FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Advanced Shadow Algorithms

Filtered Hard Shadows

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Matthias Adorjan
Matrikelnummer 0927290

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Wien, 15.03.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Advanced Shadow Algorithms

Filtered Hard Shadows

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Matthias Adorjan
Registration Number 0927290

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Vienna, 15.03.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Matthias Adorjan
Primelgasse 3, 7400 Oberwart

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Shadows play a very important role in enhancing realism of rendered scenes. Scenes without them look unnatural and flat. If you look at them you feel like there is something missing. Furthermore it is hard to determine a spatial relationship between the objects in a scene without shadows. In recent years a lot of papers and articles have been published on the topic of rendering realistic shadows in real time. Many of the presented techniques are based on shadow mapping, which has become widely accepted as a method for shadow rendering.

This thesis focuses on giving an overview of commonly used methods to fight aliasing and produce soft-edged shadows in real time by filtering hard shadows created on the basis of shadow mapping. These filtering techniques are integrated into an existing framework, which allows the user to modify different essential parameters to find the best solution regarding shadow creation for a particular scene.

Additionally gamma correction as another realism and image quality improving mechanism is explained. We present a tutorial on what to consider when implementing a gamma correct rendering pipeline in DirectX® 11.

Contents

CHAPTER 1

Introduction

Achieving a high amount of realism or even physically correct results in real time renderings is one big goal of research in the field of computer graphics. Hence lighting and shadows play an essential role because they are important effects for generating realistic scenes. In this thesis we take a closer look at shadow creation in real time.

The term *shadow* is not as easy to define as it seems. There are a lot of more or less precise definitions. The MacMillan Dictionary for example simply states [1]:

Definition 1 *an area of darkness that is created when something blocks light.*

The Oxford English Dictionary gives a more detailed definition [2]:

Definition 2 *a dark area or shape produced by a body coming between rays of light and a surface.*

According to this description shadows have a *shape* produced by occluders and are not just an arbitrary area of darkness. Furthermore it is not just light that is blocked by something but there is explicitely stated that *rays of light* are affected. There surely exist more precise and complete definitions but this one is sufficient for our work.

It's out of the question that shadows are essential for realistic scene renderings. Cast shadows give humans a better overview and perception of the scene. Spatial relations between objects can be derived. Figure 1.1 shows that the understanding of the concerned objects geometry is improved by created shadows too [3].

(a) spatial relation between occluder and receiver **(b)** geometry of the receiver **(c)** geometry of the occluder

Figure 1.1: Shadows provide information about the concerned objects geometry. Image courtesy of Hasenfratz et al. [3]

1.1 Goal definition

The main goal of this thesis is to compare different hard shadow filtering techniques and ways of implementing them. We want to analyze these methods in order to state their strengths and weaknesses in terms of quality and performance. The results are shown in form of rendered images.

To achieve that, we integrated an implementation of the commonly used filtering techniques *Variance Shadow Mapping* (with and without *Summed Area Tables*), *Exponential Shadow Mapping* and as a sort of hybrid solution *Exponential Variance Shadow Mapping* into the existing *Robust Hard Shadows Framework* developed by Martin Stingl during his master's thesis on hard shadows [4]. As a reference for these filtering methods we additionaly realized *Percentage Closer Filtering* with a variable filter kernel size.

In addition we want to state how important *gamma-correction* in the field of computer graphics is. So we implemented it in the framework too. The results can be seen in the appendix.

1.2 Structure

This thesis is built up of different chapters. After the introducing words of this chapter there are following upcoming parts:

Chapter 2: Related Work
This chapter reviews the theory behind the basic shadow mapping algorithm and different filtering techniques.

Chapter 3: Implementation
Here the implementation is described in more detail. The framework in which our code is

integrated is introduced in this chapter and some interesting parts of the code are picked out and explained.

Chapter 4: Results

This part of the thesis shows the results of our implementation. The images of this chapter depict the strengths and weaknesses of the implemented techniques.

Chapter 5: Conclusion

Finally the conclusion sums up the thesis and introduces some suggestions on what can be done in future works regarding the used framework and generally on the topic of shadow rendering.

Appendix A: Gamma Correction

Additionally there is a chapter on gamma-correction at the end of this thesis. Here is explained what we understand under the term gamma-correction and a tutorial is given on how to implement it in DirectX® 11.

CHAPTER 2

Related Work

The nowadays commonly used shadow creation techniques in real time rendering are mainly based on two methods. These methods are called *Shadow Volumes* and *Shadow Mapping*. The first mentioned algorithm was introduced by Crow in 1977. Because this thesis isn't based on shadow volumes we will not go in further detail on this topic. More information can be found in the corresponding paper [5].

The more interesting algorithm in the context of this thesis is shadow mapping. It was firstly described by Williams in 1978 [6]. The following sections state how the technique works and what problems occur when using shadow mapping. Furthermore the theory behind some filtering techniques which deal with one of the problems will be introduced.

2.1 Basic Shadow Mapping

In the aforementioned shadow definitions (cf. definition 1, definition 2) is declared that a shadow is casted on a surface if something blocks at least one light ray on its way to this surface. Instead of a geometric approach like shadow volumes, shadow mapping manages the shadow calculations without any "shadow geometry" creation. It is an image-space technique which consists of two render passes and produces hard shadows:

1. In the first pass the depth map, called *shadow map*, is created (cf. figure 2.1). For this purpose the scene is rendered from the light's point of view and the depth values are stored in the shadow map by using a z-buffer.

2. The second render pass renders the scene from the camera's point of view. Additionally to simply rendering the scene objects a so called depth test is done to determine if a point is in shadow or not. Each point of the scene is transfered from scene space into light space

and its depth value is compared to the corresponding value stored in the previously created shadow map. If its value is greater than the shadow map value the point is considered in shadow, otherwise it is unshadowed.

Figure 2.1: Shadow map consisting of depth values determined from the light's point of view

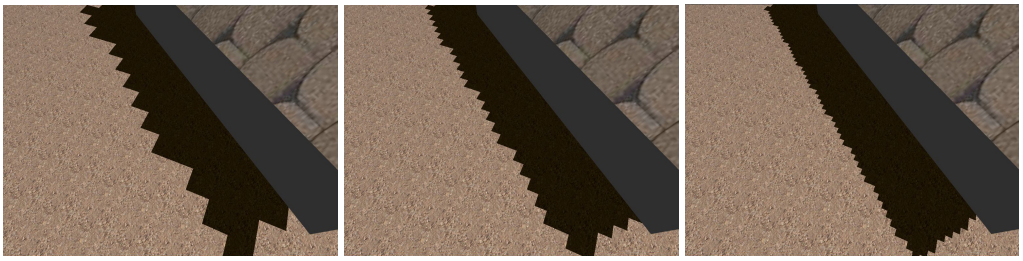
Not needing any geometry calculations is an advantage of that image-space approach. But there are some drawbacks too. Two of the most annoying are *self-shadowing artifacts* and *aliasing artifacts*. The latter occur due to the limited shadow map resolution and will be handled by this thesis. For more information have a look at the paper by Scherzer [7] which discusses the problems in more detail.

2.1.1 Fighting aliasing

As can be seen in figure 2.2 the artifacts can be reduced by higher-resolution shadow maps. Unfortunately we have limited resources regarding the graphics hardware which don't allow us to use infinitely high resoluted shadow maps. Therefore we need another solution for this problem.

Focusing the shadow map as introduced by Brabec et al. [8] allows an almost optimal usage of the shadow map space. Here the light frustum is adjusted to the camera's frustum. Stair-stepping-artifacts are reduced but another artifact is introduced. Flickering at the shadow boundaries occurs when moving the light or the camera. Moreover shadows of shadow casters which lie outside the camera's frustum are not rendered even if they intersect the view frustum.

Stamminger and Drettakis introduced *Perspective Shadow Maps (PSM)* in 2002 [9] and Wimmer et al. introduced *Light Space Perspective Shadow Maps (LiSPM)* as an enhancement of PSM in 2004 [10]. Both methods try to reduce the artifacts by redistributing shadow map samples. They use a perspective projection to warp the light space in order to achieve a redistribution of

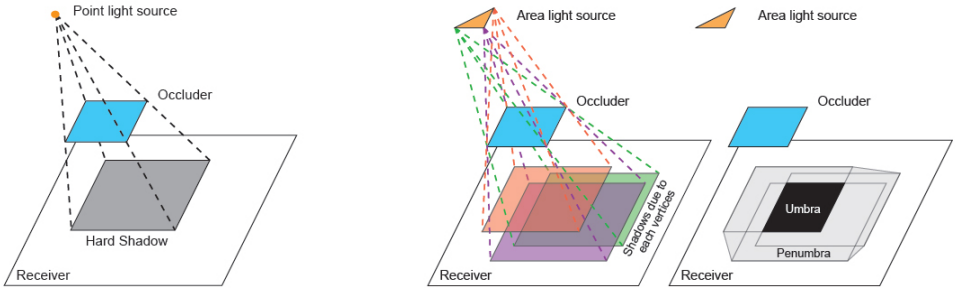


(a) 512x512 **(b)** 1024x1024 **(c)** 2048x2048

Figure 2.2: Aliasing artifacts of shadow mapping with different shadow map resolutions

samples towards the near plane so that nearby objects receive more samples than objects further away. However both have problems if the light direction is nearly parallel to the view direction. This scenario leads to numerical issues when using PSM and LiSPM falls back to a standard uniform shadow mapping applying no warping, which again suffers from aliasing artifacts.

Another strategy to solve this problem is to filter shadow maps. Filtering methods like *Percentage Closer Filtering* by Reeves et al. [11] as a reference reduce the stair-stepping artifacts and as a nice side effect soften the shadow boundaries too. The result appears like soft shadows, but they are of course not physically correct. We still assume a point light as light source and not a light area which is needed for correct soft shadows as can be seen in figure 2.3.



(a) **(b)**

Figure 2.3: Hard Shadows (a) and Soft Shadows (b). Image courtesy of Hasenfratz et al. [3]

This thesis describes three filtering methods to fight aliasing artifacts. The theory behind them is covered in the next section of this chapter.

2.2 Filtered Shadows

The filtering methods we show in this section are *Variance Shadow Maps*, *Exponential Shadow Maps* and *Exponential Variance Shadow Maps*. All of the three are based on a light-space blur of the shadow map in order to remove the jaggy stair-stepping artifacts at the shadow boundaries.

Blurring shadow maps is not so straightforward as it seems. If the shadow map is simply blurred and a standard shadow test as described in section 2.1 is done the result will still have the aliasing artifacts, because the shadow test outcome is still 0 or 1. So the result of the depth test has to be blurred [12].

Instead of $shadowContribution = avg(shadowMapDepth) \geq fragmentDepth$, the equation needs to be $shadowContribution = avg(shadowMapDepth \geq fragmentDepth)$.

Another approach is to blur the shadow map before the depth test and adapt the test to get a meaningful outcome. This is done by the following filtering methods.

2.2.1 Variance Shadow Maps (VSM)

Variance Shadow Maps by Donnelly and Lauritzen [13] allow a pre-blurring of the shadow map by linearizing the depth test. A separable blurring is usually faster than PCF-Filtering [11] in the fragment shader especially for larger filter kernels [14]. This method aims at estimating the PCF result by using simple statistics. Variance shadow mapping is done in three render passes.

The first pass is nearly equal to the basic shadow mapping algorithm. Instead of storing a depth value of a single point in one texel of the shadow map a distribution of depths is stored at each texel. To store a distribution its parameters have to be saved in the shadow map. These are the depth and the squared depth which results in a two-component variance shadow map.

The second pass filters the shadow map with a specific kernel size. After that pass the shadow map contains the first and second moments m_1, m_2 of the depth distribution over the filter region.

$$m_1 = E(x) = \int_{-\infty}^{\infty} x f(x) dx \tag{2.1}$$

$$m_2 = E(x^2) = \int_{-\infty}^{\infty} x^2 f(x) dx \tag{2.2}$$

The third pass contains the actual depth test. From the two moments stored in the depth map the mean μ and variance σ^2 of the distribution can be computed as follows.

$$\mu = E(x) = m_1 \tag{2.3}$$

$$\sigma^2 = E(x^2) = m_2 - m_1^2 \tag{2.4}$$

To determine the probability that an arbitrary point at depth *t* is shadowed Chebyshev's inequality is applied.

$$P(x \geq t) \leq p_{max}(t) = \frac{\sigma^2}{\sigma^2 + (t - \mu)^2} \tag{2.5}$$

$P(x \geq t)$ represents the percental amount of depth values inside the filter region which are not closer to the light source than an arbitrary point with depth value *t*. This is exactly the value of the shadow factor PCF computes. $p_{max}(t)$ is an upper bound to that percentage.

The inequality is valid for $t > \mu$. So the shadow factor of VSM is p_{max} if $t > \mu$, otherwise it is 1 and the surface at depth *t* is fully lit.

If the receiver and occluder are planar and perpendicular to the light source the inequality becomes an equality and $P(x \geq t) = p_{max}(t)$ is valid. This leads to the same result as computed by PCF but with less computation time. In a small neighbourhood around the shaded point (with depth *t*) this planarity is mostly given and therefore $p_{max}(t)$ is a very good approximation of the real shadow factor $P(x \geq t)$. [15]

Because the depth representation of VSM is linear and the shadow map linearly filterable, mipmapping, trilinear filtering, anisotropic filtering, and so on can be used.

Light-bleeding problem

One big disadvantage though is light-bleeding which can occur when using VSM. Figure 2.4 illustrates the problem.

As can be seen light-bleeding occurs if there is more than one occluder between the light source and the shadow receiver. Then a soft shadow is visible on a first receiver and on a second receiver, which should be in complete darkness, because it is fully occluded by the first receiver. The artifact is stronger the larger the ratio of the relative distances Δa to Δb is.

That is due to the fact that VSM calculates the mean and variance of the filter region's depth values just by using the values of the objects nearest to the light source (cf. 2.4 object 1 and 2). The second receiver (object 3) is ignored in this calculation, because fully occluded by object 1 and 2. So there is no information about object 3 in the shadow map and therefore the depth value of object 3 cannot be used to calculate a correct variance when shadowing its surface. The calculated variance σ^2 is greater than 0 in the penumbra region at object 2. The same σ^2 is used when creating the shadow in that region on object 3, which actually causes the light-bleeding, because $\sigma^2 \neq 0$ means that the surface is at least partially lit. [15]

A solution to this problem is using *Layered Variance Shadow Maps* [15], which segments the scene so that in each layer there is only one shadow caster and one receiver. This is very per-

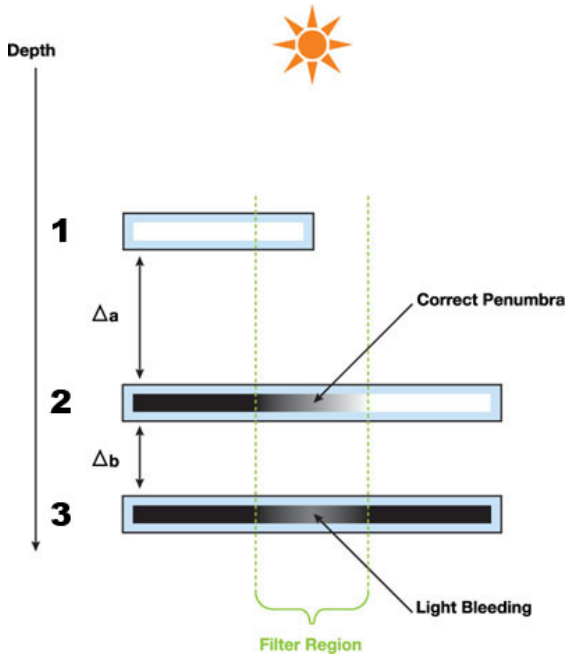


Figure 2.4: Light bleeding problem. Image courtesy of Andrew Lauritzen [14]

formance intensitive and therefore not really recommendable. Another approach is to simply adapt p_{max} of the Chebyshev's inequality (cf. equation 2.5). Intensities below a minimum value are mapped to 0 and the remaining are rescaled to fit in an range from 0 to 1. This results in reduced light-bleeding, but also in darker penumbra regions and loss of shadow details. An implementation of that adaption can be seen in chapter 3.

Summed-Area Variance Shadow Maps (SAVSM)

Lauritzen found out that summed-area tables introduced by Crow in 1984 [16] can be used to filter Variance Shadow Maps [14]. A summed-area table is an array or texture in which each element stores the sum of all values between the sample position and the bottom or top (depends on position of the origin) left corner of the texture. The formula needed to build up a summed-area table (SAT) is shown in equation 2.6. $s(x, y)$ describes the SAT and $t(u, v)$ the original texture.

$$s(x, y) = \sum_{u=0}^{x} \sum_{v=0}^{y} t(u, v) \tag{2.6}$$

It allows us to calculate any sum over an arbitrary sized rectangular area in constant time *O(1)* with the help of formula 2.7. Further the average can be computed easily by dividing the sum by the rectangle size.

$$sum = s(x_{max}, y_{max}) - s(x_{max}, y_{min}) - s(x_{min}, y_{max}) + s(x_{min}, y_{min}) \tag{2.7}$$

Instead of using hardware filtering like in the original VSM method the filtering is done manually via the summed-area table. Due to the fact that averages in SATs can be computed in constant time the performance of SAVSM is independent of the filter width. The filter width can also be adjusted for each pixel, which makes it possible to create more realistic soft shadows.

Lauritzen shows that standard VSM is still better for static filter widths. For dynamic widths SAVSM should be used [14]. It has a higher setup cost than PCF because of the SAT calculation, but with increasing filter widths its performance is far better than that of PCF (cf. section 4.1.2).

Although SAVSM delivers nice results it has some drawbacks too. Especially the need for high precision is a problem even on latest GPUs. Summed area table consume $log(width * height)$ precision, which would be 20 bits when using shadow maps with a size of 1024x1024. So there are finally only 3 bits left for the data itself, which is insufficient and causes an inaccurate result when sampling the SAT. The situation can be improved by distributing the precision over four components as described by Donnelly and Lauritzen [13]. Then a four component shadow map is needed instead of a two component map, which increases the memory usage. Another option would be to use a 32-bit-integer texture to store the shadow map. This allows us to use additionally 8 bit of the exponent part of the 32-bit-floating-point for data storage. A far better solution would be double-precision floating-point formats. However, since double-precision calculations on current GPUs are slower than computations with integer or floating point formats, 32-bit-integers should be used.

2.2.2 Exponential Shadow Maps (ESM)

Exponential Shadow Maps introduced by Annen et al. [17] and Salvi [18] try to approximate the original depth test by expanding it by an exponential term. This expansion is applied in order to linearize the shadow test, which allows pre-blurring and the usage of hardware-side filtering methods like mentioned in the VSM section 2.2.1.

ESM is based on the assumption that the shadow map represents the depth values of the nearest surface to the light source. That means each visible point has at least the depth value of its corresponding entry in the shadow map. Expressed in a formula that looks like this

$$d(\mathbf{x}) - z(\mathbf{p}) \geq 0 \tag{2.8}$$

$\mathbf{x}$ is here defined as a position in world space. $\mathbf{p}$ is the corresponding position in shadow map space. Therefore $d(\mathbf{x})$ is the distance from the light source to $\mathbf{x}$ and $z(\mathbf{p})$ the distance from the light source to the nearest blocker along the direction from $\mathbf{x}$ to the light source.

Following this assumption the shadow test is then defined as

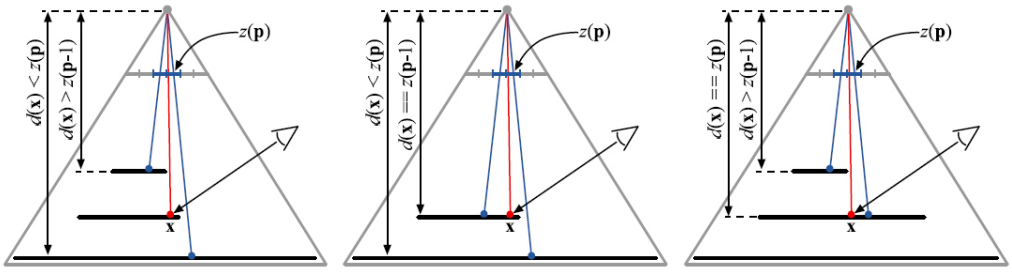
$$f(d, z) = e^{-c(d-z)} = e^{-cd}e^{cz} \tag{2.9}$$

where c is a positive constant. It can be seen, that the terms for the shadow receiver e^{-cd} and the shadow caster e^{cz} are separated. That allows us to handle them independently of each other.

In the first render pass the shadow caster values e^{cz} are rendered into a shadow map. They can then be filtered/blurred in a second pass by convolving them with a filter kernel w. The final pass includes the scene rendering and the depth test. To get the shadow factor the filtered values of the depth map just have to be multiplied by the shadow term of the shadow receiver. This finally results in the following calculation

$$shadowContribution(\mathbf{x}) = e^{-cd(\mathbf{x})}(w * e^{cz})(\mathbf{p}) \tag{2.10}$$

One advantage of ESM is that there is no light-bleeding like in VSM, due to the fact that no variances are used anymore. But it suffers from another form of light-bleeding. It occurs at the contact points of the shadow caster and the shadow receiver. Here the term $d - z$ tends to 0 and so $e^{-c(d-z)}$ tends to 1, which means the pixel is unshadowed. To reduce this a higher value for constant c can be chosen. The higher it is the later the exponent becomes 0. Because we have limited resources regarding memory and precision c cannot get as high as sometimes needed. It is shown that a value greater than 88 results in visible precision problems. To bypass this problem just the linear depth value can be saved in the shadow map and the filtering of the second render pass can be done in logarithmic space as described in [19]. This allows higher values for c, but mipmapping and hardware filtering methods wouldn't be correct because they need to be done in logarithmic space too.



(a) failure case 1 **(b)** failure case 2 **(c)** correct case

Figure 2.5: ESM filtering cases. Image courtesy of Annen et al. [17]

Another drawback of ESM is that its assumption (cf. equation 2.8) is not always right in practice as shown by Annen et al. [17]. It is only true, if the shadow receiver is over the region of the

filter kernel planar and orthogonally aligned to the light source and no depth discontinuities are present. Otherwise it is possible that a depth value of the shadow caster is higher than that of a shadow receiver. Two scenarios where this is the case are depicted in figure 2.5a and figure 2.5b. The red line/dot always shows the point observed by the camera, while the blue lines/dots represent shadow map samples. Figure 2.5c follows the assumption and is therefore handled correctly by ESM filtering.

If the assumption is not valid ESM has to fall back to PCF-style filtering. To detect the failure cases Annen et al. introduced two methods: *Z-Max classification* and *Threshold classification*. The first approach needs an extra texture to store maximum depth values of a given neighbourhood. A failure case is detected by comparing $d(\mathbf{x})$ with the stored z_{max}. The threshold classification checks if the calculated ESM-shadow-factor exceeds $1 + \epsilon$, where ϵ is the threshold. If this is the case a fallback to PCF needs to be done.

Z-max classification leads to more accurate results but due to the extra texture map and sampling overhead it is not as fast as threshold classification. The latter can introduce some artifacts, but these decrease with increasing shadow map size. Because of its better performance and at currently used shadow map sizes not noticable quality issues, threshold classification should be favored.

2.2.3 Exponential Variance Shadow Maps (EVSM)

Exponential Variance Shadow Maps are a combination of VSM and ESM. It was introduced by Lauritzen [15] in order to solve the light-bleeding issues of both VSM and ESM.

This approach uses a four-component shadow map. Instead of just storing the depth and squared depth in the shadow map as in VSM, the EVSM method stores those values warped by the exponential function e^{cz}. The third and fourth component store the depth values scaled by the negative exponential function $-e^{-cz}$:

The warp with the positive exponential function reduces the relative distance between overlaying shadow casters and increases the distance between the shadow receiver and its nearest shadow caster. This has the effect that the ratio Δa to Δb (cf. figure 2.4) decreases and therefore the VSM light-bleeding vanishes if *c* is chosen high enough.

The negative exponential function helps to fight the nonplanarity problem of ESM by decreasing the distance between the shadow receiver and its nearest shadow caster.

The filtering in the second render pass is done as in the above mentioned methods, resulting in the positive moments m_1^+ and m_2^+ in the first two components of the shadow map and the negative moments m_1^- and m_2^- in the last two components.

During the shadow test Chebyshev's inequality (cf. equation 2.5) is evaluated for the positive and negative moments. This results in two shadow factors p_{max}^- and p_{max}^+. The minimum of

these two values is finally used as the shadow factor of EVSM.

This method produces shadows with superior quality compared to VSM and ESM. However it still has precision issues when high values are used for the constant c.

CHAPTER 3

Implementation

3.1 Overview

During the work on this thesis we implemented the filtering techniques explained in section 2.2. The programming language used was Visual C++ 10.0 together with the DirectX® 11 framework and HLSL Shader Model 5 for shader development.

We realized Percentage Closer Filtering (PCF) with a variable filter kernel size as a reference for the implemented filtering methods. Here hardware PCF sampling, which uses bilinear filtering, and simple BoxPCF without bilinear filtering are realized.

To hide aliasing artifacts we blur the VSM, ESM and EVSM shadow map by using a gaussian or box filter with a constant filter size. To handle bandlimiting for the minification case mipmapping for VSM, ESM and EVSM and an adaptive box filter kernel for the summed-area-table approaches and the reference PCF method is implemented. The variable filter kernel size depends on the calculated mip-level and the kernel size of the separate blurring stage. The maximum number of samples of the reference PCF method is limited by a configurable value. This needs to be done because the filter size and further the number of samples are calculated from the used mip-level ($filtersize = 2^{miplevel}$). A low level can lead to a very high filter size (large amount of samples), which may crash some GPU drivers. Details on the implementation of the filter size calculation can be found in section 3.3.3.

To avoid precision problems we realized some simple solutions too. For ESM we implemented filtering in logarithmic space in addition to filtering in linear space. This allows the usage of higher values for the constant c which is also known as overdarkening factor. The summed-area table VSM approach as another precision hungry method is implemented like standard VSM with a two-component shadow map. This method results in very ugly artifacts due to precision issues. Therefore we distributed the precision over four components which leads to a higher

quality output. Because this method has still artifacts a third approach was implemented using a 32-bit-integer shadow map texture, which results in a nearly precision-artifact free rendering as can be seen in chapter 4.

A light-bleeding reduction constant (lbr-constant) was introduced to reduce light-bleeding of VSM and SAVSM methods (see section 3.3.1). Threshold classification as described in section 2.2.2 was implemented to fight artifacts of ESM. Regions where the ESM calculation exceeds a specified threshold are filtered in BoxPCF-style with a 2x2 filter kernel size.

Furthermore a possibility to visualize the mip-levels and regions where ESM falls back to PCF is given.

All the parameters like filter kernel size, lbr-constant-value, overdarkening constants, logspace/linspace filtering, etc. can be modified via a user-friendly Graphical User Interface (cf. figure 3.1).

(a) Reference PCF **(b)** VSM **(c)** SAVSM

(d) ESM **(e)** EVSM

Figure 3.1: Graphical User Interface to adjust shadow mapping parameters

3.2 The Framework

The *Robust Hard Shadows* framework by Martin Stingl [4] is the basis of our implementation. During our work we migrated the framework from DirectX® 10 and Visual C++ 9.0 to DirectX® 11 and Visual C++ 10.0. The Graphical User Interface rendering, window and input handling is accomplished with the help of the DXUT framework, which is shipped with the DirectX® SDK [20].

The framework is built up of dynamically linked libraries which are used by the main application. Different parts of the framework like the model loader or scenegraph representation are in separate libraries and can be exchanged easily. This modular structure guarantees a high level of flexibility and allows even to switch to another viewer application with minimal effort. The modules of the framework are the following:

- Common
- D3dRenderSystem
- GraphicsEngine
- PluginCollada
- SceneGraph
- Viewer

The framework already included the basic shadow mapping implementation, several PCF variants and advanced shadow mapping methods like focusing, warping and z-partitioning via Parallel Split Shadow Maps (PSSM) [21].

Apart from the migration to VC++ 10.0 and DirectX® 11, our implementation mainly affected the D3dRenderSystem module, the Viewer module and of course the shader files which belong to the framework. In the D3dRenderSystem module the DirectX® implementation of the filtering techniques is done. The changes of the Viewer module include user interface extensions.

For more information on the framework see Stingls master's thesis [4].

3.3 Implementation details

In this section we go into more detail and explain some selected code snippets.

3.3.1 VSM light-bleeding reduction

As already mentioned in section 2.2.1 Variance Shadow Maps suffer from light-bleeding. According to Lauritzen this artifact can be reduced by adapting the p_{max} value of Chebyshev's inequality [14]. Values below a defined minimum intensity are mapped to 0 and the remaining intensities are rescaled to fit in a range from 0 to 1. An implementation of the mapping can be seen in listing 3.1. Here *p* is the p_{max} value of the inequality, *lbramount* is the defined minimum intensity. If *p* is below *lbramount* it is mapped to 0. If it is between *lbramount* and 1 a smooth Hermite interpolation is applied and the interpolated value returned [22]. Otherwise the return value is 1. This method cuts off the lower tail of the Chebyshev's inequality, which results in darker penumbrae and loss of shadow details, but also reduces the light-bleeding. A higher *lbramount* value results in a more aggressive light-bleeding reduction. Scenes with more light-bleeding usually need a higher value for *lbramount*. This value can be adjusted via the Graphical User Interface. It is called *light-bleeding reduction constant*.

```
1 float ReduceLightBleeding(float p, float lbramount)
2 {
3     return smoothstep(lbramount, 1.0f, p);
4 }
```

Listing 3.1: HLSL code to reduce VSM light bleeding

3.3.2 Filtering in logarithmic and linear space

Section 2.2.2 already stated that there are two approaches to implement ESM. The first one is to simply write $e^{c*depth}$ to the shadow map and filter it linearly. When doing the depth test equation 3.1 has to be evaluated. *occluder* is the depth value from the shadow map and *receiver* the depth value of the currently processed fragment.

$$shadowContribution = \frac{occluder}{e^{(c*receiver)}} \tag{3.1}$$

The second approach writes $c * depth$ to the shadow map. Filtering then needs to be done in logarithmic space. At the depth test stage equation 3.2 must be solved.

$$shadowContribution = e^{(occluder - c*receiver)} \tag{3.2}$$

The second approach uses less precision for storing the depth values, which allows higher values for the overdarkening factor c. Method 1 limits c to 88, when a 32 bit floating point texture is

used [19]. Values above lead to overflow errors. This can be too low especially when large outdoor scenes are rendered. Light-leaking artifacts may appear if c is chosen too low.

Listing 3.2 shows the functions used to filter in linear (*lin_space*) and logarithmic space (*log_space*). *w0* and *w1* are the weights applied to the filtered depth values *d0* and *d1*.

```
1 float4 log_space(float w0, float4 d0, float w1, float4 d1)
2 {
3   return (d0 + log(w0 + (w1 * exp(d1 - d0))));
4 }
5
6 float4 lin_space(float w0, float4 d0, float w1, float4 d1)
7 {
8   return (w0 * d0 + w1 * d1);
9 }
```

Listing 3.2: HLSL code for logarithmic and linear filtering

We implemented a separable box filter and a separable gaussian filter. Both filters can be used to filter in linear space or in logarithmic space. These filters are used by VSM, ESM and EVSM. Which of the two filtering methods is used can be configured via the Graphical User Interface. The filtering is implemented in two render passes. The first pass blurs the shadow map in horizontal direction while the second pass filters vertically. The only difference between the two implementations are the weights which are passed to the methods shown in listing 3.2. While the weights of the box filter are uniform the gaussian weights are calculated on the CPU and passed to the corresponding blur shader. The reference PCF implementation and the summed-area table approaches use a simple box filter.

3.3.3 Manual level-of-detail calculation for filter-size computation

In order to combine our separate blur filter with mipmapping we have to manually calculate the appropriate mip-level. Lines 5-8 of listing 3.3 calculate the mip-level according to [23]. The two functions in line 5 and 6 of listing 3.3 compute the partial derivatives of the given texture coordinates with respect to the screen-space x- and y-coordinate.

We blur the mip-level 0 of the shadow map texture with the help of a separable gaussian blur filter. This filtering moves the ”starting” mip-level for the actual mipmapping calculation. For example, if the filter size of the blurring is 2x2 the texture saved in mip-level 0 is already in mip-level 1. Generally it can be said that blurring a texture with kernel size x moves it from its original mip-level 0 to mip-level $log_2(x)$. Therefore, after blurring, mip-level 0 of the original mipmap-chain saves mip-level $log_2(x)$ of the corresponding texture. If the calculated mip-level is below[1] $log_2(x)$ further mip-levels are needed, otherwise the blurred mip-level 0 is used. In practice, this means that lower mipmap levels are needed only in more distant regions.

[1] In this context ”below” means that the actually calculated value is greater than $log_2(x)$. Mip-level 0 is the highest level; 1, 2, 3, etc. are lower levels

```
1  float GetMipLevel(float2 uv, int filtersize, float texsize)
2  {
3    float maxmiplevel = round(log2(max(1,filtersize)));
4
5    float2 dx = ddx(uv * texsize);
6    float2 dy = ddy(uv * texsize);
7    float d = max(dot(dx,dx), dot(dy,dy));
8    float miplevel = 0.5f * log2(d);
9
10   if(miplevel <= maxmiplevel)
11     miplevel = 0;
12
13   return miplevel;
14 }
```

Listing 3.3: HLSL code for manually calculating the level-of-detail

3.3.4 Summed area table generation

We implemented our summed-area table (SAT) generation according to the paper by Hensley et al. [24]. The technique is called *recursive doubling* and allows to construct a SAT in *O(log(n))* instead of *O(n)* when using a line-by-line approach.

The algorithm works in a horizontal and a vertical phase. The horizontal phase (line 10-22 of listing 3.4) sums up the values along the rows of the input texture. The vertical phase (line 25-32 of listing 3.4) accumulates along the columns. The variables *n* and *m* defined in line 5-6 contain the number of render passes used for the horizontal/vertical phase.

At the beginning the render target *src* holds the input texture from which a SAT should be created. In each pass *_samplesPerPassSat* texels are read from the input texture, summed up and written to the output texture. The offset between the read texels is defined as $\_samplesPerPassSat^i$, where *i* is the current render pass counter. After each pass input and output textures are swapped (ping pong rendering; line 21/36) - the first pass reads from *src* and writes to *dest*, the second reads from *dest* and writes to *src*, and so on. This is done until all texels in horizontal direction are summed up. After that the vertical passes proceed in the same way.

When the horizontal and vertical summing is finished the *src* render target contains the generated summed-area table.

Listing 3.5 shows the fragment shader code of the vertical render pass. The source texture needs to be point sampled with a black border around it to prevent errors when summing up texels at the image boundaries. The if-statement in line 12-13 is just a hacky workaround for the SAT generation of PSSM shadow maps. *g_texBounds* contains the texture space borders of the corresponding texture in the PSSM texture atlas. To avoid summing up across the borders in the texture atlas it is checked if the current texture coordinates are within the corresponding textures boundaries.

```
void D3dShadowMapping::CreateSAT(D3dRenderTarget* src, D3dRenderTarget* dest,
     const string technique, const unsigned int split)
{
  ID3DX11EffectShaderResourceVariable* depthMap = _genSATEffect.map;

  int n = log(_shadowMapSz)/log((float)_samplesPerPassSat);
  int m = log(_shadowMapSz)/log((float)_samplesPerPassSat);
  int pass = _nbrOfSplits > 1 ? 2 : 0;

  //horizontal phase
  _renderSystem->GetEffectManager()->Use(_genSATEffect,technique, pass);
  for(int i = 0; i < n; i++)
  {
    dest->Bind(*_renderSystem);
    _genSATEffect.offset->SetInt(pow((float)_samplesPerPassSat,i));
    src->Bind(*_renderSystem, *depthMap);
    _renderSystem->Draw(*_viewports[split]);
    src->UnBind(*_renderSystem, *depthMap);
    dest->UnBind(*_renderSystem);

    //ping pong
    std::swap(dest, src);
  }

  //vertical phase
  _renderSystem->GetEffectManager()->Use(_genSATEffect,technique, pass+1);
  for(int i = 0; i < m; i++)
  {
    dest->Bind(*_renderSystem);
    _genSATEffect.offset->SetInt(pow((float)_samplesPerPassSat,i));
    src->Bind(*_renderSystem, *depthMap);
    _renderSystem->Draw(*_viewports[split]);
    src->UnBind(*_renderSystem, *depthMap);
    dest->UnBind(*_renderSystem);

    //ping pong
    std::swap(dest, src);
  }
}
```

Listing 3.4: DirectX code for SAT generation

```
float2 GenerateSATY_PS(GenSAT_PSIn input) : SV_Target
{
  float2 s = input.tex;
  float2 offset = float2(0.0f, 1.0/ float(g_texHeight) * float(g_offset));

  float2 sum = 0;
  for(int i=0;i<g_samples;i++)
  {
    float2 off = s + i * offset;
    float2 sample = map.Sample(SampPointBorder, off).rg;
```

```
11
12      if(off.x >= g_texBounds.x && off.y >= g_texBounds.y &&
13         off.x < g_texBounds.z && off.y < g_texBounds.w)
14      {
15        sum += sample;
16      }
17    }
18
19    return sum;
20  }
```

Listing 3.5: HLSL code for SAT generation (vertical pass)

3.3.5 Summed-Area VSM precision distribution

The following listing 3.6 shows a method to distribute/recombine the precision of the VSM moments into/from four components. This was suggested by Donnelly and Lauritzen [13] to reduce overflow errors when using Summed-Area VSM.

DistrFactor defines where the values are split. We used 256.0f, which means that the value is finally split after 8 bits. The function *modf* in line 6 splits the moments into fractional and integer parts.

```
1  float4 DistributePrecision(float2 moments)
2  {
3    float FactorInv = 1 / DistrFactor;  //DistrFactor = 256.0f
4
5    float2 IntPart;
6    float2 FracPart = modf(moments * DistrFactor, IntPart);
7
8    return float4(IntPart * FactorInv, FracPart);
9  }
10
11 float2 RecombinePrecision(float4 moments)
12 {
13   float FactorInv = 1 / DistrFactor;
14   return (moments.zw * FactorInv + moments.xy);
15 }
```

Listing 3.6: HLSL code for precision distribution/recombination

CHAPTER 4

Results

In this chapter we show the results of the implemented filtering techniques. We compare the outputs in terms of quality and performance. The effect of different parameter values (ESM constant, light bleeding reduction parameter, filter kernel size,...) on the rendering result is depicted on the following few pages.

4.1 Performance tests

4.1.1 Test setup

We made our performance tests on a system composed of an AMD Phenom II X6 1055T processor, 16 GiB RAM and an AMD Radeon HD 6770 GPU with 1 GiB video memory.

Figure 4.1 depicts our test scene. It is made up of approximately 10800 triangles. The measurements are based on a viewport size of 1024x768 and a shadow map resolution of 1024x1024. The Eiffel Tower model increases the scene complexity significantly. The shadows casted by this tower were the most challenging ones. It was very hard to render them without any errors as can be seen in section 4.2.

4.1.2 Benchmarking results

Table 4.1 gives information on the benchmark results taken from our test scene.

It can be seen that filtering techniques with lower memory consumption like ESM/VSM produce higher framerates than the one with higher usage like the four-component shadow map method EVSM.

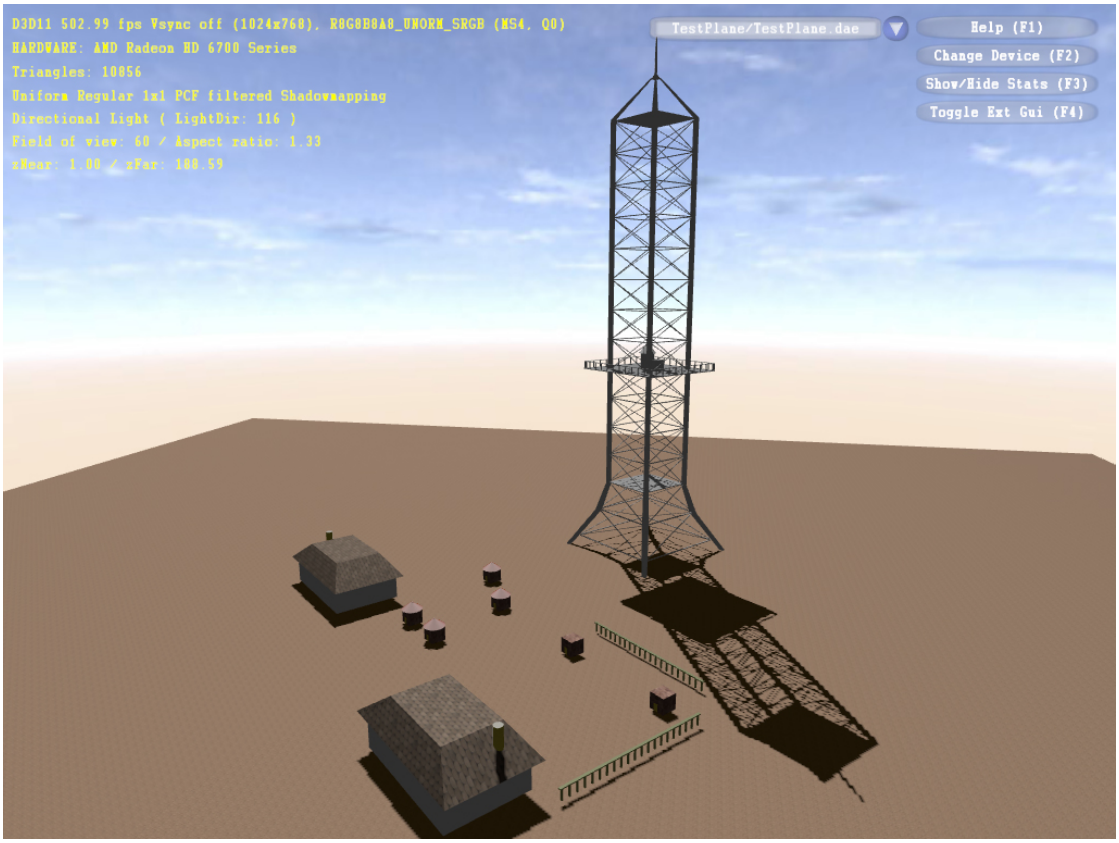


Figure 4.1: Test scene with Eiffel Tower

Furthermore the results show that the frames drop with increasing filter size when not using summed-area tables. This is due to the fact that a greater filter kernel needs more shadow map samples when blurring. Summed-area tables just need 4 (16 if bilinear filtering is active) samples independent of the filter size. Therefore the framerate stays constant over all kernel sizes.

The separable blur pass of ESM/VSM/EVSM is more efficient than the PCF-style filtering as can be seen on the framerates. Unfortunately this separate blurring doesn't allow variable filter sizes per pixel. If this is needed summed-area tables should be the first choice if high filter kernel sizes are considered, because the framerate of PCF filtering drops very fast with increasing kernel sizes as the results show.

Additionally we have compared the filtering techniques with activated mipmapping for ESM/VSM/EVSM respectively variable filter sizes for PCF/SAVSM (implementation details see chapter 3). The summed-area table and PCF methods use variable filter sizes derived from the calculated mip-level. The framerates of ESM/VSM/EVSM are lower because the mipmaps have to be generated at each frame. For stronger minifications greater filter kernels are needed which lowers the framerate of PCF significantly. The SAT methods have a constant framerate because,

as already mentioned, the number of samples is independent of the filter size.

Shadow map lookups are done bilinearly. For ESM/VSM/EVSM and PCF this is achieved via the hardware. For the summed-area table approaches we implemented the bilinear filtering in the shader code. To be able to compare the results a box filter is used for shadow map filtering, because it is implemented for all techniques.

Filtering technique	Filter kernel size				
	1x1	3x3	9x9	15x15	33x33
VSM	607 fps	401 fps	253 fps	184 fps	99 fps
+ mipmapping	486 fps	341 fps	227 fps	170 fps	96 fps
ESM	604 fps	395 fps	253 fps	184 fps	100 fps
+ mipmapping	508 fps	349 fps	233 fps	173 fps	97 fps
EVSM	493 fps	327 fps	213 fps	157 fps	85 fps
+ mipmapping	373 fps	271 fps	187 fps	143 fps	81 fps
Reference PCF	502 fps	357 fps	106 fps	45 fps	9 fps
+ var. filter size	133 fps	124 fps	72 fps	39 fps	9 fps
SAVSM (int32)	84 fps	84 fps	84 fps	84 fps	84 fps
+ var. filter size	84 fps	84 fps	84 fps	84 fps	84 fps
SAVSM (fp32)	86 fps	86 fps	86 fps	86 fps	86 fps
+ var. filter size	86 fps	86 fps	86 fps	86 fps	86 fps
SAVSM (distr. fp32)	58 fps	58 fps	58 fps	58 fps	58 fps
+ var. filter size	58 fps	58 fps	58 fps	58 fps	58 fps

Table 4.1: Benchmarking results of different filtering techniques with and without mipmapping/variable filter size and bilinear shadow map lookups

4.2 Rendering results

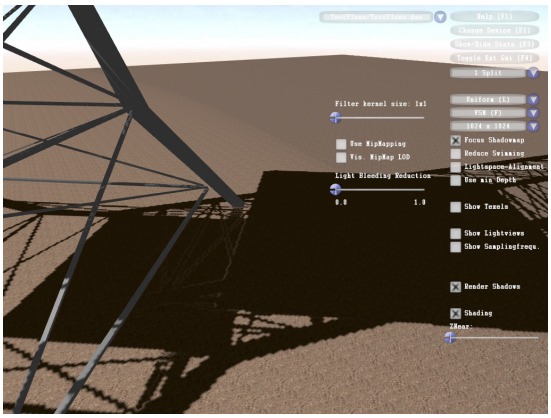
4.2.1 Variance Shadow Maps

Figure 4.2 depicts casted shadows created by VSM-filtered hard shadows with different filter kernel sizes. It can be seen that the light-bleeding problem increases with the kernel size. To reduce this light-bleeding we introduced a reduction constant *lbr* (see section 3.3.1). With its help the artifacts can be reduced, but the shadows and penumbrae get darker as figure 4.3 shows.

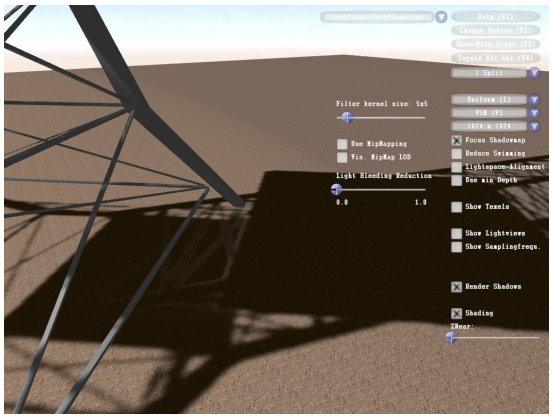
To reduce minification artifacts we implemented mipmapping. Figure 4.4 shows the effect of activated mipmapping on the cast shadows. Regions where the standard blurring isn't sufficient to reduce minification artifacts use a lower mip-level in order to achieve that. Which mip-level is eventually used can be seen in figure 4.5. Here the mip-levels are visualized with a color overlay. The red color indicates that the highest mip-level (0) is used. The colors yellow, green and blue present lower mip-levels. With increased filter kernel size the need for smaller mip-levels gets lower, because the blurring is already enough to eliminate the artifacts. Figure 4.5b

depicts that fact. Here most of the area uses the blurred mip-level 0, only the region far away from the shadow caster samples from a lower mip-level.

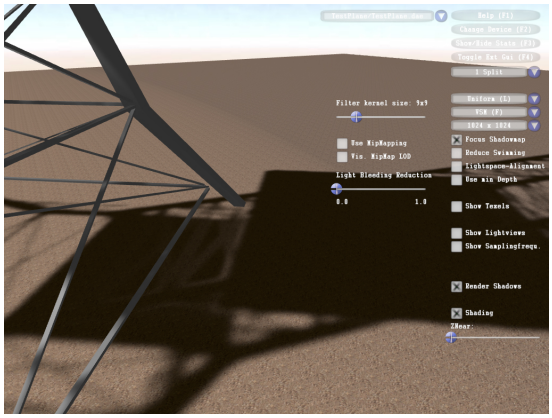
Figures 4.6 and 4.7 show results from our Summed-Area VSM implementation. They present the precision problems of current hardware when creating/sampling from large summed-area tables. Distributing the VSM moments over four components is sufficient for a shadow map size of 512x512. With a size of 1024x1024 there are still precision artifacts visible, even when distributing the precision. Here a 32 bit integer texture should be used for achieving sophisticated results.



(a) Gaussian filter kernel size 1x1

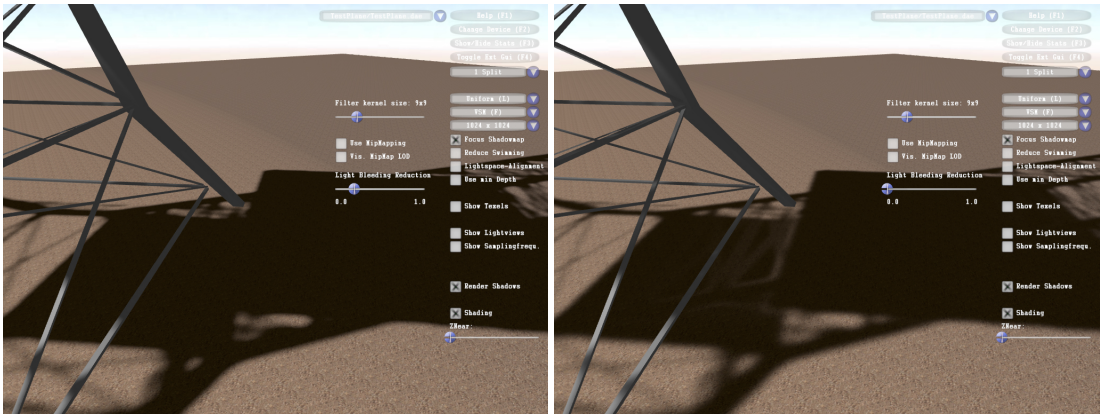


(b) Gaussian filter kernel size 5x5



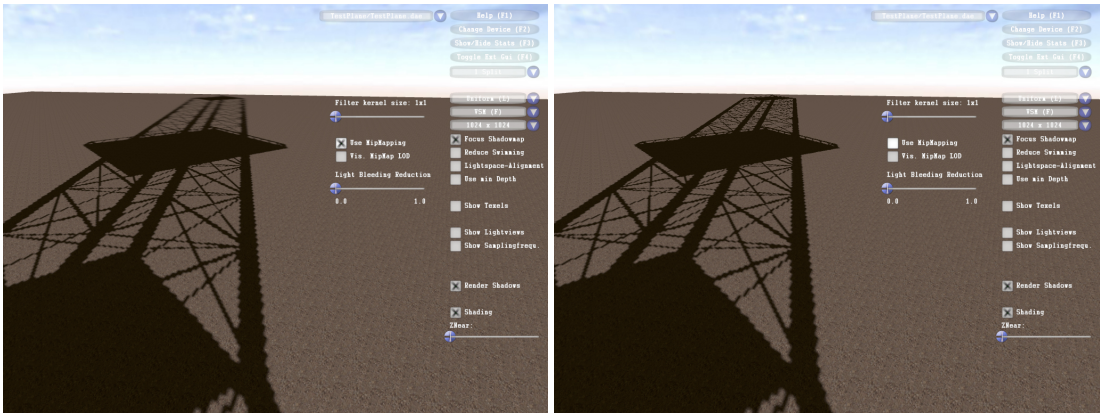
(c) Gaussian filter kernel size 9x9

Figure 4.2: Variance Shadow Mapping with different filter kernel sizes



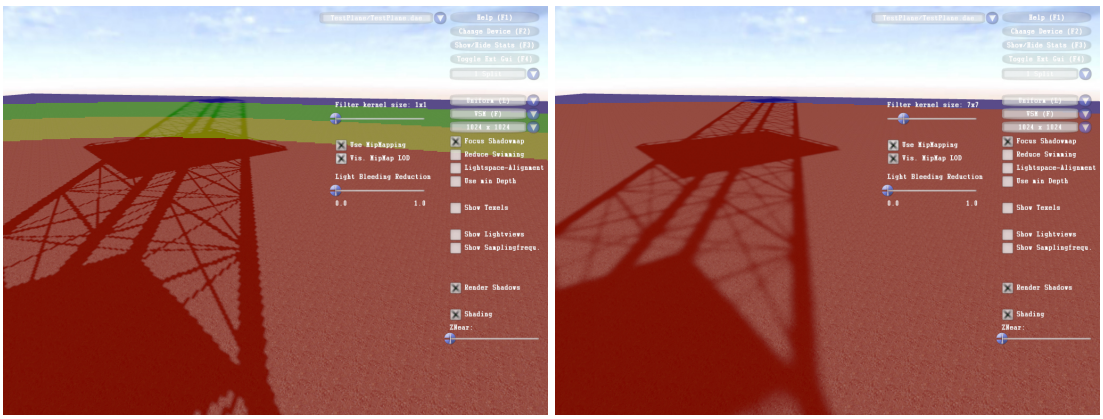
(a) light-bleeding reduction: $lbr = 0.2$ **(b)** no light-bleeding reduction

Figure 4.3: Variance Shadow Mapping with and without light bleeding reduction



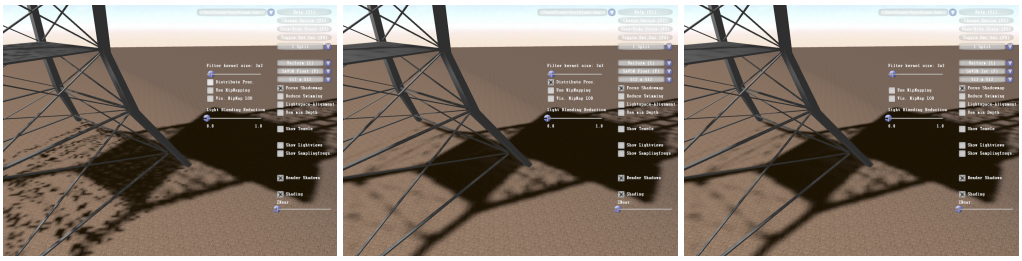
(a) MipMapping On **(b)** MipMapping Off

Figure 4.4: Variance Shadow Mapping with activated MipMapping and without MipMapping



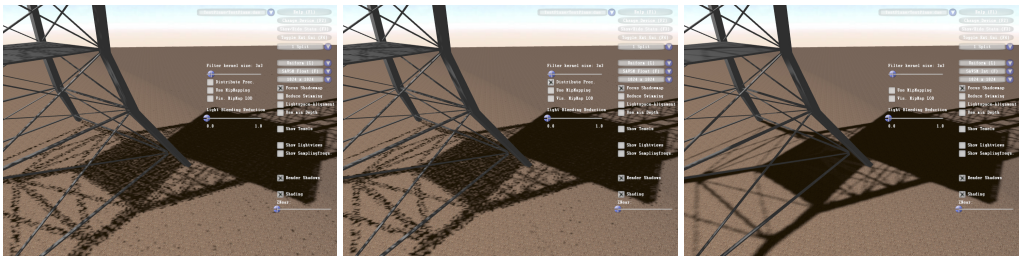
(a) Gaussian filter kernel size 1x1 **(b)** Gaussian filter kernel size 7x7

Figure 4.5: Visualization of Mip-Levels for different filter kernel sizes (VSM)



(a) 2-component 32bit float texture **(b)** 4-component 32bit float texture **(c)** 2-component 32bit integer texture

Figure 4.6: Summed-Area VSM: shadow map size 512x512



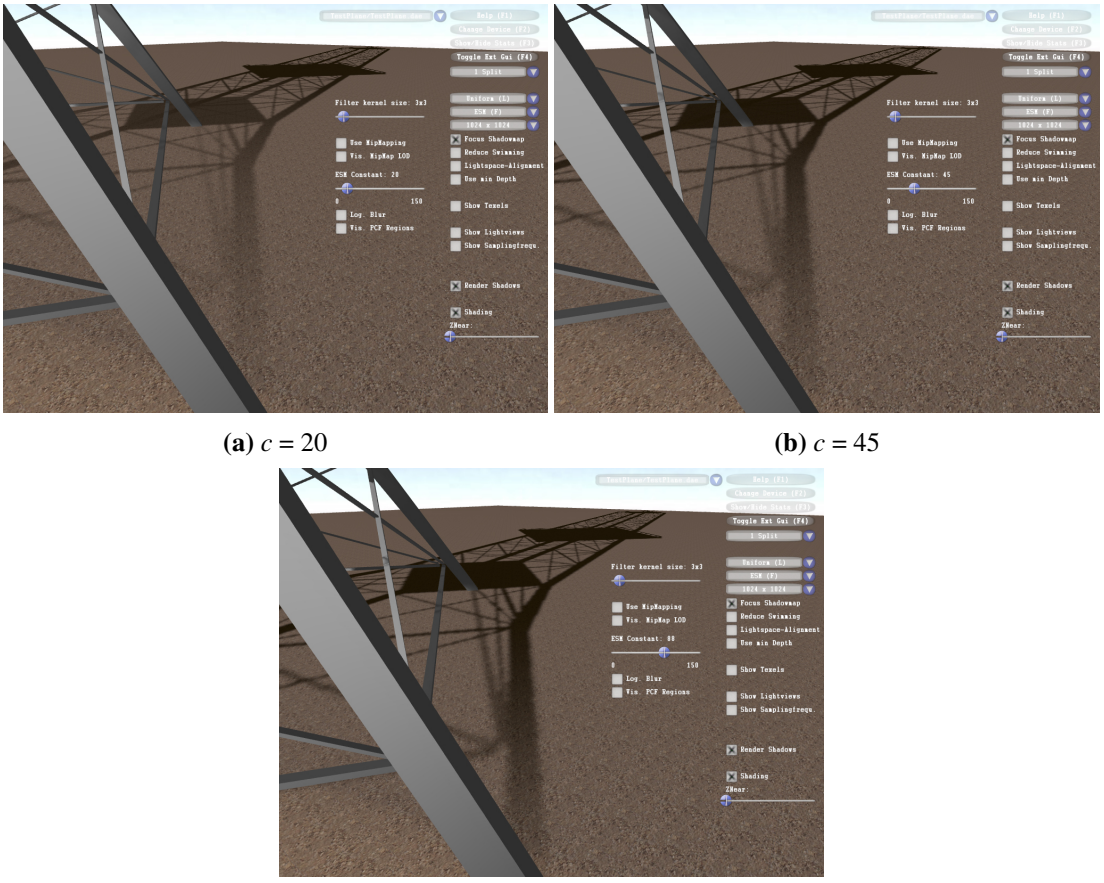
(a) 2-component 32bit float texture **(b)** 4-component 32bit float texture **(c)** 2-component 32bit integer texture

Figure 4.7: Summed-Area VSM: shadow map size 1024x1024

4.2.2 Exponential Shadow Maps

Figure 4.8 shows the impact of the overdarkening factor on the rendering result. The higher c the darker get the shadows and the more aggressive light-bleeding is fought, which occurs on contact-points to the shadow caster. With the standard ESM implementation the value of this factor can reach 88 until overflow errors occur as can be seen in figure 4.9a. The previously described logarithmic approach solves this problem (cf. figure 4.9b) and allows a higher value for c.

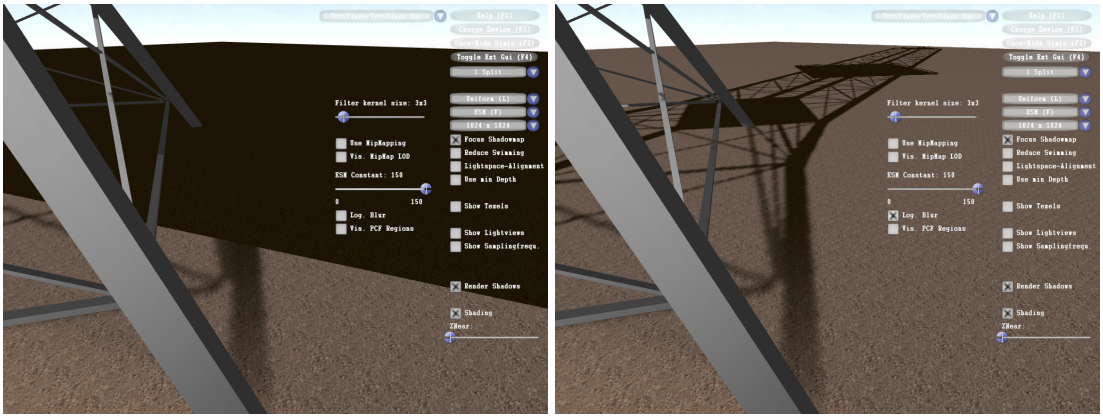
As already mentioned ESM has to fall back to PCF-style filtering if its general assumption is not valid. Figure 4.10 presents some renderings where this is the case. The areas where this fall-back occur are visualized with a red color overlay. It can be seen that these areas are mainly slanted surfaces and that they grow with increasing filter kernel size.



(a) $c = 20$ **(b)** $c = 45$

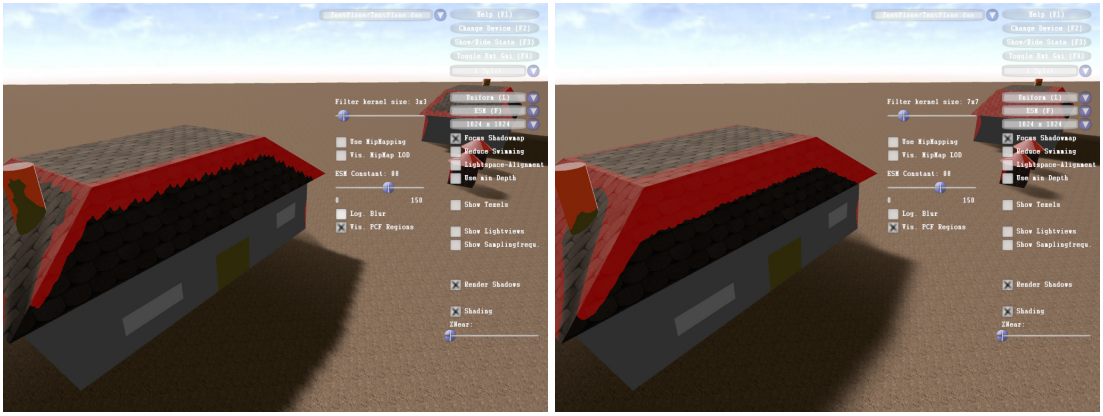
(c) $c = 88$

Figure 4.8: Exponential Shadow Mapping with different overdarkening factor (c) values

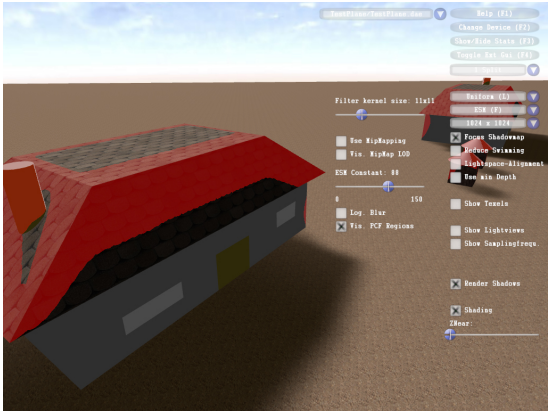


(a) Linear filtering with $c = 150$ **(b)** Logarithmic filtering with $c = 150$

Figure 4.9: Exponential Shadow Mapping with linear and logarithmic filtering



(a) Gaussian filter kernel size 3x3 **(b)** Gaussian filter kernel size 7x7

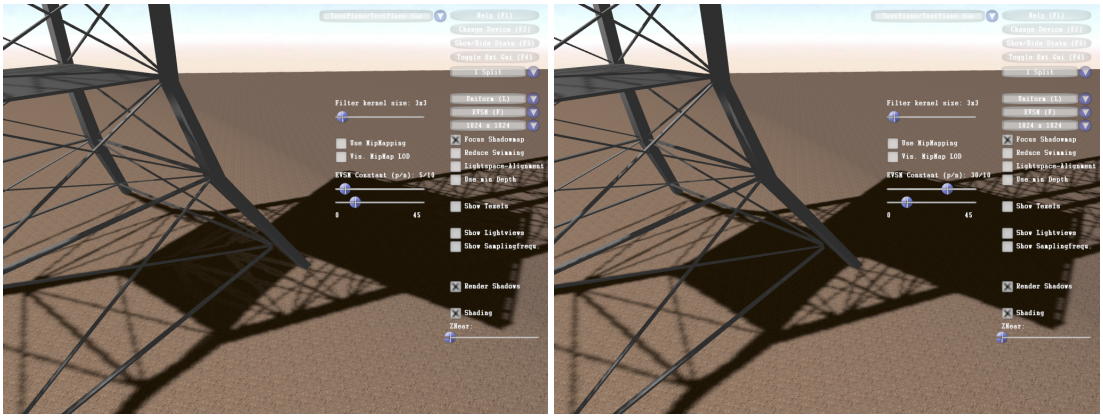


(c) Gaussian filter kernel size 11x11

Figure 4.10: Exponential Shadow Mapping with visualized PCF-fallback regions (*threshold* = 0.020)

4.2.3 Exponential Variance Shadow Maps

In figure 4.11 the effect of the EVSM constants on the output can be seen. The positive constant fights the VSM artifacts. By increasing its value light-bleeding is reduced. Unlike the usage of the light-bleeding reduction constant 3.3.1 in VSM, increasing this constant doesn't lead to overdark shadows. As shown in figure 4.11a a value of 5 isn't sufficient to remove the VSM light-bleeding. Figure 4.11b results from using 30 for the positive constant, which finally removes the artifact.

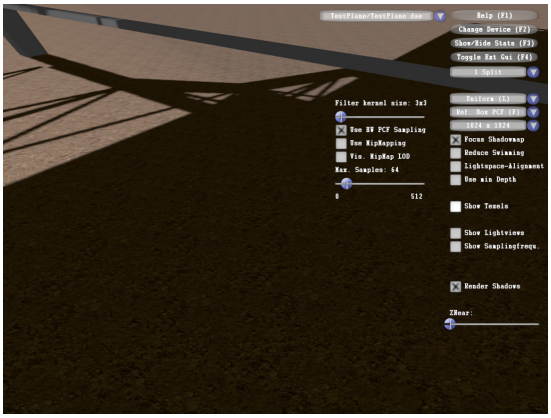


(a) $c^+ = 5$, $c^- = 10$ **(b)** $c^+ = 30$, $c^- = 10$

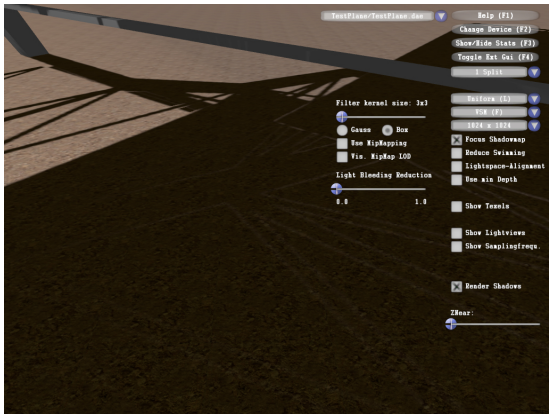
Figure 4.11: Exponential Variance Shadow Mapping with different constant (c^+ / c^-) values

4.2.4 Comparison of ESM, VSM and EVSM

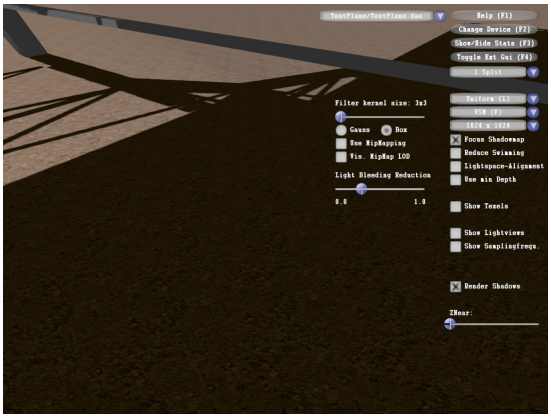
Figure 4.12 compares the output quality of box-filtered ESM, VSM and EVSM. As a reference output hardware-filtered BoxPCF is used (cf. figure 4.12a). VSM produces ugly light-bleeding which can be observed in figure 4.12b. When increasing the light-bleeding reduction constant the light-bleeding can be reduced, but the shadows become too dark and the soft shadow boundaries disappear as can be seen in figure 4.12c. ESM has light-leaking issues at the contact-points to the shadow caster as can be seen in 4.12d. EVSM combines the two approaches and nearly removes all light-bleeding artifacts. The result of the EVSM-filtered shadows is depicted in 4.12e. However, if you compare it to the reference PCF output some small light-bleeding areas can still be found.



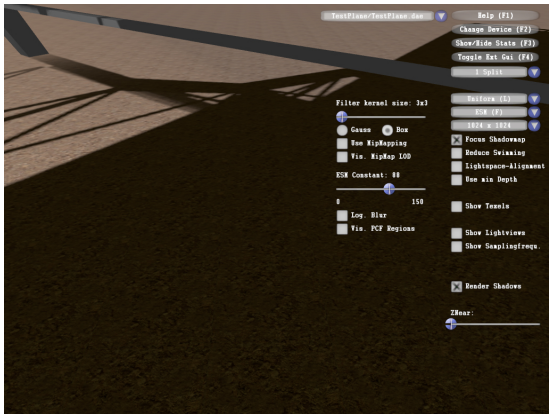
(a) Reference PCF



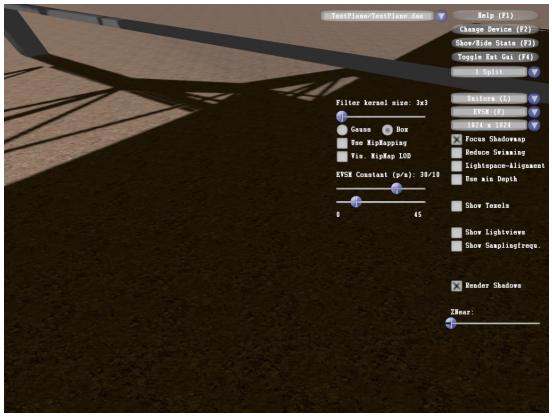
(b) VSM



(c) VSM with $lbr = 0.3$



(d) ESM with $c = 88$



(e) EVSM with $c^+ = 30$, $c^- = 10$

Figure 4.12: Comparison of box-filtered ESM/VSM/EVSM and reference BoxPCF (3x3 filter kernel)

CHAPTER 5

Conclusion

This thesis focused on the three hard shadow filtering techniques *Variance Shadow Mapping*, *Exponential Shadow Mapping* and *Exponential Variance Shadow Mapping*. Altough they all have more or less light bleeding issues, they allow us to reduce the stairstepping/aliasing artifacts at the shadow boundaries and produce a softshadow-like appearance as a nice side effect. We took a closer look on the way they work in chapter 2 and integrated them into the existing framework from Martin Stingl [4]. The results of this implementation and a comparison of the techniques can be seen in chapter 4.

Furthermore the framework was migrated from DirectX® 10 and Visual C++ 9.0 to DirectX® 11 and Visual C++ 10.0.

As an additional feature we implemented gamma correction in order to make the output look right. For more information on gamma correction and the implementation results see appendix A.

5.1 Future work

As long as there is no shadow calculation method available which produces perfect shadows, free of any artifacts, in real time, shadows and especially shadow mapping will not lose its place in research work. Research on the topic of shadow mapping has to aim on the goal of completely removing aliasing artifacts without introducing new errors like light bleeding. Soft shadows are another area of research where improvements have to be made. There is no way around soft shadows when considering the creation of realistic and physically correct shadows.

Work that needs to be done to the extended framework include the better integration of the introduced filtering techniques into the already existing Parallel Split Shadow Mapping [21] implementation. Especially the summed area table methods need to be improved to avoid artifacts

at the split boundaries. One way to do that would be to replace the currently implemented texture atlas by texture arrays. This should be no problem now because we ported the framework to DirectX® 11, which fully supports PCF on texture arrays. This support was introduced with DirectX® 10.1 [25], so the already existing PCF variants wouldn't be destroyed. The reference PCF method should be extended with Poisson-disk sampling and further filtering approaches like gaussian blur. Of course the framework can be further extended with any other advanced shadow mapping or filtering approach like Convolution Shadow Maps [26] or Exponentially Warped Gaussian Filtering [27].

APPENDIX A

Gamma Correction

A.1 Introduction

When lighting equations or other calculations regarding color intensities are solved in the shaders, it is generally assumed that these color intensities are linear. That means, for example, if the light intensities are doubled the resulting color appears twice as bright as the original one on the screen. Unfortunatley this assumption is not valid.

In this chapter we will explain why it is not valid and take a closer look on the terms gamma and gamma-correction. Finally a short tutorial on how to properly implement gamma-correction with the help of the DirectX® 11 API is given.

A.2 About monitors, human vision and linearity

To follow the upcoming part, we must make a little digression into linearity. Mathematically a linear transformation is defined as follows [28]:

- The output of the sum of inputs is equal to the sum of the outputs of the individual inputs: $f(x + y) = f(x) + f(y)$
- The output scales by the same factor as a scale of the input (for a scalar k): $f(k \times x) = k \times f(x)$

The problem when displaying digital images on screen is that cathode-ray-tubes (CRT) are non-linear. The relationship between the voltage input and the reproduced light intensities on the output side is not linear.

Although current display technologies like LCDs don't inherently have this behavior, they simulate this effect due to compatibility and perceptual issues. Not only the display of images is a nonlinear process, but the capture of images too. Therefore most of the captured media (digital photography, DVDs, Blu-Rays,...) is adjusted so that nonlinear monitors can show them correctly.

As mentioned above, there are also perceptual issues why hardware manufacturers alter the pixels that make up the image on the screen. They adjust them in a way the human vision works and the human perception of lightness is nonuniform. [29] [30]

A.3 Gamma and Gamma Correction

We already know that display manufacturers adjust pixel values to better accommodate human vision. That's the point where the terms *gamma* and *gamma-correction* come into play. The manufacturers alter the pixels by applying a function to them, which makes the monitors response similar to an exponential curve (cf. figure A.1). The exponent of this curve is called *gamma*. A common default value for gamma, which is assumed if the exact value of the monitor isn't known, is $^1/_{0.45} \approx 2.2$.

To calculate the eventual output intensity of a pixel following function, called *gamma function*, has to be evaluated:

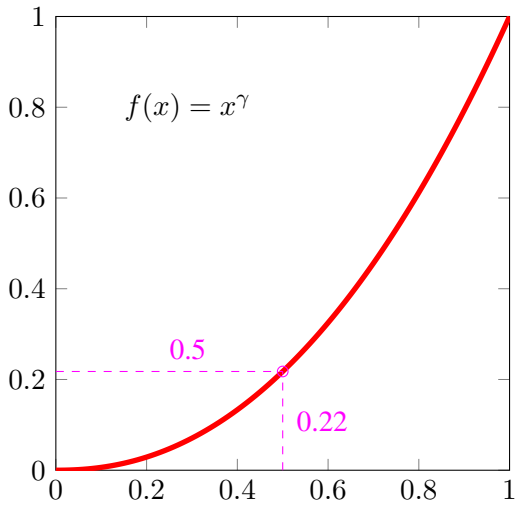


Figure A.1: Gamma function of a monitor, $\gamma = 2.2$

A pixel with an intensity value of 0.5 has therefore an output value of just $0.5^{2.2} \approx 0.22$ (assuming monitor gamma is 2.2). That means a pixel with 50% intensity just emits approximately 22% of the maximum light intensity of the monitor. So 50% of the colorspace is used to cover intensities from less than a quarter of the maximum possible intensity. This shows that the gamma output of the monitor is much darker than the input intensities.

But we don't want the monitor to make our output darker than intended. An intensity value of 0.5 should be displayed as 0.5 and not 0.22!

The simple solution is to apply an inverse transformation (*inverse gamma function*, cf. figure A.2) to the intensities that cancels out the monitors calculation and makes the final (displayed) intensities remain in linear colorspace. The function which has to be evaluated before sending the intensity to the display is the following:

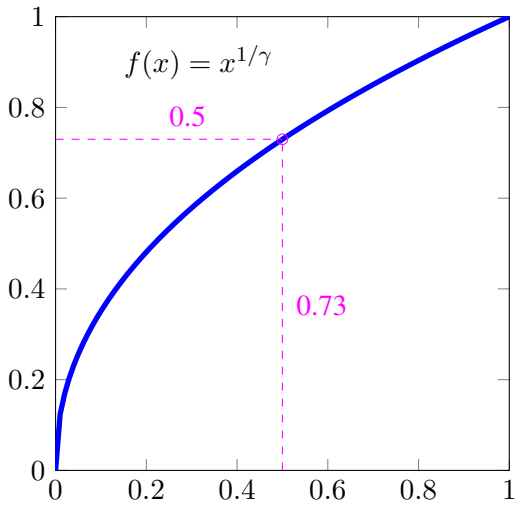


Figure A.2: Inverse Gamma function, $\gamma = 2.2$

The application of this function to the pixel values is called *gamma-correction*. If a light intensity of 0.5 is inserted as input value, we don't store 0.5 but $0.5^{1/2.2} = 0.73$ in the pixel. When this value is sent to the monitor it applies its gamma function which results in the calculation of $0.73^{2.2} = 0.5$. The finally displayed intensity is then as intended 0.5.

To sum up see figure A.3: Gamma correction at the output stage works as follows:

1. apply inverse gamma function (blue line) to the pixel values and send the results to the monitor...
2. ...monitor applies its gamma function (red line) to the received pixel values...
3. ...resulting in a linear function (yellow line) which represents the finally displayed pixels as calculated by the shader.

If the shader output isn't gamma-corrected the red curve will be displayed, which results in a too dark image (cf. figure A.4)!

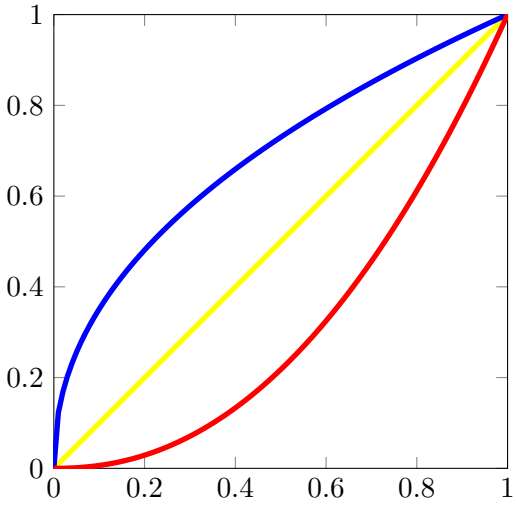


Figure A.3: (Inverse) Gamma function and linear output, $\gamma = 2.2$

Figure A.4: A linear image gamma-corrected (a) and uncorrected (b). Image courtesy of L. Gritz and E. d'Eon [28]

Wrong or no gamma-correction not only makes illumination calculations wrong but destroys mipmapping too. When creating mipmaps usually the pixel value of lower levels is calculated as an average of four neighbouring pixels in the level above. That's a simple linear transformation which leads to wrong results when calculated in nonlinear space.

Not just the output stage is affected by gamma-correction, but the input stage too. Most of the digitally captured and stored images are already gamma-corrected (e.g. JPEGs). It can be assumed that an image which looks right in a web browser is already gamma-corrected and therefore in nonlinear space. Before such images are used as textures they have to be brought back to linear space, because the lighting calculations are always done in that space! Intensity values from different model-files should be uncorrected (applying gamma function) too before

used in lighting calculations. Alpha channels and automatically created maps, such as normal or displacement maps, are already linear and can be used without any further special handling.

The nonlinear space is often called Gamma-space or sRGB-space (standard RGB), while the linear space is known as linear-RGB-space.

A.4 Gamma Correction in DirectX® 11

When implementing a gamma-correct rendering pipeline you have to consider that both input and output are affected by gamma-correction. In the next few lines we will explain what you have to do that the above stated conversions are correctly applied in DirectX® 11 applications.

A.4.1 Backbuffer

To implement gamma-correction the backbuffer needs to be in sRGB-space. This is accomplished by setting the buffer format in the swapchain-descriptor to an *_SRGB* format before creating the D3dDevice and swapchain with this descriptor.

```
1 DXGI_SWAP_CHAIN_DESC desc;
2 ZeroMemory(&desc, sizeof(DXGI_SWAP_CHAIN_DESC));
3 desc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM_SRGB;
4
5 //...set other descriptor members...
6
7 //create the d3ddevice and the swapchain
8 D3D11CreateDeviceAndSwapChain(...,&desc,...);
```

Listing A.1: Gamma-correction of the backbuffer

A.4.2 Textures

The gamma-correction of the input images needs to be done before they are loaded into the 8-bit framebuffer (which is also gamma encoded) to make best use of the available limited buffer precision. Another advantage of this early correction is that the relationship between the intensities stored in the framebuffer and the intensities perceived by a human being is linear, because human perception follows the gamma curve. That means, if the intensity value in the framebuffer is doubled we also recognize a doubled intensity. Therefore nowadays most of the 8-bit media, like JPEGs, is already gamma encoded so that they can be loaded directly into the sRGB framebuffer.

So the image read from the disk (e.g. with FreeImage) must be firstly loaded into a sRGB-texture. In order to be able to access a texture in the pixel shader it has to be wrapped into a shader resource view and the BindFlags member of its descriptor has to be set to *D3D11_BIND_SHADER_RESOURCE*. The format flag in the descriptor needs to be set to an *_SRGB* format to create a sRGB-texture. To avoid that DirectX® converts the loaded images

to sRGB space the filter flag must have the value *D3DX11_FILTER_SRGB*. This is important if you load images that are already saved in sRGB space on the disk, which is mostly the case as mentioned above.

```
D3DX11_IMAGE_INFO imgInfo;
ZeroMemory(&imgInfo, sizeof(D3DX11_IMAGE_INFO));
D3DX11GetImageInfoFromFile(imgfile,0,&imgInfo,0);

D3DX11_IMAGE_LOAD_INFO loadInfo;
ZeroMemory(&loadInfo, sizeof(D3DX11_IMAGE_LOAD_INFO));
loadInfo.Width = imgInfo.Width;
loadInfo.Height = imgInfo.Height;
loadInfo.Depth = imgInfo.Depth;
loadInfo.BindFlags = D3D11_BIND_SHADER_RESOURCE;
loadInfo.Format = DXGI_FORMAT_R8G8B8A8_UNORM_SRGB;
loadInfo.Filter = D3DX11_FILTER_SRGB | D3DX11_FILTER_NONE;
loadInfo.pSrcInfo = &imgInfo;

//...set other descriptor members...

ID3D11ShaderResourceView* srv = 0;

//create the shader resource view
D3DX11CreateShaderResourceViewFromFile(d3dDevice,imgfile,&loadInfo,0,&srv,0);
```

Listing A.2: Gamma-correction of textures

A.4.3 Other color material

For material intensities loaded from model-files you have to do the adjustments directly in the shader. You just need to uncorrect the nonlinear values in order to move them into linear-RGB-space. Of course, if the intensities are already linear no transformation into linear-RGB-space needs to be done.

```
diffuseColor = float4(pow(abs(matDiffuse.rgb), 2.2), matDiffuse.a);
```

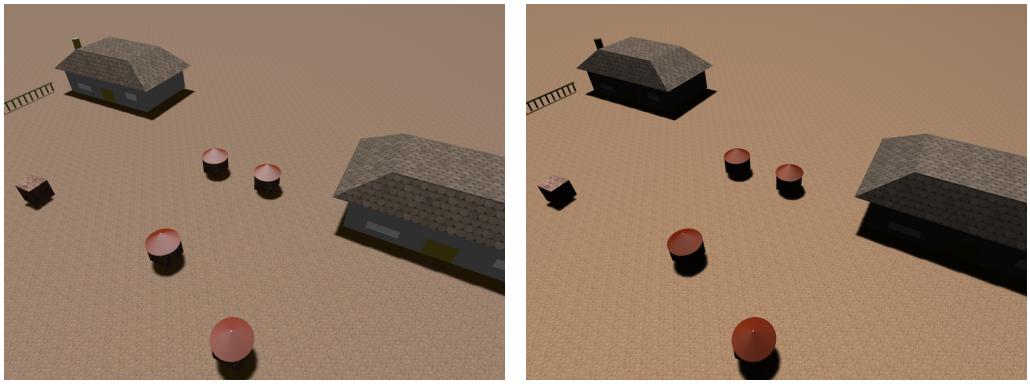
Listing A.3: Gamma-correction of diffuse materials

The gamma-correction at the output stage is done automatically by the sRGB-backbuffer. If you use the DXUT framework, which is shipped with the DirectX® SDK [20], set the gamma mode in DXUT. This is done via *DXUTSetIsInGammaCorrectMode*. The value is by default true. This has to be called before the creation of the D3dDevice, because the backbuffer format is set at the device creation. If DXUT is in gamma-correct mode it will override all texture/backbuffer formats with an *_SRGB* format. The above mentioned filter flag is not set by DXUT! Therefore each used image is converted into sRGB-space when loaded into a sRGB-texture, even if it is already in that space. The result is then brighter than intended because the *inverse gamma function* is applied twice. Make sure you set the filter flag to *D3DX11_FILTER_SRGB* when using sRGB-images. With the DirectX® framework it is not implicitly possible to detect whether the

loaded image is in sRGB-space or in linear-RGB-space. You need to implement an own routine to achieve this.

A.5 Results

In this last section we will present the results of our implementation of gamma-correction.



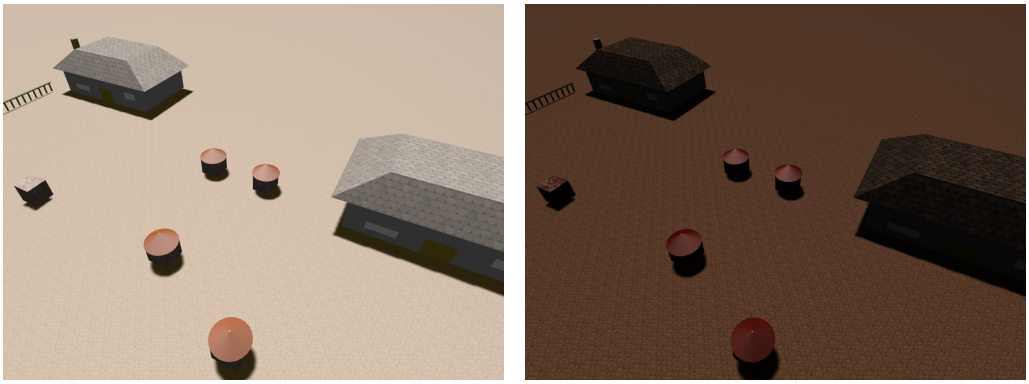
(a) **(b)**

Figure A.5: The framework scene gamma-corrected (a) and uncorrected (b)

In figure A.5a you can see the result of a correct implementation of gamma-correction. The nonlinear JPEG pictures are loaded into sRGB-textures which are automatically converted into linear space by the DirectX® framework when sampled. The material colors from the model-file are linearized manually in the pixel shader. Then the shading and lighting is done in linear space. Finally the resulting image is automatically gamma-corrected due to the sRGB format of the backbuffer and sent to the monitor.

Figure A.5b shows the uncorrected image. Here the input (textures and material colors) as well as the output (backbuffer) are not corrected in any way. The shadows and shadowed regions appear too dark. Details in that regions can't be seen. The color tone of the nonlinear textures and materials is wrong too. This can be seen on the plane texture which appears much more yellow than in the gamma-corrected version. Also the highlights on the small cylindric roofs are yellow instead of white.

The rendering in figure A.6a is the result of an unhandled input and gamma-correct output/back-buffer. Since the textures and materials remain in sRGB-space when sampled and are finally gamma-corrected due to the sRGB-backbuffer the intensities sent to the monitor are twice corrected, which results in a too bright rendering. Figure A.6b depicts a wrong gamma-correction where the input is handled correctly and the backbuffer is not corrected. Here the linear shader results are directly sent to the monitor (which applies its gamma curve) and the image is too dark.



(a) Input not corrected, backbuffer corrected **(b)** Input corrected, backbuffer not corrected

Figure A.6: The framework scene with wrong gamma-correction

Figure A.7 is rendered gamma-correct without setting the *D3DX11_FILTER_SRGB* flag. Here the *inverse gamma function* A.2 is applied to the input images which are already in sRGB-space. So they are twice corrected, which finally results in a too bright image. The non-textured areas are rendered correctly.

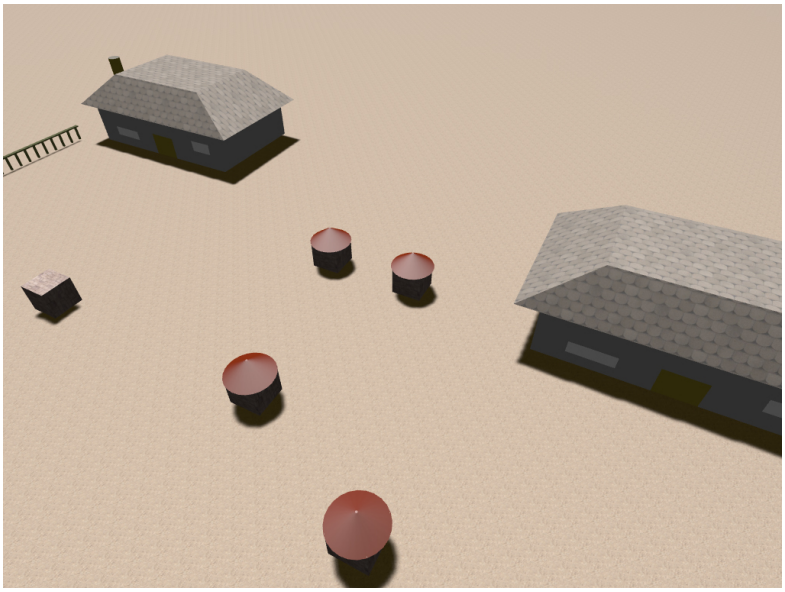
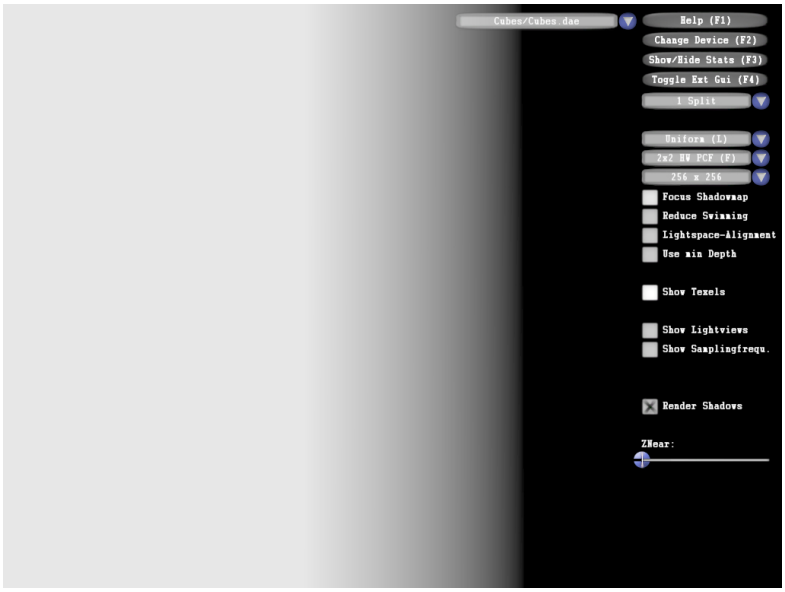
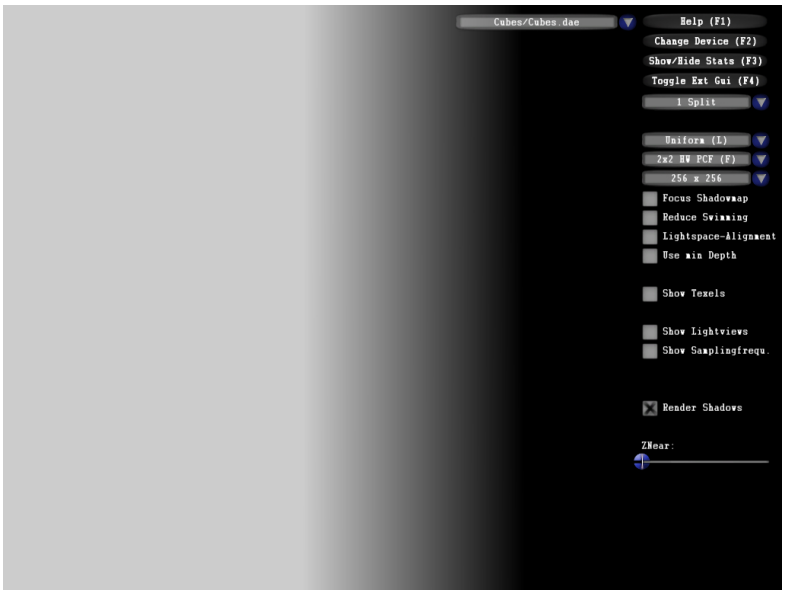


Figure A.7: D3DX11_FILTER_SRGB not set

Figure A.8 shows the difference between a corrected and uncorrected bilinear filtered PCF shadow gradient. The gradient of the uncorrected version is too dark and has a linear fall-off, while the gradient of the corrected shadow has an exponential fall-off.



(a) Gamma corrected bilinear filtering



(b) Uncorrected bilinear filtering

Figure A.8: Bilinear PCF-Filtering with and without gamma correction

Bibliography

[1] MacMillan Dictionary. shadow. Retrieved February 09, 2013, from `http://www.macmillandictionary.com/dictionary/british/shadow`.

[2] Oxford Dictionaries. shadow. Retrieved February 09, 2013, from `http://oxforddictionaries.com/definition/english/shadow`.

[3] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François X. Sillion. A survey of real-time soft-shadow algorithms. In *Eurographics State-of-the-art reports*, volume 22(4), pages 753–774, September 2003.

[4] Martin Stingl. Robust Hard Shadows. Master's thesis, Vienna University of Technology, September 2009.

[5] Franklin C. Crow. Shadow algorithms for computer graphics. In *SIGGRAPH '77 Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, volume 11(2), pages 242—-248, 1977.

[6] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78 Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, volume 12(3), pages 270—-274, August 1978.

[7] Daniel Scherzer. Robust shadow maps for large environments. In *Proceedings of the Central European Seminar on Computer Graphics*, 2005.

[8] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Practical shadow mapping. In *Journal of Graphics Tools*, volume 7, pages 9—-18, 2000.

[9] Marc Stamminger and George Drettakis. Perspective shadow maps. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, volume 21(3), pages 557—562, 2002.

[10] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In *Proceedings of the Eurographics Symposium on Rendering 2004*, pages 143—-152, 2004.

[11] William Reeves, David Salesin, and Robert Cook. Rendering antialiased shadows with depth maps. In *Computer Graphics (Proceedings of SIGGRAPH 1987)*, volume 21(3), pages 283—291, 1987.

[12] Anirudh.S Shastry. Soft-edged shadows. Retrieved February 13, 2013, from http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/soft-edged-shadows-r2193.

[13] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 161—-165, 2006.

[14] Andrew Lauritzen. Summed-area variance shadow maps. In H. Nguyen, editor, *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 8, pages 157–182. Addison-Wesley Professional, August 2007.

[15] Andrew Lauritzen. Rendering Antialiased Shadows using Warped Variance Shadow Maps. Master's thesis, University of Waterloo, 2008.

[16] Franklin Crow. Summed-area tables for texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 1984)*, volume 18(3), pages 207—212, 1984.

[17] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. Exponential shadow maps. In *Proceedings of Graphics Interface 2008*, pages 155—161, May 2008.

[18] Marco Salvi. Rendering filtered shadows with exponential shadow maps. In Wolfgang Engel, editor, *ShaderX6: Advanced Rendering Techniques*, pages 257—274. Charles River Media, February 2008.

[19] Kris Olhovsky. Exponential shadow map filtering (in log space). Retrieved February 16, 2013, from http://www.olhovsky.com/2011/07/exponential-shadow-map-filtering-in-hlsl/.

[20] Microsoft Corporation. Directx developer center. Retrieved February 10, 2013, from http://msdn.microsoft.com/en-us/directx/.

[21] Fan Zhang, Hanqiu Sun, Leilei Xu, and Kit-Lun Lee. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of ACM International Conference on Virtual Reality Continuum and Its Applications 2006*, pages 311–318, 2006.

[22] Microsoft Corporation. Smoothstep. Retrieved March 13, 2013, from http://msdn.microsoft.com/en-us/library/windows/desktop/bb509658%28v=vs.85%29.aspx.

[23] Mark Segal and Kurt Akeley. The opengl graphics system: A specification 4.2. Retrieved March 11, 2013, from http://www.opengl.org/registry/doc/glspec42.core.20120427.withchanges.pdf, page 249, chapter 3.9.11, equation 3.21.

[24] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, volume 24(3), pages 547—555, 2005.

[25] Microsoft Corporation. Cascaded shadow maps. Retrieved March 13, 2013, from http://msdn.microsoft.com/en-us/library/windows/desktop/ee416307%28v=vs.85%29.aspx.

[26] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Convolution shadow maps. In *Proceedings of the Eurographics Symposium on Rendering 2007*, pages 51–60, June 2007.

[27] Jesús Gumbau, Mateu Sbert, László Szirmay-Kalos, Miguel Chover, and Carlos González. Smooth shadow boundaries with exponentially warped gaussian filtering. *Computers and Graphics*, 37(3):214–224, 2013.

[28] Larry Gritz and Eugene d'Eon. The importance of being linear. In H. Nguyen, editor, *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 24, pages 529–543. Addison-Wesley Professional, August 2007.

[29] Charles A. Poynton. Gamma. In *A technical introduction to digital video*, chapter 6, pages 91–114. John Wiley & Sons, Inc., 1996.

[30] Jason L. McKesson. Linearity and gamma. Retrieved February 05, 2013, from http://www.arcsynthesis.org/gltut/Illumination/Tut12%20Monitors%20and%20Gamma.html.