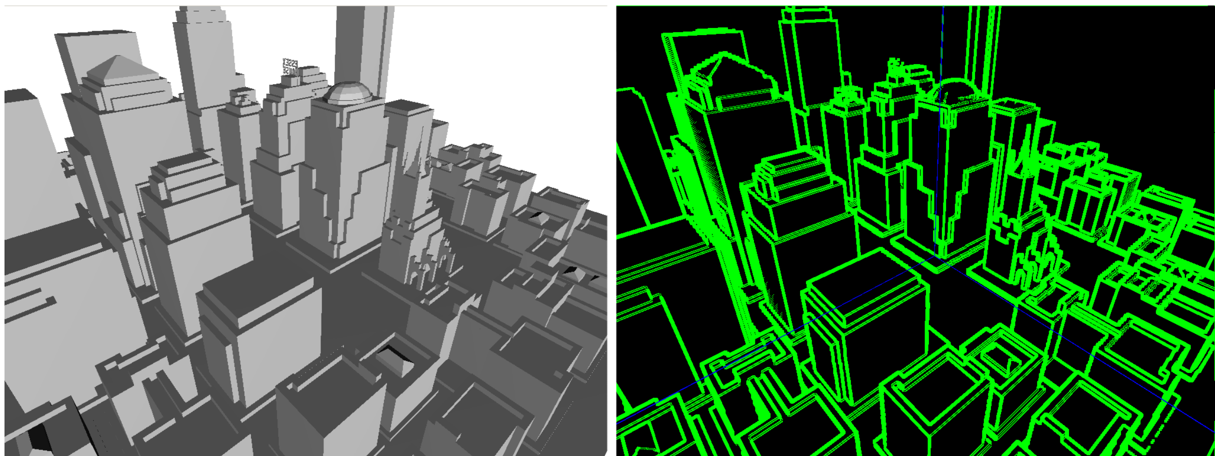


Figure 4.16: Edge detection based on the depth and normal information of the current frame. The right image shows the edge image computed for the scene shown on the left. Edges are visualized using a green colour. The thin blue lines show the outlines of a cube map.

Listing 4.15: GLSL code of an edge aware lowpass blurring filter.

```
// Define the size of the filter . For higher performance the number may be
// lowered at the cost of a lower quality .
const int filter_size = 7;
const int half_filter_size = filter_size /2;

float ssao_average = 0;
int num_summed_fragments = 0;

// Fetch the depth and hashed normal of the center pixel.
float center_depth = texture2D(tex1, fragment_pos / screen_res_).z;
float center_normal = texture2D(tex1, fragment_pos / screen_res_).y;

// Helper variable to reduce computations later.
vec2 screen_res_factor = 1 / screen_res_;

// Filter loop.
for(int i = -half_filter_size ; i <= half_filter_size ; i++)
{
  for(int j = -half_filter_size ; j <= half_filter_size ; j++)
  {
    vec2 texture_fetch_location = (fragment_pos+vec2(i,j)) * screen_res_factor;

    float neighbour_depth = texture2D(tex1, texture_fetch_location).z;
    float neighbour_normal = texture2D(tex1, texture_fetch_location).y;

    // Compare the depth and normal of the center pixel to the ones
    // of the neighbouring pixel. Add the pixel only in case both values
    // are close enough together.
    if (abs(center_depth - neighbour_depth) < 0.01
    && abs(center_normal - neighbour_normal) < 0.1)
    {
      ssao_average += texture2D(tex1, texture_fetch_location).x;
      num_summed_fragments++;
    }
  }
}
ssao_value = ssao_average;
if (num_summed_fragments > 0)
  ssao_value /= num_summed_fragments;
```

4.6 Automated cube map placement

At this point the ambient occlusion computation is basically finished and the solution is already being displayed on the screen. This is the right time to find out how many of the visible pixels are actually shaded using ambient occlusion. As long as the camera stays in the same place as a cube map, all pixels should be shaded. However once the camera is moved away and there are no other cube maps close to the new camera position, chances are that parts of the scene will become visible that are not shaded by any cube map. As long as it is just a few pixels that are unshaded it will be hard for the user to notice. However, when large unshaded areas become visible it will be necessary to place a new cube map that covers this area. For this purpose an algorithm is implemented that automatically counts the number of shaded and the number of unshaded pixels that are currently visible and not part of the empty background. Using this information allows to compute the percentage of ambient occlusion shaded pixels and compare it to a manually defined threshold. The exact algorithm works as follows.

The algorithm makes use of the full screen texture containing the ambient occlusion information that is also used to blend with the rendered image. Aside from the ambient occlusion this texture contains an info flag in a separate channel. The flag contains information whether or not the current fragment is shaded by any cube map or not. Figure 4.17 shows a visualization of this flag. Shaded fragments are displayed in the correct colour while unshaded fragments are shown in bright red. The flag can be used to find the percentage of fragments that are shaded. One can simply read the data

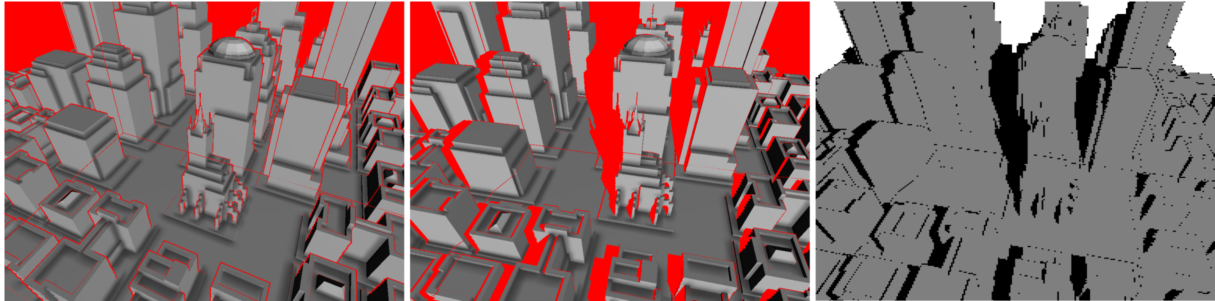


Figure 4.17: First image: Image of the town scene from the exact position where the only cube map is currently placed. Ideally all fragments except for background should be covered by the cube map and illuminated by ambient occlusion. As can be seen however there is a fine line of red fragments around each building. This error is caused by the fact that the cube maps typically have a different image resolution than the rendered image. During the up- or downscaling process an inaccuracy is introduced around the edges. In these cases the depth test incorrectly returns the fragments as not being part of the cube map. In practice this error does not have any influence on the perception of the renderd solution as can be seen in the result images shown in this work. Second: The camera is moved to the side a little revealing parts of the scene that are not covered by the cube map. Third: Depiction of the low resolution image used to count the shaded and unshaded fragments. Note that here the background has a different colour than the unshaded fragments. This is necessary to be able to compute the correct percentage of covered versus uncovered fragments.

of this texture back from the graphics memory to the application memory and use the CPU to compute the percentage. Instead of using the full resolution texture, its contents are rendered into a small sized render target first. In the current implementation an 8 x 8 pixel render target is used. After this is done the data of the small render target is transferred to the application memory and the number of shaded and unshaded fragments are counted. Doing it this way reduces the amount of data that needs to be transferred from the graphics memory to the system memory. Also it lowers the amount of fragments that need to be counted on the CPU side. The right side of figure 4.17 shows a depiction of the small sized render target. Note that for the purpose of this depiction a 256 x 256 pixel target is used because otherwise the image would be hard to comprehend. In the implementation this does not matter and the 8 x 8 pixel render target is found to be working well.

The data is read back from the texture memory using the following OpenGL command.

Listing 4.16: Reading back texture data from graphics memory to system memory using OpenGL and C++.

```
glGetTexImage(GL_TEXTURE_2D, 0, GL_LUMINANCE, GL_BYTE, data);
```

The *data* buffer from the listing is iterated through and the amount of shaded and unshaded fragments are counted. Using a custom GLSL shader, the colour value 0 is assigned to unshaded fragments while 0.5 is assigned to shaded ones. The background colour is set to 1 so this allows to count only visible parts of the scene. When the algorithm finds that less than 75% of the fragments are covered by ambient occlusion a new cube map is placed at the current position of the camera.

4.7 Library Interface

This work includes a software library, called *aolib2*, that allows to make use of ambient occlusion in regular applications. The library is written in C++ with the aim of allowing for an easy integration into existing rendering frameworks. The integration consists of two parts. First the scene geometry has to be defined and handed over to the library. This is done by using a polygonal data representation. Then, at runtime, the render calls of the host application need to be wrapped by the ambient occlusion rendering routine.

4.7.1 Defining the scene

The following shows an example of how to hand over the description of a simple square surface to the aolib2. The data needed by the library consists of indexed triangle sets of the scene. Vertex positions, which define the corner points of the surface are given first, followed by indices that tell what vertices every triangle is made of.

Listing 4.17: Example of defining a scene for the aolib2 to use for computing ambient occlusion.

```
// Tell the aolib2 that a new scene is being defined now.
AmbientOcclusion().meshBegin();

// Give all the 4 corner vertices of the square.
AmbientOcclusion().aoVertex3f(0.f, 0.f, 0.f);
AmbientOcclusion().aoVertex3f(0.f, 0.f, 1.f);
AmbientOcclusion().aoVertex3f(1.f, 0.f, 1.f);
AmbientOcclusion().aoVertex3f(1.f, 0.f, 0.f);

// Tell the aolib2 that the geometry is defined in terms of triangles.
AmbientOcclusion().aoBegin(AO_TRIANGLES);

// Definition of the first triangle to make up the square.
AmbientOcclusion().aoVertexIndex(0);
AmbientOcclusion().aoVertexIndex(1);
AmbientOcclusion().aoVertexIndex(2);

// Definition of the second triangle to make up the square.
AmbientOcclusion().aoVertexIndex(0);
AmbientOcclusion().aoVertexIndex(2);
AmbientOcclusion().aoVertexIndex(3);

// Finish definition of the geometry.
AmbientOcclusion().aoEnd();

// Finish definition of the entire mesh. Note that a mesh can be made
// from different types of geometry by using multiple aoBegin/End blocks
// within a meshBegin/End block.
AmbientOcclusion().meshEnd();
```

Apart from this very direct kind of triangle definition the user may also choose to pass the face indices in the form of a triangle strip, triangle fan, quads or a quad strip. In order to enable one of these other ways line 11 in listing 4.17 needs to be changed. Instead of giving *AO_TRIANGLES* as identifier one may give any of the following: *AO_TRIANGLE_STRIP*, *AO_TRIANGLE_FAN*, *AO_QUADS*, *AO_QUAD_STRIP*. The idea is borrowed from OpenGL and has the purpose of saving unnecessary calls and data. Figure 4.18 shows how geometry can be defined using the different ways of indexing.

4.7.2 Wrapping the render calls

In order to show ambient occlusion on top of a rendering it is necessary to wrap the render calls in the following way.

Listing 4.18: Wrapping host application render calls to enable the ambient occlusion overlay.

```
AmbientOcclusion().aoDrawBegin();

// Original render calls.
// ...

AmbientOcclusion().aoDrawEnd(camera_world_position.x, camera_world_position.y,
                             camera_world_position.z);
```

As can be seen the world position of the camera needs to be given as parameter. This is necessary to be able to sort all cube maps by their distance to the current camera position. Remember that only the n closest cube maps are used by the projection step described in chapter 4.4. This is what the camera world position is needed for.

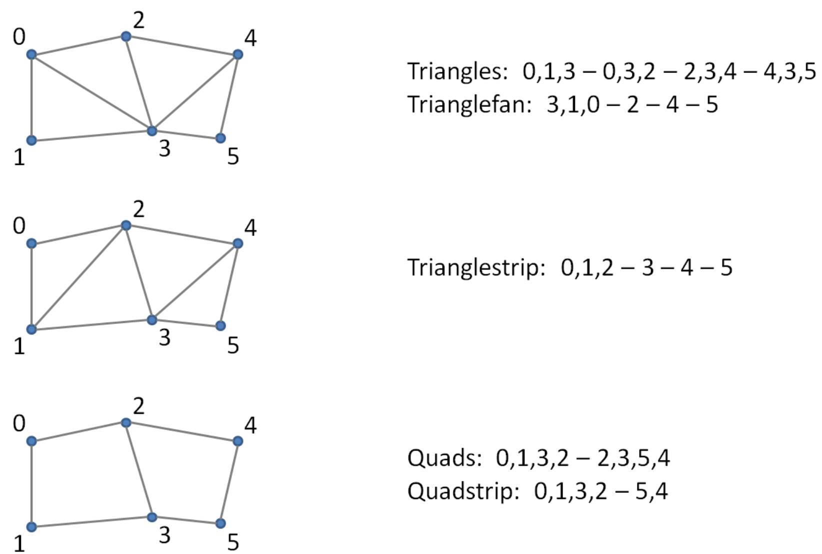


Figure 4.18: Different methods of indexing faces to describe a surface. As can be seen, the same surface can be described with varying amounts of face indices. While the straight forward triangle description needs a total of 12 indices, the strip and fan descriptions only need 6 indices in this example.

Chapter 5

Discussion

The following discusses the performance of the algorithms described in this work. After that an insight is given into the advantages as well as the weaknesses of this approach to compute ambient occlusion. Finally some outtakes are shown and ideas for future work are discussed.

5.1 Benchmarks

In order to run the various benchmarks three scenes with different properties are used. Figure 5.1 shows these scenes. The town model has a low polygon count, consisting of roughly 12000 faces and around 8000 vertices. Most of this scene consists of rectangular blocks and planar surfaces. The Stanford bunny has a higher polygon count with roughly 70000 triangles. Also it has a much more organic shape and is more compact than the town scene. The dome is the most complex model used in the benchmark tests. It consists of 564000 triangles and contains very small and fine grained details.

The PC system used during the benchmarks is a Windows XP computer with an Intel Core2 Quad CPU, 4GB of RAM and a GeForce 9800 GTX with 512 MB of memory. *gDebugger*, an OpenGL debugger and profiler, is used to measure the runtimes of the various shader programs and the speed of the rendering algorithms in general.

5.1.1 Ambient occlusion computation

Before starting to do the actual performance analysis it is useful to have a look at the factors that will influence the computation time. First the geometry of the scene needs to be rasterized six times for one cube map. In this phase a high polygon count may influence the computation time negatively. After that a rectangular polygon that covers the entire screen of the cube map is rendered for every side of a cube map. This procedure ensures that every fragment of the cube map is sent to the ambient occlusion fragment shader. In this phase the computation time depends mainly on the resolution of the cube map and on the complexity of the fragment shader. The higher the resolution the more fragments need to be shaded and the more complex the shader the bigger the computation time for each fragment.

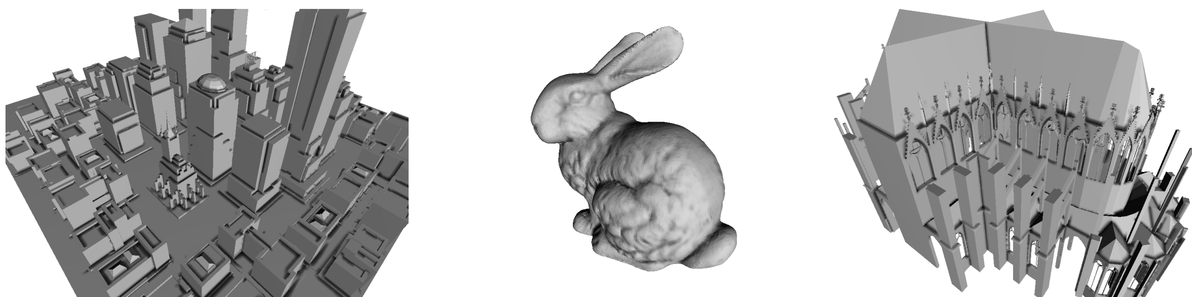


Figure 5.1: First image: Model of a town center. Second: Stanford bunny. Third: Model of a dome, courtesy of Dr.-Ing. Sven Havemann.

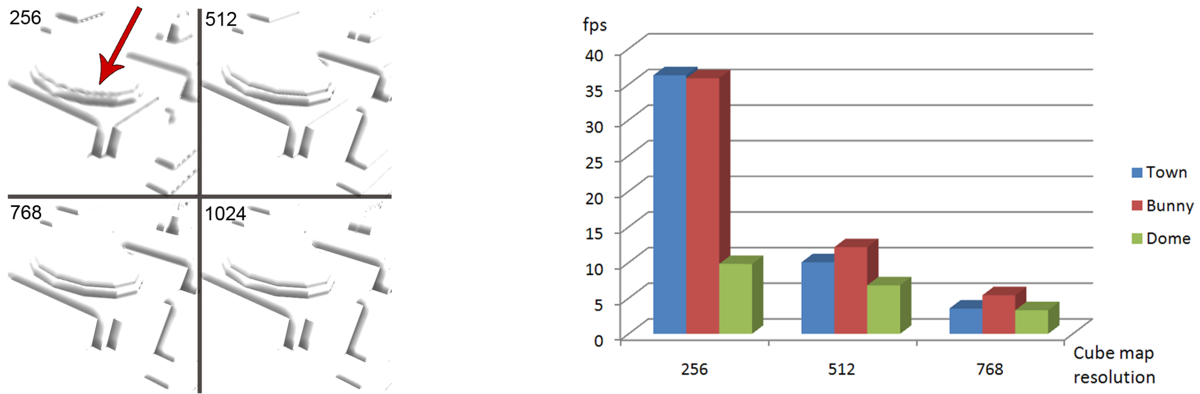


Figure 5.2: Left: Extraction of a scene at different cube map resolutions. It can be seen that the 256 x 256 cube map suffers from the low resolution and is not able to reconstruct fine geometry details. Quality increases gradually with increasing cube map resolution. Right: Computation times measured in frames per second for three different cube map resolutions at different input scenes. The 1024 x 1024 resolution is left out of the benchmark as it leads to framerates too low to be measured correctly.

Both the fragment shader complexity as well as the cube map resolution can be varied. Experiments are run to find settings that yield a good visual quality as well as low computation time. The experiments start by comparing the visual quality of the results when using different cube map resolutions and looking at the variations in the computation times.

For the following test a single cube map is placed in a central area of the scenes where as many sides as possible are covered by geometry. Figure 5.2 shows computation times and output quality for varying cube map resolutions. Quality wise it is obvious that the visual quality increases along with the resolution. Although it depends heavily on the placement of the cube maps, the tests show that a resolution of 256 x 256 pixel is not high enough in general. One should strive to use a resolution of 512 x 512 or higher. Unfortunately, as can be seen at the right side of Figure 5.2, refresh rates decrease heavily with higher resolutions. However, the computation carried out in this phase usually needs to be performed only once as long as the geometry does not change. So higher computation times may be acceptable as they stall the application for just a moment. What is interesting to see is that for low resolutions the dominant part of the computation consists of the rasterization of the geometry. At the 256 x 256 resolution the low polygon models are rendering significantly faster than the more complex dome model. The higher the cube map resolution the more dominant however the fragment computation time becomes. This is an expected result as the ambient occlusion shader program is rather complex and therefore eats up more computation time the more fragments there are to shade.

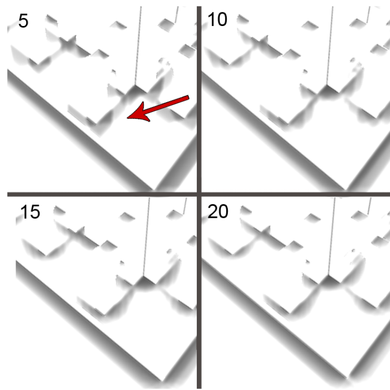
Next up is a comparison of different quality settings of the ambient occlusion shader along with their computation times. There are two parameters in the shader that allow for an intuitive tradeoff between fast and slow computation and good and bad quality. The main computational task in this work is to find the horizon line for the surroundings of a point on the surface. In order to find this information the geometry of the scene is sampled along finite amounts of directions. The amount of directions is the first parameter. Then for a chosen direction a discrete amount of sampling steps along the direction is used. The more steps the better the sampling along this direction, which is also the second parameter. It is worth mentioning that in this work the sampling step size is fixed. Increasing the sampling number therefore increases the radius of influence but does not change the resolution of the sampling. For the purpose of this benchmark the cube map resolution is set to 512 x 512 and not changed.

Figure 5.3 shows the way in which the amount of directions influences the quality of the computed solution. Artefacts can be seen that show a distinct directional pattern when the number of directions is too low. In this example the artefacts start to vanish at around 15 sampling directions. Typically this pattern does not show as much as in this example so a number of 10 directions may be just as fine for many applications.

Figure 5.4 compares the outcome of the algorithm for different amounts of sampling steps. It can be seen clearly that the number of sampling steps directly influences the amount of shadow in the scene. Experiments show that a value of 15 is a good tradeoff between execution speed and ambient occlusion quality.

5.1.2 Projective texturing

Next the performance of the projective texturing step is discussed. In order to test the performance an increasing number of cube maps are placed on a regular grid that surrounds the scene. Figure 5.5 shows the placement of 25 cube maps in the



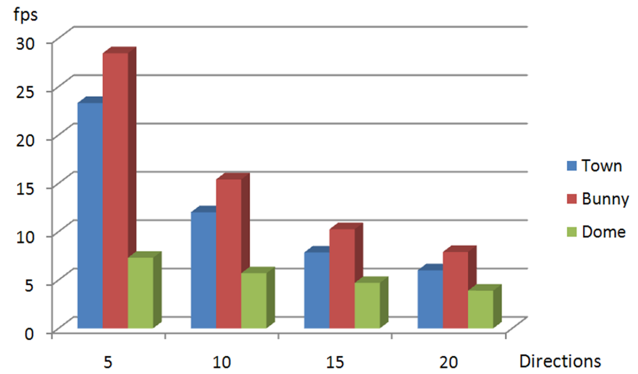
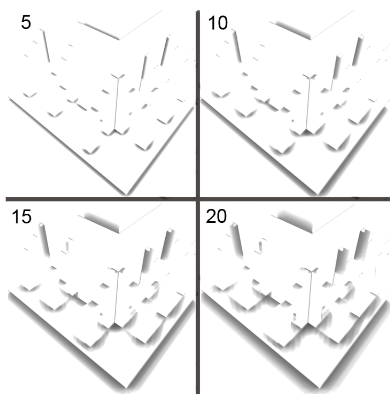


Figure 5.3: Left: Comparing the quality of the ambient occlusion at different amounts of sampling directions. Artefacts can be seen when the amount of directions is too low. Right: The higher the amount of directions, the lower the framerate.



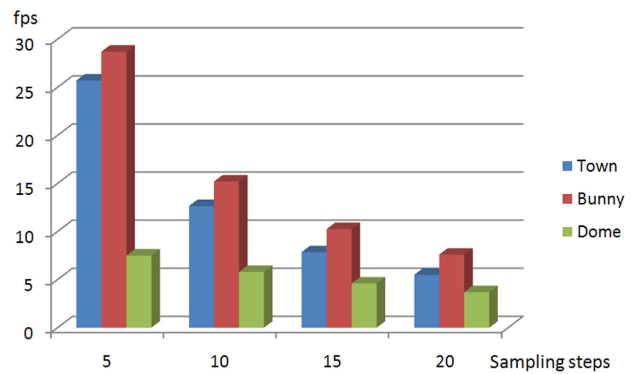


Figure 5.4: Left: Changing the amount of sampling steps has a direct influence on the radius of the ambient occlusion. Low amounts lead to very little shadowing while higher amounts lead to bigger shadow areas. Right: Just as the amount of directions the number of sampling steps has a penalizing influence on the framerate.

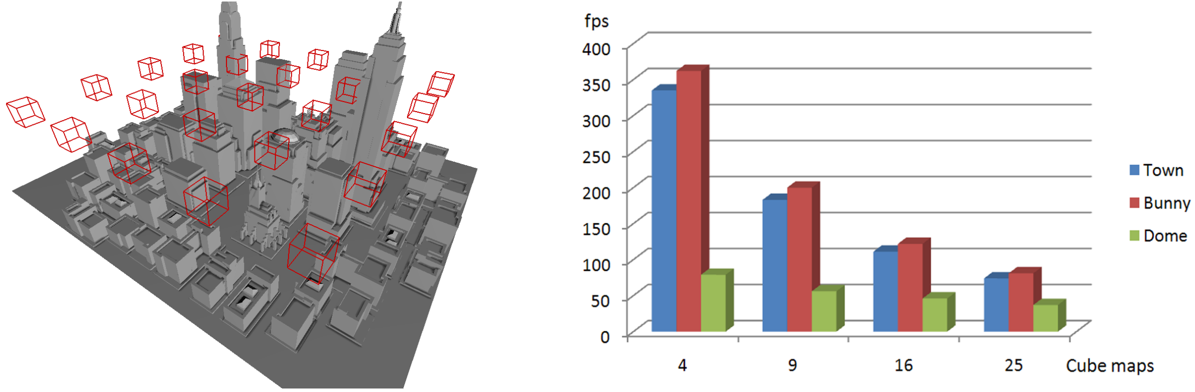


Figure 5.5: Left: 25 cube maps placed for the benchmark test of the town scene. Right: For all scenes the framerate decreases with increasing amounts of cube map.

town scene as an example. As can be seen on the right side of figure 5.5, the amount of cube maps has a direct influence on the framerate. In the reference implementation every additional cube map means one additional rendering pass. So it is easy to see where the decreasing framerate comes from. A way to keep a more stable framrate is to use just the n closest cube maps instead of all cube maps during the projection. This limits the amount of rendering passes to n even if the total amount of cube maps is bigger. The downside is that one may notice when a cube map is suddenly excluded and another one is included in the rendering. The higher the number n, the less noticeable this gets but of course the lower the framerate gets. What number to pick depends on the scene and the placement of the cube maps. In the experiments for this work a number around 10 has proven to be a good tradeoff.

5.1.3 Cube map memory consumption

Aside from the computation times of the various algorithms it is also important to talk about the memory consumption. The following discussion will focus on the memory that is needed to compute and store the ambient occlusion values in a cube map as this is the most expensive operation. The memory consumption is split up into two parts. The first part consists of the data needed temporarily during the computation. This data is deleted as soon as the ambient occlusion is computed and stored. The second part is the actual ambient occlusion data that needs to be kept in memory as long as the cube map exists.

5.1.3.1 Temporary data

The temporary data needed during the computation consists of the world position and surface normal of each fragment in the cube map. Both these vectors are stored in a floating point render target. The temporary memory consumed by one cube map can be computed using the following formula.

$$S_{temp}[Byte] = n_t \cdot n_s \cdot n_c \cdot n_{Bpf} \cdot n_f$$

n_t denotes the number of render targets used which is two in this work as the position and normal data is stored separately. n_s denotes the number of sides per cube map which is 6. n_c is for the number of channels per cube map texture. Both position and normal data take up 3 channels. n_Bpf denotes the number of Bytes used per texture fragment. Since the data is stored as floating point numbers this value typically becomes 4. Finally the number of fragments per texture is needed. As an example here is what the consumption looks like for a 512 x 512 sized cube map.

$$S_{temp}[Byte] = 2 \cdot 6 \cdot 3 \cdot 4 \cdot 512 \cdot 512 = 37748736\,Byte = 37.7\,MB$$

As mentioned before this memory is used temporarily and deleted after the computation has finished.

5.1.3.2 Ambient occlusion data

More important than the temporary data is the data that needs to be kept in memory at all times. For each fragment of the cube map 4 values need to be stored. The ambient occlusion value, the quality factor, the hashed normal and the depth of the fragment. In order to save memory the precision is kept as low as possible. The ambient occlusion value, the quality factor and the hashed normal do not need to be stored at floating point precision. They can be stored at single

byte precision. The ambient occlusion value is used to modulate a single byte per channel output image. Both the quality factor and the hashed normal are used mainly to find big differences in the values. For example it is more important to find a cube map that has a much higher quality for a specific fragment. It is not so important to find a cube map that is just a little better as the impact on the quality will not be big in that case. The same applies to the hashed normal. The only value that needs to be encoded at floating point precision is the depth of the fragment. It is crucial to find small differences in depth for the shading to be correct. In order to save memory two separate textures are used to save these 4 values. The first texture contains a single floating point channel to store the depth using 4 bytes per value. The second texture contains 3 channels that use one byte each to store the remaining values.

Using the above formula the runtime memory consumption of a single 512 x 512 sized cube map can be computed.

$$S_{cubemap}[Byte] = 1 \cdot 6 \cdot 1 \cdot 4 \cdot 512 \cdot 512 + 1 \cdot 6 \cdot 3 \cdot 1 \cdot 512 \cdot 512 = 11010048\,Byte = 11\,MB$$

In total a single cube map of size 512 x 512 takes up 11 MB in the memory of the graphics card. Placing 10 cube maps consequently takes up 110 MB and so on. One can see that there is a limit of cube maps to place depending on the amount of memory available on the graphics hardware.

5.2 Discussion: Strengths and Weaknesses

After all of the technical discussion the question is now how well this work performs in practice. In order to find an answer to this question the quality of the outcoming result is compared to two other implementations. First there is a raytraced version that is the most true to the original formulation of the ambient occlusion equation (3.4). Given a big enough number of rays, the sampling of the hemisphere becomes close to the analytic integration. In practice the number of rays has to be kept low in order to get a result within a finite timeframe. This may cause artefacts but the outcoming results still allows to get a good sense of how the analytic result would look like. The second reference implementation is based on the use of discrete surface elements to approximate a scene. Form factors between these elements are computed in order to find the influence of all surface elements on a point of the scene. This method is described in a paper by NVIDIA, [Bun05], and is is also used in a seminar work of mine [Osw08]. Figure 5.6 shows a comparison of the outcoming results for these 3 different implementations. Two things are interesting in this comparison. Due to the limited amount of sampling points the cube map ambient occlusion lacks of the soft shadows that are cast by objects that are too far away. In the raytraced solution for example the ear of the bunny casts a subtle shadow on the neck. These types of shadows are missing from the cube map solution. On the other hand fine grained details are captured more than well. The surface structure of the bunny with its dents on the back comes out a lot better than in the surface element based implementation. In terms of quality the cube map based algorithm yields good results for this scene. This has to do with the fact that the structure of the scene can be represented well by a heightfield. There are only few parts that are hidden behind other geometry which would be something that cannot be handled well by the cube map ambient occlusion.

The one-time computation time for setting up a cube map that shades all currently visible fragments is typically below one second for the examples shown here. The surface element based solution typically takes from 2 seconds to up to 7 seconds for average settings on these examples. The implementation that makes use of raytracing is not optimized for performance and can take from minutes to hours depending on the complexity of the scenes.

A comparison for the town scene is shown in figure 5.7. Again the soft shadows casted by big structures far away are missing from the cube map based solution. More importantly a problem is unveiled in this sample. The ambient occlusion values along the border of 2 faces of a cube map are not continuous. When projected back onto the scene this border can be seen sometimes. The red arrow in the figure points at this problem. It can be mitigated by blurring the ambient occlusion values along the edge. Apart from this problem the solution is satisfying. The main property of highlighting coves and ridges is working fine.

Figure 5.8 shows a comparison for the dome scene. What sticks out the most is the missing distance of the computed solution. Although this partly depends on the selected attenuation function, the cube map solution currently does not allow for a wider radius of influence as this would introduce more artefacts. This again has to do with the rather low amount of sampling points used. It is worth noting that the raytraced comparison image is not computed per fragment but instead only for selected points on the surface. This was done to keep the rendering time down. However it is also the reason why the solution is lacking at small details like the windows. At these spots the cube map based version does provide a better quality as it operates at a finer resolution in this example.

Figure 5.9 shows some examples for the outcome of the integration of the ambient occlusion library into another software project.

As mentioned before the structure of the scene plays an important role in the quality of the outcoming solution. This has to do with the fact that a cube map only takes into account the visible geometry for the computation of the ambient occlusion. Geometry that is hidden behind a surface does not contribute to the computation even though it may have an

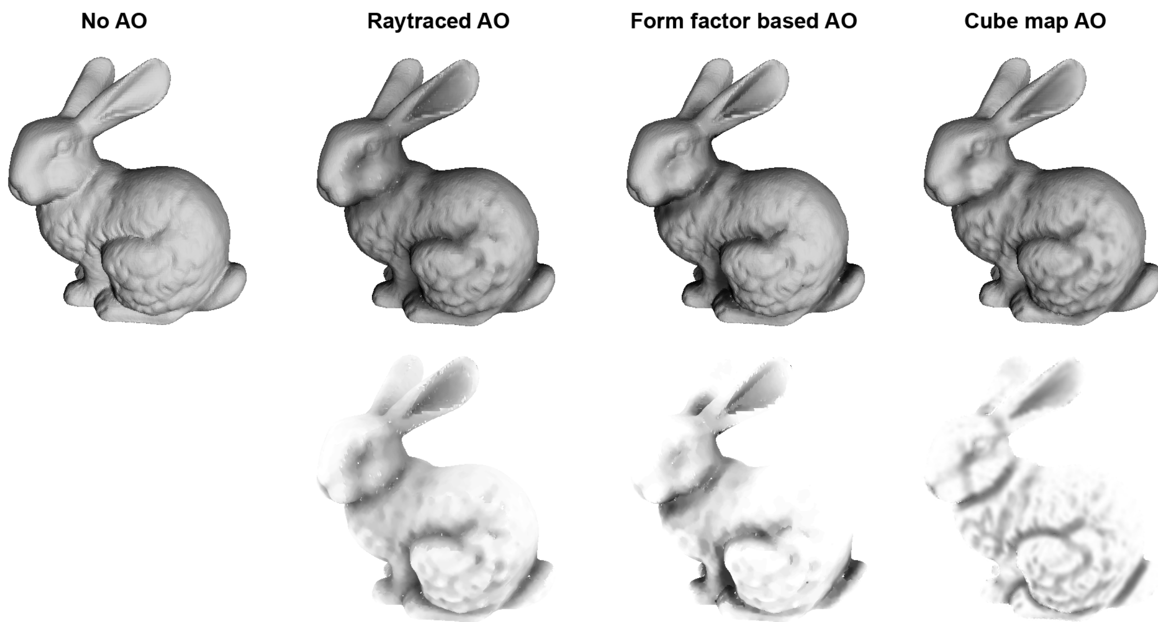


Figure 5.6: Comparison of the quality of the outcoming solution for various different methods. On the very left the original model of the Stanford bunny is shown with no ambient occlusion added. The second image shows a more or less exact solution computed by use of raytracing. The image below shows the pure ambient occlusion. The third image shows the results of my previous, surface element based algorithm. The fourth image shows the outcome using the cube map ambient occlusion presented in this work. The top row shows the ambient occlusion already blended with the original diffuse illumination while the bottom row shows the plain ambient occlusion values.

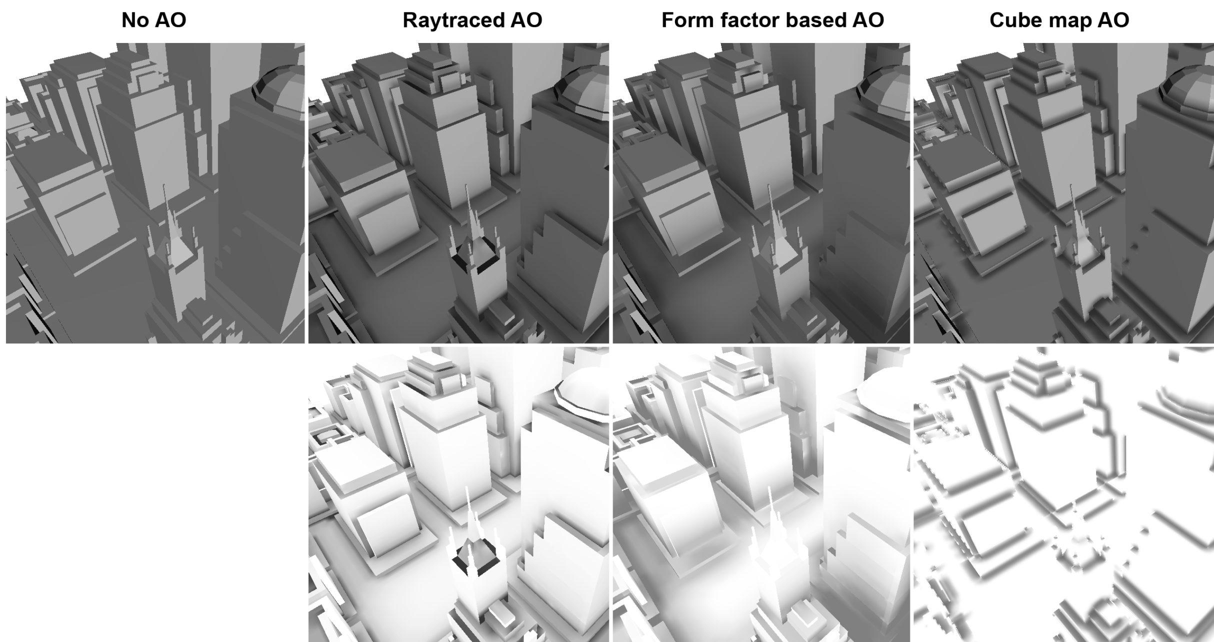


Figure 5.7: First image: Original town scene with no ambient occlusion. Second: Raytraced ambient occlusion. Third: Surface element based ambient occlusion. Fourth: Cube map based ambient occlusion. The red arrow shows the discontinuity along the border edge of the two faces of a cube map.

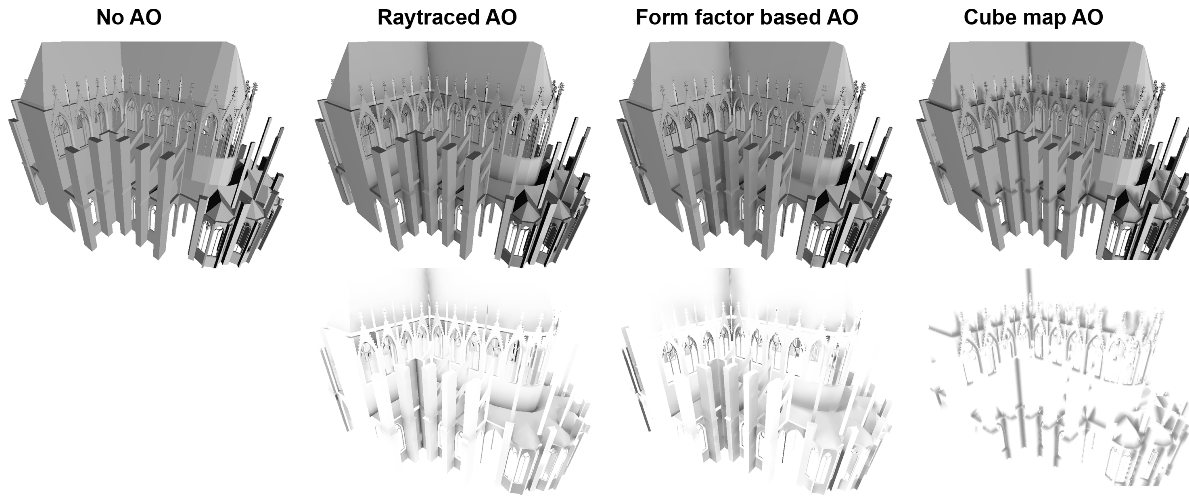


Figure 5.8: First image: The complex dome scene with no ambient occlusion. Second: Raytraced ambient occlusion. Third: Surface element based ambient occlusion. Fourth: Cube map based ambient occlusion. The shadows in the cube map based version are shorter which takes away a lot of the typical ambient occlusion look.

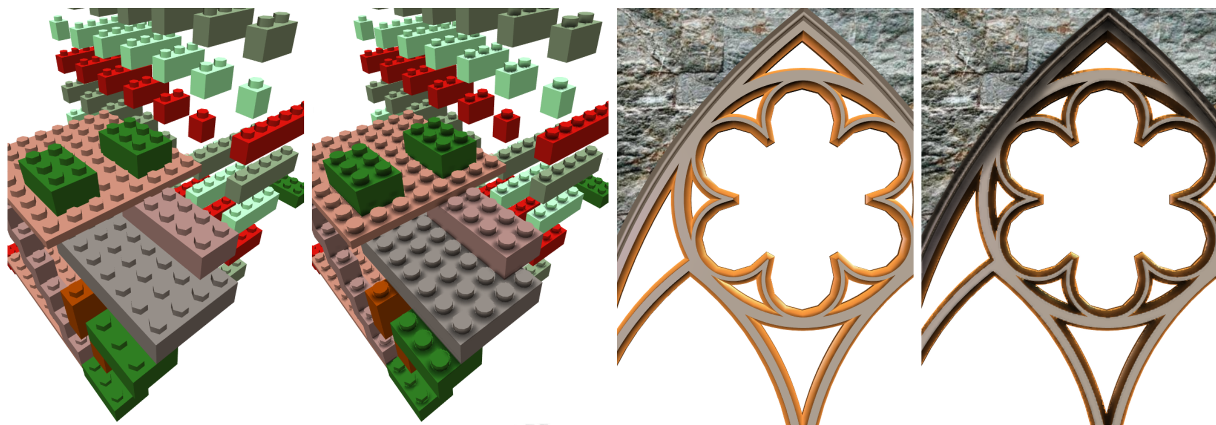


Figure 5.9: Screenshots of two different scenes with and without cube map ambient occlusion. The images are the result of an integration of the ambient occlusion library into another software project.

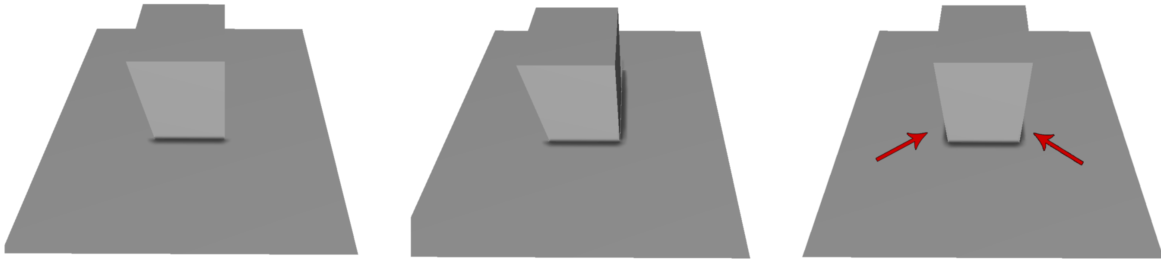


Figure 5.10: First image: The left and right side of the cube do not cast any shadow on the ground as long as they are not visible to the cube map. In this image one cube map is placed at the same position as camera. Second: Once a side becomes visible to the cube map it also casts a shadow. Third: Two cube maps are placed to the left and to the right of the cube which is why the sides are casting a shadow in this image.

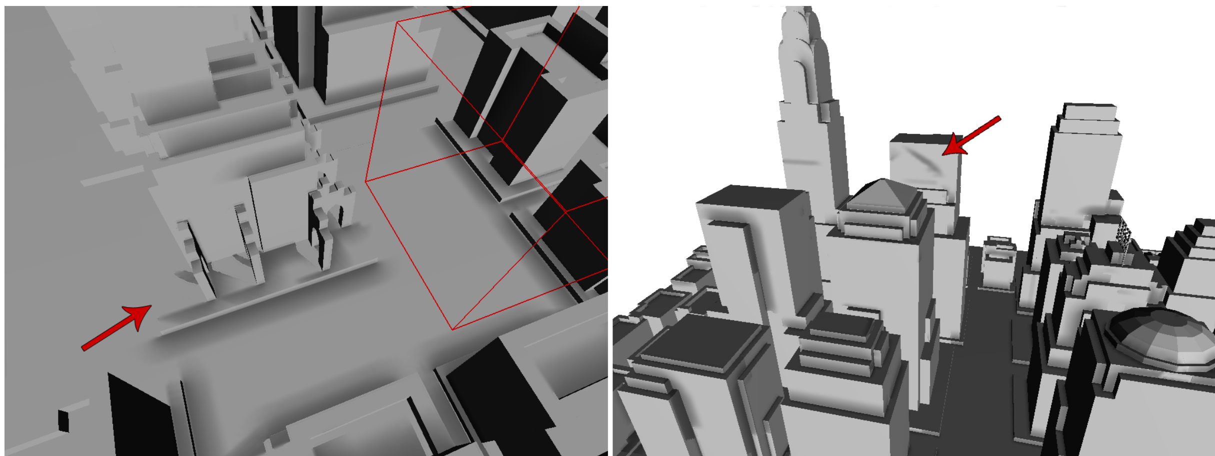


Figure 5.11: Left: A cube map is placed at the position of the red wire cube. The camera is then moved to the left to show an area of the scene that does not have complete ambient occlusion coverage. The pillars are partly blocking the sight of the cube map to the ground. These parts of the ground show a bright white lighting while the cube map projected parts show a darkened lighting. Right: Again a cube map is placed and the camera move away slightly. In this example the roof of a building is the cause of an unnatural shadow on another building far away.

impact on the solution. This is a typical problem to most screen space based techniques. This issue may be mitigated by the placement of more than 1 cube map. Surfaces not seen by one cube map may be visible to another cube map and thus cast shadows that may contribute to the final image. Figure 5.10 demonstrates this problem and shows how it is mitigated by the placement of multiple cube maps.

5.2.1 Out-Takes

The above section shows the problems that may arise when using the algorithms described in this work. Some of the problems can be worked around by clever placement of the cube maps. When this fails, severe rendering inconsistencies may show. Figure 5.11 shows two of these problems.

The problem at the right side of figure 5.11 can be mitigated by employing an attentuation function $\rho(L)$ that leaves away geometry that is too far away. The problem is to find the right function for every type of scene. Some scenes may be very compacted and have a tiny radius while others may stretch very far. Given these different types of settings a fixed attenuation function may work for some scenes but fail for others. A parameter could be added to allow the user to choose between differing cutoff distances but this would introduce a lot more tweaking work for the user. The current implementation does contain an attenuation function based on an estimate of the size of the scene, however this may not always work perfectly. In order to show the problem this function has been disabled when taking the screenshots for figure 5.11.

The problem at the left side of the figure may be mitigated by clever placement of the cube maps. The problem does not show for example when a cube map is placed at the position of the camera when it comes to rest.

5.3 Conclusion and Future Work

This work has shown how to derive a screen-space based ambient occlusion solution that allows to store and restore the results in cube maps placed in the scene. Computing the ambient occlusion for one cube map is a quick process that is best performed by the user at the push of a button whenever it is needed. No expensive precomputation is necessary. The resulting lighting information is projected in all directions around the location of the cube map. This is why it is best suited for scenes where the geometry is surrounding the user and the user is not moving too fast. It still works for scenes where the geometry is only at the center and the user is rotating around the object but may require more cube maps to be placed.

Even though a lot of work has been put into this project there is still a range of things that can be improved. Here is an attempt to list all of the important points.

In the projective texturing step a full screen quad is rendered in order to invoke a fragment shader program for every pixel of the screen. This is done multiple times, one time for every cube map. In case the current scene does not fill the entire screen a lot of unnecessary fragments are shaded. Background fragments should not be shaded but discarded instead. This goal can be achieved by using the *stencil buffer*. The stencil buffer can be used to control the rendering into the colour buffer [AMHH02], where each image is created before it is shown on the screen. In an initial pass, all fragments that do contain actual scene geometry are marked separately in the stencil buffer. In the following passes the stencil buffer can be used to exclude background fragments from being shaded. This technique is typically used in *deferred shading* applications to prevent fragments from being shaded when they are not within the radius of a light source. Deferred shading denotes a technique that allows to reduce the workload when lots of light sources are used in a rendering. More information along with some critical concerns about this topic can be found in [Koo07].

In order to speed up computation times further it may be useful to investigate the use of upsampling techniques such as *joint bilateral upsampling* [KCLU07]. The idea here is to perform the computation at a lower resolution and then use upsampling to get to the screen resolution. A crude up- or downsampling is already carried out in this work in case the cube map resolution does not equal the screen resolution. However there exist much more elaborate algorithms that allow to retain a lot more detail than by simply upscaling the image as done in this work. This may even make the blurring step superfluous. The blurring step in fact is similar to a downsampling of the solution followed by an immediate upsampling. Tiny artefact start to vanish because of this. It may be more useful to just compute the low resolution image which is possibly lacking the high frequency artefacts and then perform upsampling.

The blurring step itself can also be optimized further in future versions. For a filter of size $n x n$ currently n^2 computations are needed per pixel. This is currently limiting the size of the filter and also the quality of the resulting solution. Every computation involves fetching data from a texture as well as performing the necessary arithmetic operations. It is desirable to keep the amount of these two necessary operations as low as possible. In computer vision there are well known and well studied methods of reducing the workload. One of these methods is to use separable filter kernels. Instead of using one filter performing n^2 computations per pixel this technique is using two filters performing n computations each. For plain gaussian linear filters the results of these two techniques are identical. Unfortunately the theory behind this does not apply to bilateral filters. Research is showing however that it may still be applied at the risk of lower quality [PKTD08]. However there may be better alternatives altogether so it might be worth doing more research in this direction.

Another issue is the relatively large amount of memory used up by every single cube map. OpenGL as well as DirectX both support the use of compressed textures for rendering. The idea is to compress the raw texture data upfront and use realtime decompressing schemes to get to the uncompressed texture data. A commonly used compression technique is the S3TC texture compression [AMHH02]. It is a lossy, block based, scheme that allows for compression ratios of up to 8:1 [vWC07]. Algorithms exist that are capable of creating compressed images quickly. Information on these is found for example in [vW06] and [vWC07]. In a future work it may be interesting to bring down the memory consumption of the cube maps using a real time S3TC compression scheme. The question is if this type of compression suits the type of data needed by the cube maps. Chances are that it does not behave very well on the numeric values for the quality factor or on the hashed surface normal and the other values. This means research needs to be done to find out if it is suitable in the first place. In case it is, one still needs to implement a fast enough compression technique that produces good quality images.

Another way to bring down memory consumption a little bit is to leave away the storage of the hashed surface normal. This data is used primarily for the implementation of the edge aware blurring filter. In image processing, where the normal of a picture is typically not available, algorithms make use of the image intensity instead to find the edges. It may be worth investigating the performance of such an implementation. The savings achieved by this would be 1 Byte per cube map pixel. Currently the total consumption is exactly 7 Byte per pixel. So the saving may not be much but still useful

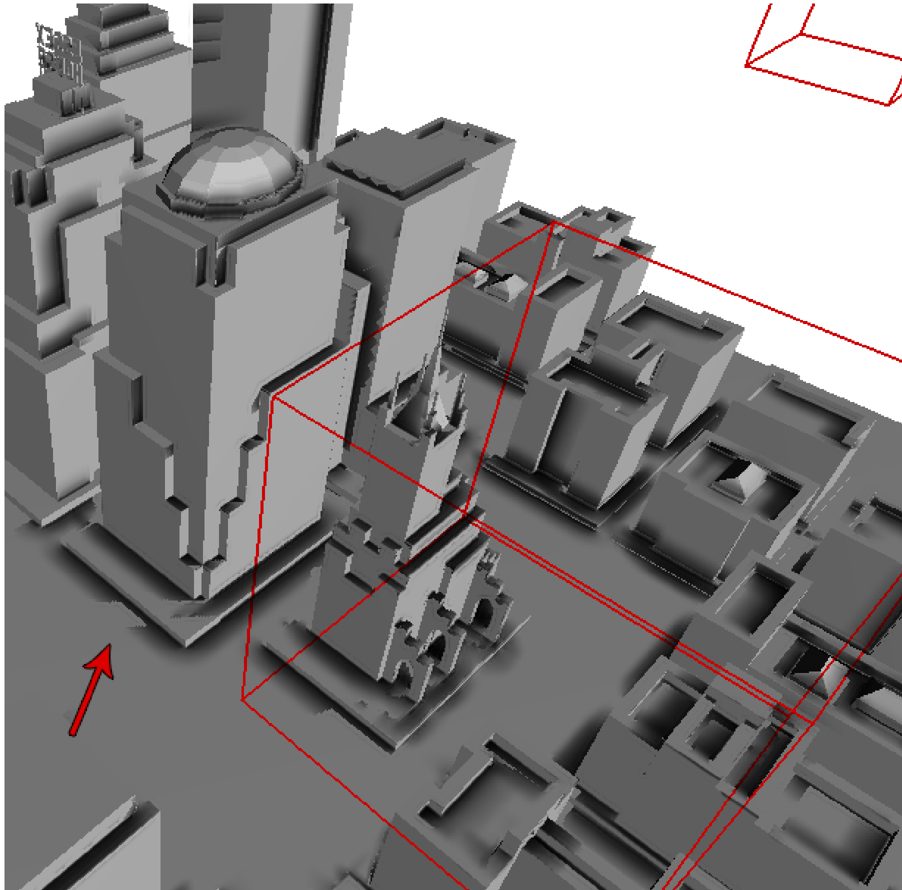


Figure 5.12: The arrow hints to a spot where incorrect ambient occlusion information is projected on a surface. The cube map in the back is chosen over the one in the front for this particular spot. The chosen cube map does not contain a correct lighting information. The cube map in the front is likely to contain a better solution but it is not picked by the algorithm.

considering that the implementation is not very complicated. The quality is something that would need to be evaluated however.

Another point that needs to be discussed is the library interface that is explained in chapter 4.7. The downside of this interface is that it is not ideal for a dynamic update of the ambient occlusion. In case the scene geometry changes, the surface description needs to be handed to the library again. It is possible to create an interface that does not depend on the surface description. A couple of different ways could be explored. A straight forward approach would be to have the host application render the scene 6 times; one time for every side of a cube map. The 6 render calls could be wrapped by a library interface. However this may be more complicated for people to use. Passing a function pointer that allows the library to call the render routine by itself may be another idea worth exploring. This would probably give rise to other problems however. The user would have to have a separate rendering routine in his application which may not always be given or wanted.

Problems also arise sometimes when blending the results of multiple cube maps placed in close proximity. The quality factor is supposed to ensure that only the best-placed cube map is used for a specific part of the scene surface. Unfortunately this does not always work out as can be seen in figure 5.12. The way the quality factor is computed in this work is based solely on heuristics. More research can be put into finding ways around this problem. Instead of using just one cube map for a particular spot one may also try to blend the values of multiple cube maps to get an averaged result. This may work better than the current solution however it may also increase the complexity and the memory requirements of this rendering step.

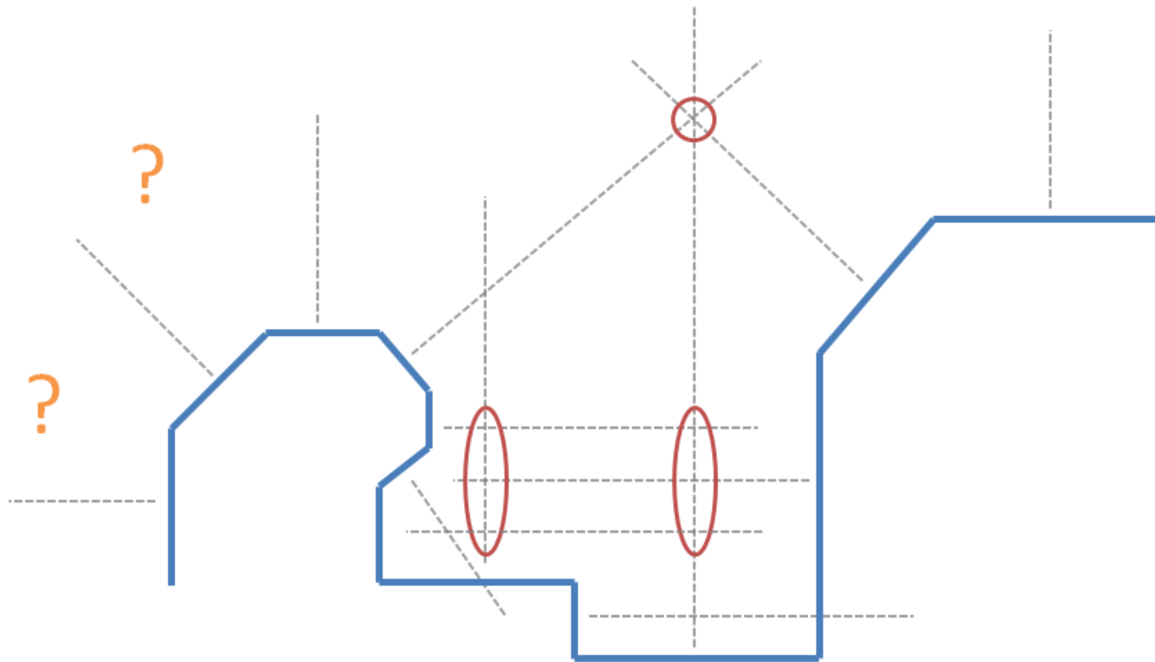


Figure 5.13: Depiction of a voting scheme to find good cube map locations automatically. The red circles and ellipses mark areas of higher voting count. In this depiction rays are only shot from the center of a polygon. One may also shoot more than one ray, distributing the other rays across the polygon. The orange question marks show areas where this algorithm would possibly fail.

5.3.1 Automated cube map placement

There is still one important point that needs to be discussed. It is the question of how and where to place the cube maps. A system would be ideal if it was able to shade all parts of the surface with little distortion and as few cube maps as possible. Leaving distortions and limited quality aside one may ask for a solution to the question how many cube maps are needed to cover all surfaces. This is a similar question as the one posed by the *art gallery problem*. This well known problem arises from the question how many cameras are necessary to guard a gallery. The goal is to find the minimum number of cameras and their placement that allows to view all parts of the gallery. Information on this topic can be found in [O'R87]. Research concerning this topic focuses mainly on the two dimensional problem. Very little is known about this problem in three dimensions which is what is needed in this work.

A possible but stupid solution for the placement would be to place the cube maps on a regular three dimensional grid. A bit more refined is the approach of using an octree like subdivision of the scene to focus on areas of higher geometric complexity. Ideas on this approach are presented in [Gam05].

Another possible idea to find a good cube map placement would be to employ a voting scheme where the surface normals are voting for discrete voxels in space. A ray originating from every polygon of the scene in the direction of the normal may vote for every voxel it passes through. In the end the voxels with the highest voting count may serve as an indication for good cube map positions. One can see this as a way of solving an optimization task for the term Q_c of the quality factor described in this work. Figure 5.13 shows how this scheme may work.

5.4 Interesting bugs

During the making of this thesis not everything always worked out immediately. Figure 5.14 shows three of the more interesting images resulting from erroneous code.

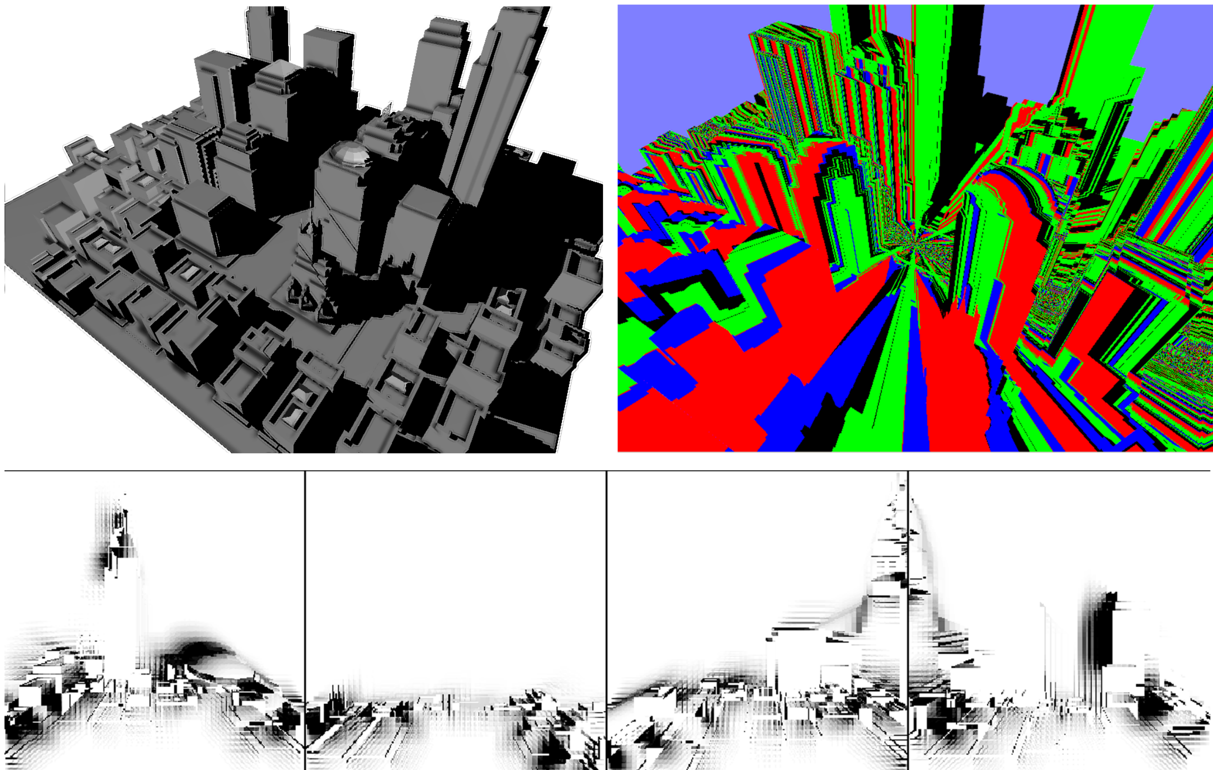


Figure 5.14: Top left: What can be seen in this image can be labeled "It is not a bug, it is a feature". The interesting thing is that, due to an error in the projective texturing algorithm, realistic hard shadows appeared in addition to the ambient occlusion. This is because the projective texturing is in fact similar to the shadow mapping technique. Top right: This is what happens when computing the wrong texture coordinates for projective texturing and at the same time binding the texture containing the normal data instead of the ambient occlusion data. Bottom: Interesting aliasing patterns occurred when playing around with the parameters of the scene sampling.

Bibliography

[AMHH02] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering 2nd Edition*. A. K. Peters, Ltd., 2002. 1, 19, 53

[BSD08] BAVOIL L., SAINZ M., DIMITROV R.: Image-space horizon-based ambient occlusion. In *SIGGRAPH '08: ACM SIGGRAPH 2008 talks* (New York, NY, USA, 2008), ACM, pp. 1–1. 12, 32

[Bun05] BUNNELL M.: *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005, ch. Dynamic ambient occlusion and indirect lighting, pp. 223–233. 9, 10, 49

[DBS08] DIMITROV R., BAVOIL L., SAINZ M.: Horizon-split ambient occlusion. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM, pp. 1–1. 12, 31

[Eva06] EVANS A.: Fast approximations for global illumination on dynamic scenes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses* (New York, NY, USA, 2006), ACM, pp. 153–171. 14

[Gam05] GAME DEVELOPERS CONFERENCE:. *Irradiance volumes for games* (2005). 55

[GTGB84] GORAL C. M., TORRANCE K. E., GREENBERG D. P., BATTAILE B.: Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 213–222. 5

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F.: A survey of real-time soft shadows algorithms. *Computer Graphics Forum 22*, 4 (dec 2003), 753–774. 7

[IKSZ03] IONES A., KRUPKIN A., SBERT M., ZHUKOV S.: Fast, realistic lighting for video games. *IEEE Comput. Graph. Appl. 23*, 3 (2003), 54–64. 9, 18

[JH07] JARED HOBEROCK Y. J.: *GPU Gems 3*. Addison Wesley, 2007, ch. High-Quality Ambient Occlusion. 9

[Kaj86] KAJIYA J. T.: The rendering equation. *SIGGRAPH Comput. Graph. 20*, 4 (1986), 143–150. 1, 17, 18

[KCLU07] KOPF J., COHEN M., LISCHINSKI D., UYTTENDAELE M.: Joint bilateral upsampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26*, 3 (2007), to appear. 53

[KL05] KONTKANEN J., LAINE S.: Ambient occlusion fields. In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games* (2005), ACM Press, pp. 41–48. 11

[Kne07] KNECHT M.: *State of the Art Report on Ambient Occlusion*. techreport, Vienna Institute of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, Nov. 2007. human contact: technical-report@cg.tuwien.ac.at. 7

[Koo07] KOONCE R.: *GPU Gems 3*. Addison Wesley, 2007, ch. Deferred Shading in Tabula Rasa. 53

[McG10] MCGUIRE M.: Ambient occlusion volumes. In *Proceedings of High Performance Graphics 2010* (June 2010). 13, 14

[MFS09] MENDEZ-FELIU A., SBERT M.: From obscurances to ambient occlusion: A survey. *Vis. Comput. 25*, 2 (2009), 181–196. 7

[Mil94] MILLER G.: Efficient algorithms for local and global accessibility shading. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), ACM, pp. 319–326. 5

[Mit07] MITTRING M.: Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), ACM, pp. 113–115. 12

[MK05] M. KNUTH A. F.: Self-shadowing of dynamic scenes with environment maps using the gpu. In *WSCG'2005 - Full Papers* (2005). 7, 8

[MMAH07] MALMER M., MALMER F., ASSARSSON U., HOLZSCHUCH N.: Fast precomputed ambient occlusion for proximity shadows. *Journal of Graphics Tools 12*, 2 (2007), 59–71. 11

[O'R87] O'ROURKE J.: *Art gallery theorems and algorithms*. Oxford University Press, Inc., New York, NY, USA, 1987. 55

[Osw08] OSWALD J.: *Pre-processed ambient occlusion*. Tech. rep., Institute of Computer Graphics and Knowledge Visualization - Graz University of Technology, Austria, 2008. 10, 49

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. 9

[PKTD08] PARIS S., KORNPROBST P., TUMBLIN J., DURAND F.: A gentle introduction to bilateral filtering and its applications. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes* (New York, NY, USA, 2008), ACM, pp. 1–50. 20, 53

[RBA09] REINBOTHE C., BOUBEKEUR T., ALEXA M.: Hybrid ambient occlusion. *EUROGRAPHICS 2009 Areas Papers* (2009). 12, 13

[RGS09] RITSCHEL T., GROSCH T., SEIDEL H.-P.: Approximating dynamic global illumination in image space. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), ACM, pp. 75–82. 12, 13

[RMSD*08] ROPINSKI T., MEYER-SPRADOW J., DIEPENBROCK S., MENSMANN J., HINRICHS K. H.: Interactive volume rendering with dynamic ambient occlusion and color bleeding. *Computer Graphics Forum (Eurographics 2008) 27*, 2 (2008), 567–576. 14

[Ros06] ROST R. J.: *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006. 28

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM, pp. 527–536. 5

[SSZK04] SATTLER M., SARLETTE R., ZACHMANN G., KLEIN R.: Hardware-accelerated ambient occlusion computation. In *Vision, Modeling, and Visualization 2004* (Nov. 2004), Girod B., Magnor M.,, Seidel H.-P., (Eds.), Akademische Verlagsgesellschaft Aka GmbH, Berlin, pp. 331–338. 8

[SZ98] S. ZHUKOV A. IONES G. K.: Using light maps to create realistic lighting in real-time applications. In *In Proceedings of WSCG'98 - Central European conference on Computer Graphics and Visualization'98* (1998), pp. 464–471. 5, 38

[vW06] VAN WAVEREN J.: Real-time dxt compression. Whitepaper found on the Intel Software Network.), 2006. 53

[vWC07] VAN WAVEREN J., CASTAÑO I.: Real-time ycocg-dxt compression. Whitepaper found on the NVIDIA developer zone.), 2007. 53

[Was05] WASSENIUS C.: *Accelerated Ambient Occlusion Using Spatial Subdivision Structures*. Tech. rep., Department of Computer Science and Electrical Engineering - UMBC Honors University in Maryland, 2005. 10

[ZIK98] ZHUKOV S., INOES A., KRONIN G.: An ambient light illumination model. In *Rendering Techniques '98* (1998), Drettakis G., Max N., (Eds.), Eurographics, Springer-Verlag Wien New York, pp. 45–56. 5

List of Figures

1.1 Various illumination models . . . 2
1.2 Ambient Occlusion Motivational . . . 3

2.1 Real life light transport phenomena . . . 6
2.2 Ambient Occlusion in real life . . . 6
2.3 Outside In and Inside Out Approach . . . 7
2.4 Ambient Occlusion using Shadow Mapping . . . 8
2.5 Hardware Accelerated Ambient Occlusion . . . 8
2.6 NVIDIA's Dynamic Ambient Occlusion . . . 10
2.7 NVIDIA's Dynamic Ambient Occlusion Disc Hierarchy . . . 10
2.8 AO Fields Spherical Cap . . . 11
2.9 Screen Space Directional Occlusion . . . 13
2.10 Hybrid Ambient Occlusion . . . 13
2.11 Ambient Occlusion Volumes . . . 14

3.1 Weighting function . . . 19
3.2 Cubemap . . . 20
3.3 Projective texturing sample . . . 21
3.4 Gaussian and bilateral filtering comparison . . . 21
3.5 Algorithm Sketch . . . 22

4.1 Reduced Class Diagram . . . 24
4.2 RenderToTexture2D Enumerations Explanation . . . 25
4.3 Cube map containing world position and normal data . . . 29
4.4 Rendering Pipeline . . . 30
4.5 Horizon Samples . . . 32
4.6 Piece-wise linear horizon example . . . 32
4.7 Iterative Horizon Computation . . . 32
4.8 Sampling step size . . . 34
4.9 Delta-x computation . . . 35
4.10 Cube map quality factor . . . 36
4.11 Cube map SSAO Data . . . 36
4.12 Projective texturing explanation . . . 37
4.13 Projective texturing side selection . . . 38
4.14 Blending ambient occlusion . . . 39
4.15 Artefacts and blurring filter . . . 40
4.16 Edge detection in town scene . . . 40
4.17 Ambient occlusion fill percentage . . . 42
4.18 Face indexing schemes . . . 44

5.1 Benchmark test scenes . . . 45
5.2 Profiling Cube map resolution . . . 46
5.3 Profiling shader params directions . . . 47
5.4 Profiling shader params distance . . . 47
5.5 Profiling amount of cube maps . . . 48
5.6 Quality analysis for the bunny model . . . 50

5.7 Quality analysis for the town model 50
5.8 Quality analysis for the dome model 51
5.9 GML integration images 51
5.10 Analysis of the hidden surface problem 52
5.11 Analysis of unseen areas and false shadows problem 52
5.12 Cube map blending problem 54
5.13 Cube map placement voting scheme 55
5.14 Bug images 56