Master Thesis

Illumination for Real-Time Rendering of Large Architectural Environments

by
Markus Fahlén

LITH-ISY-EX--05/3736--SE

2005-12-19

Master Thesis

Illumination for Real-Time Rendering of Large Architectural Environments

by **Markus Fahlén**

LITH-ISY-EX--05/3736--SE

Supervisor : **Josep Blat**
Department of Technology
at Universitat Pompeu Fabra

Examiner : **Ingemar Ragnemalm**
Department of Electrical Engineering
at Linköpings universitet

Avdelning, Institution
Division, Department

ICG,
Department of Electrical Engineering
581 83 LINKÖPING

LINKÖPINGS UNIVERSITET

Datum
Date

2005-12-19

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LITH-ISY-EX--05/3736--SE

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version
http://www.ep.liu.se/exjobb/isy/2005/dd-d/3736/

Titel Illumination för realtidsrendering av stora arkitektoniska miljöer

Title Illumination for Real-Time Rendering of Large Architectural Environments

Författare Markus Fahlén
Author

Sammanfattning
Abstract

This thesis explores efficient techniques for high quality real-time rendering of large architectural environments using affordable graphics hardware, as applied to illumination, including window reflections, shadows, and ”bump mapping”. For each of these fields, the thesis investigates existing methods and intends to provide adequate solutions. The focus lies on the use of new features found in current graphics hardware, making use of new OpenGL extensions and functionality found in Shader Model 3.0 vertex and pixel shaders and the OpenGL 2.0 core. The thesis strives to achieve maximum image quality, while maintaining acceptable performance at an affordable cost.

The thesis shows the feasibility of using deferred shading on current hardware and applies high dynamic range rendering with the intent to increase realism. Furthermore, the thesis explains how to use environment mapping to simulate true planar reflections as well as incorporates relevant image post-processing effects. Finally, a shadow mapping solution is provided for the future integration of dynamic geometry.

Nyckelord illumination, real-time rendering, large architectural environments, affordable graphics hardware
Keywords

Abstract

This thesis explores efficient techniques for high quality real-time rendering of large architectural environments using affordable graphics hardware, as applied to illumination, including window reflections, shadows, and "bump mapping". For each of these fields, the thesis investigates existing methods and intends to provide adequate solutions. The focus lies on the use of new features found in current graphics hardware, making use of new OpenGL extensions and functionality found in Shader Model 3.0 vertex and pixel shaders and the OpenGL 2.0 core. The thesis strives to achieve maximum image quality, while maintaining acceptable performance at an affordable cost.

The thesis shows the feasibility of using deferred shading on current hardware and applies high dynamic range rendering with the intent to increase realism. Furthermore, the thesis explains how to use environment mapping to simulate true planar reflections as well as incorporates relevant image post-processing effects. Finally, a shadow mapping solution is provided for the future integration of dynamic geometry.

Keywords : illumination, real-time rendering, large architectural environments, affordable graphics hardware

Acknowledgements

I would like show my appreciation to Josep Blat (Director of the Institut Universitari de l'Audiovisual), Daniel Soto, and Juan Abadía of Universitat Pompeu Fabra for their time and guidance, making possible the realization of this thesis.

I also want to thank the members of the `OpenGL.org`, `GPGPU.org`, and `GameDev.net` forums for their help and quick replies on questions regarding more recent features found in OpenGL 2.0 and current extensions, among other things.

Lastly, but not least, I would like to thank Eduard and Sergi González for their valuable input on various topics and Toni Masó, with whom I worked on the project, for his collaboration.

The Barcelona city block used in the demo application is courtesy of Anima 3D S.L. (`http://www.anima-g.com/`) and the indoor Cloister model was created by Adriano del Fabbro (`http://www.3dcafe.com/`).

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Objectives . . . 2
- 1.3 Problem Description . . . 3
- 1.4 Document Overview . . . 4
- 1.5 Reading Instructions . . . 5

2 Deferred Shading **7**
- 2.1 Requirements . . . 8
- 2.2 G-buffer . . . 9
- 2.3 Optimizations . . . 10
- 2.4 Anti-Aliasing . . . 13
  - 2.4.1 Edge Detection . . . 13
    - Color Gradient . . . 13
    - Depth and Normal Discontinuities . . . 15

3 Reflection **19**
- 3.1 Optics in a Window Pane . . . 20
  - 3.1.1 Fresnel Equations . . . 20
  - 3.1.2 Multiple Reflections and Refractions . . . 23
    - Total Reflection Coefficient . . . 23
  - 3.1.3 Blur . . . 25
- 3.2 Computation of Reflections . . . 26
  - 3.2.1 True Planar Reflections . . . 26

"Level-of-Detail" 26
3.2.2 Cube Environment Mapping 27
3.2.3 Paraboloid Environment Mapping 29
3.2.4 Accurate Reflections 34
Using a Distance Cube Map 35
Approximating Distance with a Plane 36
3.3 High Dynamic Range 37
3.3.1 Tone Mapping 40
Average Luminance 40
Scaling and Compression 42
Parameter Estimation 43
Alternative Formats 44
3.4 Bloom 44
3.4.1 Convolution on the GPU 46
3.4.2 Different Approaches 48
Repeated Convolution 48
Downsampling Filtered Textures 49

4 Lighting **53**
4.1 Global Illumination 53
4.2 Light Mapping 54
4.2.1 High Dynamic Range 54
4.3 Ambient Occlusion 55

5 Shadows **57**
5.1 Common Methods 58
5.1.1 Stenciled Shadow Volumes 58
5.1.2 Projected Planar Shadows 59
5.1.3 Shadow Mapping 59
5.2 Shadow Mapping 59
5.2.1 Theory 61
5.2.2 Shadow Acne 61
5.2.3 Dueling Frusta 62
5.2.4 Soft Shadows 62
5.2.5 Omni-Directional Shadows 64
Cube Shadow Mapping 66

Dual-Paraboloid Shadow Mapping 66
Sampling Rate 69
Non-Linear Depth Distribution 70

6 Surface Detail **71**
6.1 Common Methods 71
6.1.1 Displacement Mapping 71
6.1.2 Bump and Normal Mapping 72
6.1.3 Ray-Tracing Based Methods 72
6.2 Normal Mapping 72
6.2.1 Tangent Space 72
6.2.2 Implementation Details 74
6.3 Parallax Mapping 75

7 Discussion **79**
7.1 Requirements 79
7.1.1 Hardware 79
7.1.2 Software 80
7.2 Evaluation 80
7.2.1 Performance 80
7.2.2 Image Quality 83
7.3 Future Work 85
7.4 Conclusion 86

A Tools **87**

Bibliography **88**

Index **93**

Chapter 1

Introduction

This chapter gives an overview of the document and explains the objectives of the thesis.

1.1 Background

Interactive high quality real-time 3D graphics have traditionally been restricted to very expensive hardware. Recent industrial developments driven by the gaming industry have made available mainstream graphics cards with staggering computational power and capabilities. Affordable high quality graphics is now expanding very quickly and the new possibilities provided by the advances in technology are very much worth exploring in fields outside the world of computer games. An example of the exploitation of these possibilities in a seemingly very disconnected field, drug discovery, is provided by the OpenMOIV development (`http://www.tecn.upf.es/openMOIV/`) related to the Link3D project (`http://www.tecn.upf.es/link3d/`) although the approximation in that development is quite different from the approach in this thesis.

As the processing power of GPUs keeps increasing, so does the demand for handling ever more complex geometry and texture detail. One area where this holds true is the visualization of very large architectural envi-

ronments, such as a city block or even an entire city. Architects and city planners alike look for ways to further increase the realism of models used of already existing and still non-existing buildings, parks, etc. to better visualize and more easily be able to foresee the outcome of construction projects. The projects developed by the company Anima 3D S.L., which has provided models for tests in this thesis, is one example of commercial use of these aspects in the architecture and urban planning fields.

With the current developments in graphics hardware, the two major bottlenecks in the rendering pipeline are the CPU and the fragment processor. As GPUs get faster and pixel shaders get more complex, this trend is not expected to change. Poor batching of data sent to the graphics card for processing leads to excessive draw calls. Improved functionality provided by Shader Model 3.0 [1], features found in OpenGL 2.0 [2], and extensions exposing new hardware functionality can help remove potential bottlenecks, greatly improving performance.

1.2 Objectives

This thesis should explore efficient techniques for high quality real-time rendering of large architectural environments using affordable graphics hardware, as applied to illumination, including window reflections, shadows, and surface detail. For each of these fields, the thesis should investigate existing methods and intend to provide adequate solutions. The focus lies on the use of new features found in current graphics hardware, making use of new OpenGL extensions and functionality found in Shader Model 3.0 vertex and pixel shaders and the OpenGL 2.0 core. The thesis should strive to achieve maximum image quality, while maintaining acceptable performance at an affordable cost.

The thesis was done for Universitat Pompeu Fabra (`http://www.upf.es/`), which collaborates with Anima 3D S.L., and for this reason the purpose of the thesis was visualization of architectural environments. The final result of the thesis is represented by the written report and example code.

1.3 Problem Description

While meeting the above objectives, the thesis should take into consideration the following requisites:

- The environment will be both outdoor and indoor, though mainly outdoor and never the two simultaneously.
- There is no restriction on the number of light sources.
- The geometry will be both static and dynamic, e.g. buildings and people respectively.

In order to visualize very large architectural environments in real-time, an extensive amount of vertices must be processed, possibly leading to excessive draw calls, making the application CPU bound and thus decreasing performance. This becomes more of a problem when algorithms require multiple rendering passes. Attention must also be paid to the reduction of fragment processing bottlenecks by cutting down on shader execution time, possibly by utilizing new hardware features. Modern graphics applications do not tend to be vertex bound, so the volume of geometry needed to be processed should not pose any problem, unless multiple rendering passes are necessary.

For realistic illumination of outdoor and indoor environments, one would normally opt for a global illumination model such as radiosity, photon mapping, or spherical harmonics. However, these models would prove too computationally expensive for scenes with dynamic geometry and multiple light sources, especially when taking into account the complexity of the intended geometry. The mentioned methods require multiple passes and are more suited for offline rendering. The dynamic geometry makes the option of accurate pre-calculation for these models impossible.

For architectural walkthroughs in urban environments, planar window reflections play an important role. Unfortunately, completely accurate planar reflections require an additional rendering of the scene for every reflecting plane. At any given moment, the number of visible reflecting planes can range anywhere from one to three or more. Computing these reflections in real-time significantly degrades performance. Other important considerations when modeling window reflections include how window glass reflects

light differently for different angles and how very bright reflections are perceived by the human eye.

A common problem when applying more or less complex lighting calculations, is the amount of overdraw[1]. Expensive calculations are performed for pixels never ending up in the final image, making poor use of the GPU. For geometrically complex environments, the amount of overdraw could be significant.

No prior framework is available for the implementation and the thesis should provide solutions to the above (and possibly other) issues related to the visualization of architectural environments for the following areas:

- reflections
- lighting
- shadows
- surface detail

1.4 Document Overview

Below follows an overview of the chapters in this document.

Chapter 1
This chapter gives an overview of the document and explains the objectives of the thesis.

Chapter 2
This chapter introduces the concept of deferred shading and motivates its use within context of the thesis.

Chapter 3
This chapter explains difficulties and possible solutions for generation of window reflections.

[1] The number of pixels passing the z-test divided by the screen area [3], i.e. a measure of how many times a pixel is written to the framebuffer before being displayed

Chapter 4
This chapter talks about implications of lighting as applied to high quality rendering of large architectural environments.

Chapter 5
This chapter talks about problems involved with current shadow techniques and good choices within the current context.

Chapter 6
This chapter reviews methods often used for adding small-scale surface detail to objects without increasing their geometric complexity, and explains in greater detail the methods used in the implementation part of the thesis.

Chapter 7
This chapter evaluates the results of the thesis and gives a conclusion.

Appendix A
This appendix lists the tools used for the implementation part of the thesis.

1.5 Reading Instructions

The reader is encouraged to read chapters 1 and 2 before any of the following chapters in order to ensure an understanding of the underlying objectives of the thesis and the framework of deferred shading. Chapter 7 can be read at any time after having read the two introductory chapters if one quickly wants to find out the conclusion of the thesis.

If previously unacquainted with computer graphics, the reader will benefit from skimming through an introductory book on the topic in order to gain a basic understanding of standard concepts and terminology.

It should be mentioned that all screenshots in this text were generated by my implementation, as this may otherwise not be obvious since it has not been noted later on.

Chapter 2

Deferred Shading

This chapter introduces the concept of deferred shading and motivates its use within the context of the thesis.

Some reading about available methods for shading indicates that for the intent of this thesis *deferred shading* (or quad shading) [3, 4, 5] may show promise, and it was decided to study the viability of using this method on current graphics hardware and for the rendering of large architectural environments. Deferred shading was first introduced by Michael Deering et al. at SIGGRAPH 1988, but it is only recently becoming practical for real-time graphics applications because of its high hardware requirements.

Traditional *forward shading* of a scene performs shading calculations as geometry passes through the graphics pipeline. This often results in hidden surfaces needlessly being shaded, difficulties in handling scenes containing many lights, and lots of repeated work such as vertex transformations and anisotropic filtering. Deferred shading solves these issues while introducing some other limitations. With this method of shading, vertex transformations and rasterization of primitives is decoupled from the shading of generated fragments. All geometry is rendered only once and necessary lighting properties are written into a so-called *G-buffer* (or geometric buffer) in this first pass. Later, all shading calculations are performed as 2D post-processes on the G-buffer.

Deferred shading has significant advantages over standard forward shad-

ing, but at the same time this technique introduces new difficulties. Listed below are the key advantages of using deferred shading:

- simplified batching
- each triangle is rendered only once
- each visible pixel is shaded only once
- easy integration of post-process effects (tone mapping, bloom, etc.)
- works well with different shadow techniques
- many small lights $\sim$ one big light
- new types of lighting are easily added

As with any method there are also disadvantages and of these the principal ones are:

- alpha blending is difficult
- memory and bandwidth intensive
- no hardware multi-sampling
- all lighting calculations must be per-pixel
- very high hardware requirements

2.1 Requirements

In order to properly take advantage of deferred shading in OpenGL, one needs to make use of features made available in OpenGL 2.0 and more recent extensions. These include:

- floating point buffers and textures for storage of position; made available in `ARB_color_buffer_float` [6], `ARB_texture_float` [7], and `ARB_half_float_pixel` [8]

- framebuffer objects (FBOs) to avoid expensive context switches as with the widespread pbuffers; made available in `EXT_framebuffer_object` [9]

- multiple render targets (MRTs) to be able to write all G-buffer attributes in a single pass; made available in `ARB_draw_buffers` [10], now part of the OpenGL 2.0 core

- non-power-of-two textures (NPOT textures) to allow rendering to buffers of the same dimensions as the framebuffer; made available in `ARB_texture_non_power_of_two` [11], now part of the OpenGL 2.0 core

The use of multiple render targets limits the application to GeForce 6 Series type graphics cards or better.

2.2 G-buffer

An important consideration when implementing deferred shading is the layout of the G-buffer. Using too high precision for storing elements leads to a performance hit in terms of memory bandwidth, while using too low precision results in deterioration of image quality. There is also one aspect one must take into account when using MRTs (at least on NVIDIA hardware). Each render target may have a different number of channels, but all render targets *must* coincide in the number of occupied bits. As an example R8G8B8A8 and G16R16F differ in number of channels, but have the same number of bits. Another limitation related to the use of MRTs is the maximum number of active render targets, which at the time of writing is limited to four.

Some common attributes stored in the G-buffer are: position/depth, normal, tangent, diffuse color, albedo[1], specular and emissive power, and a material identifier. According to the *NVIDIA GPU Programming Guide* [1], GeForce 6 Series graphics cards exhibit a performance cliff when going from three to four render targets, and in most circumstances it would be wise to follow this guideline. One could trade storage for computation and use hemisphere remapping for both normals and tangents as they are both

RT0	position.x	position.y	position.z	material id
RT1	normal.x	normal.y	normal.z	FREE
RT2	diffuse.r	diffuse.g	diffuse.b	nmap_normal.x
RT3	tangent.x	tangent.y	tangent.w	nmap_normal.y

Table 2.1: G-buffer layout used in the implementation

known to have unit length. By storing only the x- and y-coordinates, the z-coordinate can be calculated as $z = \sqrt{(1 - x^2 - y^2)}$. Another space optimization is to store only the depth of the eye space position and compute the x- and y-coordinates by using the screen space position and "unproject". This is however a bit more expensive than the simple hemisphere remapping. In any case, for the given application a reduction of the size of the G-buffer had no impact on the observed framerate.

When deciding on whether to pack the G-buffer tightly or not, one should first locate the bottleneck for this rendering pass. Late on in this project when code for different modules were integrated, it was observed that the G-buffer pass of the application was bandwidth limited and bound by the total size of textures shuffled from CPU to GPU. The very large quantity of textures needed for the visualization of the buildings can not all fit in the on-board VRAM[2] and this appears to be the reason a reduction in number of render targets does not have any impact on the observed framerate. The layout currently used in the implementation can be seen in table 2.1. It deserves to be mentioned that the limitations imposed by the maximum number of attributes in the G-buffer has led to doubt of the usability of deferred shading on more than one occasion, but for this implementation the maximum size does suffice.

2.3 Optimizations

The basic algorithm for deferred shading is quite simple:

[1] A measure of reflectivity of a surface or body, usually expressed as a fraction in the interval [0,1]

[2] Video Random Access Memory

```
1) For each object:
     Render lighting properties to G-buffer
2) For each light:
     framebuffer += brdf(G-buffer, light)
3) For entire framebuffer:
     perform image space post-processes
```

In the above algorithm, BRDF stands for Bidirectional Reflection Distribution Function, which specifies the ratio of light reflected from a surface and incident luminosity. One such function is the diffuse or Lambertian BRDF, for which light is reflected evenly in all directions.

There are however a few things to bear in mind, apart from API specific details and limitations in current driver implementations. For each light, one is only interested in performing lighting calculations for affected pixels. By representing each light with a bounding geometric shape, unnecessary calculations can be avoided. A spot light can be represented by a cone, a point light by a sphere, and a directional light source by a screen-aligned rectangle. In the second pass, where lighting is calculated, the bounding volume for each light is passed to the shader and this way only the screen space projection of each light volume is shaded. The contribution to the final image from each light source is accumulated in the framebuffer or an intermediate buffer using additive blending.

It is important that each pixel is not shaded more than once for the same light source. To ensure this, each bounding volume must be convex and face culling enabled. Back-face culling skips the rasterization process for those polygons facing away from the viewer and guarantees that there are no overlapping of polygons. When the camera is located inside one of these volumes, front-facing polygons must be culled instead.

Other optimization opportunities include *occlusion query* and *early-z culling*. Many times, a light volume is partially or completely occluded by scene geometry. Occlusion queries make it possible to see how many pixels of rendered geometry end up visible on screen. By issuing a query for each light with color and depth writes turned off, one can then decide whether or not to perform lighting calculations for this light. Early-z culling is a feature which allows fragments to be discarded before they are passed to the pixel shader, unlike the standard depth test which is performed after shader

Method	Time (100,000 iterations)
`glBindFramebufferEXT()`	2.54 s
`glDrawBuffer()`	1.04 s
`glViewport()`	0.03 s

Table 2.2: Comparison of test execution times

execution, and can be used as a computation mask [12]. Unfortunately, early-z culling is not supported for framebuffer objects by current drivers. This type of depth test could be used to avoid shading for parts of a light volume suspended in midair without touching any of the scene geometry.

Performance can suffer when switching render targets frequently, as is often necessary when applying various post-process effects. When using FBOs to do so-called *ping-ponging*[3] [13] one has a few alternatives of how to switch render target. See table 2.2 for a comparison of execution times for different methods. When using the slower alternative, the reported performance gain is in the 5-10% range when compared to standard pbuffers, which imply a context switch. `glViewport()`, which only modifies a matrix, has here been included although it is not directly related to FBOs, but by reading and writing to the same texture and only change viewport and texture coordinates one could in theory speed up ping-ponging even more. Some people testify to have used this successfully, but it is officially unsupported and only works on some hardware and driver configurations, thus making it a poor solution. `glDrawBuffer()` is approximately 2.4 times faster than `glBindFramebufferEXT()` and was used wherever permitted.

Shader Model 3.0 supports *dynamic branching*[4] for the implementation of deferred shading and this has been used for the lighting pass to do a so-called early-out for those pixels which represent the sky. No lighting calculations are performed for the sky and the pixel shader is allowed to exit after only writing the fragment color. Potentially, early-z culling can be used instead when this is supported for FBOs by graphics drivers. Dynamic branching is also used for the sky in the geometry pass, where only one

[3]Render-to-texture scheme where two textures are alternately used as read source and render target respectively

render target is written to instead of four.

2.4 Anti-Aliasing

According to the sampling theorem (Nyquist's rule), when sampling a signal the sampling frequency must be greater than twice the bandwidth of the input signal in order to be able to reconstruct the original signal perfectly from the sampled version. Aliasing is caused by the under sampling of a signal and is often observed in computer graphics as moiré patterns when textures contain higher frequencies and so-called "jaggies" or *rasterization aliasing artifacts* (due to the mapping of the defined geometry onto a discrete grid of pixels displayable on a computer screen).

Anti-aliasing is the process of reducing these artifacts. For textures this can be done by using mipmaps and activating bilinear, trilinear, or anisotropic filtering and for object outlines this can normally be accomplished either by multi-sampling or filtering. Multi sample rasterization samples geometric primitives on a sub-pixel level whereas the filtering approach detects object edges in the final image and smooths these.

As mentioned in the beginning of the chapter, one of the disadvantages with deferred shading is the lack of hardware multi-sampling. The OpenGL API does not allow anti-aliasing of MRTs, nor would it be possible to apply multi-sample anti-aliasing (MSAA) on the G-buffer [5]. Because of this, it is up to the programmer to perform adequate anti-aliasing calculations. The rasterization aliasing artifacts can be reduced by filtering only object edges in an image. The next section goes into detail about how these edges can be detected.

2.4.1 Edge Detection

Color Gradient

An naive solution to the problem of correctly detecting object edges would be to apply a standard edge detection filter, such as the Sobel operator [14]

[4]Dynamic branching in a pixel shader allows for run-time evaluation of control flow on a per-fragment basis, making it possible to skip execution of redundant code

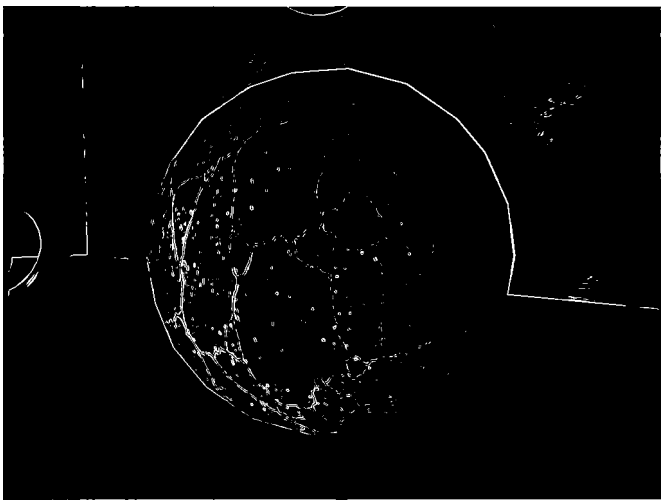

Figure 2.1: Edges detected with the sobel operator

shown below, to the color values of the final image. This was tried because of its simplicity, hoping that maybe one could find a good enough threshold for the magnitude of the gradient to blur object edges and leave texture detail untouched.

The Sobel_x and Sobel_y operators, shown below, were used for edge detection.

$$\mathbf{D}_x = \frac{1}{8}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\mathbf{D}_y = \frac{1}{8}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The magnitude of the gradient is calculated as

$$\|\nabla_{sobel}\| = \sqrt{\mathbf{D}_x(x,y)^2 + \mathbf{D}_y(x,y)^2}$$

and then a rather high threshold can be applied to discard edges below a certain value. The results of this method can be seen in figure 2.1.

Depth and Normal Discontinuities

Using the above method would however miss edges where there are small differences in color values, although this should not have much impact on the end result for the same reason. A second and more severe problem is the unwanted effect of detecting edges within rendered geometric primitives due to details in the applied textures. This is not at all desirable as blurring these false edges would lead to loss of texture detail. The applied textures will most likely already have been bilinearly, trilinearly, or anisotropically filtered, and further blurring only leads to loss of image quality.

A better solution for the purpose of detecting object edges is to use depth and normal values already available in the G-buffer [5]. In GPU Gems 2, Oles Shishkovtsov presents a method for doing this [3]. Nine samples (the pixel itself and its eight nearest neighbors) are taken of the depth and these values are then used to find how much the depth at the current pixel differs from the straight line passing through points of opposite corners. This alone will have problems with scenarios such as a wall perpendicular to a floor, where depth forms a perfect line or is equal at all samples. To remedy this, the normals at each sample were used and the dot products of these calculated to detect an edge. These values are then multiplied and the resulting value is used to offset four texture lookups in the image being anti-aliased. Edges detected by this alternative approach are displayed in figure 2.2 and the final result of the two methods are shown in figure 2.3. The reader is reminded that it is the loss of texture detail which is shown in figure 2.3. The texture has already been anisotropically filtered and any further filtering is unwanted.

A solution with Sobel edge detection on the z-buffer was not tested, and no mention can therefore be made as to what the result would be, nor performance wise nor quality wise. Any further work on anti-aliasing for deferred shading could perhaps benefit from a closer look at this.

Figure 2.4 intends to give an overview of how the deferred shading framework functions and integrates with used post-processing effects, presented later in this text. Shadow maps are generated when shadow casting geometry or lights change position or orientation. The rendered image itself is created in various steps. First, the G-buffer is created and needed attributes are written into the buffer. In this pass, reflection, normal, and

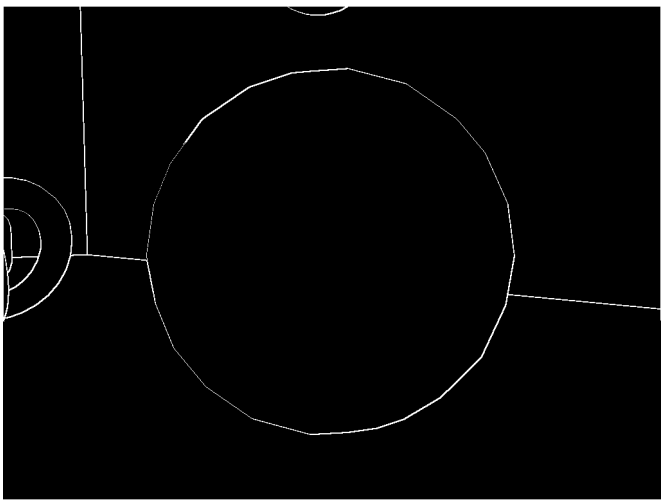

Figure 2.2: Edges found by detecting discontinuities in depth and normals

height maps are used if available. Second, lighting calculations are performed, taking into account information from shadow maps. The resulting image contains HDR values and needs to be mapped to the interval [0,1]. Consequently, the average luminance of the image is calculated and tone mapping is performed to generate a displayable image. As a last step, object edge aliasing is reduced and the result is written to the framebuffer. Those concepts which are unknown to the reader at this time are mentioned later on in this report.

Figure 2.3: Cut out image section (top) and anti-aliased image using sobel edge detection (middle) and depth/normal discontinuity detection (bottom)

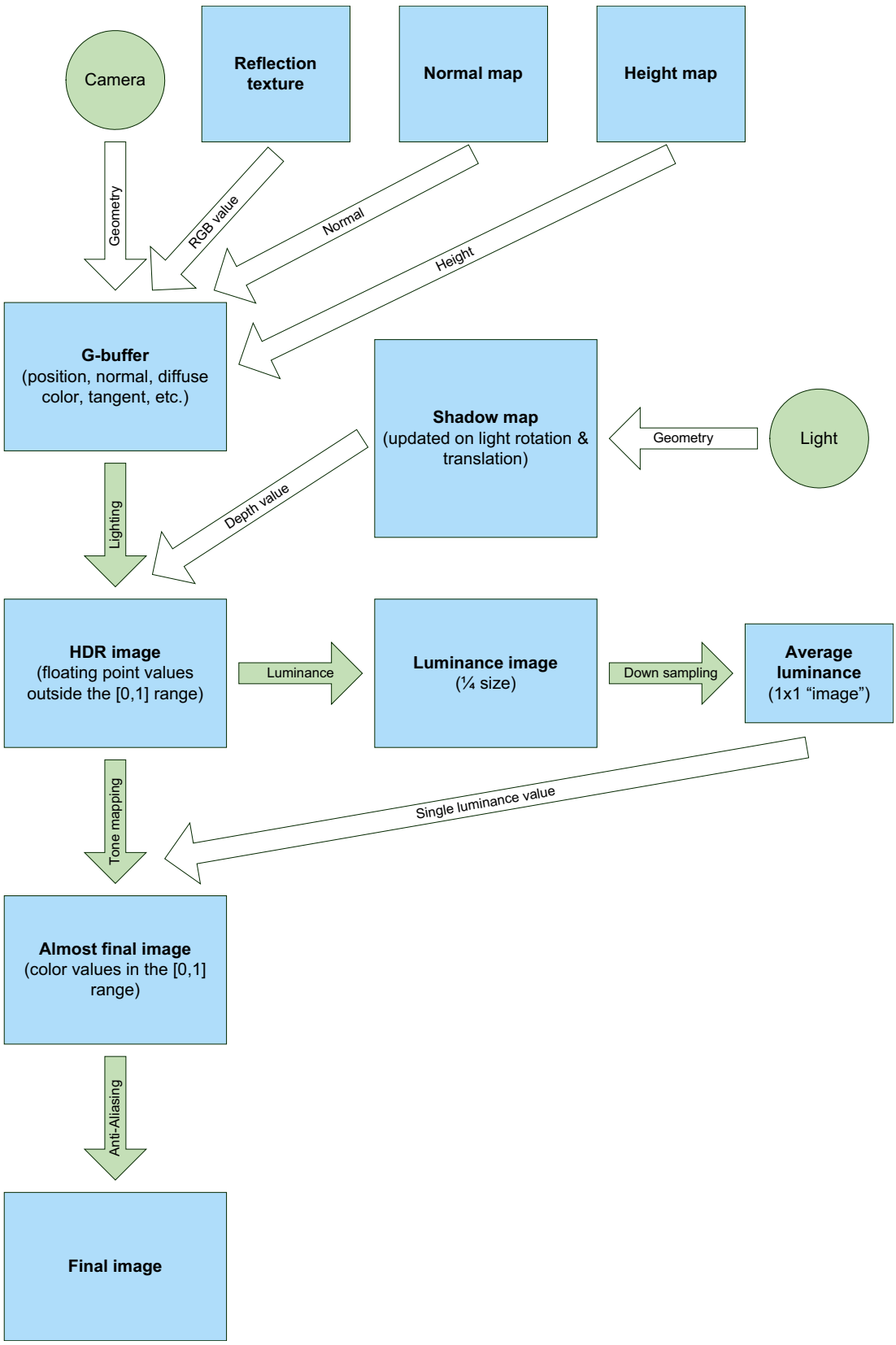

Figure 2.4: Flow chart showing how the framework functions

Chapter 3

Reflection

This chapter explains difficulties and possible solutions for generation of window reflections.

Reflections are eye candy used in most any computer game today. For reflections on small objects, methods like sphere environment mapping and cube environment mapping are sometimes used. For correct reflections on larger surfaces other approaches need to be taken, since standard environment mapping would create a reflection with visible artifacts. Sections 3.2.2 and 3.2.4 explain why. For water and planar surfaces, people have often used projective texture mapping as a tool for creating realistic reflections. Doing so would however require an extra rendering pass for every reflecting plane.

The reflections considered in the context of this architectural walk-through application are limited to the planar reflections observed in the windows of buildings. Also, inter-reflections have been discarded as online ray-tracing in the given context would not achieve interactive frame rates.

Any discussion of refraction in the following section is only brought up in the context of how it affects what an observer sees reflected in a window pane. Refraction, as it relates to what an observer would see *through* a window pane, has not been covered in this thesis, nor is it part of the stated objectives.

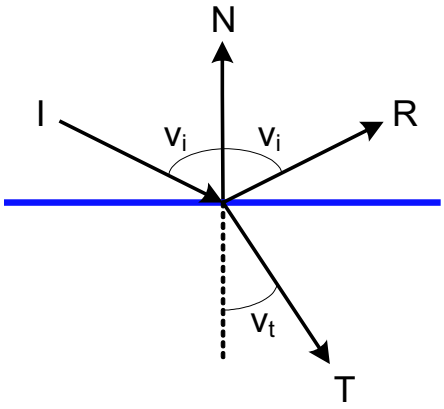

Figure 3.1: Angles of incidence and refraction

3.1 Optics in a Window Pane

When light crosses the boundary between two media with different indices of refraction, the relation between the angles of incidence and refraction are given by Snell's law, specified in equation 3.1 and illustrated in figure 3.1. Here, θ_i is the angle of incidence and θ_t the angle of refraction. n_i and n_t are the indices of refraction for the two media. The average index of refraction for uncoated window glass is $n_{glass} \approx 1.52$ and the index of refraction for air is $n_{air} \approx 1.00$ [15].

$$n_i \sin(\theta_i) = n_t \sin(\theta_t) \tag{3.1}$$

3.1.1 Fresnel Equations

When light travels between media of different indices of refraction, not all light is transmitted. The fraction of incident light which is reflected and transmitted is given by the reflection coefficient and transmission coefficient respectively. The Fresnel equations can be used to calculate these coefficients in a given situation, if the angle of incidence and indices of refraction for the two materials of a surface boundary are known. Light with its electric field parallell to the plane of incidence is called p-polarized and that with its electric field perpendicular to this plane is called s-polarized.

These two components reflect/refract differently when striking a surface boundary. Equations 3.2 and 3.3 shows how the reflection coefficients for s- and p-polarized light are calculated respectively.

$$R_s(\theta_i) = \Big(\frac{\sin(\theta_i - \theta_t)}{\sin(\theta_i + \theta_t)}\Big)^2 \tag{3.2}$$

$$R_p(\theta_i) = \Big(\frac{\tan(\theta_i - \theta_t)}{\tan(\theta_i + \theta_t)}\Big)^2 \tag{3.3}$$

If we assume the light striking a surface between two materials with different indices of refraction is non-polarized (containing an equal mix of s- and p-polarizations) and of the same wavelength, the reflection coefficient is given by equation 3.4, expanding to equation 3.5. These simplifications are acceptable as they only introduce a small error, though neither assumption is true [15].

$$R(\theta) = \frac{1}{2}\Big(R_s(\theta) + R_p(\theta)\Big) \tag{3.4}$$

$$R(\theta_i) = \frac{1}{2}\left(\Big(\frac{\sin(\theta_i - \theta_t)}{\sin(\theta_i + \theta_t)}\Big)^2 + \Big(\frac{\tan(\theta_i - \theta_t)}{\tan(\theta_i + \theta_t)}\Big)^2\right) \tag{3.5}$$

The reflection coefficients for different angles are shown for air to glass in figure 3.2 and for glass to air in figure 3.3. At an angle of incidence greater or equal to approximately 41.8 ° when light passes from glass to air, there is a total internal reflection and no light passes into air. This does however not affect the reflections in this case.

The reflection coefficient can be approximated well using equation 3.6 [15], but due to multiple reflections in a window pane, it's faster to pre-compute the combined reflection coefficient and use $N \cdot E$ and $N \cdot L$ to index this 1D texture for the environment and specular reflections respectively.

$$R(\theta) = R(0) + \Big(1 - R(0)\Big)\Big(1 - \cos(\theta)\Big)^5 \tag{3.6}$$

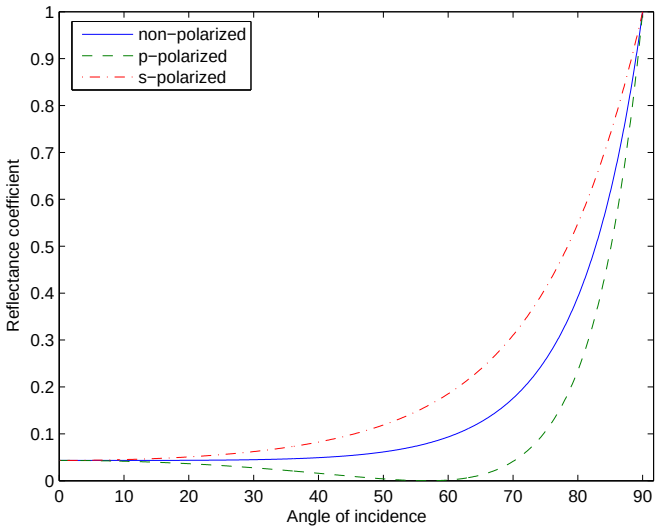

Figure 3.2: Reflection coefficients for different angles when light passes from air to glass

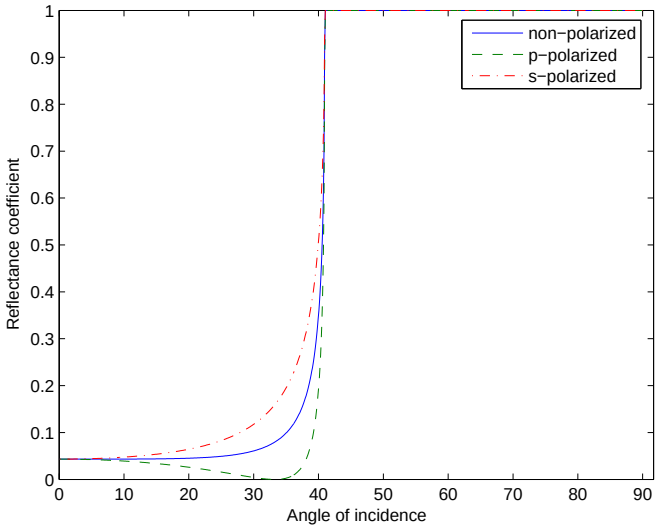

Figure 3.3: Reflection coefficients for different angles when light passes from glass to air

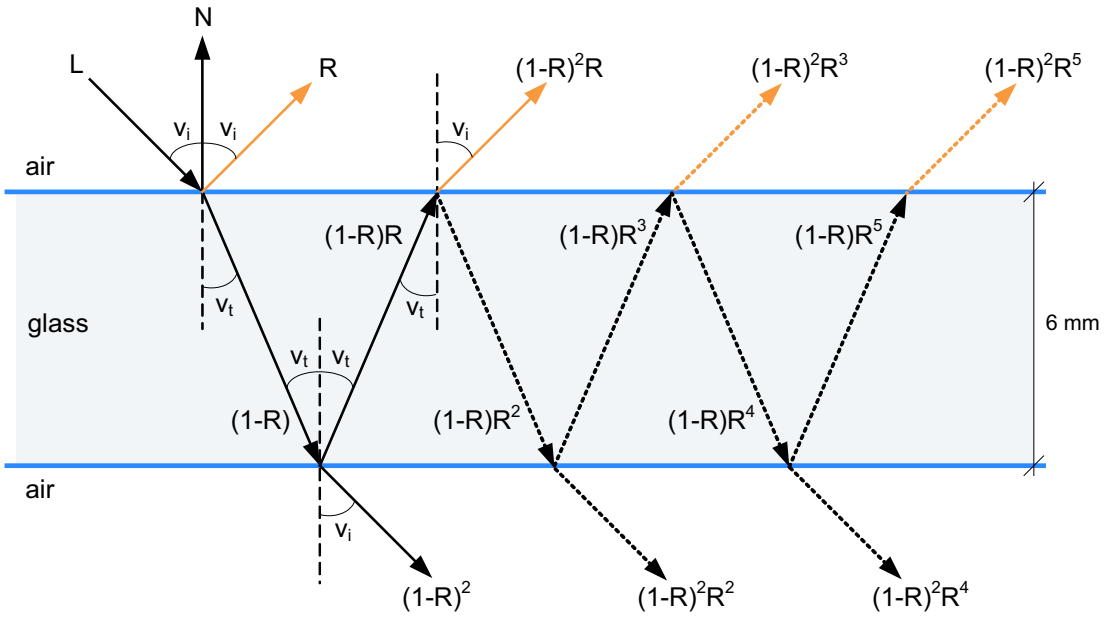

Figure 3.4: Reflection and refraction in a window pane

3.1.2 Multiple Reflections and Refractions

Snell's law and the Fresnel equations determine that a single window pane returns not one, but multiple reflections, as seen in figure 3.4, blurring the reflection slightly in the plane of incidence. There is a *reflection loss* with each internal reflection. The reflectance coefficient for the first three reflections are shown in figure 3.5.

Total Reflection Coefficient

The reflection and transmission coefficients sum to one and the reflection coefficient for light passing from glass to air equals that of one minus the reflection coefficient for light passing from air to glass. Looking at figure 3.4 one can see that the reflection coefficients form a geometric series (see equation 3.7). Since $|R| < 1$, the series converges and the total reflection

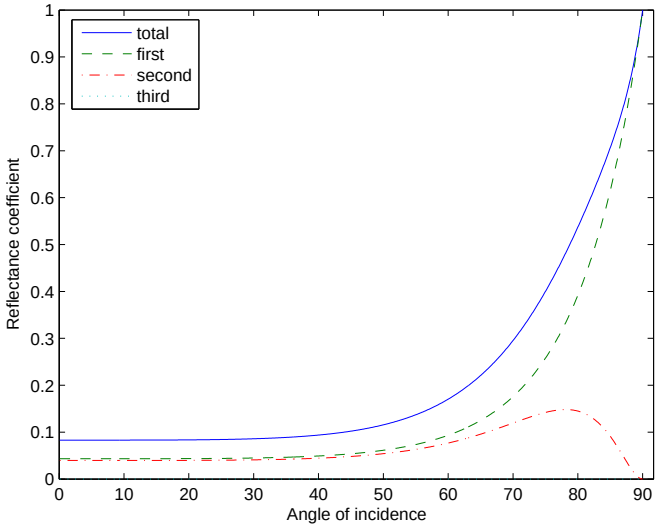

Figure 3.5: Contributions of first, second, and third reflection

coefficient $\frac{2R}{1+R}$ is given by equation 3.8.

$$R_{total} = R + (1-R)^2 R + (1-R)^2 R^3 + (1-R)^2 R^5 + \ldots \tag{3.7}$$

$$\begin{aligned} R_{total} &= R + (1-R)^2 R + (1-R)^2 R^3 + (1-R)^2 R^5 + \ldots \\ &= R + (1-R)^2 R \sum_{k=0}^{\infty} (R^2)^k \\ &= R + \frac{(1-R)^2 R}{1-R^2} \\ &= R + \frac{(1-R)R}{1+R} \\ &= \frac{(1+R)R + (1-R)R}{1+R} \\ &= \frac{2R}{1+R} \end{aligned} \tag{3.8}$$

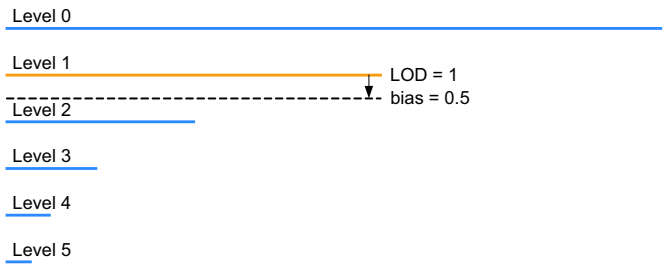

Figure 3.6: Mipmapping by adding a bias to the pre-computed LOD

3.1.3 Blur

The displacement of the reflections in the window pane makes the reflections slightly blurred in the plane of incidence. Many modern windows have double window panes with a hermetically sealed space filled with gas in between. This results in additional reflections, further blurring the perceived reflection. One could perform various texture lookups for the reflection, each slightly displaced in the direction of the internal reflections in texture space and weighted with its corresponding reflection coefficient. Compared to the already ”poor” quality of window reflections, it would be difficult to justify the extra expense of doing this.

Instead the OpenGL feature *texture lod bias* (part of OpenGL 2.0) is used to produce a slightly blurred reflection. This OpenGL feature makes use of mipmapping and the given bias is added to the texture LOD (level-of-detail) computed by OpenGL to select mipmaps of lower or higher levels. By adding a positive bias, a lower level low-pass filtered mipmap is selected for the texture lookup. If trilinear filtering is enabled, the texture lookup is performed with a linear interpolation not only between neighboring pixels of the closest mipmap level, but also between neighboring mipmap levels. See figure 3.6. There is an OpenGL command for adding a constant bias to all texture lookups performed on a certain texture, but as the bias should be varied as a result of the angle of incidence, the angle of incidence is computed on a per-pixel basis and the texture lookups performed in the fragment shader thus use a bias corresponding to the angle of incidence for the current fragment.

Another factor affecting the reflection is the negative or positive pressure

of the gas between the window panes (in the case of double or triple panes) due to the current temperature and the expansion/contraction of the gas. This makes the surface of the window panes slightly curved and distorts the reflection. There are also other factors such as dirt on the glass surface and not quite planar window surfaces (in older window panes), which lead to less perfect reflections. Also, there is the effect of changed indices of refractions as part of the light striking a window being absorbed by the window panes and the isolating gas in the space between them. All these factors have been ignored, since the effects of these on the resulting reflection are difficult to estimate and differ widely between windows.

3.2 Computation of Reflections

There are two potential paths to take when implementing window reflections. Either one can strive for correct and expensive true planar reflections, or opt for cheap pre-computed environment mapped reflections. The latter reflections are mathematically incorrect, but can be made to give a rather good approximation.

3.2.1 True Planar Reflections

At first, the feasibility of implementing mathematically correct planar reflections was considered. David Blythe mentions three possible ways of how to this using OpenGL [16]. These methods either use the stencil buffer (1), OpenGL's clip planes (2), or texture mapping (3) when rendering a reflecting plane. Each of these techniques require an extra pass. Since the windows of a single facade of a building are co-planar, these could be grouped together to minimize the number of additional rendering passes needed. As a matter of fact, the facades of buildings along the same street usually have near co-planar window planes, and where possible these could be treated as a single reflective plane.

"Level-of-Detail"

There is a tradeoff between detail and performance, and by regulating the level-of-detail one can achieve interactive speeds even for very complex

models. The basic principle of LOD is to use less detail for the representation of areas of a 3D scene which are either small, distant, or otherwise less important [17]. In the case of window reflections, the observer pays closer attention to reflections within close proximity while mostly ignoring far away reflections occupying a very small amount of screen space in the final rendered image.

The most important question for management of LOD is how to manage transitions between the different LODs. Common LOD selection factors include distance from viewpoint to object in world space, projected screen coverage of an object, priority, hysteresis, etc. For the purpose of window reflections, the screen coverage approach could be used to create a list of reflection planes occupying more screen space than a threshold specified by the application. These reflection planes would then be updated using a round robin scheduling scheme, one reflection plane being updated every frame. Those reflection planes which do not qualify for a full calculation of the planar reflection, are left to use their respective "inactive" reflection textures or alternatively a pre-computed environment cube map as described in more detail in section 3.2.2.

To avoid popping artifacts when switching from one LOD to another, once would use alpha blending to achieve smooth transitions. Each LOD is associated with an alpha value, with 1.0 meaning opaque and 0.0 completely transparent. Within the transition region, both LODs are rendered simultaneously and blended together, see figure 3.7.

3.2.2 Cube Environment Mapping

Computing true planar reflections for the windows would be mathematically correct. However, as mentioned earlier, this would be very time consuming for an application already struggling to achieve interactive frame rates. The number of visible reflecting planes can range anywhere from one to three or more. Instead it was decided to use pre-computed environment maps for every reflective entity (in this case mesh of window panes), and use an already existing texture manager to minimize bandwidth usage for the texture switches. One immediate disadvantage is that no dynamic geometry (when incorporated) is reflected, but this was a conscious choice, trading accuracy for speed.

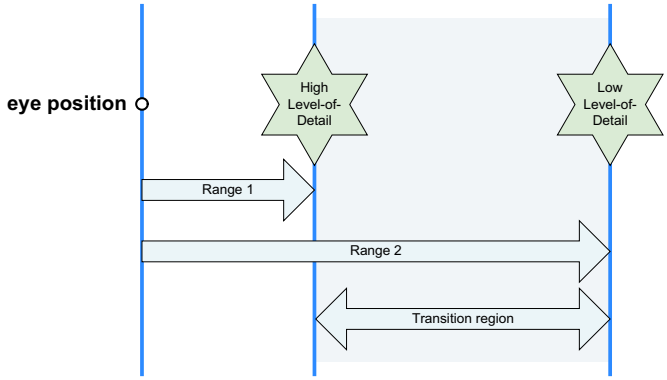

Figure 3.7: Alpha blending to avoid popping artifacts

Environment maps only give correct reflections for very small reflective objects or with the reflected environment infinitely far away. This criteria does not hold for window reflections. The reflective entity has a significant extension within a plane and the reflected geometry is quite close, often no further away than across a street. The reflection vector used to lookup a texel in the reflection map is the same regardless of whether the reflection originates at the center point for which the environment map was calculated or if the reflection occurs at an extreme of the window cluster. These two cases will generate the same reflection, having the reflected geometry ”follow” the viewer as he/she walks along a street. This can be alleviated by adjusting the reflection vector. See section 3.2.4 for a more in depth explication of the problem and its solution.

The type of environment mapping first looked into was *cube environment mapping*, due the simplicity of its generation and the hardware support for texture lookups. One has only to set the field-of-view to 90 °, change the viewport to the dimension the cube sides and then render the environment once for each side. Also, the texture lookup constitutes only a single instruction. The cube map is generated in world space, making it *view-independent*. All one needs to do before performing the texture lookup is to rotate the reflection vector, defined in eye space, into world space to have both vector and texture defined in the same space. These cube maps could then be generated offline or at each startup.

A big disadvantage with cube maps (for the intended purpose) is that

half the cube map contains irrelevant information and thus wastes valuable memory. As said before, the G-buffer creation phase is limited by the size and quantity of the textures. Also, one would also have to adapt the already implemented texture manager for the use of cube maps. These problems can be at least partially solved by adopting a different parameterization for the environment map, as talked about in the following section.

3.2.3 Paraboloid Environment Mapping

First off, let's look at the subject of wasted space when using cube environment maps. The excessive amount of memory used limits the cube map resolution, giving a blocky reflection. Two other commonly used parameterizations are sphere maps and dual paraboloid maps. One known problem with sphere environment mapping is so-called "speckle" artifacts along the silhouette edges of the reflecting object. Sphere mapping also assumes that the center of the map faces towards the viewer and is thus *view-dependent*, meaning reflection maps would have to be regenerated every time the camera moves. On the other hand, *paraboloid environment mapping* shows promise. The paraboloid is defined by equation 3.9 and is illustrated in figure 3.8.

$$f(x) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2),\ x^2 + y^2 \leq 1 \tag{3.9}$$

At a closer look, paraboloid mapping appears to fit the bill almost perfectly. We can use a single paraboloid map to represent the half-space reflected in a window pane. There is still some space wasted of the environment map, as the reflected geometry will be mapped to what's commonly referred to as the *sweet circle*, see figure 3.9. If the 2D texture has a side of n texels, the effective area of the paraboloid map is $\pi\left(\frac{n}{2}\right)^2/n^2 = \pi/4 \approx 79\%$, which is a great step up from 50% for the cube map. One minor disadvantage is the extra math involved to perform the necessary image warping.

This parameterization also makes it possible to easily plug the textures into the existing texture manager. To further decrease the memory imprint of these textures, they are generated offline and converted to DDS images using DXT1 compression. Allegedly, DXT1 has a compression ratio of 8:1,

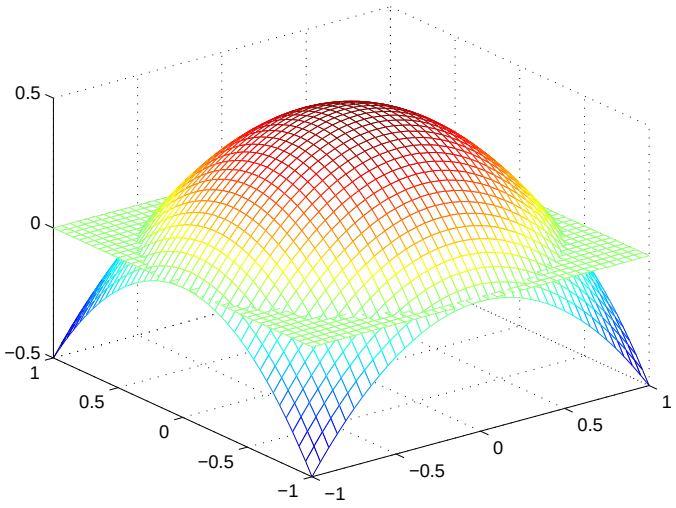

Figure 3.8: Paraboloid mapping for positive hemisphere

Figure 3.9: Sweet circle of a paraboloid map

but the observed compression was only 4:1. No further investigation was made into this.

All seems well, but there are two major disadvantages to paraboloid mapping. The paraboloid mapping of geometric primitives is non-linear and can only be done on the vertex processor, which allows for warping of vertex positions. Between the vertex processor and the fragment processor is located the rasterization unit, which performs the task of converting geometric primitives to fragments. This process is done in hardware and uses linear interpolation. As a consequence, low tessellation of a model leads to very visible artifacts. Straight lines which are meant to be mapped to curves are instead mapped to straight lines. See figure 3.10.

The second disadvantage is that geometric primitives spanning the two hemispheres will be completely discarded by the rasterization unit and no fragments are generated. This can clearly be seen in the missing pieces of the asphalt in figure 3.10.

Neither of these artifacts are acceptable. But what about using a high resolution cube environment map and then warp half of it into a paraboloid map? That works very well indeed. All straight lines are now perfectly mapped onto their corresponding curves. See figure 3.11. The process can then be summarized as follows:

1. Create a high resolution cube environment map for each window mesh

2. Create a paraboloid environment map, sampling the cube map

3. Convert the textures to compressed DDS images

In step two above, one first has to map the front side (facing in the direction of $\overrightarrow{d0} = (0, 0, -1)^T$) 2D texture coordinates (s, t) to the corresponding world space reflection vector $\overrightarrow{R}$, see equation 3.10 [18]. Secondly, the reflection vector must be rotated around the y-axis to make the normal

Figure 3.10: Linear rasterization artifacts for paraboloid mapping

Figure 3.11: Paraboloid generated by sampling cube map

vector $\overrightarrow{N}$ coincide with $\overrightarrow{d0}$. See equation 3.11.

$$\begin{pmatrix} R_x \\ R_y \\ R_z \end{pmatrix} = \begin{pmatrix} \frac{2s}{s^2+t^2+1} \\ \frac{2t}{s^2+t^2+1} \\ \frac{-1+s^2+t^2}{s^2+t^2+1} \end{pmatrix} \tag{3.10}$$

$$\theta = \begin{cases} arccos(\overrightarrow{N} \cdot \overrightarrow{d0}),\ (\overrightarrow{N} \times \overrightarrow{d0})_y \leq 0 \\ -arccos(\overrightarrow{N} \cdot \overrightarrow{d0}),\ (\overrightarrow{N} \times \overrightarrow{d0})_y > 0 \end{cases} \tag{3.11}$$

Before doing lookups in the paraboloid environment maps, one performs the inverse of the above mapping. See equation 3.12 for the mapping of the reflection vector $\overrightarrow{R}$ to the front side 2D texture coordinates (s, t).

$$\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} \frac{R_x}{1-R_z} \\ \frac{R_y}{1-R_z} \end{pmatrix} \tag{3.12}$$

3.2.4 Accurate Reflections

Environment mapping uses textures to encode the environment surrounding a specific point in space. If the reflecting object can not be approximated by a point, such as the case with window panes, the reflections will display artifacts. Straight forward environment mapping does not take into consideration that the reflection vector used to index the environment map can originate from points other than the center. The reflection vector varies little over a flat surface, which leads to only a small region of the environment map being indexed and thus magnified. In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks* [19], Chris Brennan presents a technique to compensate for the inaccuracy introduced when using environment mapping for reflections of objects not infinitely far away.

He gives the environment map a finite radius closely approximating the actual distance of reflected objects. The reflection vector is then adjusted by the vector from the center of the environment map to the actual origin of the reflection vector, scaled by the reciprocal of the radius. Quoting the

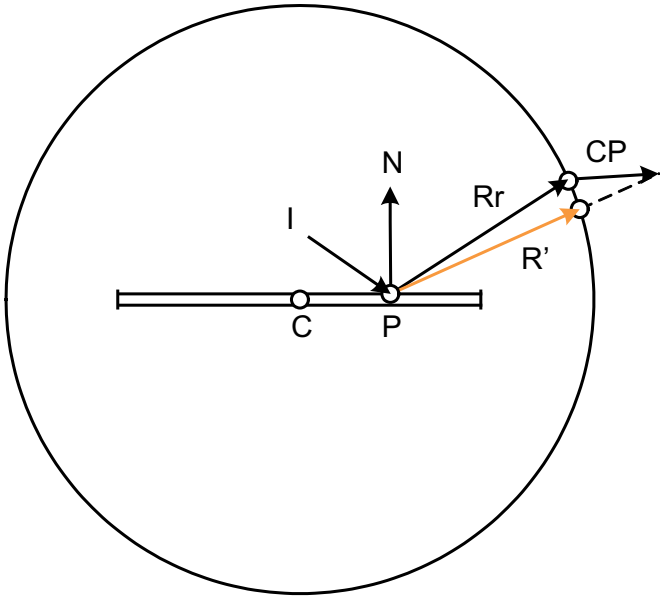

Figure 3.12: Correction of reflection vector

equations given by Brennan, the new reflection vector is $\overrightarrow{R''} = \overrightarrow{R} + \frac{\overrightarrow{CP}}{r}$, which is derived from $\overrightarrow{R'} = \overrightarrow{R}r + \overrightarrow{CP}$. See figure 3.12 for an illustration.

Needless to say, this works well if the distance of reflected objects is more or less the same, but quite badly if the reflected objects are the buildings across a street. See figure 3.13. In this case, the distance varies greatly, especially in the direction of the street. The problem with using a constant radius is especially accentuated when the camera is in motion.

Using a Distance Cube Map

At first, the possibility of using a low resolution floating point cube map to store the distances to all surrounding objects was explored. It was believed that by using the reflection vector to look up an approximate distance (only accurate for reflection vectors originating at the center) and then use this to scale the correction vector $\overrightarrow{CP}$. If necessary, the process could be iterated two or more times to achieve better approximations of the true reflection vector. However, the approximation of the reflection vector does not always

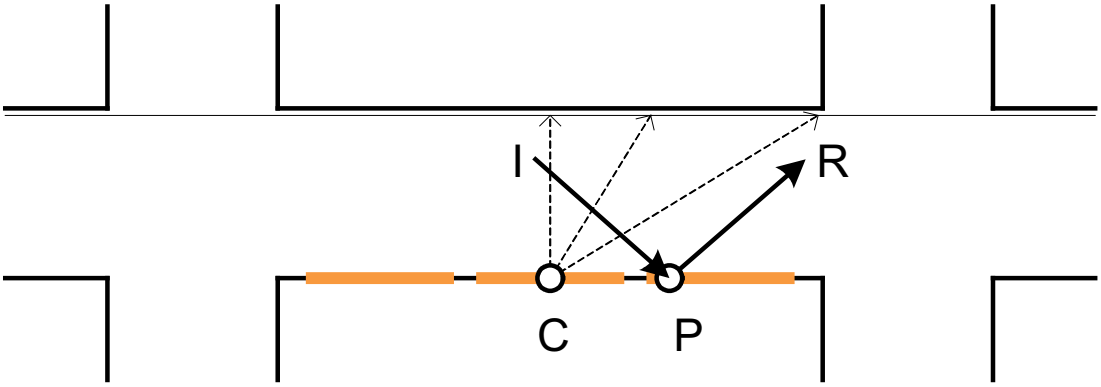

Figure 3.13: Correction of reflection vector in a city environment

converge with its true direction. Also, the reflection became very distorted where there were discontinuities in depth, e.g. along building silhouettes.

Approximating Distance with a Plane

Better results were achieved by approximating the distance to reflected objects with a plane, extending itself in the direction of the reflected building facades across the street. See figure 3.13. For the given city model, the width of a street is more or less eight meters. The sought distance forms the hypotenuse of a right triangle, where the adjacent has a length equal to eight and the hypotenuse extends in the direction of the true reflection vector. Basic trigonometry tells us the length of the hypotenuse is equal to eight divided by the dot product of the unit length normal $\overrightarrow{N}$ and uncorrected reflection vector $\overrightarrow{R}$.

This model gave better results, but shows quite noticeable artifacts for reflections at grazing angles. Adjusting the scaling function $\cos(x)$ by preserving its value for small angles and reducing its value for larger angles, compensates for this. Simply using the square of the dot product removed artifacts at grazing angles, but introduced new ones when the incident vector is near normal. The sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-a(x-k)}}$$

gave satisfying results for $a = -4.7$ and $k = 0.9$. With these parameters, the sigmoid function approximates $\cos(x)$ for smaller angles and looks more like $\cos^2(x)$ for angles close to $90\,^\circ$. See figure 3.15. For a comparison between the results using the dot product and a sigmoid function, see figure 3.14 and pay special attention to the reflected horizon.

Another artifact appeared when the center of a window cluster (and its environment map) was situated much above the height of the camera, making the reflected ground appear tilted. Fortunately, also this was alleviated when using the above specified sigmoid function. The reflection correction still is not perfect and one can perhaps find another model to better approximate true planar reflections. Yet another solution would be to generate all environment maps at a height equal to that of a walking person and then lock the camera to this same height. The impact of window reflections in an architectural walkthrough would be considered to be at its maximum when the trajectory of the camera is close to that of a walking person. When "flying" over buildings, the reflections usually are of less importance to the viewer, unless of course there are facades completely covered by glass windows.

3.3 High Dynamic Range

Dynamic range is defined as the ratio of the largest value of a signal to the lowest measurable value. In real world scenarios, the *dynamic range* can be as high as 100,000:1. In *high dynamic range* (HDR) rendering [20], pixel intensity is not clamped to the [0,1] range. This means that bright areas can be very bright and dark areas very dark, while still allowing for details to be seen in both. In order to more accurately approximate the effect of very bright light, such as that of the sun reflecting off a window pane, it was decided to use HDR rendering, although its use of floating point buffers and textures increases memory and bandwidth requirements. To minimize these requirements, a 16-bit floating point format also known as *half* was chosen. Another very important reason for choosing this format is that graphics cards still don't support blending for single precision 32-bit floats.

HDR requires floating point precision throughout the entire rendering pipeline, including floating point blending and filtering support, limiting its

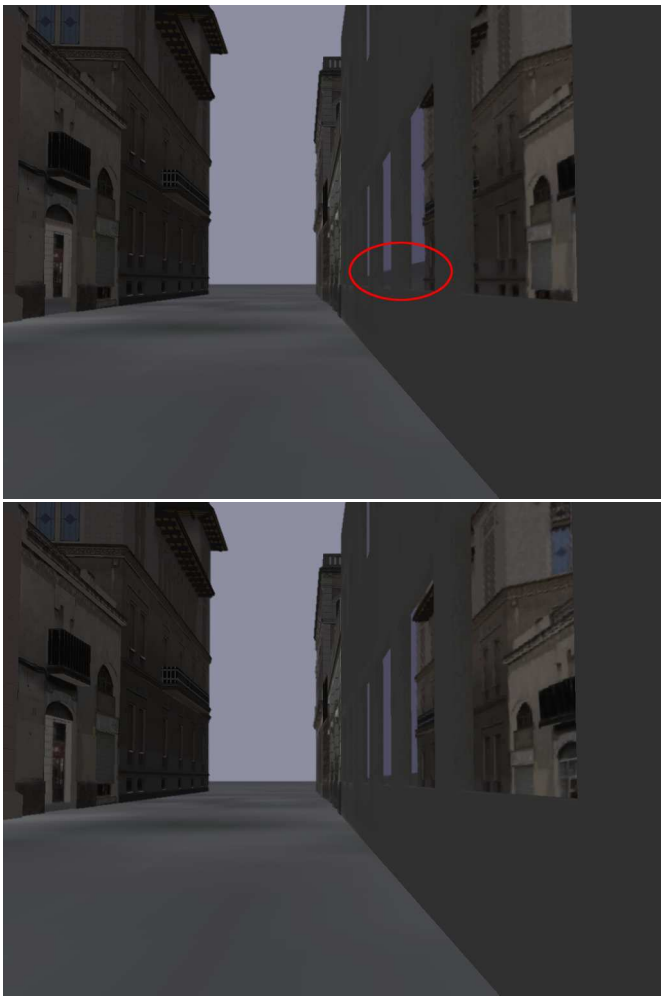

Figure 3.14: Reflection correction with dot product (top) and sigmoid function (bottom)

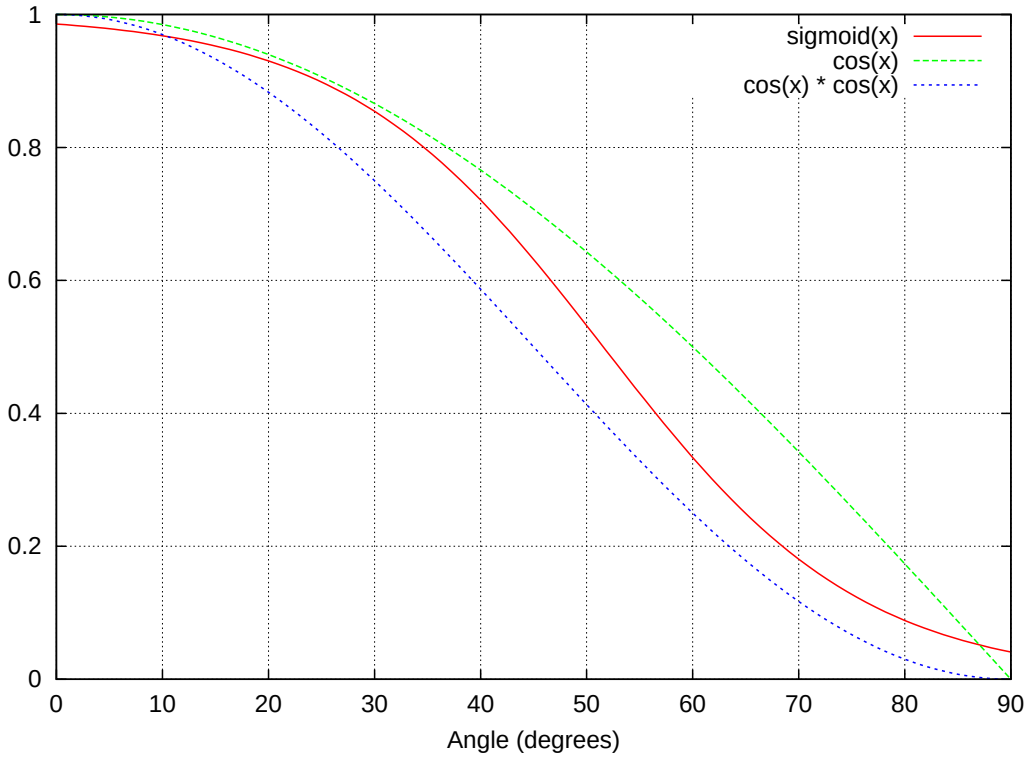

Figure 3.15: Sigmoid function and dot products

use to GeForce 6 Series type graphics cards or better. Since HDR rendering produces floating point pixel values with unlimited range, these values must be mapped to a range displayable by normal low dynamic range displays. This process is called *tone mapping* and the resulting image quality of combining HDR rendering with tone mapping is usually superior to that of using only 8 bits per channel. By using high dynamic range and tone mapping, phenomena such as bloom, glare, and blue shift[1] can be simulated on normal displays.

3.3.1 Tone Mapping

There are a few terms one should be acquainted with before reading on about tone mapping. The full range of luminance values can be divided into a set of *zones*, where each zone represents a certain range. The term *middle gray* is another word for the middle brightness of a scene and the *key* is a subjective measurement of its lighting.

Average Luminance

When performing tone mapping the first step is to compute the *logarithmic average luminance* as in equation 3.14, a value which is then used as the key of the scene. One could usually get away with a normal average, but according to Adam Lake and Cody Northrop [22], luminance values form an exponential curve and the logarithmic average luminance is therefore often used instead.

To calculate the luminance for a set of RGB values, people have traditionally used the coefficients defined by the NTSC standard, calculating the luminance as $Y = 0.299R + 0.587G + 0.114B^2$. The standard was drafted in 1953 and does no longer accurately calculate luminance for modern monitors. A better approximation is found in the HDTV color standard (defined in ITU-R BT.709 - Parameter Values for the HDTV Standards for Production and International Programme Exchange) with weights applied to the

[1] Bloom is the effect of colors with high luminance values bleeding into neighboring parts of the image and glare is caused by diffraction or refraction of incoming light [21]. Blue shift happens in the human eye and is the effect of low light conditions and a biochemical adaptation in the human eye[22].

different color components as in equation 3.13 [23]. These values correspond rather well to the physical properties of modern day CRT displays.

$$Y = 0.2125R + 0.7154G + 0.0721B \tag{3.13}$$

A quick way of calculating the logarithmic average luminance of a high dynamic range image is to:

1. calculate the log() value of the luminance for every pixel
2. scale the rendered scene to $1/16^{\text{th}}$ of the original image size by calculating the average of 2x2 or 4x4 blocks of pixels
3. keep downsampling by averaging pixel blocks until reaching an image size of 1x1
4. calculate the exp() value of the value found in the 1x1 texture

The GeForce 6 Series graphics cards support floating point filtering, so one rather straight forward way of downsampling the luminance image (step 3 above) would be to use mipmapping to generate mipmap levels all the way down to 1x1 and then read the average luminance from the lowest level mipmap. This approach was tested using a framebuffer object with a single luminance texture attached as a render target and automatically generating corresponding mipmaps. Shader Model 3.0 supports *vertex texture fetches* (VTF) and allows texture lookups at specific mipmap levels. However, the read value seemed to be extremely sensible to light changes in the scene due to camera movement. Perhaps this can be attributed to incorrect mipmap generation of the luminance texture, since only fp16 filtering is supported for GeForce 6 Series graphics cards.

An alternative way of downsampling the image is to use a shader to iteratively reduce the texture size, calculating the average for 4x4 blocks of pixels. This approach produced more stable values and is also the method used in the implementation. Using any of the above specified methods, all calculations are kept on the GPU.

[2]RGB and YUV are two different color space formats, with YUV = luminance/brightness and two chrominance (color difference) signals.

Vertex texture fetches have quite limited support for different texture formats, and although this approach would have avoided transfers of data to the CPU, it was decided to use `glReadPixels()` to read the average luminance instead. Among other things, when using textures of the type `GL_TEXTURE_RECTANGLE`, which use unnormalized texture coordinates ([0, width] x [0, height]), shaders using VTFs fall back on software rendering. The `GL_LUMINANCE32F_ARB` texture format was chosen, even though an fp16 format would have cut the texture size in half, because of difficulties with using so-called "half floats" on the CPU.

Another issue encountered when implementing the necessary calculations for the logarithmic average luminance was that of *floating point specials* [24]. Calling log(Y) for a luminance value equal or close to 0.0 will give $-Inf$ as result. This will propagate through the entire process of downsampling until reaching the 1x1 texture. A pixel value of $-Inf$ will produce a black color, while $+Inf$ will appear as white. To resolve such issues for black pixels, one has to add a small value before computing the logarithm of the luminance as in equation 3.14.

In equations 3.14 through 3.17, $L_w(x, y)$ is the world luminance value, $\bar{L}_w$ the average world luminance, $L_s(x, y)$ the scaled luminance value, and $L_d(x, y)$ the tone mapped displayable luminance value. L_{white} is the smallest luminance which is mapped to pure white and if this value equals L_{max}, no burn-out will occur at all.

$$\bar{L}_w = exp\Big(\frac{1}{N}\sum_{x,y}\log(L_w(x, y) + \delta)\Big) \tag{3.14}$$

Scaling and Compression

Once the scene key has been calculated, one can map the high dynamic range values to the interval [0,1]. First the average luminance is mapped to a middle gray value by using the *scaling operator* in equation 3.15. This gives a *target scene key* equal to α. Once this has been done, a *tone mapping operator* is applied in order to compress the values to the displayable [0,1] range. One function often used for this purpose is found in equation 3.16 [25]. This operator scales high luminance values by $1/L$ and low luminance

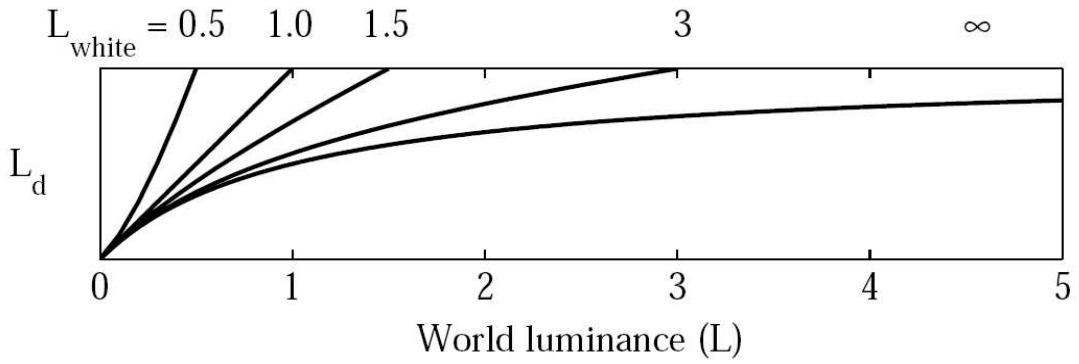

Figure 3.16: Mapping of world luminance to a displayable luminance for different values of L_{white}. Image courtesy of Erik Reinhard et al. [25]

values by 1. See figure 3.16.

$$L_s(x,y) = \frac{\alpha}{\bar{L}_w} L_w(x,y) \tag{3.15}$$

$$L_d(x,y) = \frac{L_s(x,y)}{1 + L_s(x,y)} \tag{3.16}$$

This mapping does not always achieve the effects one strives for and Reinhard presents an alternative operator (see equation 3.17) which gives a *burn-out* effect for pixels with luminance values above a certain threshold.

$$L_d(x,y) = \frac{L_s(x,y)\left(1 + \frac{L_s(x,y)}{L^2_{white}}\right)}{1 + L_s(x,y)} \tag{3.17}$$

Parameter Estimation

In his paper *Parameter Estimation for Photographic Tone Reproduction* [26], Reinhard presents empirically determined solutions to how the parameters α in equation 3.18 and L_{white} in equation 3.19 can be determined if L_{min} and L_{max} of the scene are known. Although this could avoid the

manual tweaking of parameters, this never made it into the final implementation due to lack of time.

$$\alpha = 0.18 \times 4\left(\frac{2\log_2(\bar{L}_w) - \log_2(L_{min}) - \log_2(L_{max})}{\log_2(L_{max}) - \log_2(L_{min})}\right) \tag{3.18}$$

$$L_{white} = 1.5 \times 2\Big(\log_2(L_{max}) - \log_2(L_{min}) - 5\Big) \tag{3.19}$$

Alternative Formats

According to the HDRFormats Sample in the DirectX 9.0 SDK [27], it is not necessary to use floating point textures to perform HDR rendering. This example demonstrates how to use standard integer formats for storing compressed HDR values, thus enabling the use of HDR on graphics cards lacking support for floating point textures. This approach is however not a perfect replacement for floating point textures, as it leads to loss of precision and/or loss of range. The implementation uses true floating point formats.

3.4 Bloom

When watching a very bright light source or reflection, the light scattering in the human eye or a camera lens makes these bright areas bleed into surrounding areas. The scattering of light occurs in the cornea, the crystalline lens, and the first layer of the retina, as seen in figure 3.17 taken from the paper *Physically-Based Glare Effects for Digital Images* by Greg Spencer et al. [28]. These three locations of scattering contribute more or less equally to this so-called "veiling luminance" or *bloom*.

Another effect, mainly caused by scattering in the lens, is called flare. Flare is in turn composed of the *lenticular halo* and the *ciliary corona*. The ciliary corona looks like radial streaks or rays, while the lenticular halo is seen as colored concentric rings. The lenticular halo manifests itself around point lights in dark environments. Also the ciliary corona is a visual effect extending from point light sources, but it's often also observed in daylight

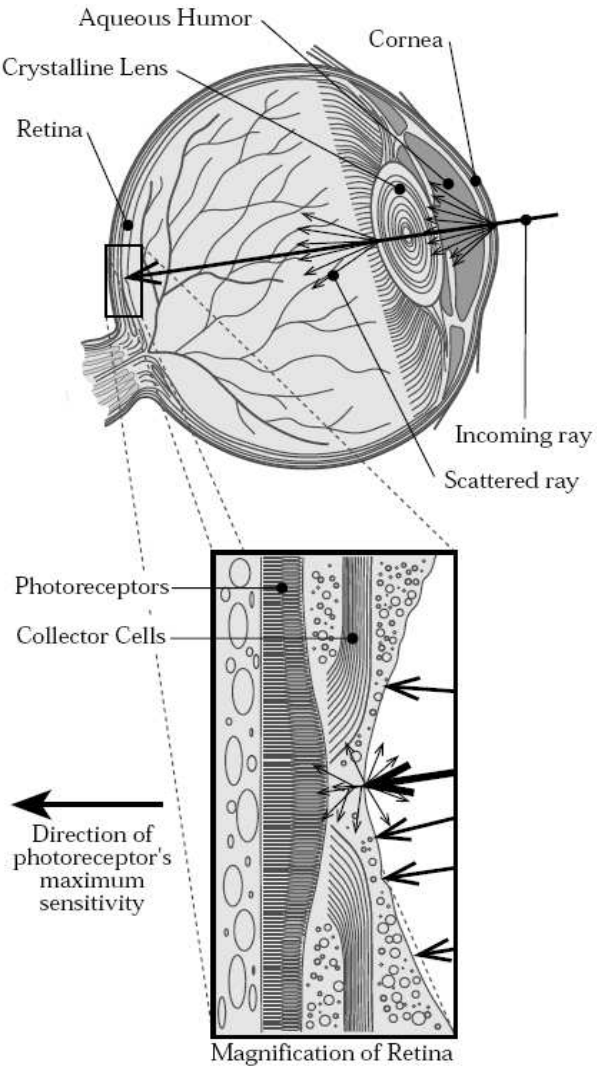

Figure 3.17: Light scattering in the human eye. Image courtesy of Greg Spencer et al. [28].

environments, as for example when the sun is seen through the leaves of a treetop.

Both bloom and flare contribute to the perceived brightness of a light source, but since this thesis will focus on a daylight environment where the bright areas of an image largely are limited to the sun and window pane reflections, only bloom will be taken into consideration. Simulating the streaks of the ciliary corona is also a computationally rather expensive post-process, at least if more than four or six streaks are to be generated. Of course it all depends on how well one wants to approximate this effect. Erik Häggmark discusses the underlying theory of bloom and flare in his master thesis *Nighttime Driving: Real-time Simulation and Visualization of Vehicle Illumination for Nighttime Driving in a Simulator* [29].

The cone formed by the light scattered in the cornea and the lens approximately has a Gaussian distribution. The resulting bloom effect can be simulated by low-pass filtering a thresholded high dynamic range image, and then adding the result to the original image. The areas which are given this bloom effect are made up of those pixels with a luminance above a certain value. For the implementation this threshold was set to 1.5, but it should be considered a tweak factor and a value which works well in one context may not give good results in another, depending on the dynamic range. Slightly different ways of doing this quickly are discussed in section 3.4.2.

3.4.1 Convolution on the GPU

The modern day GPU is very capable of performing computationally expensive image processing tasks in real-time, such as that of low-pass filtering an image. The images are represented as 2D grids, where each grid cell is a pixel or texel. When calculating the convolution sum at each grid cell, one has two options for sharing information between neighboring grid cells, i.e. scatter and gather. See figure 3.18 for an illustration of the difference between the two. Gather and scatter correspond to conventional convolution and impulse summation respectively.

Looking more closely at the architectural limitations of the GPU, one sees that the vertex processor is capable of scatter but not gather, whereas the fragment processor is capable of gather but not scatter [30]. The vertex

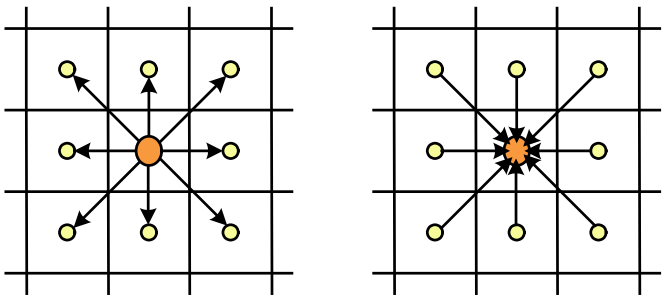

Figure 3.18: Scatter vs gather for grid computation

processor can change the position of a vertex (scatter), but as one vertex is being processed, it can not access other vertices. The fragment processor can perform random access reads from textures and the output is fixed to the pixel corresponding to the current fragment. As modern GPUs have more fragment pipelines than vertex pipelines and the fragment processor provides direct output for the result, the use of pixel shaders for the computation is preferred. Mapping convolution to the vertex processor would be complicated and I doubt the speed would come close to that of the fragment processor.

When filtering an image, the image itself is bound as a texture for read-only random access. The convolution kernel could be either bound as a texture or stored as a uniform vector passed to the shader. The uniform qualifier declares a variable as global and is a type of variable for which its value stays constant for the entire geometric primitive being processed [2]. Storing the kernel as a texture would only increase the bandwidth requirements of the computation. The result of the convolution is preferrably stored in a texture by using the framebuffer object framework for rendering directly to the texture instead of first rendering to a framebuffer and the copying the written information to a texture with `glCopyTexSubImage2D()`, which forces an indirection on the process of rendering to a texture. To invoke the computation for each texel in the destination texture, one has only to draw screen-aligned rectangle.

To avoid unwanted wrapping when filtering pixels near the edges of an image, the texture wrapping mode should be set to `GL_CLAMP_TO_BORDER` and the border set to black. Depending on the intended purpose of the

filtering, other wrap modes may be more appropriate.

It seems like the current OpenGL driver implementation does not allow for the number of iterations in a loop to be specified by a uniform variable. It appears the driver needs to know the number of iterations at compile time. This means one has to provide a different shader for each size kernel. Probably the easiest way to do this, is to write a function taking the kernel size as an argument and returns the corresponding shader as a string.

3.4.2 Different Approaches

A few different approaches can be taken to efficiently low-pass filter an image and imitate the effect of light scattering in the eye.

Repeated Convolution

In *Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless)* [21], Masaki Kawasi introduces one way of minimizing the number of texture lookups while still achieving an acceptable quality low-pass filter. According to Kawase, a single binomial filter does not produce acceptable results, as it gives a small effective radius and the bloom is not sharp enough around the light. Instead he suggests repeated filtering with small kernels with slightly different radii.

The method he presents limits itself to the use of kernels with four taps, but performs multiple passes, each time with a slightly wider kernel. The kernel used in the first pass is a standard 3x3 binomial kernel. By using bilinear filtering only four texture lookups are needed instead of nine as with nearest neighbor filtering. See figure 3.19. By sampling at the intersection of four pixels, these are all sampled and given an equal weight in the bilinear interpolation. Adding together the result of the four texture reads and dividing by four gives the nine texels weights equal to those of the kernel.

$$\mathbf{B}^2 = \left(\frac{1}{4} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \frac{1}{4} \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \frac{1}{4} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} + \frac{1}{4} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \right) / 4 = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

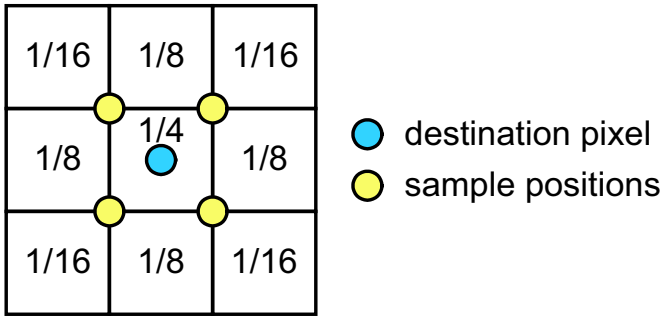

Figure 3.19: 3x3 binomial filter kernel with only four texture lookups

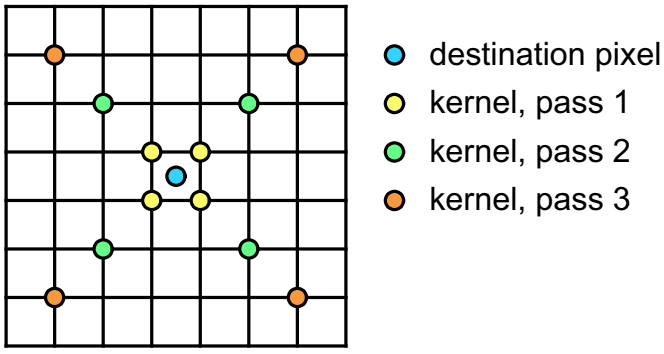

Figure 3.20: Sample points for first, second, and third pass

For each pass the four taps are moved further apart as shown in figure 3.20. To get a visually appealing result, one should do at least five passes. This method was discarded because of problems with flickering of the bloom when the camera was in motion, most likely due to imperfections in the resulting filter kernel. In any case, the method did give acceptable results for still images. See figure 3.21.

Downsampling Filtered Textures

In *Practical Implementation of High Dynamic Range Rendering* [31], Kawase presents an alternative way of efficiently performing this filtering while not introducing any noticeable artifacts. Instead of changing the kernel width, the thresholded high dynamic range image is downsampled a number of

Figure 3.21: Bloom with Kawase's first method

times, using a 2x2 box filter. These images are then each filtered with a relatively large Gaussian filter. Each filtered low resolution image is finally magnified to the same size as the original image using bilinear filtering, and then added to the original image. By changing the image resolution instead of the filter radius, a strong low-pass filtering can be done quickly, while artifacts due the linear interpolation when magnifying the result are difficult to discern.

By taking advantage of the separability of Gaussian 2D filters, the kernel is decomposed into two 1D kernels, further reducing the number of texture lookups from n^2 to $2n$. A horizontal and a vertical pass is performed, giving the same result as the 2D filter would have. See equation 3.20 for an example of the decomposition of a 2D kernel.

$$\mathbf{B}^2 = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} = \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} * \frac{1}{4}\begin{bmatrix}1\\ 2\\ 1\end{bmatrix} \tag{3.20}$$

Combined with the method discussed in GPU Gems 2 [3], chapter 20, this is further reduced to only $(n+1)$ div $2 \approx n$ linearly interpolated texture lookups. Figure 3.22 illustrates how the number of samples are reduced to about half by adjusting the sample positions and kernel weights. The

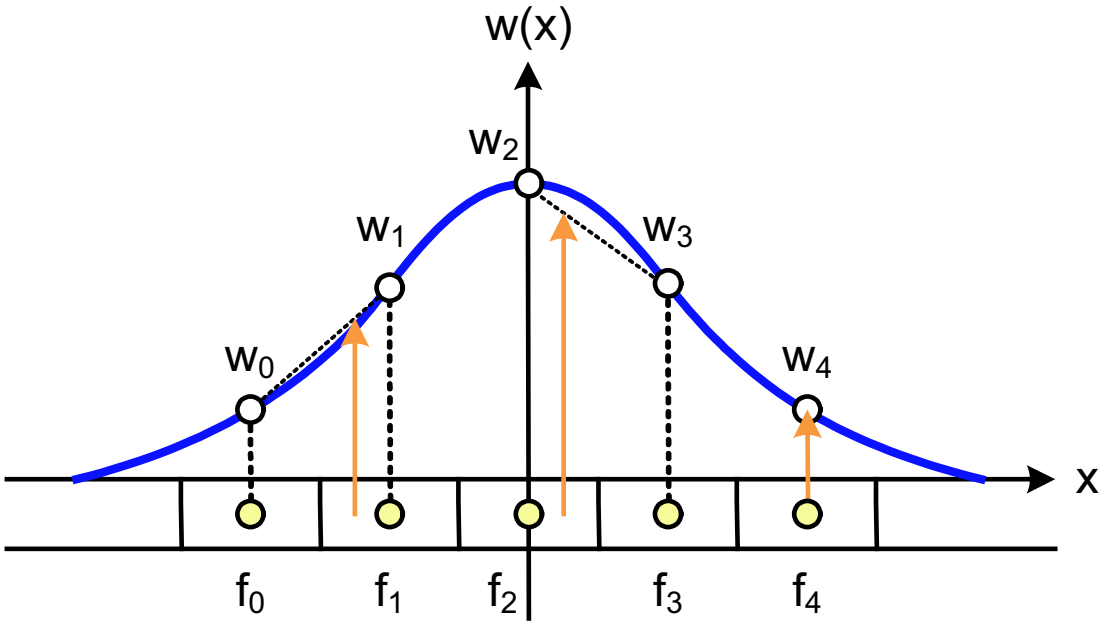

Figure 3.22: Convolution taking advantage of linear texture filtering

unadjusted convolution sum can be written as

$$f_x = w_i \times f_i + w_{i+1} \times f_{i+1} + \dots$$

If the convex combination property $0 \leq b/(a+b) \leq 1$ holds, then the convolution sum for each pair of samples can be written as

$$f_x = (a+b) \times \big((1-\alpha) \times f_i + \alpha \times f_{i+1}\big)$$

In other words, we perform a texture lookup at $i + b/(a+b)$ and multiply with $(a+b)$.

After tweaking parameters and sizes, it was decided to use four down-sampled images, the largest of which was $1/8^{\text{th}} \times 1/8^{\text{th}}$ the size of the original image. A Gaussian filter with standard deviation of 3 was used for each of these and gave good results. Smaller filters showed visual artifacts for the smaller of the filtered images when magnified.

Chapter 4

Lighting

This chapter talks about implications of lighting as applied to high quality rendering of large architectural environments.

4.1 Global Illumination

In order to achieve a certain level of realism and a perceived effect of immersion, a more complex lighting model must be used to better approximate the physical characteristics of the interaction of light with matter. Standard *local illumination* models provide a poor alternative, while *global illumination* models are difficult to use in real-time graphics.

Global illumination takes into account, not only light coming directly from a light source (*direct illumination*), but also light reflected off other surfaces in the environment (*indirect illumination*). Examples of such algorithms are radiosity, ray tracing, and photon mapping. By allowing light to scatter throughout an environment, much more natural tones and shadows are achieved.

4.2 Light Mapping

John Carmack was first in applying an approach called *light mapping* to real-time graphics in the computer game Quake [32]. Contrary to standard per-pixel lighting, where the correct illumination is calculated for every fragment as a function of the light, eye, and normal vectors, light mapping decouples lighting from rasterization. In a static lighting environment (where both the geometry and the light sources remain still), the diffuse component is view-independent and can thus be stored in a texture. The idea is to perform lighting in two steps. First, textures which capture the light contributions from surrounding geometry and light sources are pre-calculated offline. A scene is then lit by multiplying the value found in a light map with the diffuse texture of the underlying surface. This way, a more accurate lighting model, such as radiosity, which takes into account indirect lighting, can be used in the pre-calculation step while still maintaining real-time performance. Shadow mapping uses a similar approach, using textures to store information about shadows instead of reflected brightness, as explained in greater detail in chapter 5.

Since the brightness of diffuse lighting usually changes relatively slowly over a surface, it is sufficient to use low resolution textures for the light maps [33]. This observation saves a great amount of bandwidth considering the amount of geometry in an architectural walkthrough.

Building surfaces, except for window panes, are quite well approximated with a diffuse bidirectional reflectance distribution functions (BRDF) and works well with light mapping. Light mapping was used in the implementation for the lighting of building facades. See chapter 3 for a discussion about planar reflections on buildings.

4.2.1 High Dynamic Range

Light maps could be generated as high dynamic range floating point textures (and stored using a fileformat such as OpenEXR [34]), this way taking advantage of the HDR rendering discussed in section 3.3 and used in the implementation. According to Carsten Wenzel in his presentation [35] at the Game Developer's Conference 2005, this approach was successfully implemented by Crytek in a prototype of Far Cry, and gave very pleasing

results. Unfortunately, this would increase the bandwidth requirements and within the context of this thesis it was decided to limit the application to the use of low dynamic range light maps. 3ds max was used for the generation of light maps and the corresponding texture coordinates.

4.3 Ambient Occlusion

A method known as *ambient occlusion* can be used to approximate global illumination [36] for dynamic geometry. It is a simplified special case of spherical harmonics. The ambient occlusion term is a value in the interval [0,1] which approximates the light attenuation resulting from surrounding geometry by estimating the fraction of the hemisphere at that point not obstructed by other geometry. It could be thought of as a number representing how visible or accessible a point is from the outside. Usually an ambient occlusion term is calculated for every vertex of an object. This term is often calculated using a Monte-Carlo algorithm, sampling the hemisphere by casting random rays and calculating the fraction not hitting any object. Often, one also calculates a so-called *bent normal*, which points in the average direction of unoccluded samples. The bent normal can be used to look up values in an environment map.

As the method does not take into account the orientation of an object relative to surrounding geometry and light sources, it's very much suitable for animated geometry where pre-calculated light mapping can not be used. By calculating the ambient occlusion term for key frames of an animation sequence and interpolating these values, the method can be used for real-time graphics. This was never implemented as priority was given to the illumination of static geometry. The method should in any case give good performance and quality.

Chapter 5

Shadows

This chapter talks about problems involved with current shadow techniques and good choices within the current context.

The use of light maps to account for the effects of indirect lighting works well for static geometry and materials with reflection properties being mainly diffuse [33], but does not work with dynamic geometry as the movement of any part of the geometry would invalidate the correctness these lighting calculations.

One of the criteria for the implementation was to accommodate for the possible incorporation of dynamic objects, such as people and vehicles. Approximating a global illumination model for vast amount of dynamic geometry is not feasible, nor would the use of light maps for static geometry, such as buildings, and only a direct illumination of dynamic objects give a very realistic overall appearance. One approach sometimes used in these cases is to keep using light maps for the lighting of buildings, as this is not affected much by smaller dynamic objects, and then perform a very crude approximation of the effects of global illumination on people, using a method such as ambient occlusion [36] to estimate diffuse lighting (see section 4.3) and implement a technique for dynamic shadows.

Shadows are often considered to be one of the most important parts of a real-time graphics engine and some of the questions that arise are which objects cast shadows, which ones receive shadows, and what technique is

most suitable for the intended purpose. For the context of this thesis, dynamic shadows would not only allow dynamic geometry to cast shadows onto itself, but also give a visual queue to help ”anchor” this geometry to the ground.

5.1 Common Methods

Global illumination models implicitly include the generation of very realistic shadows. For local illumination models, different methods have been tried to generate penumbras (or soft shadows) to increase realism. Different techniques for generation of dynamic hard shadows include stenciled shadow volumes, projected planar shadows, and shadow mapping. Each method has its pros and cons and complex dynamic geometry complicates the matter.

5.1.1 Stenciled Shadow Volumes

Stenciled Shadow volumes [37] are a mathematically elegant way to compute omni-directional shadows. A shadow volume is the half-space defined by a light source and a shadowing object. A surface inside the shadow volume is shadowed, whereas a surface outside is illuminated. To compute the shadow volumes one must first compute the silhouettes of occluding objects and extrude the volumes from these. The big question when using shadow volumes is how to determine these silhouettes, a task which can be quite demanding for geometrically complex objects.

To determine whether a fragment should be lit or shadowed, the shadow volumes are rendered and the stencil buffer is modified on a per-pixel basis, depending on whether the fragment passed or failed the depth test. In a following lighting pass, a non-zero value in the stencil buffer means surface should be shadowed. There are many aspects to how this should be done and an entire chapter has been dedicated to stenciled shadow volumes in the book *Mathematics for 3D Game Programming and Computer Graphics* [37].

The vast amount of geometry used in the implementation makes this method less suitable.

5.1.2 Projected Planar Shadows

Projected planar shadows is probably the easiest way of creating dynamic shadows and is done by creating a matrix which projects the geometry onto a given plane. For a long time, this was the dominant method for shadow generation. The downside with projected planar shadows is that they only work for flat surfaces and it's also not an easy task to achieve soft shadows as this method makes use of the stencil buffer. The interested reader can look at Mark J. Kilgard's article on the subject for an in depth discussion of the algorithm [38].

5.1.3 Shadow Mapping

Shadow mapping is an image based technique for creating shadows [39]. It's a two pass algorithm and in a first pass the scene is rendered from the point-of-view of the light source to generate a shadow map, storing the depth for each fragment. In the second pass, the scene is rendered from the camera's point-of-view and each fragment is transformed from camera eye space into clip space of the light source and then finally scaled and biased to the interval [0,1]. The depth of the fragment is then compared to the depth stored in the shadow map and if the comparison yields greater, it's shadowed as there must be an occluding object closer to the light source. The basic principle is that: *light can "see" point* $\Leftrightarrow$ *point is not shadowed*. Figure 5.1 tries to illustrate the concept of shadow mapping and depicts the situation where a fragment is shadowed. The depth for the current fragment is here greater than the depth stored in the shadow map, signifying that there must be something between the light source and the fragment.

5.2 Shadow Mapping

Shadow mapping was the method of choice for the generation of dynamic shadows and it has many advantages over the otherwise elegant shadow volumes [36]:

- better scaling with GPU power
- softer edges

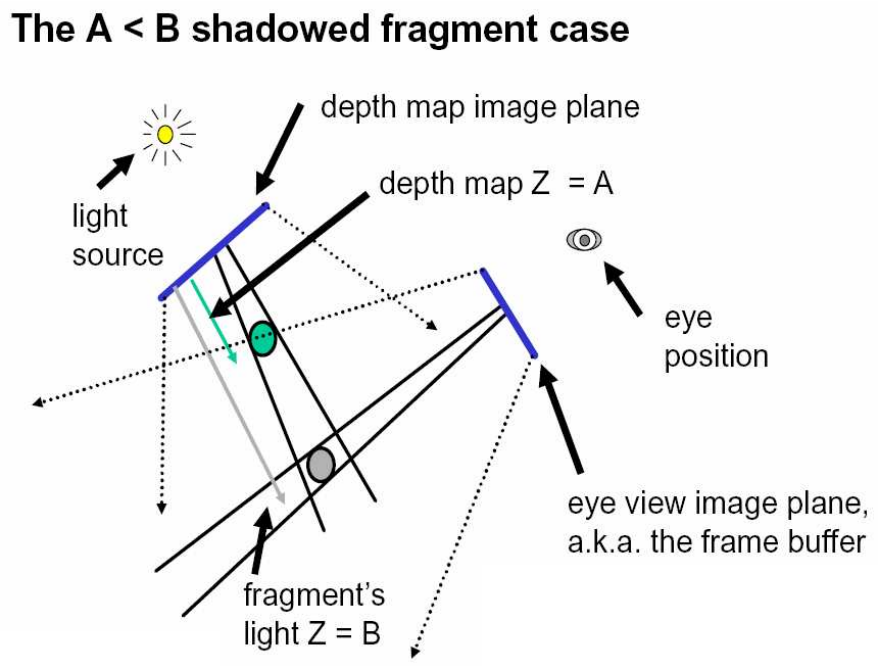

Figure 5.1: The concept of shadow mapping. Image courtesy of Mark J. Kilgard.

- applicability to different kinds of geometry

There are however also disadvantages:

- incorrect self-shadowing of surfaces (shadow acne)
- blocky shadows when shadow map resolution is low
- flickering due to changes in shadow map resolution
- point lights cast shadows in all directions and no *single* linear transform can handle all occluders

5.2.1 Theory

As briefly touched upon earlier, shadow mapping makes use of *projective texture mapping*, more elaborately described by Cass Everitt in his paper with the same title [40]. Basically, what it boils down to is the use of the world or eye space position of a fragment as texture coordinate, a transformation of this texture coordinate into projector clip space, and then finally a scale and bias to the texture space interval [0,1]. See equation 5.1. When applied to shadow mapping, this has to be done since the fragment and the texel have to be in the same space for a depth comparison to make any sense.

$$\begin{pmatrix} s \\ t \\ p \\ q \end{pmatrix} = \begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} light \\ projection \\ matrix \end{pmatrix} \begin{pmatrix} light \\ view \\ matrix \end{pmatrix} \begin{pmatrix} inverse \\ camera \\ view \\ matrix \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} \tag{5.1}$$

5.2.2 Shadow Acne

So-called shadow acne (i.e. artifacts due to the discretization of the depth space) takes away any illusion of shadows. The standard way of removing these artifacts is to add a small constant value to the fragment depth when creating the shadow map. This constant is know as *depth bias* and has no

perfect value; it's a tweak factor, i.e. a value one must adjust until shadow acne is minimized. Shadow acne can be seen on the sphere in the top image of figure 5.3.

There is one gotcha when doing shadow mapping with a non-linear mapping and adding a constant depth bias. Results are not what one would expect at a first glance. This is due to the resulting non-linearity of space. Hence, one must use an adaptive depth bias [36]. An orthographic projection is used for both directional lights and spot lights to avoid such issues.

5.2.3 Dueling Frusta

When the frusta of the camera and light source are aligned, there can be almost a 1:1 mapping between pixels and shadow map texels. The concept *dueling frusta* occurs when these two frusta are aligned, but face opposite directions. Here, distant objects get high resolution shadows while objects close to the camera (which occupy a large portion of the screen) get very poor resolution. In this situation there is a significant mismatch between the sampling frequencies for the shadow map and the eye.

Recently, *trapezoidal shadow mapping* [41] was developed to deal with problems with shadow map resolution, continuity (shadow map quality changes from frame to frame, producing flickering), and polygon offset (shadow acne). The methods is however patented and was not used in the implementation.

5.2.4 Soft Shadows

The area of a shadow is divided in *umbra* and *penumbra*, where umbra is the completely shadowed region and the penumbra is located along the shadow edges and forms a gradual transition between the fully lit region and the umbra. See figure 5.2. The size of the penumbra depends on the size of the light source and the distance to the occluder and the receiver. In reality, shadows harden (the penumbra gets smaller) on contact between occluder and receiver. This is however often overlooked in favor of performance, as was also done in the implementation.

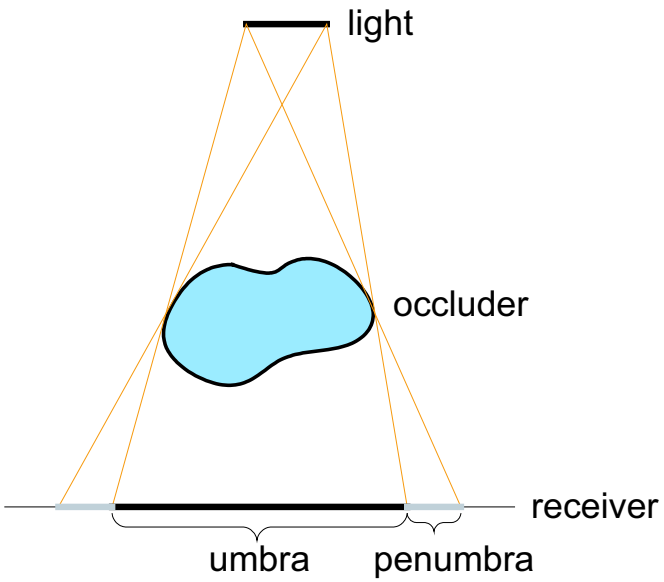

Figure 5.2: Umbra vs penumbra for area lights

The ease of achieving *soft shadows* (penumbra) was one of the main reasons for choosing shadow mapping above other shadowing methods. A popular technique known as *percentage closer filtering* (PCF) can be used to create nice looking gradual penumbras. In this methods, the shadow map is super sampled in some specific pattern and the result of each sample is then averaged to decide the darkness of the shadow.

Advanced theories for how one should perform the super sampling exist. One of these consists in dividing the sampling space into a rectangular grid, *jittering* the sample within each square, warping the grid into a circle, and then finally performing the sampling with these offsets. By jittering (i.e. randomizing the location of a sample within its allocated square) the aliasing/banding is substituted by high frequency noise, which the human eye easily filters out. It is also usually recommended to perform at least 16-32 samples to achieve acceptable quality of shadow edges or maybe even as many as 64 samples as described in GPU Gems 2 [3].

At the time of writing, a new method named *percentage-closer soft shadows* (PCSS) was presented by Randima Fernando at SIGGRAPH [42]. Instead of having uniformly soft shadows as with standard PCF, the kernel

width is varied according to the light size, and occluder and receiver depth. A larger kernel width gives softer shadows. Fernando used 72-292 texture lookups depending on penumbra size, which must be considered unrealistic for graphics cards older than the GeForce 7 Series and especially for an application where a lot more things are going on than shadow generation.

Tests performed with percentage closer filtering show that one in most cases can achieve high quality soft edges by performing only 8 samples per pixel, performing an additional *4-sample PCF* (2x2) for each of these in hardware. NVIDIA has the patent on performing a bilinear interpolation of the 4 nearby shadow values for every shadow map lookup at the speed of a single normal lookup (UltraShadow) by means of special purpose hardware [43]. The comparison of a fragment depth value against the corresponding depth value in the shadow map is performed in hardware, but one must make sure textures have their depth format and depth comparison mode set, or else results are undefined. We effectively get 32 lookups at the expense of 8.

The current implementation of soft shadows lowers the frame rate by 2 FPS when performing deferred shading at a resolution of 800x600 for a single directional light. More advanced techniques provide a higher quality penumbra, but will likely perform worse as far as speed goes. The result of hard shadows versus PCF is shown in figure 5.3. As can be seen, percentage closer filtering also does a good job of removing shadow acne. The sampling pattern used in the implementation is shown in figure 5.4. No jittering was performed since no disturbing aliasing or banding effects were observed.

Naturally, the shadow map for each light only needs updating when the light position/direction or shadow casting geometry changes. Even when the position or direction of a light changes from frame to frame, the shadow map can be updated every n^{th} frame to increase speed. On GeForce FX and 6 Series hardware one can also do *z-only rendering* at double speed by disabling color writes and alpha testing, which is particularly useful for dynamic shadow mapping.

5.2.5 Omni-Directional Shadows

There are at least two well known methods for omni-directional shadow mapping, which are *dual-paraboloid shadow mapping* [44] and *cube shadow*

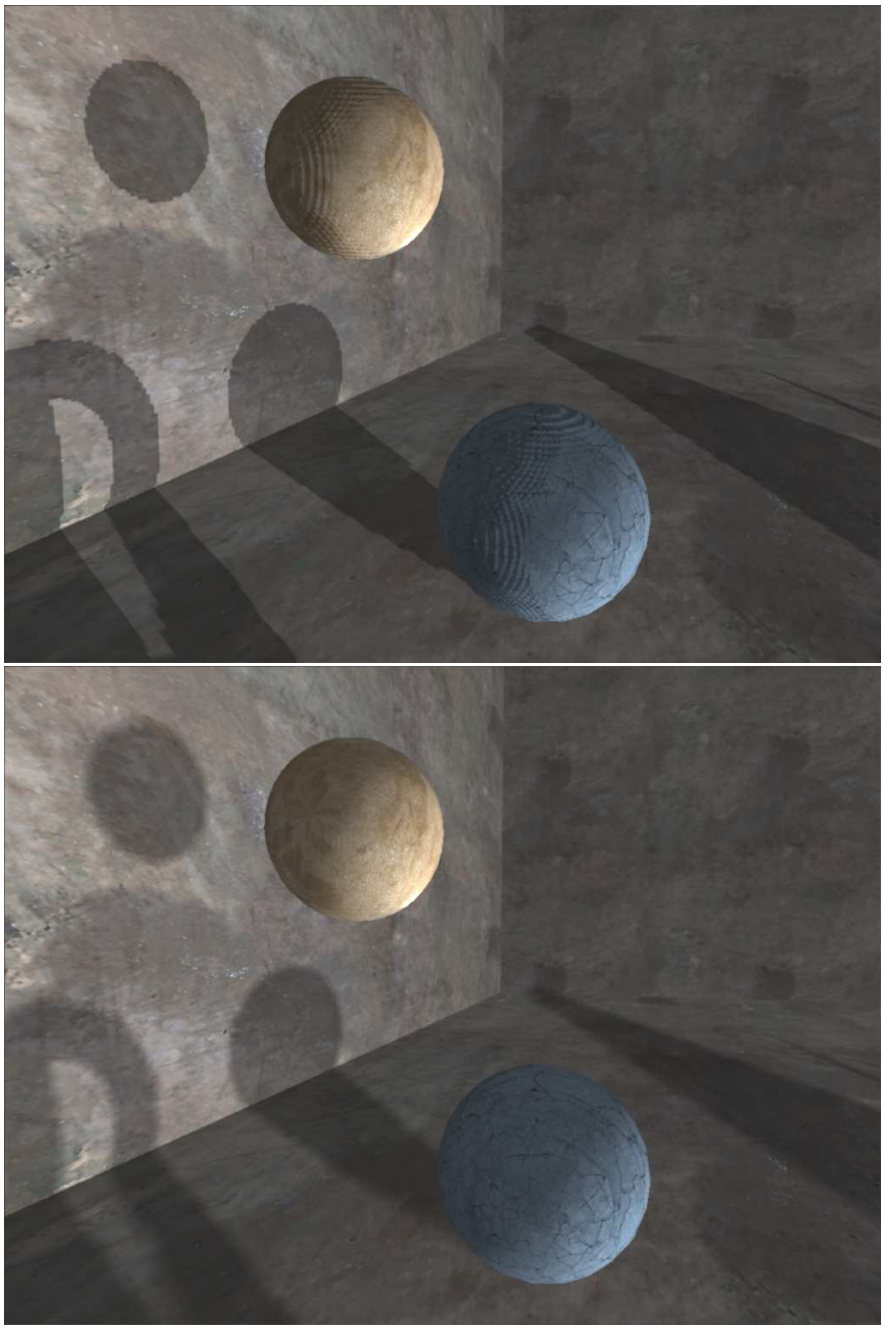

Figure 5.3: Hard shadows (top) and PCF soft shadows (bottom)

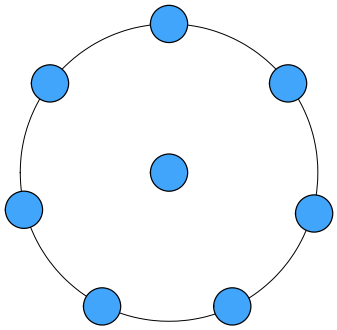

Figure 5.4: Sampling pattern used for percentage closer filtering

mapping. Both of these parameterizations have already been commented in chapter 3, although then in the context of environment mapping. Common to both of these methods is that more than one rendering pass is used to account for all occluders.

Cube Shadow Mapping

Cube shadow mapping uses the same parameterization as cube environment mapping, described in section 3.2.2, but stores depth information instead of colors for surrounding geometry. While only a single texture lookup is needed to do the depth comparison, no less than six render passes are needed to generate the textures forming the sides of the cube map, making this method rather expensive even for a small number of point lights.

Dual-Paraboloid Shadow Mapping

Dual-paraboloid shadow mapping was introduced by Stefan Brabec et al. in their paper *Shadow Mapping for Hemispherical and Omnidirectional Light Sources* [44]. Geometry is here mapped to a paraboloid and one advantage with this method over others is that the sampling rate only varies by factor of four between the center of the shadow map and its outer regions. The paraboloid used for this mapping is defined in equation 5.2 and is the same

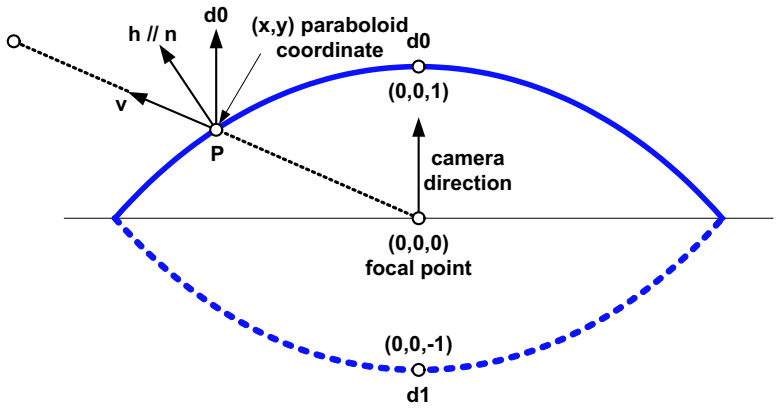

Figure 5.5: Mapping to (x,y) paraboloid coordinates

as for the environment paraboloid mapping in chapter 3.

$$f(x) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2),\ x^2 + y^2 \leq 1 \tag{5.2}$$

The rest of the discussion in this section will refer to figure 5.5. The *hemisphere* for geometry with positive z-coordinates is centered at $(0, 0, 0)$ and is oriented towards the camera, located at $(0, 0, 1)$. The paraboloid captures from light from its entire hemisphere and reflects this in the direction $\overrightarrow{d0} = (0, 0, 1)$ (or $\overrightarrow{d1} = (0, 0, -1)$ for the opposite hemisphere and paraboloid). To capture the entire environment, two paraboloids are attached back to back.

The 3D-to-2D mapping is performed by finding the point P on the paraboloid which reflects a given direction $\overrightarrow{v}$ in the direction $\overrightarrow{d0}$ (or $\overrightarrow{d1}$). The normal vector at the point P can be shown to be equal to the expression in equation 5.3.

$$\overrightarrow{n} = \frac{1}{z}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{5.3}$$

The halfway vector points in the same direction as the normal, but is

scaled by a factor. See equation 5.4.

$$\overrightarrow{h} = \overrightarrow{d0} + \overrightarrow{v} = k \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \; v_z \geq 0 \tag{5.4}$$

Scaling the halfway vector to $z = 1$ gives us a 2D *paraboloid coordinate* (x, y) which can be used for a shadow map lookup. To get the 3D-to-3D mapping needed to perform shadow mapping, one takes advantage of the fact that dual-paraboloid mapping divides the world in two half-spaces, one with positive z-coordinates and the other with negative z-coordinates. As a consequence, one can use the distance from the center of the paraboloid $(0, 0, 0)$ to the surface point as depth. The approach presented by Stefan Brabec et al. then uses a type of alpha testing to cull pixels not being part of the currently rendered hemisphere.

One drawback of dual-paraboloid shadow mapping has to do with the *non-linear parabolic projection* which maps straight lines to curves. Rasterization hardware performs a linear interpolation when generating fragments and this causes artifacts, but by tessellating geometry finely enough, the effects are unnoticeable. Walls, floor/ground, and ceiling, which are usually not very tessellated, can be ignored during shadow map generation as this is bounding geometry (for indoor environments) and will not cast any visible shadows. For the context of outdoor architectural environments, occluders were limited to dynamic and well tessellated geometry (at this time not incorporated in the implementation).

Using the multiple render target (MRT) capabilities provided by current hardware we could in theory limit shadow map generation for omnidirectional light sources to one rendering pass. Instead of performing two passes, one for each hemisphere, we could bind two shadow map render targets to the framebuffer object and decide which one to render to for every rasterized fragment, using dynamic branching provided by Shader Model 3.0 compatible graphics cards [1]. Alternatively, we could bind one texture with two channels (e.g. G16R16F) and decide which channel to render to. Both alternatives would however make it impossible to get 4-sample PCF and comparison for free; instead this would have to be done in the shader. Nor would we get double speed z-only rendering, as binding two shadow

maps or one with two channels implies binding these to the framebuffer object as color attachments.

Staying with the idea of single pass dual-paraboloid shadow mapping, another option would be to use a rectangular texture of proportions 2:1 (e.g. 2048x1024) instead of two textures of proportions 1:1, one for each hemisphere. One would then index either the left or right half depending on the sign of the z-coordinate. This would allow for 4-sample PCF with free depth comparison in hardware, plus double speed z-only rendering. Geometry would only have to be passed through the graphics pipeline once, and there would also be a reduction in the number of texture binds, possibly having a positive effect on frame rate, especially when various point lights are present in a scene.

Unfortunately, a big problem was run into. There is currently no way to handle the case when a triangle spans the boundary between the two hemispheres. As mentioned in chapter 2, the fragment processor can not perform scattering (i.e. choose what pixel to write to), nor can the rasterization unit handle such geometric primitives. Perhaps new developments in graphics hardware will provide a solution and allow for single pass dual-paraboloid shadow map generation. In any case, it's possible that similar results can be achieved by not only culling geometry outside the radius of the light source, but also that located in the opposite hemisphere, at the time of shadow map generation.

Sampling Rate

As mentioned earlier, the sampling rate for different directions varies by a factor of four when using paraboloid mapping. Pixels in the outer region of the paraboloid map cover only $1/4^{\text{th}}$ of the solid angle covered by pixels in the center [45].

$$\overrightarrow{f}(x,y) = \left(x, y, \frac{1}{2} - \frac{1}{2}(x^2 + y^2)\right)^T$$

The solid angle covered by the pixel is the projection of dA onto the unit sphere:

$$\omega(x,y) = \frac{dA}{\|\overrightarrow{f}(x,y)\|^2} sr$$

$$f(0,0) = 1/2$$

$$\omega_r = \frac{4sr}{m^2} dA$$

$$\omega(x,y)/\omega_r = \frac{1}{4\|\overrightarrow{f}(x,y)\|^2}$$

To cope with this one could vary the PCF kernel radius, but at this time no disturbing artifacts have been observed and the subject is overlooked.

Non-Linear Depth Distribution

The non-linear projection in paraboloid mapping complicates the matter of depth bias as depths in the shadow map are non-linearly distributed. The addition of a constant at one location in the depth texture during the shadow map generation phase will not have the same effect as adding this same constant somewhere else. The constant will translate to different lengths depending on the z-value of the current fragment. This has not been taken into account in the implementation, but as of now, no visible artifacts have been noticed that can be attributed to this.

Chapter 6

Surface Detail

This chapter reviews methods often used for adding surface detail to objects without increasing their geometric complexity, and explains in greater detail the methods used in the implementation part of the thesis.

6.1 Common Methods

There are several algorithms with the goal of adding small-scale surface detail to a model without the need to increase its geometric complexity. These include bump mapping, normal mapping, displacement mapping, distance mapping, etc.

6.1.1 Displacement Mapping

Displacement mapping (introduced by Robert Cook in 1984) is a method for adding small-scale detail to surfaces by adjusting the position of surface elements by iteratively tessellating the base surface, pushing vertices out along the normal of the base surface making use of a height map. This method requires multiple rendering passes.

6.1.2 Bump and Normal Mapping

In 1978, James Blinn introduced a technique called *bump mapping*. This method requires a height map and uses the partial derivatives of the height in the s and t directions to perturb the surface normal. But the most commonly used variation of bump mapping is *normal mapping* where a normal map is used instead of a height map to perturb the surface normals on a per-pixel basis. Unlike some other methods, bump mapping only affects the shading of a surface and does not allow for non-polygonal silhouettes nor for bumps to occlude each other.

6.1.3 Ray-Tracing Based Methods

Other very accurate ray-tracing based methods have very recently been developed, such as Per-Pixel Displacement Mapping with Distance Functions [3] or Parallax Occlusion Mapping [46]. These methods rely on multiple iterations in the pixel shader and were considered too demanding.

6.2 Normal Mapping

Many methods have been developed to cope with the silhouette and bump occlusion problems, but on the other hand, normal mapping adds very little to the computational complexity of the shaders. Almost all additional work required is two matrix multiplications for transformation of the light and eye vectors from eye space into *tangent space*, and one texture lookup plus unpacking of the *TBN-matrix*. The TBN-matrix defines the transformation from eye space to tangent space. Normal mapping was the method of choice for the given application because of its efficient execution.

6.2.1 Tangent Space

Tangent space, also sometimes called texture space, is a coordinate system oriented in such a way that the z-axis always points in the direction of the surface normal (not the one fetched from the normal map) and the x- and y-axes are tangents in the surface plane pointing in the direction of increasing s and t texture coordinates respectively. Since the normal map

is mapped to the surface as a texture, it's defined in tangent space. To be able to compute lighting the light vector must lie in the same vector space. By multiplying a light vector defined in eye space by the TBN-matrix, see equation 6.1, it's transformed (rotated) into tangent space.

$$\mathbf{L_{tangent}} = \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix} \mathbf{L_{eye}} \tag{6.1}$$

The TBN-matrix, defining the transformation from eye space into tangent space, can be found using equations below.

$$\begin{aligned} \mathbf{Q_1} &= \mathbf{P_1} - \mathbf{P_0} \\ \mathbf{Q_2} &= \mathbf{P_2} - \mathbf{P_0} \end{aligned} \tag{6.2}$$

$$\begin{aligned} (s_1, t_1) &= (s_1 - s_0, t_1 - t_0) \\ (s_2, t_2) &= (s_2 - s_0, t_2 - t_0) \end{aligned} \tag{6.3}$$

$$\begin{aligned} \mathbf{Q_1} &= s_1\mathbf{T} - t_1\mathbf{B} \\ \mathbf{Q_2} &= s_2\mathbf{T} - t_2\mathbf{B} \end{aligned} \tag{6.4}$$

To get a orthogonal coordinate system one can use *Gram-Schmidt orthogonalization*. Before finally creating the TBN-matrix, the vectors should be normalized. These vectors are then passed as vertex attributes and interpolated by rasterization hardware.

The determinant of the TBN-matrix represents the handedness TBN coordinate system and can be stored as a fourth component T_w in the tangent vector. Only the normal and tangent are then needed to calculate the binormal, which then no longer has to be passed as a vertex attribute. See equation 6.5.

$$\mathbf{B} = T_w(\mathbf{N} \times \mathbf{T}) \tag{6.5}$$

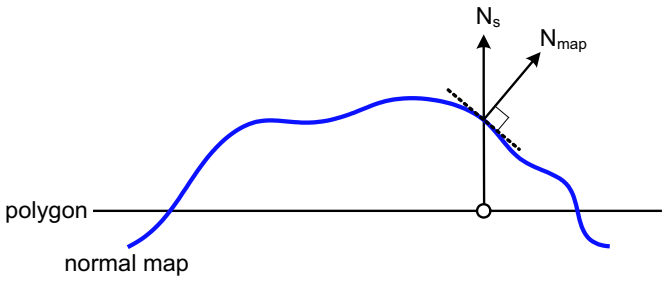

Figure 6.1: The concept of normal mapping

When calculating the lighting for a pixel, the perturbed normal N_{map} from the normal map is used instead of the surface normal N_s. As mentioned earlier, the light vector has to be transformed from eye space into tangent space before the dot products are calculated. Figure 6.1 intends to give an illustration of this.

6.2.2 Implementation Details

One problem quickly arose when implementing normal mapping within the framework of deferred shading. The normal and tangent (and the determinant of the TBN-matrix stored in the fourth tangent component) need to be written to the G-buffer and then used in the lighting pass. However, the normal map lookup needs to be performed in the lighting pass, using the same texture coordinates used selecting the diffuse color in the first pass. The G-buffer is already full with other attributes and to perform 2 passes for the creation of the the G-buffer is not really an option, as only having to transform and rasterize geometry once was one of the reasons for choosing deferred shading in the first place.

Taking into account that both the normal and the tangent are unit length, it's enough to store the x- and y-components of the vectors and calculate the z-component as $z = \sqrt{1 - x^2 - y^2}$ (i.e. hemisphere remapping [47]). This way we free up 2 slots in the G-buffer and these can use for storing the (s,t) texture coordinates. This would only make sense if every surface uses the same normal map, which would hardly be the case. Instead, two different normals needs to be stored in the G-buffer: the surface normal

and the normal stored in the normal map.

Also the eye space position can be ”packed” and later ”unpacked” by unprojecting the window (x,y) coordinates, storing only the z-coordinate and using the modelview and projection matrix. This would free up an additional 2 slots in the G-buffer but add several instructions per pixel. A tradeoff has to be made between space and speed, but since a reduction in the number of render targets did not show an increase in performance and the four component G-buffer still has space, the full position was stored.

For the different 1-dimensional attributes, such as specular and emission, these could be stored together in a single slot, combined into a single fp16 value and later unpacked. This would free up yet another slot in the G-buffer. The G-buffer is discussed further in section 2.2.

6.3 Parallax Mapping

As mentioned earlier, normal mapping has some flaws. This section will discuss one of these problems. While lighting computations are performed with a perturbed surface normal, the texture is still applied to the surface as if it was flat. By adding *parallax mapping*, the correct appearance of the uneven surface is better approximated [48]. Parallax is the effect of areas of a surface appearing to move relative to one another as the view position changes. Parallax mapping is a method for approximating this parallax using a height map to perturb texture coordinates as a function of the tangent space view vector. See figure 6.2. This method will not, however, simulate occlusion, silhouettes, nor self-shadowing. Combined with normal mapping, results can nevertheless give a quite convincing illusion uneven surfaces.

When looking at a normal mapped and textured surface, one would see point/texel A (see figure 6.2) just like one would for a truly flat surface. This will effectively remove some of the illusion of an uneven surface. With the help of a height map, an offset is calculated for the texture coordinates used to index the diffuse texture. The technique displaces texture coordinates individually and shifts elevated areas away from the eye while lowered areas are shifted towards the eye. The method does not give the correct texel for position B, but gives a good approximation of its location.

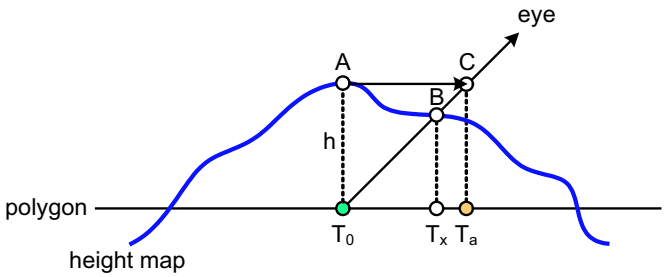

Figure 6.2: The concept of parallax mapping

Figure 6.3 shows the effect of normal and parallax mapping of an otherwise flat surface. Both a diffuse and specular component has here been included in the lighting. The difference between the middle and bottom image is especially notable in the right half. None of the models made available for the project contained any normal maps or height maps, but the functionality for normal and parallax mapping has nonetheless been included in the implementation since added surface detail formed part of the requirements for the thesis.

Figure 6.3: No added surface detail (top), normal mapping (middle), and normal mapping with parallax mapping (bottom)

Chapter 7

Discussion

This chapter evaluates the results of the thesis and gives a conclusion. It also comments on future work from which the application could benefit.

7.1 Requirements

The nature of the methods used in this project and the current implementation impose some requirements for both hardware and software.

7.1.1 Hardware

The implementation discussed throughout this thesis makes thorough use of hardware features only made available by GeForce 6 Series GPUs (or equivalent) and later. Among these are fragment level branching, floating point buffers and textures, multiple render targets, and non-power-of-two textures. Some of these have driver support on GeForce FX GPUs, but rendering would be performed on the CPU rather than on the GPU.

The hardware configuration used for implementation and testing had an Athlon 2400+ CPU, 1 GB of RAM, and a GeForce 6800 GT (AGP) graphics card.

7.1.2 Software

The application should run with an NVIDIA graphics driver of version 1.0-7667 for Linux (or Forceware 77.72 for Windows) or newer. To be able to compile and run the application, one must also install GLEW 1.3.2+ (`http://glew.sourceforge.net/`) and SDL (`http://www.libsdl.org/`), preferable 1.2.8 or newer.

7.2 Evaluation

This section evaluates the performance for a few aspects of the implementation. Deferred shading is tried to be put in perspective against the traditional forward shading for the rendering of large architectural environments.

7.2.1 Performance

As a consequence of using deferred shading, vertex transformations and texture transfers are only performed once per frame. Lighting calculations are also only performed for pixels ending up in the final image, thus not wasting expensive calculations on pixels later being overdrawn. Indoor environments with many small lights are rendered at framerates comparable to those of outdoor daylight environments with a single directional light, since only pixels affected by a light source are shaded.

Figure 7.1 shows how the execution times for the G-buffer pass at different resolutions. Execution times are averaged over 200 frames to give more accurate timings. Apparently, the execution time scales linearly against the number of pixels in the buffer, indicating that this pass is framebuffer bandwidth limited. Studying the graph further we see that this pass has a constant overhead of approximately 20 ms which most likely can be attributed to texture AGP transfer time. All textures do not fit in VRAM all at once and the used texture manager solves this by approximating necessary texture detail and downloading only lower mipmap levels to the graphics card when possible.

Figure 7.2 compares execution times for deferred shading and forward shading when rendering the Barcelona city block. Execution times are

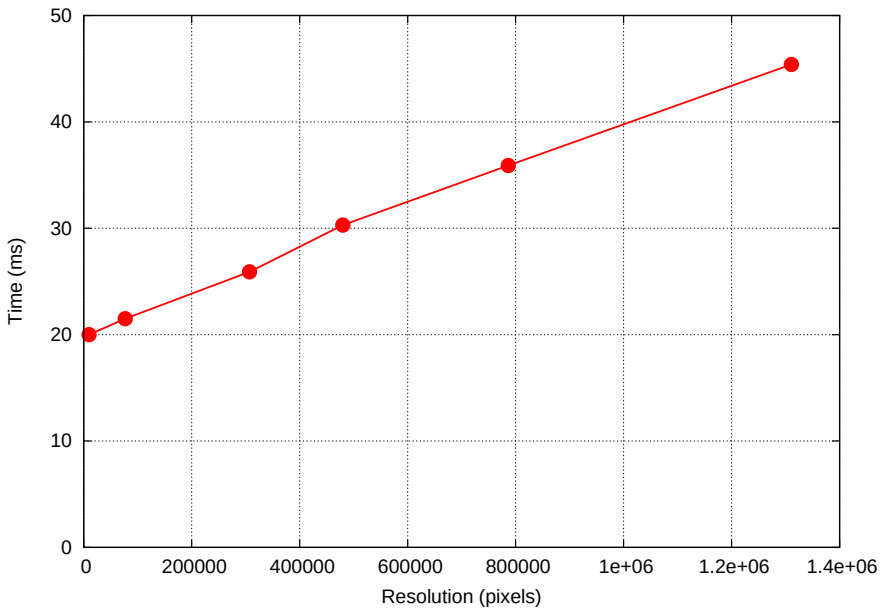

Figure 7.1: Execution times for G-buffer pass plotted against resolution

averaged over 200 frames here also. Both resolutions and number of light sources are varied, but all lights are directional since the implementation of deferred shading is optimized for small point and spot lights, which would bias the comparison in favor of deferred shading. Apart window reflection calculations, also PCF shadow mapping was enabled, just like it would have been if dynamic geometry had been present in the scene.

The time difference between deferred and forward shading with one light at smaller resolutions is most likely due to shaded fragments being overdrawn with forward shading, while deferred shading only performs calculations for fragments visible in the final image. The setup time is the same for deferred shading when going from one light source to two, while this is almost doubled for forward shading. All in all, differences in execution times between deferred shading and forward shading should be attributed to overdraw, vertex transformations, and texture transfers.

These figures were given when lighting calculations were more or less complex. Needless to say, when simply outputting textured geometry, forward shading performs much better. Deferred shading also has disadvan-

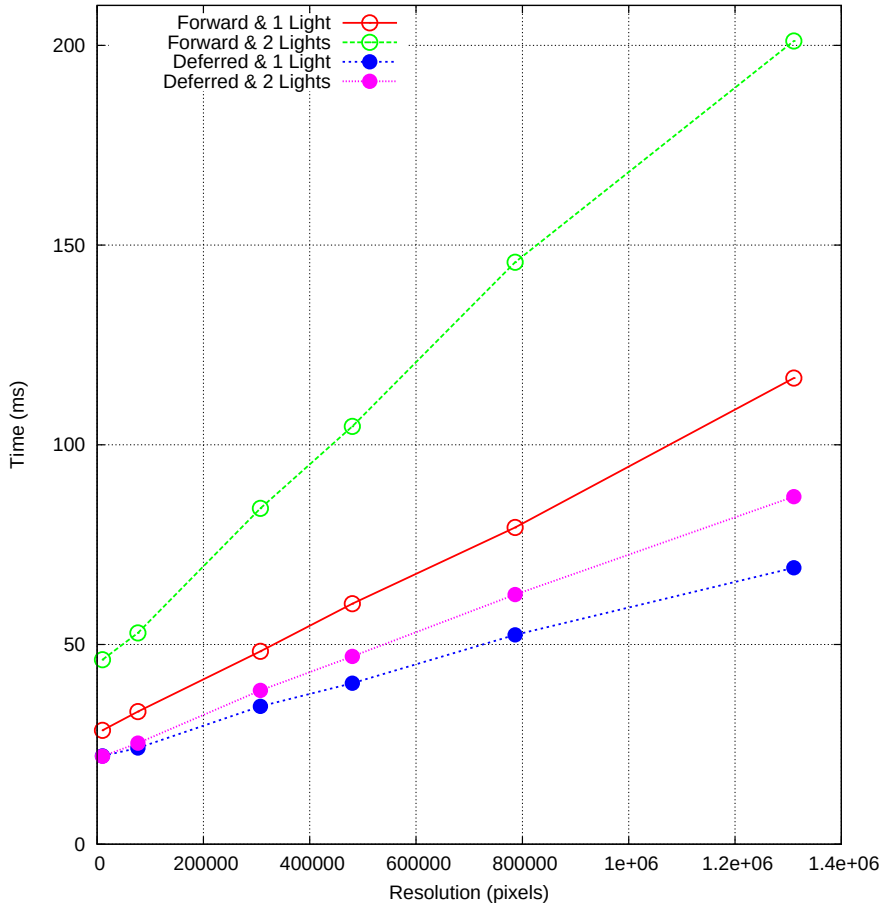

Figure 7.2: Comparison of deferred and forward shading execution times, plotted against resolution

tages when compared with forward shading, some of which are difficulties with object edge aliasing, problematic alpha blending, and high hardware requirements.

The gain in performance by using pre-calculated reflections compared to true planar reflections was never measured, but the incorporation of window pane reflections did almost not affect the frame rate when compared to using the static diffuse textures accompanying the model. True planar reflections would most definitely have put a big dent in the performance and were never implemented.

7.2.2 Image Quality

The demo application does not look as good as it could, mainly because of less perfect light maps and the lack of normal maps and height maps for the used model. Generated light maps turned out really dark and this was compensated in the shader by adding an ambient term. The generation of light maps using 3ds max was looked into at great length, but without finding a solution to the problem. The tweaking of parameters in 3ds max should in any case be considered a task for which 3D artists have more knowledge and experience, unlike an application programmer.

High dynamic rendering increases the realism for very bright window reflections, facilitating post-process effects such as bloom which simulates the light scattering taking place in the human eye. Tone mapping reduces color values in the rendered image to an interval displayable on a modern day monitors.

Figures 7.3 to 7.5 show current results for outdoor and indoor environments. The city block used for the outdoor environment is courtesy of Anima 3D S.L. (`http://www.anima-g.com/`) and shows a portion of the Gracia neighborhood of Barcelona. The indoor Cloister model was created by Adriano del Fabbro and downloaded from 3D Café (`http://www.3dcafe.com/`).

Figure 7.3: Window reflection showing the Fresnel effect

Figure 7.4: Window reflection with bloom

Figure 7.5: Indoor environment

7.3 Future Work

One important area of interest which was not explored in this project is the full-blown use of high dynamic range rendering. Neither the light maps nor the environment maps for the window reflections are high dynamic range. Bandwidth and on-board memory limits the applications to use of low dynamic range textures. Perhaps two GeForce 7800 GTX cards in an SLI configuration could handle the necessary quantity of textures. HDR textures were simulated by scaling texel color values by a given factor.

The function used to estimate the distance to reflected geometry still is not perfect and the application could benefit from improved distance approximation. Alternatively, one may want to study more thoroughly the possibility of using a forward image warping technique like the one presented by Rui Bastos et al. [49], even though their method has at least two severe drawbacks.

The final implementation runs at approximately 40 FPS with a resolution of 1024x768 on a GeForce 6800 GT, which must be considered to be an

interactive frame rate. People report a 2-3 speed increase on the GeForce 7800 GTX for applications making heavy use of floating point textures. It would therefore be interesting to measure performance on a graphics card of this latest generation, which has a texture cache optimized for 64-bit floating point textures.

The current implementation uses standard backward shadow mapping. Forward shadow mapping is supposed to be more texture cache friendly, and it may be of interest to implement the algorithm and compare performances figures. One may also want to implement physically more correct soft shadows, i.e. shadows that harden on contact between occluder and receiver.

7.4 Conclusion

The broad focus of the thesis has made it difficult to dig deep into every topic and has for some areas rather been an investigation of recent trends and available techniques with relevance to the objectives of the thesis. Emphasis has been given to problems related to deferred shading and window reflections in an architectural walkthrough. For shadow generation and added surface detail, suitable methods have been chosen and implemented.

The thesis has shown the feasibility of using deferred shading on current hardware and has applied high dynamic range rendering with the intent to increase realism. Furthermore, the thesis has explained how one can use environment mapping to simulate true planar reflections and has incorporated relevant image post-processing effects. Finally, a shadow mapping solution has been provided for the future integration of dynamic geometry. The achieved performance should be considered reasonable for the given context.

Appendix A

Tools

This appendix lists the tools used for the implementation part of the thesis.

For the realization of this thesis, the following tools were used for different aspects of the implementation:

- Linux operating system
- GCC (GNU Compiler Collection) for standard C++
- OpenGL 2.0
- SDL (Simple DirectMedia Layer) cross-platform multimedia library
- GLEW OpenGL extension loading library
- Doxygen[1] for documentation of source code
- 3ds max for 3D modeling and lightmap creation

[1] Doxygen is a documentation system for C++, C, Java, among others. It can generate an online documentation in HTML and/or an offline reference manual in LaTeX from a set of documented source files. One can configure doxygen to extract the code structure from undocumented source files. One can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

Bibliography

[1] NVIDIA Corporation. NVIDIA GPU Programming Guide. http://download.nvidia.com/developer/GPU_Programming_Guide/GPU_Programming_Guide.pdf.

[2] *The OpenGL®Shading Language*.

[3] *GPU Gems 2*. Addison-Wesley, 2005.

[4] *ShaderX2: Shader Programming Tips & Tricks with DirectX 9*. Wordware Publishing, Inc., 2004.

[5] Mark Harris. Deferred Shading. http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf.

[6] The OpenGL Architecture Review Board. ARB_color_buffer_float. http://oss.sgi.com/projects/ogl-sample/registry/ARB/color_buffer_float.txt.

[7] The OpenGL Architecture Review Board. ARB_texture_float. http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_float.txt.

[8] The OpenGL Architecture Review Board. ARB_half_float_pixel. http://oss.sgi.com/projects/ogl-sample/registry/ARB/half_float_pixel.txt.

[9] The OpenGL Architecture Review Board. EXT_framebuffer_object. http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt.

[10] The OpenGL Architecture Review Board. ARB_draw_buffers. http://oss.sgi.com/projects/ogl-sample/registry/ARB/draw_buffers.txt.

[11] The OpenGL Architecture Review Board. ARB_texture_non_power_of_two. http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_non_power_of_two.txt.

[12] Pedro V. Sander et al. Explicit Early-Z Culling and Dynamic Flow Control on Graphics Hardware. http://www.ati.com/developer/SIGGRAPH05/ShadingCourse_ATI.pdf.

[13] GPGPU.org Wiki Glossary. http://www.gpgpu.org/w/index.php/Glossary.

[14] *Digital Image Processing.* John Wiley & Sons, 2001.

[15] Matthias Wloka. Fresnel Reflection. http://developer.nvidia.com/attach/6664.

[16] David Blythe. Planar Reflectors. http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node159.html.

[17] *Level of Detail for 3D Graphics.* Morgan Kaufmann, 2003.

[18] David Blythe. Dual-Paraboloid Environment Mapping. http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node187.html.

[19] *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks.* Wordware Publishing, Inc., 2002.

[20] Simon Green et al. High Dynamic Range Rendering on the GeForce 6800. http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_HDR.pdf.

[21] Masaki Kawase. Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless). http://www.daionet.gr.jp/~masa/archives/GDC2003_DSTEAL.ppt.

[22] Adam Lake et al. High Dynamic Range Environment Mapping On Mainstream Graphics Hardware. http://www.gamedev.net/reference/articles/article2208.asp.

[23] *OpenGL®Shading Language*. Addison-Wesley, 2004.

[24] Cem Cebenoyan. Floating-Point Specials on the GPU. http://download.nvidia.com/developer/Papers/2005/FP_Specials/FP_Specials.pdf.

[25] Erik Reinhard et al. Photographic Tone Reproduction for Digital Images. http://www.cs.utah.edu/~reinhard/cdrom/tonemap.pdf.

[26] Erik Reinhard. Parameter Estimation for Photographic Tone Reproduction. http://www.cs.ucf.edu/~reinhard/papers/jgt_reinhard.pdf.

[27] DirectX 9.0 SDK. http://msdn.microsoft.com/directx/sdk/.

[28] Greg Spencer et al. Physically-Based Glare Effects for Digital Images. http://www.cs.utah.edu/~shirley/papers/spencer95.pdf.

[29] Erik Häggmark. Mörkerkörning: Realtidssimulering och visualisering av fordonsbelysning för mörkerkörning i körsimulator. Master's thesis, Linköpings universitet, 2004.

[30] Randima Fernando. GPGPU: General-Purpose Computation on GPUs. download.nvidia.com/developer/presentations/2005/I3D/I3D_05_GPGPU.pdf.

[31] Masaki Kawase. Practical Implementation of High Dynamic Range Rendering. http://www.daionet.gr.jp/~masa/archives/GDC2004/GDC2004_PIoHDRR_EN.ppt.

[32] *Michael Abrash's Graphics Programming Black Book, Special Edition*. Coriolis Group Books, 1997.

[33] *Real-Time Rendering*. A K Peters, 2002.

[34] Technical Introduction to OpenEXR. `http://www.openexr.com/TechnicalIntroduction.pdf`.

[35] Carsten Wenzel. Far Cry and DirectX. `http://download.nvidia.com/developer/presentations/2005/GDC/Direct3D_Day/D3DTutorial08_FarCryAndDX9.pdf`.

[36] *GPU Gems*. Addison-Wesley, 2004.

[37] *Mathematics for 3D Game Programming & Computer Graphics*. Charles River Media, 2004.

[38] Mark J. Kilgard. Improving Shadows and Reflections via the Stencil Buffer. `http://developer.nvidia.com/attach/6641`.

[39] Cass Everitt et al. Hardware Shadow Mapping. `http://developer.nvidia.com/attach/8456`.

[40] Cass Everitt. Projective Texture Mapping. `http://developer.nvidia.com/attach/6549`.

[41] Tobias Martin et al. Anti-aliasing and Continuity with Trapezoidal Shadow Maps. `http://www.comp.nus.edu.sg/~tants/tsm/tsm.pdf`.

[42] Randima Fernando. Percentage-Closer Soft Shadows. `http://download.nvidia.com/developer/presentations/2005/SIGGRAPH/Percentage_Closer_Soft_Shadows.pdf`.

[43] Randima Fernando. Shader Model 3.0 Unleashed. `ftp://download.nvidia.com/developer/presentations/2004/SIGGRAPH/Shader_Model_3_Unleashed.pdf`.

[44] Stefan Brabec et al. Shadow Mapping for Hemispherical and Omnidirectional Light Sources. `http://www.mpi-sb.mpg.de/~brabec/doc/brabec_cgi02.pdf`.

[45] Wolfgang Heidrich. Environment Maps And Their Applications. `http://www.csee.umbc.edu/~olano/s2000c27/envmap.pdf`.

[46] Natalya Tatarchuk. Practical Dynamic Parallax Occlusion Mapping. http://ati.com/developer/SIGGRAPH05/Tatarchuk-ParallaxOcclusionMapping-Sketch-print.pdf.

[47] Simon Green. Bump Map Compression. http://download.nvidia.com/developer/Papers/2004/Bump_Map_Compression/Bump_Map_Compression.pdf.

[48] Terry Welsh. Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces. http://www.infiscape.com/doc/parallax_mapping.pdf.

[49] Rui Bastos et al. Forward Mapped Planar Mirror Reflections. http://www.cs.yorku.ca/~wolfgang/papers/TR98-026.pdf.

Index

4-sample PCF, 64, 68

ambient occlusion, 55
approximate distance, 35

bent normal, 55
bloom, 44
bump mapping, 72
burn-out, 43

ciliary corona, 44
cube environment mapping, 28
cube shadow mapping, 64

deferred shading, 7
depth bias, 61
direct illumination, 53
displacement mapping, 71
dual-paraboloid shadow mapping, 64
dueling frusta, 62
DXT1, 29
dynamic branching, 12
dynamic range, 37

early-z culling, 11

floating point buffers, 8
floating point specials, 42
forward shading, 7
framebuffer objects, 9
Fresnel equations, 20

G-buffer, 7, 9
global illumination, 53
Gram-Schmidt orthogonalization, 73

hemisphere, 67
hemisphere remapping, 74
high dynamic range, 37

image warping, 29
indirect illumination, 53

jittering, 63

key, 40

lenticular halo, 44
light mapping, 54
local illumination, 53
logarithmic average luminance, 40

middle gray, 40

multiple render targets, 9

non-linear parabolic projection, 68
non-linear projection, 70
non-power-of-two textures, 9
normal mapping, 72

occlusion query, 11

p-polarized, 20
paraboloid coordinate, 68
paraboloid environment mapping, 29
parallax mapping, 75
penumbra, 62
percentage closer filtering, 63
percentage-closer soft shadows, 63
ping-ponging, 12
projected planar shadows, 59
projective texture mapping, 61

reflection coefficient, 20
reflection loss, 23

s-polarized, 20
sampling rate, 69
scaling operator, 42
scene key, 42
Shader Model 3.0, 68
shadow acne, 61
shadow volumes, 58
Snell's law, 20
soft shadows, 63
sphere environment mapping, 29
sweet circle, 29

tangent space, 72
target scene key, 42
TBN-matrix, 72
texture lod bias, 25
tone mapping, 40
tone mapping operator, 42
transmission coefficient, 20
trapezoidal shadow mapping, 62
trilinear filtering, 25

UltraShadow, 64
umbra, 62

vertex texture fetches, 41
view-dependent, 29
view-independent, 28, 54

z-only rendering, 64, 68
zone, 40

Copyright

Svenska

English