Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Clustered Deep Shadow Maps for Multiple Volumes and Geometry Using CUDA

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik & Digitale Bildverarbeitung

eingereicht von

Wolfgang Knecht
Matrikelnummer 0427560

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg
Mitwirkung: Dipl.-Ing. Dr.techn. Alexander Bornik
Dipl.-Ing. Dr.techn. Markus Grabner
Dipl.-Ing. Dr.techn. Markus Hadwiger

Wien, 08.04.2011

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Wolfgang Knecht
Mottaweg 72, 6822 Röns

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: ______________________

Abstract

In computer graphics shadows play an essential role. Besides visual cues they put a lot of atmosphere into a three dimensional scene. The complex interaction of shadows casted by volumetric objects like clouds or smoke as well as shadows casted by polygonal geometry is very challenging in real-time graphics.

This thesis presents an integrated solution for high quality shadows in scenes involving multiple volumetric as well as translucent polyhedral objects. Shadows between the same type of object and different object types are supported in any direction. The algorithmic solution integrates *Deep Shadow Maps* and *Polyhedral Volume Rendering* implemented in CUDA. Two different memory management strategies for storing DSMs are discussed. The described implementation supports spotlights as well as omnidirectional light sources. DSM clustering further improves rendering quality.

The described method achieves interactive frame rates for shadow calculation and rendering of multiple volumetric objects and polyhedral geometry on current graphics hardware.

Kurzfassung

In der Computergrafik spielen Schatten eine bedeutende Rolle und tragen wesentlich zur Atmosphäre in einer drei dimensionalen Szene bei. Das Zusammenspiel von Schatten volumetrischer Objekte, wie etwa Wolken oder Rauch, als auch Schatten von polygonaler Geometrie stellt in der Echtzeitgrafik eine besondere Herausforderung dar.

Ziel dieser Diplomarbeit ist es, qualitativ hochwertige Schatten in Szene mit mehreren volumetrischen sowie lichtdurchlässigen polygonalen Objekten zu berechnen. Schatten werden dabei auf Objekte vom selben Typ als auch auf Objekte von unterschiedlichem Typ geworfen. Dazu wird die Implementierung von *Deep Shadow Maps* auf Basis eines *Polyhedral Volume Renderers* beschrieben. Die Implementierung unterstützt neben Spot-Lichtquellen auch omnidirektionale Lichtquellen. Des Weiteren werden zwei verschiedene Methoden beschrieben wie *Deep Shadow Maps* abgespeichert werden. Um qualitativ bessere Ergebnisse zu erzielen, werden Objekte der Szene in Gruppen zusammengefasst und für jede Gruppe eine separate *Deep Shadow Map* berechnet.

Auf aktueller Hardware können Schatten in Szenen, bestehend aus mehreren volumetrischen Objekten sowie polygonaler Geometrie, mit interaktiven Frameraten berechnet werden.

Danksagung

An dieser Stelle möchte ich mich bei meinem Professor Dieter Schmalstieg, sowie meinen Betreuern Alexander Bornik, Markus Grabner und Markus Hadwiger bedanken, die mich während meiner Diplomarbeit umfangreich unterstützt haben.

Ich danke dem *Ludwig Boltzmann Institut für Klinisch-Forensische Bildgebung* für die zur Verfügung gestellten medizinischen Datensätze und dem *VRVis Zentrum für Virtual Reality und Visualisierung Forschung*. Die Diplomarbeit wurde vom *„Scale-VS"* Projekt unterstützt, welches durch den *Wiener Wissenschafts-, Forschungs- und Technologiefonds WWTF* finanziert wird.

Bei der *Fakultät für Informatik* der *Technischen Universität Wien* bedanke ich mich für die Finanzierung der benötigten Hardware.

Ich danke meinen Freunden, vor allem meiner Freundin Anja, dass sie mir mit viel Geduld jederzeit zur Seite gestanden haben.

Ein besonderer Dank gilt meinen Eltern und Geschwistern, deren finanzielle und moralische Unterstützung mein Studium überhaupt erst möglich gemacht hat.

Contents

Abstract ii

Kurzfassung iii

Danksagung v

Contents vii

1 Introduction 1
1.1 Shadows in Computer Graphics 1
1.2 Shadows for Translucent Objects 2
1.3 Rendering Methods 4
    1.3.1 The Rendering Pipeline 4
    1.3.2 Raytracing 5
1.4 General Purpose Computing on the GPU – CUDA 6
    1.4.1 Coalesced Memory Access 8
    1.4.2 Atomic Functions 8
    1.4.3 CUDA for Graphics 9
1.5 Aims of This Thesis 9

2 Related Work 11
2.1 Volume Rendering 11
    2.1.1 Texture-Based Volume Rendering 12
    2.1.2 Splatting 12
    2.1.3 Volume Raycasting 13
2.2 Shadow Algorithms 14
    2.2.1 Shadow Volumes 14
    2.2.2 Shadow Mapping 16
2.3 Shadow Algorithms for Volumes and Hair 17
    2.3.1 Adding Shadows to a Texture-Based Volume Renderer 17
    2.3.2 Volume Rendering Using Half-Angle Slicing 18
    2.3.3 Deep Shadow Maps 19
    2.3.4 GPU-Accelerated Deep Shadow Maps 21
    2.3.5 Opacity Shadow Maps 22

    - 2.3.6 Deep Opacity Maps 23
    - 2.3.7 Hair Self Shadowing Using Occupancy Maps 23
    - 2.3.8 Fourier Opacity Mapping 25
  - 2.4 Strategies to Reduce Aliasing Artifacts 26
    - 2.4.1 Perspective Shadow Maps 26
    - 2.4.2 Light Space Perspective Shadow Maps 28
    - 2.4.3 Trapezoidal Shadow Maps 30
    - 2.4.4 Parallel-Split Shadow Maps 33

3 Clustered Deep Shadow Maps for Multiple Volumes and Geometry Using CUDA 35
  - 3.1 Polyhedral Volume Rendering 35
    - 3.1.1 Rasterization in CUDA 36
    - 3.1.2 The Fragment Kernel 37
  - 3.2 Deep Shadow Maps Generation 38
  - 3.3 Rendering Using Deep Shadow Maps 41
  - 3.4 Clustering 43
    - 3.4.1 Building Clusters 44
    - 3.4.2 Adjusting Light's Camera Parameters to Clusters 47
    - 3.4.3 Clustered Deep Shadow Map Generation 47
    - 3.4.4 Use Clustered Deep Shadow Map During Rendering 48

4 Implementation 49
  - 4.1 Polyhedral Volume Rendering 49
    - 4.1.1 Rasterization in CUDA 49
    - 4.1.2 The Fragment Kernel 51
  - 4.2 Memory Management 51
    - 4.2.1 3D Texture 51
    - 4.2.2 Chunked Memory 55
  - 4.3 Deep Shadow Map Generation 59
  - 4.4 Rendering Using Deep Shadow Maps 59
  - 4.5 Clustering 61

5 Results 65
  - 5.1 Deep Shadow Maps and Conventional Shadow Maps 65
  - 5.2 Shadow Interaction of Volumes and Geometry 66
  - 5.3 Shadows of Multiple Volumes 66
  - 5.4 Multiple Spot Lights and Omnidirectional Light Sources 66
  - 5.5 Clustering 67
  - 5.6 Filtering 68
  - 5.7 Performance 71
    - 5.7.1 DSM Resolution 71
    - 5.7.2 Volumetric Dataset Resolution 73
    - 5.7.3 Error Tolerance ϵ 74
    - 5.7.4 Number of Objects 76

5.7.5 Summary . . . 76
5.8 Memory Consumption . . . 77

6 Conclusion 81
6.1 Future Work . . . 81

List of Figures 83

List of Tables 85

Bibliography 87

CHAPTER 1

Introduction

This thesis is about casting shadows in computer graphics. Shadows in computer graphics mostly refer to shadows cast by opaque polyhedral objects, based on a binary decision whether a location on an object is visible from the light source or not. This thesis covers a more general type of shadows – shadows cast by semitransparent objects and objects with a volumetric character like clouds or smoke. An important aspect of this thesis is the shadow interaction of volumetric and polygonal objects.

Chapter 2 presents related work. Algorithms to render volumetric objects as well as shadow algorithms and strategies to improve their quality are discussed. Chapter 3 explains the idea of this thesis and Chapter 4 deals with the implementation details. Results are presented in Chapter 5 and Chapter 6 concludes this thesis.

1.1 Shadows in Computer Graphics

In the real world shadows are omnipresent. We hardly recognize them because they are so common. Mainly we recognize them if they are missing. An image without shadows looks unnatural to us, because shadows give us a lot of additional information about the scene. They implicate a few important visual cues.

For an observer it is hardly possible to estimate distances between objects without shadows, especially if there is no motion of the objects and/or the camera and the user has no additional knowledge of the scene. In Figure 1.1 (a) a rendering without shadows is shown. No one can tell if the blocks are on the ground or in the air. Figure 1.1 (b) shows the same setting but with shadows. Now it's obvious that the left box is on the ground and the right one is in the air. Shadows sometimes also reveal objects that cannot be seen from the camera's point of view but from the light source. Figure 1.2 shows an example of a human model hiding behind a box and the shadow reveals it. Moreover, the shape of other objects can be clarified if they receive shadows. An example is shown in Figure 1.3.

Besides the visual cues shadows are an essential effect to improve image quality in computer graphics. They add realism and atmosphere to the rendered scene.

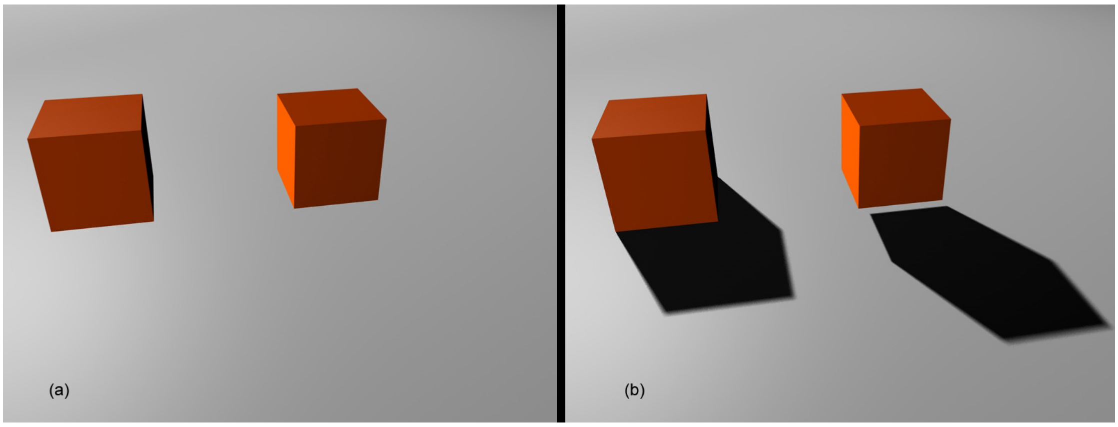


Figure 1.1: Comparison of the same image without (a) and with shadows (b). Shadows are a visual cue to guess distances between objects.

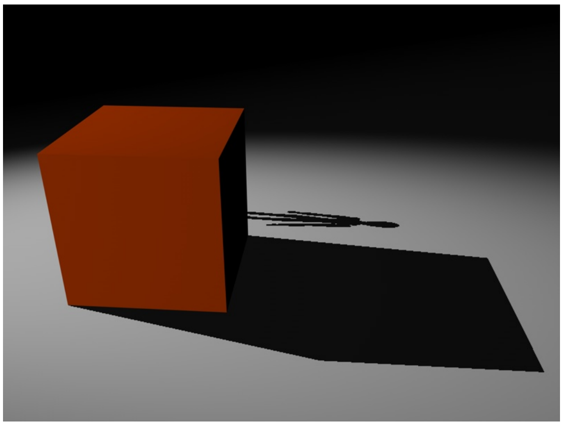


Figure 1.2: Shadows reveal objects that cannot be seen from the camera's point of view.

1.2 Shadows for Translucent Objects

Conventional shadow techniques are able to detect whether an observed point in the scene is in the shadow or outside of the shadow. These techniques are just able to distinguish between the two cases of illuminated and unlit areas (see Figure 1.4 (a)). There is nothing in between. For translucent objects such binary shadow techniques are not sufficient. Smoke for example absorbs just a part of the light. This results in illuminated, partially illuminated and unlit areas in the scene (see Figure 1.4 (b)).

To compute the amount of light penetrating through semitransparent and volumetric objects like the mentioned smoke more advanced shadow techniques have to be used. It is not sufficient to know the first object hit by a light ray, but it is also necessary to know the translucence of all objects between the light source and the observed point. Techniques which solve these

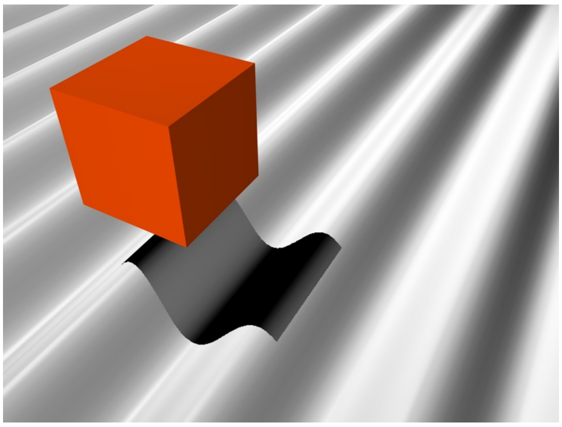


Figure 1.3: Shadows give information about the receiver's shape.

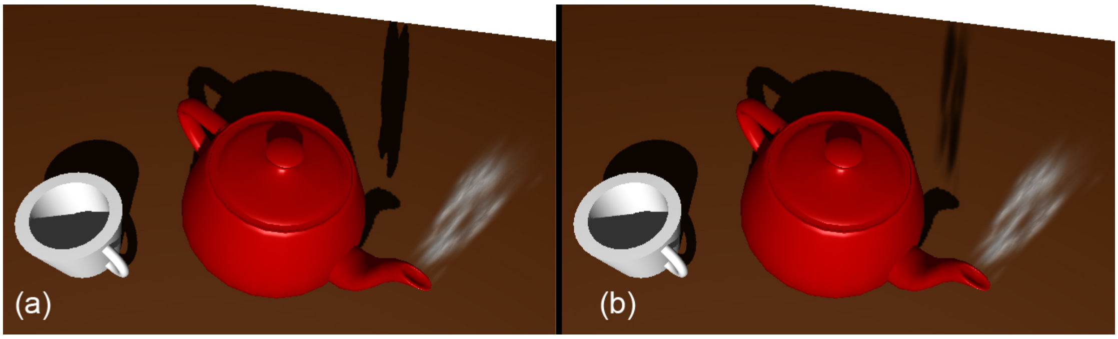


Figure 1.4: (a) shows a scene rendered with conventional Shadow Maps. (b) shows the same scene rendered with *Deep Shadow Maps*.

problems, e.g. *Deep Shadow Maps* by Lokovic and Veach [40], which we integrate into our solution in Chapter 3, will be discussed in Chapter 2.

The film industry uses offline renderers to create shadows of scenes in animation movies that include polygonal geometry and volumetric objects. Figure 1.5 shows an example of such a scene. These shadows introduce a lot of atmosphere to the scene.

In computer games shadows usually exist only for polygonal geometry and not for volumetric objects. The reason is the high computational effort for shadows of objects with volumetric character. The graphics in computer games have to be rendered in real-time which means at least 60 frames per second. Today it is possible to calculate shadows in real-time for single volumes (e.g. Hadwiger et al. [25]). These techniques are used in medical visualization to increase depth perception for example. But a complex scene in a computer game may consists of more than one volumetric object and the ordinary polygonal geometry. To create such high atmospheric scenes including multiple volumes and geometry even in applications, which require real-time performance, current shadow algorithm must be adapted to the possibilities of today's hardware.

Figure 1.5: An image of the short movie "Partly Cloudy" by Pixar [60].

Using a *‘General-purpose computing on Graphics Processing Units’* (*GPGPU*) solution like CUDA (see Section 1.4) enables the computation of *Deep Shadow Maps* for multiple volumes and geometry on current graphics hardware at interactive frame rates.

1.3 Rendering Methods

Rendering is the task of generating a digital image of a scene described by several objects. The objects can be represented in different ways. It is very common to describe objects using polygons (especially triangles) that consist of vertices which are connected by edges. Such a vertex consists of attributes like the position in the three dimensional space, normal vectors and texture coordinates. Objects can also be described by mathematical functions. Another way to describe an object in a scene is to use a volumetric representation which means the visual attributes of an object are stored in a three dimensional grid. This also allows describing the inside of an object and not only its boundary as by the polygonal representation.

There are several methods to calculate the color of a pixel in a digital image. In this section the two most important methods, the *Rendering Pipeline* and *Raytracing*, are described.

1.3.1 The Rendering Pipeline

The rendering pipeline is the most common and most important rendering model in real-time graphics. The rendering pipeline is separated into stages. The key stages are the *Geometry Stage* and the *Rasterization Stage*. In the geometry stage each vertex of a polygon is transformed to screen space. Each transformed polygon is then rasterized in the rasterization stage to determine which pixels are covered by the polygon. A part of a polygon that covers a pixel is called *fragment*. A fragment contains additional attributes, for example the color, the normal vector or texture coordinates. The attributes are interpolated values of the vertex attributes. Once a polygon covers a pixel the color of the pixel can be calculated using these attributes.

When polygons are projected to screen space it often happens that more than one fragment covers a pixel. Therefore an additional buffer called *depth buffer* is used to determine the nearest fragment which is used to calculate the color of the pixel. Each pixel is associated with an entry in the depth buffer. Each time a polygon is projected to screen its depth is compared to the value stored in the depth buffer. If the depth of the fragment is lower than the value stored in the depth buffer the color of the pixel is computed using this fragment and its depth is stored in the depth buffer. Otherwise the fragment is ignored. This guarantees that only the nearest fragment is used to calculate the pixel's color.

The rendering pipeline is a very fast rendering model and therefore usually used in real-time applications like computer games. Current graphics hardware is highly optimized for this method. Today the different stages in the rendering pipeline are freely programmable with small programs called *Shaders*. Programmers can manipulate vertex coordinates, fragment colors and other attributes with *Shaders* to simulate complex materials.

The rendering pipeline always requires polygons to generate fragments during the rasterization stage. Therefore only objects described by polygons can be rendered directly. Because volumetric objects are not represented by polygons which can be rasterized, different techniques exist to render volumetric objects using the rendering pipeline. These techniques exploit the power of *Shaders* (see Section 2.1).

1.3.2 Raytracing

The idea of raytracing is shooting a ray from the eye point through each pixel and following the ray through the scene to calculate the color of a pixel. The rays are intersected with the objects in the scene. Depending on the material parameters like reflection or refraction coefficients and surface properties like the normal vector of the hit object further rays are shot from the hit position in different directions through the scene recursively until a maximum number of *bounces* is reached. More bounces result in higher quality of the final image but increase computation time.

Raytracing can compute high realistic images of scenes including objects described by polygons, objects described by mathematical functions and volumetric objects with complex material parameters but it is a very slow rendering method compared to the rasterization method. An image rendered with raytracing is shown in Figure 1.6.

A simplified and faster variant of raytracing is raycasting. Instead of following the ray along several bounces through the scene, raycasting determines the color of a pixel by the intersections along the initial rays only. Volumetric objects and also their shadows can be rendered on current hardware using a combination of the rendering pipeline and raycasting.

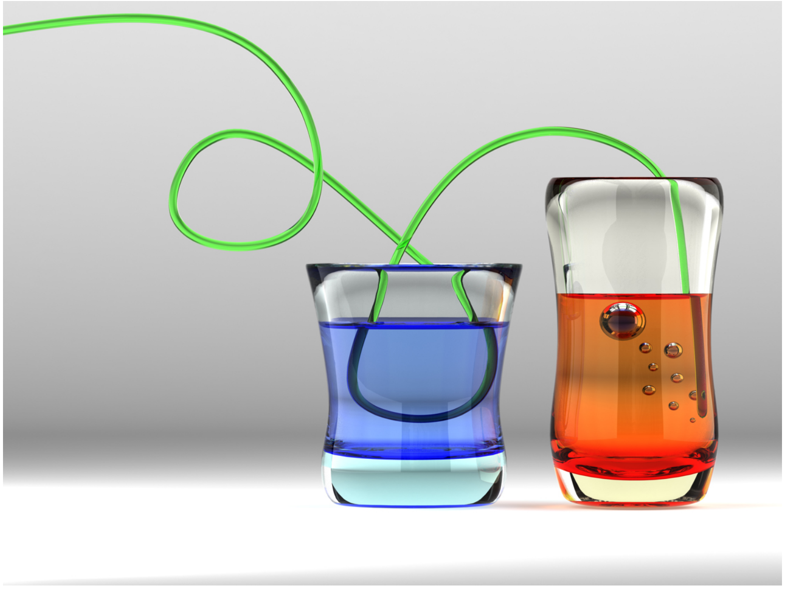


Figure 1.6: A scene rendered with ray tracing [46].

1.4 General Purpose Computing on the GPU – CUDA

Today's graphic processing units (GPUs) are highly parallel architectures because most of the calculations in the scope of computer graphics have to be done in enormous number simultaneously. Whether vertices or fragments, it's always a huge number of elements that have to be processed.

Until the year 1999 vertices and fragments where processed by the *fixed functions pipeline* [48] which means the programmer was not able to control this step. Then GPUs became programmable and programmers where not restricted to the fixed functions pipeline anymore. With *shaders* which are small programs that are executed parallel for all vertices or all fragments advanced effects where possible in computer graphics. People realized that these new possibilities of parallel computation were not necessarily restricted to the field of graphics. High parallel computing can also be used to speed up the computation of other problems like physical or economic simulations. The graphics card can be used to solve non-graphic problems with parallel character very efficient. This kind of usage of the graphics hardware is called *'General-purpose computing on Graphics Processing Units'* (*GPGPU*).

To solve non-graphic problems using the GPU it was still necessary to rasterize something and do the parallel computation in the fragment shader. Usually a simple quad was rasterized for this purpose. To avoid this indirect and inconvenient work flow Nvidia introduced *CUDA* (Compute Unified Device Architecture) [47].

With *C* for CUDA it is possible to write kernels in C-style code that are executed by many threads simultaneously. In CUDA threads are arranged in blocks while the blocks are arranged in a grid (Figure 1.7 illustrates this arrangement). Each thread can be clearly identified by a combination of the built-in variables *blockIdx* and *threadIdx*. This identification can be used to

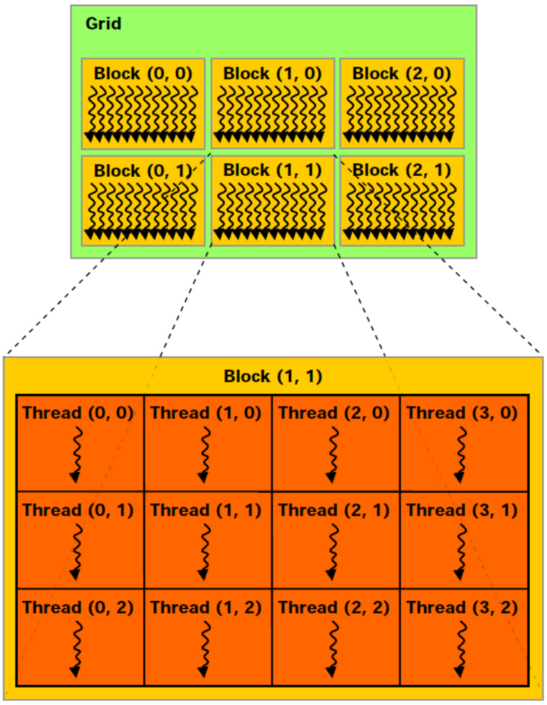


Figure 1.7: Grid of Thread Blocks in CUDA [47].

access different data in different threads. The code that is executed on the data is still the same in each thread. Usually a program that makes use of CUDA consists of serial code that is executed on the host (the CPU) and parallel code that is executed on the device (the GPU). The host code writes data to the memory of the device, initiates the execution of kernels on the device and reads back the data.

There are different types of memory which can be accessed within a CUDA kernel:

Local memory: Local memory is a per-thread memory. Each thread has its own local memory.

Global memory: The global memory can be accessed from each thread of different blocks and even from threads of different grids. Using coalesced memory access (see Section 1.4.1) improves performance.

Shared memory: Shared memory is faster than the local and global memory. The memory is shared between threads of the same block.

Constant memory: The constant memory is read-only memory. It is cached and accessible by all threads.

Texture memory: : Texture memory is also read-only. Its cache is optimized for 2D spatial accesses. Texture memory supports different addressing modes and data filtering.

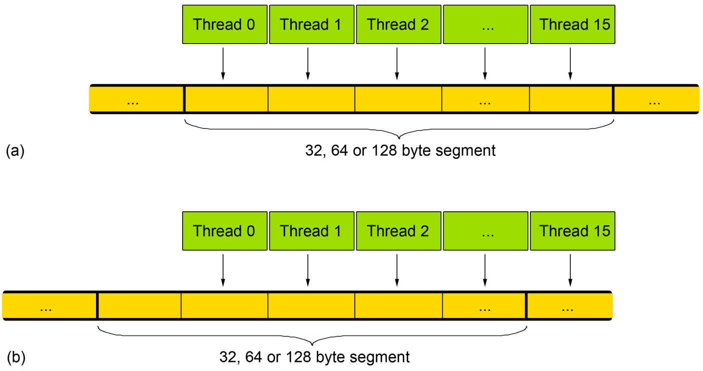


Figure 1.8: (a) well aligned memory access within a 32B, 64B or 128B segment, resulting in one memory transaction. (b) misaligned memory access, resulting in either one transaction with more bytes than necessary or two memory transactions.

1.4.1 Coalesced Memory Access

To get the best performance when accessing global memory the transaction for different threads should be coalesced. Global memory is split into segments with the size of 32 bytes, 64 bytes and 128 bytes. When all threads in a half-warp (16 threads) access memory within the same segment the data can be read/written by one single transaction. For a transaction the smallest possible segment type (32, 64 or 128 bytes) is chosen to save bandwidth.

To achieve the best performance data should be aligned to 32, 64 or 128 bytes (depending on the size of the data). Figure 1.8 illustrates threads accessing memory with different alignments. (a) is well aligned and results in one single transaction while (b) is a misaligned access that results in either one transaction with more bytes than necessary or two memory transactions. Further information about coalesced memory access can be found in the *NVIDIA CUDA C Programming Guide* [47].

1.4.2 Atomic Functions

CUDA provides a set of functions that are called *atomic functions* [47]. An atomic function is a function which manipulates data in shared or global memory with the guarantee that there are no interferences with other threads. If a thread accesses an address using an atomic function no other thread is allowed to access the same address until the operation is complete. For the implementation described in Chapter 4 the following two atomic functions are used:

atomicAdd(address, val): Adds **val** to the value stored at the address **address** and returns the old value.

atomicCAS(address, compare, val): Replaces the value at the address **address** with **val** if the value at the address **address** is equal to **compare**. The function returns the old value.

1.4.3 CUDA for Graphics

CUDA can be used for the same as a graphics card was initially intended for: rasterize polygons and shade the fragments. Doing the rasterization step with CUDA opens new possibilities. As mentioned in Section 1.3.1 usually a depth buffer is used to decide which fragment is the frontmost and thus the one that is used to calculate the color of a pixel. All the information of other fragments that cover the same pixel is lost. When doing the rasterization with CUDA one can keep the information of all these fragments covering a pixel. By sorting them from near to far it is also clear which one is the frontmost fragment, which is the next and so on.

Until recently using conventional shading language did not allow writing at arbitrary locations in a texture. With CUDA it is possible to do that indirectly.

Having the information about all fragments covering the pixels and being able to write at arbitrary locations to the memory enables the calculation of *Deep Shadow Maps* for multiple volumes and geometry simultaneously within one single render pass.

1.5 Aims of This Thesis

This thesis describes how to compute shadows in scenes with the following requirements:

- Scene includes multiple volumetric objects and polygonal geometry simultaneously.
- Polygonal objects cast shadows on them self, on other polygonal objects and on objects with volumetric character.
- Volumetric objects cast shadows on them self, on other volumetric objects and on polygonal geometry.
- The scene can include more than one spotlight and more than one omnidirectional light source.
- All types of objects, the camera and the light sources can be dynamic.
- The shadows should be of high quality with reduced aliasing artifacts.
- The scene should be rendered at interactive frame rates.

CHAPTER 2

Related Work

This chapter gives an overview of existing concepts and algorithms related to this thesis. Section 2.1 discusses volume rendering, Section 2.2 is about basic shadow generation concepts while Section 2.3 explains shadow algorithms specialized for volumetric shadow caster.

There is also a number of approaches to avoid aliasing artifacts when using shadow maps. Some of them are presented in Section 2.4.

2.1 Volume Rendering

Volume rendering handles the task to visualize volumetric 3D data on the screen. In medical context volumetric 3D data is usually acquired by CT (*Computed tomography*), MRI (*Magnetic resonance imaging*) or 3D ultrasonography. In the industry volumetric 3D data is also obtained by MicroCT scanners or FEM (*Finite Element Method*) simulations (e.g. fluid simulation). Volume rendering can also be used to analyze the underground/underwater structure obtained by seismic/sonar measurements. Clouds or smoke for example can be generated procedurally.

The volume data is usually stored as values (density, color, pressure, etc.) in a 3D grid. One data point in the 3D grid is called *voxel*. There are different types of 3D grids. A grid can be either cartesian, regular, rectilinear, curvilinear or unstructured. A cartesian grid has the property that the space between the samples is the same for all three axes while for a regular grid the spaces are constant for an axis but can vary from one axis to the others. For rectilinear grids the space between rows, columns and slices can vary. A curvilinear grid is a non-linear transformed rectilinear grid. Such curvilinear grids are the result of a 3D ultrasonography for example. FEM simulations usually generate data in unstructured grids which means the locations of the *voxels* are arbitrary. The volume rendering methods presented in this section require a regular grid. However there also exist direct volume rendering algorithms for unstructured grids (e.g. Roettger and Ertl [54]).

Direct volume rendering techniques can be categorized into *image order*, *object order* and hybrid methods. *Image order* methods (e.g. *Volume Raycasting* by Levoy [37]) render the

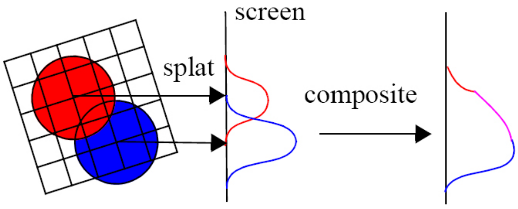


Figure 2.1: Voxels are splatted with a Gaussian distribution on the view plane [26].

volume for each pixel of the image plane while *object order* methods (e.g. *Splatting* by Westover [66, 67] or *Shear-Warp Factorization* by Lacroute and Levoy [35]) compute for each object (e.g. each *voxel*) its contribution to the image plane. Hybrid methods like *Fourier Volume Rendering* by Malzbender [43] can not exactly be categorized in either *object order* or *image order* methods.

This section describes some techniques to render volumetric data. First of all a texture-based approach (2.1.1) then splatting (2.1.2) and finally raycasting (2.1.3). For a further survey of volume rendering techniques see the chapter "Overview of Volume Rendering" by Hansen and Johnson [26].

2.1.1 Texture-Based Volume Rendering

Texture-based volume rendering (Cullip and Neumann [14], Wilson et al. [69], Cabral et al. [12]) uses hardware-accelerated texture mapping to render a volume. Each 2D slice is successively mapped on a quad and rendered in the frame buffer. While drawing a slice, it is blended with the existing values in the frame buffer which finally results in a composite volume on the screen.

The slices can be either aligned to the volume cube or aligned to the view plane (Rezk-Salama et al. [52]) which requires 3D texture support of the hardware. If the slices are aligned to the volume cube the slices have to be aligned to the plane of the cube which is most parallel to the view plane. This implies noticeable transitions when the volume is rotated.

Texture-based volume rendering is very fast on today's hardware. The image quality is not as good as the quality of a volume ray-caster. The quality can be increased by adding additional slices using multi-texturing to interpolate between two slices.

2.1.2 Splatting

Splatting first introduced by Westover [66, 67] is a technique where each voxel is projected separately from back-to-front on the view plane which results in a composite image (see Figure 2.1. Each splat is a disk with a certain diameter. The disk's opacity is usually coupled with a Gaussian distribution.

Splatting produces high quality volume renderings but is a slow method because a high number of splats have to be drawn.

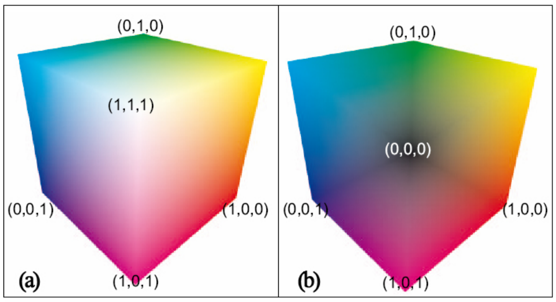


Figure 2.2: The color cube of the first (a) and second (b) pass [34].

2.1.3 Volume Raycasting

Volume Raycasting first introduced by Levoy [37] generates an image out of volumetric data by casting a ray through each pixel of the view plane and sample the volume with a constant sample rate. Then the pixels are colored by the composite samples.

Because ray-casting can be parallelized very good volume raycasting is very suitable for GPU-acceleration. GPU-based volume raycasting is presented by Kruger and Westermann [34] while Hadwiger et al. [24] present advanced shading algorithms for isosurfaces.

The basic idea of the GPU-based volume raycasting algorithm by Kruger and Westermann [34] is to render the volume in three passes:

Pass 1: In the first pass the front faces of the volume's bounding box are rendered. The cube is colored by the corresponding 3D texture coordinates which results in a colored cube (Figure 2.2 (a)).

Pass 2: Similar to the first pass the second pass renders the back faces of the volume cube. The result of this pass is shown in Figure 2.2 (b).

Pass 3: The results of the first and the second pass can now be taken to cast rays through the volume. The color cube from the first pass indicates the start positions of the rays and the cube from the second pass the end positions. Subtracting the result of the second pass from the first pass gives the ray directions. With the knowledge of the start positions and the directions rays can step through the volume and composite the color of each sample point with the Equation 2.1 (Col is the composite color, C the color of the sample and α its opacity) if back-to-front rendering is used or with Equations 2.2 if front-to-back rendering is applied.

$$Col = Col \cdot (1 - \alpha) + C \cdot \alpha \tag{2.1}$$

$$\begin{aligned} Col &= Col + (1 - \alpha_{acc}) \cdot C \cdot \alpha \\ \alpha_{acc} &= \alpha_{acc} + (1 - \alpha_{acc}) \cdot \alpha \end{aligned} \tag{2.2}$$

While sampling the volume a *transfer function* is usually applied on the samples. In its simplest form a transfer function defines the color and the opacity for each density value in the volume data. Since the color and opacity depend on one single value it is a 1D transfer function. Beside this simple form today also multidimensional transfer functions are used to be more flexible in extracting materials and boundaries from volumetric data. An additional dimension for a 2D transfer function is often the gradient magnitude. The second directional derivative could be a variable for further dimensions. Gradients and their magnitudes are pre-calculated or computed on the fly while tracing through the volume for these multidimensional transfer functions and also for shading. Kniss et al. [31, 32] demonstrates the advantages of multidimensional transfer functions and presents an intuitive user interface to manipulate them.

Volume raycasting can be speed up by *early ray termination* and *empty space skipping*. The first one means that a ray can be terminated even before it reaches the back side of the volume if the composite opacity is above a threshold. The contribution of further samples is so minimal that they can be skipped. Certainly early ray termination is only possible when the volume is rendered front-to-back. Empty space skipping can be done using an octree for example. Each node of the octree holds information about the maximum and minimum density inside. This information can be used to skip parts of the volume. Another possibility to do empty space skipping is to rasterize a tight bounding volume in the first and second pass.

The advantages of volume raycasting is its flexibility on current hardware, its speed and the high quality of the visualizations.

2.2 Shadow Algorithms

In this chapter algorithms for real-time shadow computation are discussed. Very simple shadows are *approximate shadows*. Approximate shadows are just dark textures placed underneath objects. They are not really calculated and just a simple approximation of the shadow with minimal cost. Another outdated method uses *Projected Shadows* by Blinn [6]. In this method objects are projected flat on the ground. Both methods suffer from the fact that they just cast shadows on the flat ground and not around edges or curved surfaces.

Currently there are two main concepts to compute shadows in computer graphics. The first one generates shadows using *shadow volumes* (Crow [13]) and the other method is *shadow mapping* (Williams [68]). In Section 2.2.1 the concept of shadow volumes is explained. Section 2.2.2 explains the basic shadow mapping algorithm. There are also hybrid shadow algorithms which combine these two concepts (Sen et al. [56], Govindaraju et al. [22]).

2.2.1 Shadow Volumes

Shadow volumes were first introduced by Crow [13]. It is an object based method to compute shadows.

The basic idea is to create a shadow volume for each occluder in the scene. Each point that lies inside one of these shadow volumes must be in the shadow. Otherwise the point is illuminated. To create a shadow volume for an occluder its silhouette with respect to the light source has to be found. This is done by finding each edge of an occluder which connects one

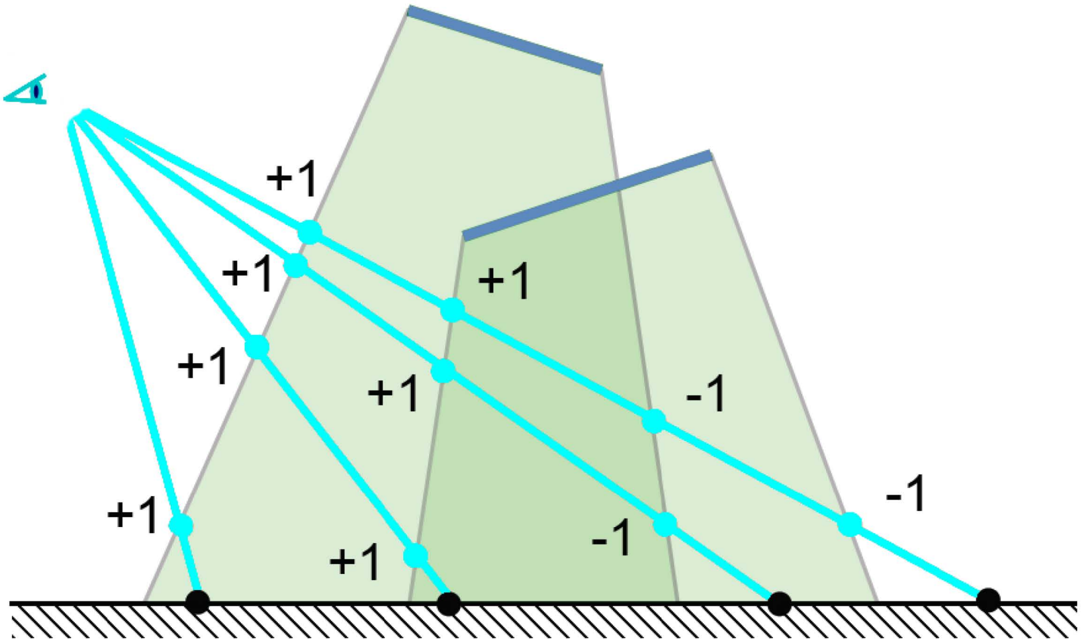


Figure 2.3: Illustration of the shadow volume algorithm [55].

face that faces towards the light source and one face that faces away from the light source. The silhouette is extruded away from the light source to form the shadow volume.

After creating the shadow volumes for each occluder the test if a point is in the shadow or not can be performed with Algorithm 2.1 using the *stencil buffer* (see Figure 2.3).

Algorithm 2.1 Shadow Volumes

```
render the unlit scene
disable writes to the depth and color buffers
use front-face culling
set the stencil operation to increment on depth fail (only count shadows behind the object)
render the shadow volumes
use back-face culling
set the stencil operation to decrement on depth fail
render the shadow volumes
render the illuminated scene where the stencil value is 0 and blend it additive with the unlit
    rendering
```

The main advantage of shadow volumes are the aliasing-free shadows. The shadows are pixel-accurate. Self-shadowing and omnidirectional light sources are supported without additional effort.

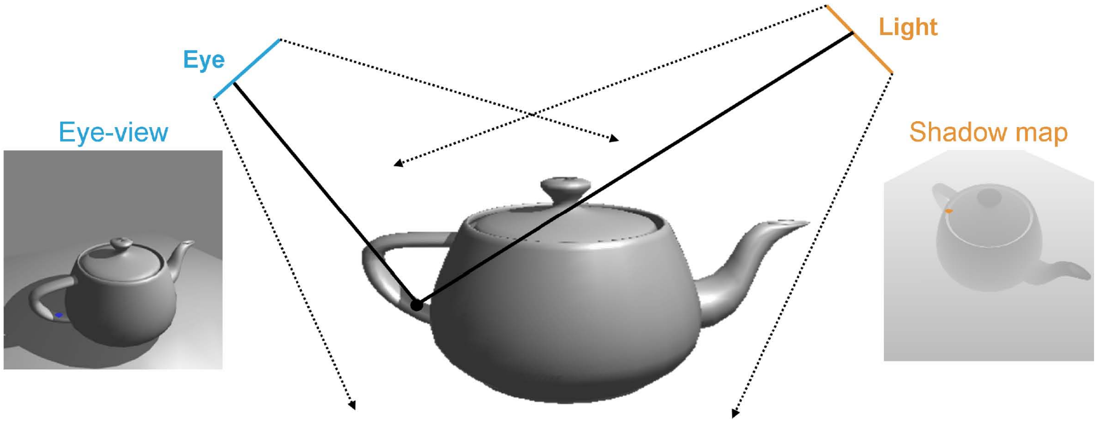


Figure 2.4: Illustration of the shadow mapping algorithm [55].

Drawbacks of shadow volumes are the high costs for the shadow volume generation. The computation time increases with scene complexity. Another drawback is the limitation on polygonal objects. Currently silhouettes of volumetric objects cannot be calculated in a suitable time for real-time applications.

2.2.2 Shadow Mapping

Shadow mapping was first introduced by Williams [68]. In contrast to shadow volumes (2.2.1), shadow mapping is an image based method to compute shadows in computer graphics.

The basic idea behind shadow mapping is to render the scene from the light's point of view. During this render pass the depth values are stored in a texture, which is called the shadow map. After this first pass the shadow map contains the distance to the frontmost object at each texel.

In a second pass the scene is rendered from the camera's point of view. For the shadow evaluation each fragment is transformed to light space to calculate the distance to the light source. The distances of the transformed fragments can then be compared to the corresponding depth values in the shadow map. If the fragments distance to the light is larger than the distance stored in the shadow map the fragment must be in the shadow. The fragment has to be illuminated otherwise. The basic concept of the shadow mapping algorithm is illustrated in Figure 2.4.

Shadow mapping is easy to implement and fits perfectly to current GPU architecture. It is a common method and often used in today's computer games. Because it is an image based method shadow mapping does not depend on scene complexity. As long as a depth value can be calculated every kind of object can cast and receive shadows. Another advantage of shadow mapping is the fact that self-shadowing is supported without additional effort.

But there are also some drawbacks which have to be considered when using shadow maps:

- The basic shadow mapping algorithm creates only hard shadows. That means each point in the scene is either in the shadow or illuminated. There is nothing in between. In real life hard shadows never occur because therefore the light source has to be one single point which is not possible. Each real light has at least a small area which results in soft shadows. There are several approaches to create/fake soft shadows using shadow maps like variance shadow maps (Donnelly and Lauritzen [16], Lauritzen and McCool [36]) or smooth penumbra transitions (Boer [7]).

- For omnidirectional light sources the scene has to be rendered six times for a cube map or two times for parabolic mapping (Brabec et al. [9], Heidrich and Seidel [27]).
- Aliasing artifacts are a big disadvantage of shadow mapping. Because a shadow map has a limited resolution it can happen that more than one pixel on the screen uses the same texel in the shadow map to identify if the fragment is in shadow or not. This results in stair-stepping artifacts which can be reduced by percentage closer filtering (Reeves et al. [51]) for example. Especially surfaces that are almost parallel to the light direction suffer from self shadowing artifacts caused by too low shadow map resolutions and depth quantization. Further information on aliasing artifacts and strategies to reduce them can be found in Section 2.4.

2.3 Shadow Algorithms for Volumes and Hair

Since hair and fur consist of very thin geometry which usually does not cover a full pixel at today's frame buffer resolutions, hair can be rendered either by supersampling which is very inefficient or by just drawing the hair semi-transparent while the alpha value depends on the amount of pixel-coverage. The alpha values can also be interpreted as densities. At least in the case of shadow calculation fine geometry like hair and fur has similarities to volumetric objects.

Computing shadows for volumes and fine geometry like hair is a more advanced task than computing shadows for simple opaque geometry. A single depth value for each light-ray like in shadow mapping is not sufficient anymore because the light can penetrate into the volume or the hair depending on its densities.

There are several algorithms to compute such shadows. Some of them are designed to compute shadows for texture-based volume rendering (Behrens and Ratering [5], Kniss et al. [32]). Section 2.3.1 and 2.3.2 present two of these methods. Other algorithms are specialized on hair rendering (Kim and Neumann [30], Mertens et al. [45], Yuksel and Keyser [71], Sintorn and Assarsson [57, 58]). See Sections 2.3.5, 2.3.6 and 2.3.7 for more details.

Some algorithms are capable to compute shadows using volume ray-casters and can be used to combine shadows of volumes and geometry. Section 2.3.3 explains *Deep Shadow Maps* (Lokovic and Veach [40]) while Section 2.3.4 discusses how to compute *Deep Shadow Maps* using GPU-acceleration (Hadwiger et al. [25]). Section 2.3.8 explains *Fourier opacity mapping* (Jansen and Bavoil [28]).

2.3.1 Adding Shadows to a Texture-Based Volume Renderer

Behrens and Ratering [5] present an algorithm which creates shadows for texture-based volume renderers (see Section 2.1.1).

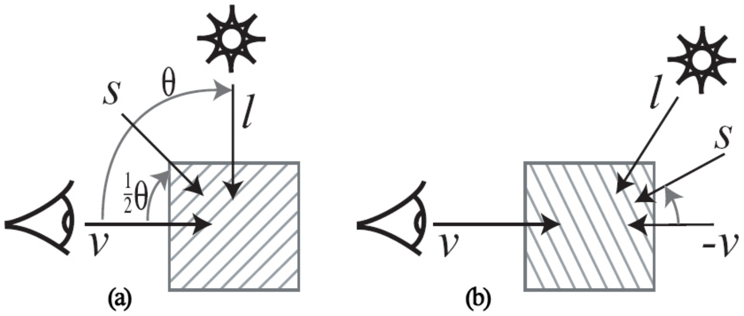


Figure 2.5: Illustration of the half-angle slicing algorithm [32].

The basic idea is to copy successively slice by slice from a given unshadowed volume V into a new volume V'. During the copy process the slices are modified in a way that the resulting volume V' is a shadowed volume. The first slice has to be the one nearest to the light source and the slices have to be oriented equal to the plane of the volume cube which is the most perpendicular plane to the light's view direction. The light penetration is decreased by each slice and stored in an extra shadow buffer. Each slice is then mixed with the shadow buffer and stored in the new volume V'. For the final rendering only the modified volume V' is used.

The advantage of the algorithm is that the shadowed volume only has to be calculated again if the light moves in respect to the volume. As long as only the camera is moving the algorithm doesn't decrease performance. Disadvantages are the artifacts caused by the texture-based approach and additional the algorithm only handles self-shadowing. The volume doesn't cast shadow on anything else than itself.

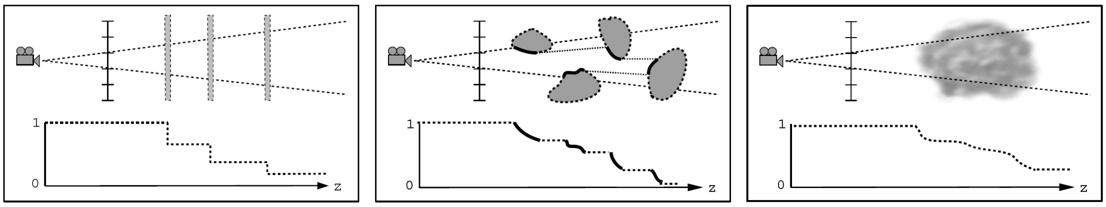
2.3.2 Volume Rendering Using Half-Angle Slicing

Half-angle slicing by Kniss et al. [32] is similar to the approach from Behrens and Ratering [5]. Instead of evaluating the shadow with slicing planes that are oriented differently from the slicing planes used for the final rendering, half-angle slicing uses the same slice orientation for both, the light attenuation and the rendering. To do so the slices are oriented perpendicular to the half-vector which is the vector halfway between the view direction and the light direction (Figure 2.5 (a)). If the dot-product between the view and light vector is negative the half-vector is defined as the vector halfway between the negative view direction and the light direction (Figure 2.5 (b)).

The slices are rendered successively alternately from the camera's point of view and the light beginning with the slice nearest to the light source to ensure front-to-back rendering in respect to the light. This implies front-to-back rendering from the camera's point of view if the dot-product between view and light direction is positive and back-to-front rendering otherwise.

When rendering the volume from the light's point of view the attenuation is accumulated and used for the next slice-rendering from the camera's point of view. This results in a shadowed final volume rendering.

The advantage compared to the algorithm presented in Section 2.3.1 is on the one hand a higher quality rendering with less artifacts and on the other hand less memory consumption be-



***(a)** A stack of semitransparent objects*

***(b)** Partial coverage by opaque blockers*

***(c)** Volume attenuation due to smoke*

Figure 2.6: Examples for visibility functions [40].

cause it only needs an additional shadow buffer and not a complete second volume. A drawback is the need for recomputation of the shadow every time the camera, the light or the volume moves. It is also not applicable for ray-casting.

2.3.3 Deep Shadow Maps

While standard shadow maps are not capable to compute accurate shadows for volumetric objects and semitransparent surfaces, *Deep Shadow Maps* (DSMs) by Lokovic and Veach [40] can be used to compute such shadows. In contrast to ordinary shadow maps a *visibility function* is created for each pixel on the shadow plane. This means that instead of saving just the depth of the first hit surface in the shadow map, every change to light intensity through attenuation by scene objects along the way is recorded in DSMs. DSMs can also be pre-filtered while sampling to reduce aliasing artifacts. This makes it possible to generate high quality shadows for volumes, semitransparent surfaces and even fine geometry like hair or fur.

Sampling

As mentioned in DSMs a pixel on the shadow plane holds a visibility function instead of a single depth value. Visibility functions represent the amount of light that penetrates to each depth. Figure 2.6 shows examples of visibility functions for semitransparent surfaces (a), partial covered pixels (b) and volumes (c).

To handle the case of partial covered pixels and doing pre-filtering one sample point is not sufficient for a pixel. A number of jittered sample points (for example 16) have to be taken for one pixel. Each sample point (x, y) produces a *transmittance function* τ as a function of z which describes the light falloff along the given ray. The visibility function of the pixel is then calculated as the weighted sum of the transmittance functions where the weights depend on the filter kernel.

A transmittance function is the product of a *surface transmittance function* τ^s and a *volume transmittance function* τ^v (see Figure 2.7). The surface transmittance function is generated by doing surface intersections along the ray through (x, y). Each intersections generates two nodes in τ^s which results in a piecewise continuous function (Figure 2.7 (a)). A volume transmittance

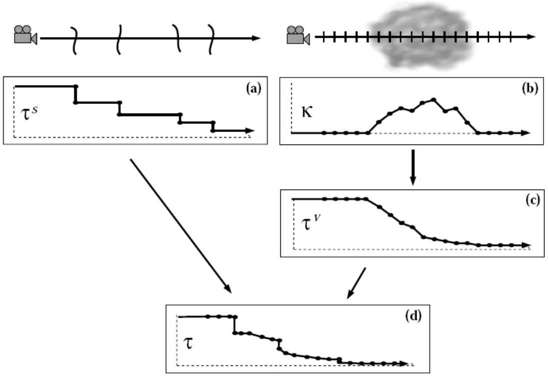


Figure 2.7: A transmittance function is the product of the surface transmittance function and the volume transmittance function [40].

function is generated by creating an *extinction function* κ first. Sampling the volumes density in regular intervals along the ray through (x, y) results in κ.

$$T_i = exp(-(z^v_{i+1} - z^v_i)(\kappa_{i+1} + \kappa_i)/2) \tag{2.3}$$

The transmittance T_i at each node of τ^v (Figure 2.7 (c)) is then computed with Equation 2.3. The pre-filtered visibility function $V_{i,j}(z)$ of the pixel (i, j) is the result of:

$$V_{i,j}(z) = \sum_{k=1}^{n} \omega_k \tau_k(z) \tag{2.4}$$

where n is the number of sampled transmittance functions and ω_k is the filter weight for the corresponding transmittance function τ_k.

Compression

A visibility function $V(z)$ is now defined by a number of nodes. To reduce memory consumption and speed up subsequent shadow map lookups the visibility functions are compressed in a further step. Because usually a visibility function is a very smooth function, it can be approximated by a function $V'(z)$ with less nodes. For details about the compression algorithm see Section 3.2.

Lookup

The lookup while rendering the scene is similar to the standard shadow map lookup. Additional the depth value z has to be searched in the DSM. This is done by either doing an initial linear or binary search through the compressed visibility function to find the two adjoining nodes for each pixel first. For further shadow estimations at the same pixel the fact that lookups are often at nearby nodes the initial search can be skipped and doing a linear forward and backward search from the current position in the DSM is sufficient.

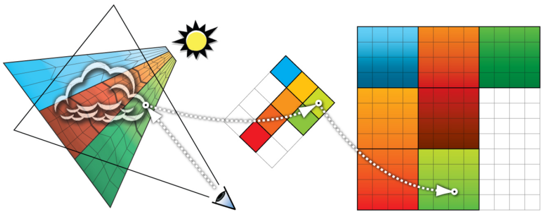


Figure 2.8: The DSM is organized in tiles. An additional low resolution texture references to the corresponding position in the tiled DSM [25].

The visibility functions can also be filtered while doing the lookup to achieve high quality shadows.

2.3.4 GPU-Accelerated Deep Shadow Maps

While the original *Deep Shadow Maps* algorithms was designed for offline rendering Hadwiger et al. [25] adapted it to make use of GPU-acceleration to reach interactive frame rates. They split up the DSM generation in multiple render passes caused by the hardware limitation of eight render targets. Each render target corresponds to one node in the visibility function.

To generate the DSM ray casting is performed including empty space skipping. The compression step is done in the same way as described by Lokovic and Veach [40] (see Section 2.3.3).

Memory Management

The nodes of the visibility functions are stored in a 3D texture. The number of nodes in the visibility function varies from pixel to pixel in the DSM. So it would be wastage of memory if the 3D texture would have as much layers as the maximum number of nodes of all visibility functions. Therefore Hadwiger et al. [25] suggest to split the DSM into tiles where each tile has a constant number of layers (e.g. 8) and dynamically allocate memory in the 3D texture for each tile if additional layers are needed. Figure 2.8 illustrates the tiling of the DSM. An additional low resolution texture is used to refer to the positions of the tiles in the 3D texture.

Pre-Computed Visibility Functions

Creating DSMs based on high frequency transfer functions introduce self-shadowing artifacts especially at isosurfaces. They can be reduced by either increasing the sampling rate or using pre-computed and compressed visibility functions similar to pre-integrated volume rendering by Engel et al. [17]. For each possible combination of two density values a visibility function is computed in the same way as for DSM generation. The maximum number of nodes is limited to a constant number (e.g. 4).

The pre-computed and compressed visibility functions are stored in a lookup texture which is used during DSM generation. Figure 2.9 shows examples of pre-computed visibility functions (red). The visibility function is shown as dashed lines and the results of regular sampling are

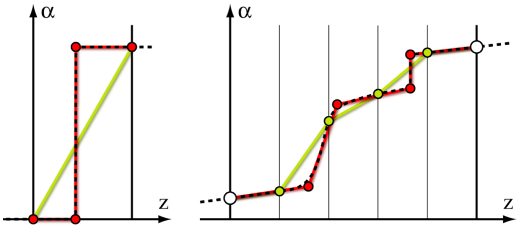


Figure 2.9: Pre-compressed visibility functions (red nodes) lead to a better approximation of the visibility function (dashed line) than regular sampling (green nodes) [25].

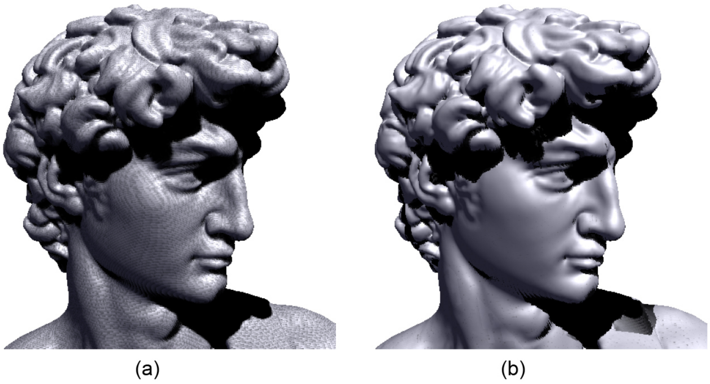


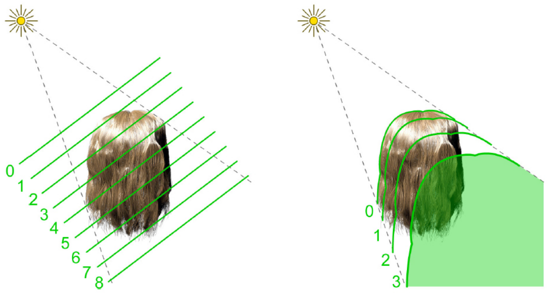
Figure 2.10: Both images are generated using the same sample rate. (a) Regular sampling without pre-computed visibility functions and with (b) pre-computed visibility functions [25].

shown in green. On the left an isosurface is captured correctly by the pre-computed visibility function while it is missed by regular sampling. Figure 2.10 shows an example of a rendered isosurface without (a) and with (b) pre-computed visibility functions. The right example in Figure 2.9 shows the difference between the pre-computed visibility function and an increased sampling rate. The pre-computed visibility function approximates the real visibility function much better with the same amount of nodes.

2.3.5 Opacity Shadow Maps

Opacity Shadow Maps by Kim and Neumann [30] are designed to render shadows of fine geometry like hair. They are simpler than DSMs but cause more artifacts than DSMs.

The basic idea is to create a constant number of slices perpendicular to the light's view direction with regular spacing between the slicing planes. Then the hair is clipped against each plane successively starting with the one nearest to the light source. The hair density defines the light falloff for the given plane. The density of the current slice is blend with the densities of the previous slices.



(a) Opacity Shadow Maps (b) Deep Opacity Maps

Figure 2.11: The difference between *Opacity Shadow Maps* and *Deep Opacity Maps* [71].

For the lookup the light penetration is calculated by interpolating between two slices. The regular arranged slices cause clearly visible artifacts which only can be reduced by a large number of slicing planes.

2.3.6 Deep Opacity Maps

Deep Opacity Maps by Yuksel and Keyser [71] are also designed for hair rendering and are similar to opacity shadow maps. The difference to *Opacity Shadow Maps* is that the slices are not perpendicular to the light's view direction. Instead they are aligned to the first hit depth of the hair geometry. Figure 2.11 illustrates the difference between *Opacity Shadow Maps* and *Deep Opacity Maps*.

The aligned slicing planes hide layering artifacts at the surface of an object even with a small number of layers. But there are still slice-shaped artifacts inside a volume.

2.3.7 Hair Self Shadowing Using Occupancy Maps

Sintorn and Assarsson [58] present a method which is designed to generate *visibility functions* for hair self shadowing. It underlies a similar idea as *Opacity Shadow Maps* or *Deep Opacity Maps*.

To reconstruct the visibility function slices along the light rays are used. But instead of storing the opacity in each slice as other methods do, in an *occupancy map* just one bit is set if at least one fragment covers the slice. As a result a simple four channel unsigned integer render target can store 128 slices for each texel in the occupancy map.

For reconstruction the total number of fragments F covering a texel must be recorded too. With the total number of set bits S of a texel the opacity increase of a slice whose bit is set is then calculated as $\frac{F}{S}$.

A further improvement presented by Sintorn and Assarsson [58] is to us a *slab map*. A slab map groups slices together and stores the number of fragments F_i covering a texel which falls in a slice of the group. So the opacity increase of a slice whose bit is set can then be calculated as $\frac{F_i}{S_i}$ where S_i is the number of set bits of slices within the slab i. This decreases the error while reconstructing the visibility function. Sintorn and Assarsson [58] suggest using a four channel

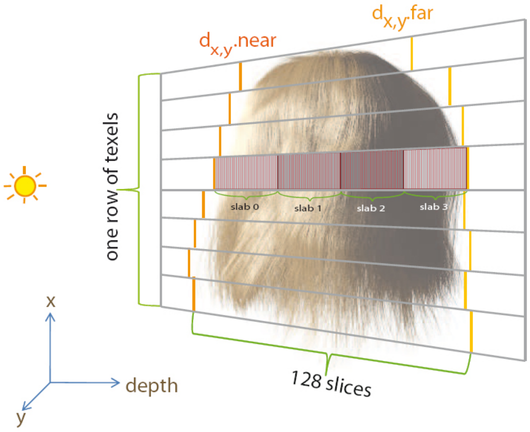


Figure 2.12: Illustration of some texels of an occupancy map, its slabs and the depth-range [58].

float texture as a slab map which results in four slabs. If a four channel unsigned integer texture is used as the occupancy map each slab holds 32 slices.

Another suggestion is to minimize the depth-range for each texel of the occupancy map to increase depth accuracy. To do this a *depth-range map* is created. The *depth-range map* stores the nearest and farthest depth value of a texel and the slices are mapped in this range (see Figure 2.12).

An overview how to generate the *depth-range-*, *slab-* and *occupancy map*:

1. Generate *depth-range map* by rendering the hair two times with alternating depth-test function from the light's point of view.
2. Render the hair. Calculate its depth d in respect to the *depth-range map*.
3. Use d to estimate in which slice the fragment falls and set the corresponding bit in the *occupancy map*. The blending operator has to be set to *bitwise-or*.
4. Estimate the *slab* using d again and set the corresponding color channel in the *slab map* to 1.0. Additive blending must be enabled.

The shadow term while rendering the hair is approximated by multiply the number of fragments in front of the current fragment by a constant shadow weight. The depth of a fragment is calculated in respect to the depth-range map as before. Using this depth the current slice and slab for the given fragment is calculated. The number of fragments in front of the current slice is then assembled by the sum of fragments in the slabs in front of the current slab and the number of fragments in front of the slice in the current slab. The number of fragments in front of the slice in the current slab is:

$$\frac{F_i}{B_{total}} \cdot B_{before} \tag{2.5}$$

where F_i is the number of fragments within the current slab, B_{total} the total number of set bits in the current slab and B_{before} the number of set bits within the current slab but in front of the current slice.

Hair self shadowing using occupancy maps needs far less memory than methods like opacity shadow maps or *Deep Opacity Maps*. The algorithm is limited to compute shadows of fragments with the same opacity which fits perfectly to hair rendering but is not suitable to compute shadows for volumes with different densities.

2.3.8 Fourier Opacity Mapping

Fourier opacity mapping was introduced by Jansen and Bavoil [28] and was inspired by *Convolution shadow maps* (Annen et al. [2, 3]). While *Opacity Maps* as well as *Deep Opacity Maps* suffer from artifacts caused by the discrete slicing, Fourier opacity maps are designed to avoid this problem.

Fourier opacity maps formulate the absorption function $\sigma(z)$ in respect to the depth z for each light ray with a constant number of Fourier coefficients. The coefficients are calculated with Equations 2.6 and 2.7 where α_i is the opacity of a primitive and d_i its depth value.

$$a_k = -2\sum_i \ln(1-\alpha_i)cos(2\pi k d_i) \tag{2.6}$$

$$b_k = -2\sum_i \ln(1-\alpha_i)sin(2\pi k d_i) \tag{2.7}$$

This can be achieved by rendering each primitive from the light's point of view with depth testing disabled and additive blending enabled. In the shader the output of each primitive which is then blended additive in the render targets is:

$$\delta a_{i,k} = -2\ln(1-\alpha_i)cos(2\pi k d_i) \tag{2.8}$$

$$\delta b_{i,k} = -2\ln(1-\alpha_i)sin(2\pi k d_i) \tag{2.9}$$

In the rendering step these coefficients can then be used to approximate the shadow term for each depth using the Absorption Function 2.10.

$$\sigma(z) \approx \frac{\alpha_0}{2} + \sum_{k=1}^{n} \alpha_k cos(2\pi k z) + \sum_{k=1}^{n} b_k sin(2\pi k z) \tag{2.10}$$

The benefit of Fourier opacity mapping is the smoothness of its absorption functions. There are no artifacts caused by discrete slicing like in other methods. It is designed to compute shadows from low frequency volumes like smoke or clouds and is not capable to compute shadows for high frequency occluders with a reasonable number of coefficients.

2.4 Strategies to Reduce Aliasing Artifacts

There are different types of aliasing artifacts when doing shadow mapping:

Perspective aliasing: The perspective projection of the viewer causes undersampling of the shadow map. Parameterization of the shadow map can reduce perspective aliasing artifacts.

Projection aliasing: Projection aliasing occurs for surfaces that are almost parallel to the light direction. One texel in the shadow map covers a big area of such a surface. To avoid projection aliasing higher resolution of the shadow map would be required at the respective areas. Therefore an expensive scene analysis would be necessary for each frame.

Incorrect self-shadowing: Self-shadowing artifacts occur because of depth quantization. Using a bias reduce self-shadowing aliasing artifacts. Also an approach called *Second-depth shadow mapping* by Wang and Molnar [64] reduces this kind of artifacts.

There exist a lot of algorithms to reduce aliasing artifacts (especially perspective aliasing artifacts). One simple approach is to focus the shadow map on the visible part of the scene (Brabec et al. [10]). Other approaches like the *Adaptive Shadow Maps* by Fernando et al. [20], *Alias-Free Shadow Maps* by Aila and Laine [1] or *Logarithmic Perspective Shadow Maps* by Lloyd [38], Lloyd et al. [39] do not fit well to current hardware. Some methods use more than one single shadow map to reduce aliasing artifacts (Tadamura et al. [61], Arvo [4], Giegl and Wimmer [21]).

In Section 2.4.1 *Perspective shadow maps* by Stamminger and Drettakis [59] are explained while Section 2.4.2 discusses *Light Space Perspective Shadow Maps* by Wimmer et al. [70]. *Trapezoidal Shadow Maps* by Martin and Tan [44] are explained in Section 2.4.3 and Section 2.4.4 is about *Parallel-split shadow maps* by Zhang et al. [72].

2.4.1 Perspective Shadow Maps

Perspective shadow maps were introduced by Stamminger and Drettakis [59]. The idea behind them is to reduce *perspective aliasing* in the context of shadow mapping. Instead of rendering the scene as usual for shadow maps from the light's point of view with a traditional perspective (for point lights) or orthogonal (for directional light sources) projection *perspective shadow maps* do an additional perspective transformation to warp the shadow map.

First the scene is transformed with the camera's projection matrix which does nothing else than projecting the view frustum of the camera to the unit cube. This has the effect that objects near the camera are bigger than farther objects. The scene is now in the so called *post-perspective space* or in the space of normalized device coordinates. Also the light source is transformed into that space. In a second step the depth map is created by rendering the scene (which is now in post-perspective space) from the transformed light's point of view. This results in a shadow maps which has a higher resolution for objects near the camera and a lower resolution for objects farther away from the camera. The effect is a noticeable reduction of perspective aliasing artifacts.

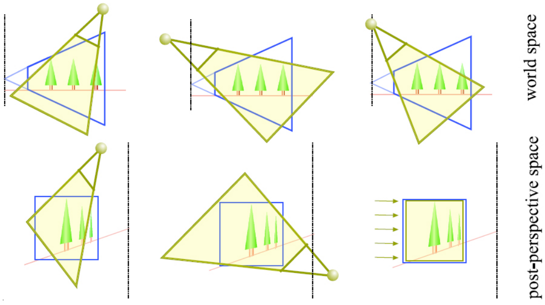


Figure 2.13: The mapping of a scene into post-perspective space. Top row: world space. Bottom row: post-perspective space. Left: light source in front of the camera. Middle: light source behind the camera. Right: light source in the same plane as the camera [59].

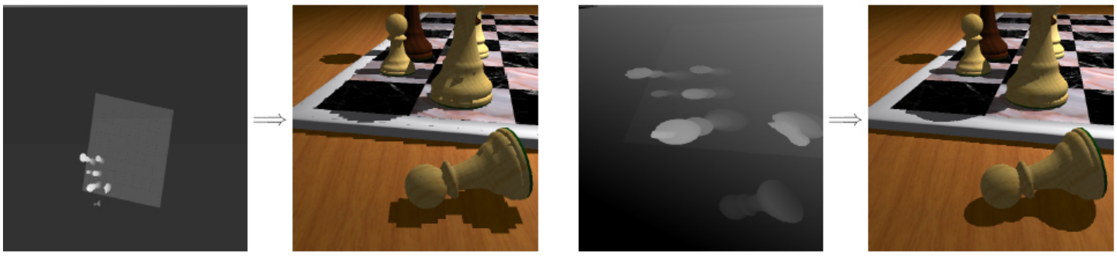


Figure 2.14: The left two images show a scene rendered with uniform shadow maps and its corresponding depth map. The right two images show the same scene rendered with Perspective shadow maps and its depth map [59].

Figure 2.13 illustrates the projection of the scene into post-perspective space. The top row shows the unprojected scene and the bottom row is the same scene after the camera perspective projection. It can be observed how the trees nearby the camera are bigger than the trees in the background. The image shows three different cases of camera, scene and point light setup. In the left column the light source is in front of the camera. In post-perspective space (bottom row) the light still comes from the same direction. The middle column shows the case of a point light behind the camera which gets inverted after perspective projection. Point lights that lie directly on the plane which goes through the camera's view point and is parallel to the view plane (right column) become directional light sources in post-perspective space.

Figure 2.14 compares a standard shadow map with perspective shadow map. The left image shows the uniform depth map, the second left the final rendered image using the standard shadow map. On the right side there is the perspective shadow map and its corresponding final rendering. Both methods use the same shadow map resolution. As one can see the perspective shadow maps improve the shadow quality significantly.

Perspective shadow maps work fine when all occluders are within the camera's view frustum. As soon as an object is outside of the camera's view frustum but still inside of the light's view

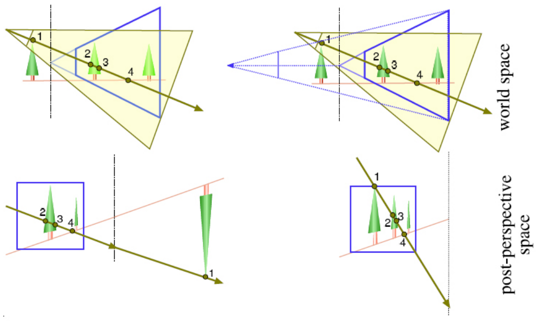


Figure 2.15: Left: one object is behind the camera but in front of the light source. This object is inverted and behind the infinite plane in post-perspective space. Right: the camera is moved backwards to avoid this problem [59].

frustum problems occur.

See Figure 2.15 for example: the top left shows a scene with three trees. One of the trees is behind the camera's view frustum (blue) but in front of the light's view frustum (yellow). In the bottom left is the same scene in post-perspective space. The tree which was outside of the camera's view frustum before is now inverted and behind the *infinity plane*. The *infinity plane* is defined as $z = \frac{f+n}{f-n}$ where f is the far plane and n the near plane. To avoid this problem Stamminger and Drettakis [59] suggest moving the camera's view point backwards until all possible occluders are inside the new view frustum (Figure 2.15 (right column)). This solves the problem but decreases the resolution of object's nearby the camera and can result in a uniform shadow map in extreme cases. Kozlov [33] has some further suggestions how to improve the quality of perspective shadow maps.

Perspective shadow maps fit very well to hardware like standard shadow maps do. It has to be mentioned that they must be drawn new whenever the camera, an object or the light moves.

2.4.2 Light Space Perspective Shadow Maps

Light Space Perspective Shadow Maps by Wimmer et al. [70] are based on the idea of *perspective shadow maps*. They observed that the perspective projection must not necessarily use the camera's view frustum to transform the scene in post-perspective space. Instead they introduced the *light space*.

Wimmer et al. [70] mentions some drawbacks of *perspective shadow maps* that mostly can be avoided by using *light space perspective shadow maps*:

- Transforming the light source into post-perspective space is not intuitive. Depending on the light/camera constellation the type of the light source changes from point light sources to directional and normal to inverted light sources in post-perspective space.

- As explained in Section 2.4.1 objects behind the camera plane but in front of the view plane of the light sources lead to problems (see Figure 2.15) and their solution decreases shadow quality.

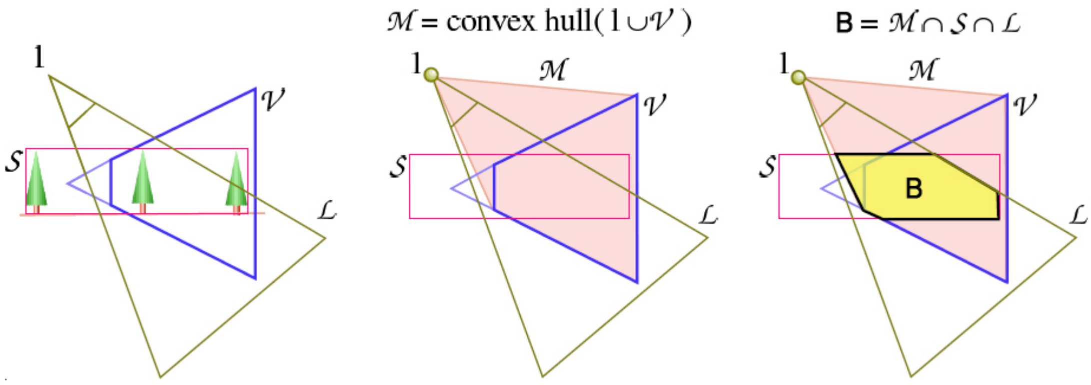

$\mathcal{M}$ = convex hull(l ∪ $\mathcal{V}$)
B = $\mathcal{M} \cap \mathcal{S} \cap \mathcal{L}$
l
$\mathcal{M}$
$\mathcal{V}$
$\mathcal{S}$
$\mathcal{L}$
B

Figure 2.16: This image illustrates the construction of the body B which includes all points that are important for the shadow map [59].

- Special cases must be implemented for practical use of *perspective shadow maps*.
- The shadow map resolution of objects far-away of the camera suffers from the high resolution of objects near the camera .

To compute light perspective shadow maps first of all the convex body B has to be generated (see Figure 2.16):

- Create convex hull $\mathcal{M}$ including the camera's view frustum $\mathcal{V}$ and the position of the light source l: $\mathcal{M} = l \cup \mathcal{V}$
- Intersect the hull $\mathcal{M}$ with the scene's bounding box $\mathcal{S}$ and the light's view frustum $\mathcal{L}$ to get the convex body B (yellow): $B = \mathcal{M} \cap \mathcal{S} \cap \mathcal{L}$

Now the convex body B has to be enclosed by a perspective frustum P where the frustum's view vector is parallel to the light's view plane. P can be constructed in *light space*. To define the light space at first the perspective transformation of the light has to be applied if the light source is a point light. This leads to a situation where a point light can be treated as a directional light source. The axes of the light space are defined as follows (see Figure 2.17):

y-axis: inverse light vector l

z-axis: perpendicular to the light vector l and on the plane containing the camera's view vector v and the light vector l

x-axis: completes the orthogonal coordinate system

The far and near plane of the perspective frustum P are perpendicular to the z-axis of the light space and the planes lie at the minimum and maximum extents of the convex body B along the z-axis in light space. The reference point of P is at:

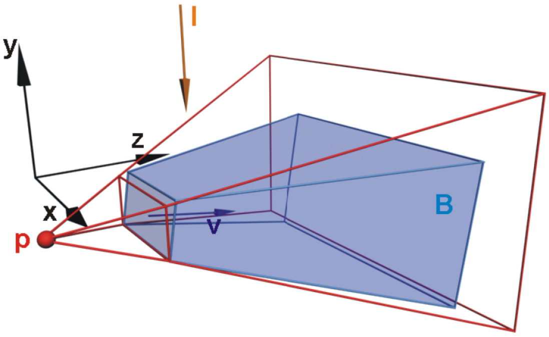


Figure 2.17: The construction of the perspective frustum P [70].

x: x of the camera's point of view in light space

y: Average of the minimum and maximum extents of B along the y-axis in light space.

z: Defined by the distance of the reference point to the near plane. The distance is a free parameter n of P. n defines the warp intensity of the shadow map. Wimmer et al. [70] suggest defining n as $n_{opt} = z_n + \sqrt{z_f z_n}$ where z_n is the near plane of the camera's view frustum and z_f the far plane of the camera's view frustum.

To complete the frustum P the missing frustum planes are obtained by projecting B on the near plane and using the minimum and maximum extents along the x- and y-axis in light space.

To generate the final shadow map the perspective projection of P in combination with the standard light transformation is applied. While rendering the scene from the camera's point of view as usual the same transformations are used for the lookup in the shadow map.

2.4.3 Trapezoidal Shadow Maps

Another method to avoid perspective aliasing are *Trapezoidal Shadow Maps* by Martin and Tan [44]. They are based on the idea of increasing shadow quality by minimizing wastage of the shadow map and are designed for light sources with view frustums that contain the whole camera's view frustum. The algorithm tries to map the camera's view frustum seen from the light's point of view in a way that it fills almost the complete shadow map.

One idea would be to approximate the view frustum by a bounding box. Approximating the view frustum by a bounding box does not lead to a minimized wastage. Martin and Tan [44] suggest a trapezoidal approximation of the camera's view frustum to minimize wastage. Figure 2.18 shows the improvement of a trapezoidal approximation compared to the bounding box approach.

Beside the mentioned wastage of shadow map space *trapezoidal shadow maps* handle a second type of shadow map wastage: the over-sampling of nearby objects. The resolution in the shadow map can be distributed flexible when using trapezoidal shadow maps.

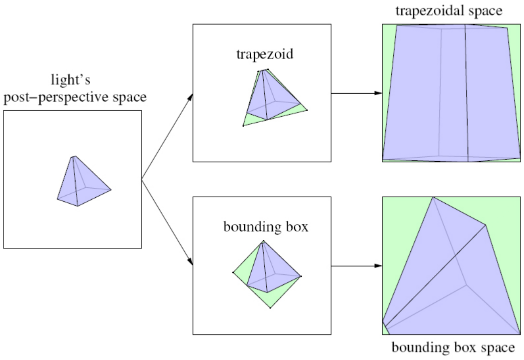


Figure 2.18: The view frustum seen from the camera is mapped to trapezoidal space (top) and to the smallest bounding box space (bottom) [44].

To create a trapezoidal shadow map the projected camera's view frustum on the shadow plane is transformed in the so called *trapezoidal space* $\mathcal{T}$ with the *trapezoidal transformation* N_T. The base, top and side lines of the trapezoid are computed to generate this transformation.

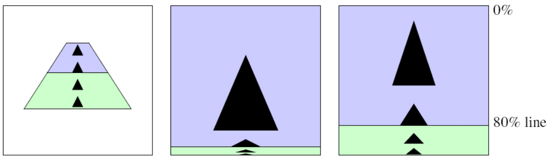
The base and top lines are calculated with the following steps:

1. Transform the camera's view frustum into the light's post-perspective space which results in E.
2. Compute the line l which includes the centers of the near and far plane of E.
3. Create the 2D convex hull of the transformed view frustum.
4. The center line l intersects the convex hull at two points. The base line l_b is perpendicular to l and goes through the intersection point that is closer to the near plane of E.
5. Also the top line l_t is perpendicular to l and goes through the intersection point closer to the far plane of E.

Martin and Tan [44] observed that a trapezoidal transformation wastes shadow map space caused by over-sampling. Therefore they suggest computing the side lines using an 80% rule to reduce this wastage. Figure 2.19 illustrates the improvement of applying the 80% rule compared to the unmodified trapezoidal transformation.

The side lines are computed as follows (see Figure 2.20):

1. A region of focus is defined by a distance δ from the camera's near plane.
2. p is a point at distance δ from the near plane and point p_L the transformed point in the light's post-perspective space on l.
3. δ' is the distance of p_L from the top line l_t.



(a) trapezoidal approximation in L (b) trapezoidal space (c) trapezoidal space due to the 80% rule

Figure 2.19: The trapezoidal approximation (a) mapped to trapezoidal space without (b) and with (c) the 80% rule [44].

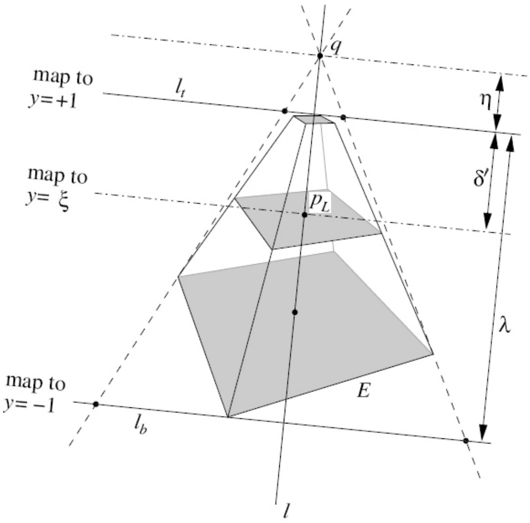


Figure 2.20: Illustration of the side-line computation [44].

4. The trapezoid transformation N_T should now be chosen in a way that p_L lies on the 80% line after applying the transformation. Therefore a perspective projection is constructed that fulfills this requirement.

5. The reference point q of this perspective projection lies on l and is obtained by calculating η which defines the distance between the top line l_t and q. The distance is $\eta = \frac{\lambda\delta' + \lambda\delta'\xi}{\lambda - 2\delta' - \lambda\xi}$ where λ is the distance between l_t and l_b and ξ defines the 80% line (i.e. $\xi = -0.6$).

6. The side lines of the trapezoid are defined as lines through q that touch the convex hull of E.

To create a *trapezoidal shadow map* the scene is transformed to the post-perspective space of the light first and then the trapezoidal transformation N_T is applied. The final rendering is done by transforming the fragment into the shadow map space as usual to obtain the shadow term.

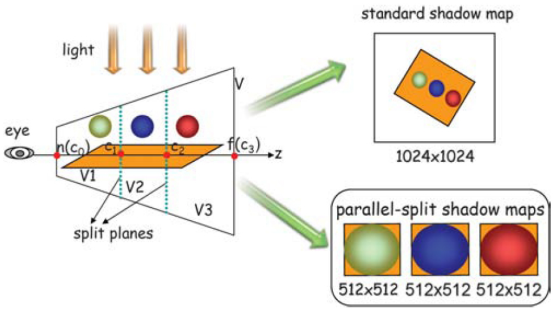


Figure 2.21: A scene is split into three parts (left). Instead of one single shadow map (right/top) three separate shadow maps are generated for each split part (right/bottom) [72].

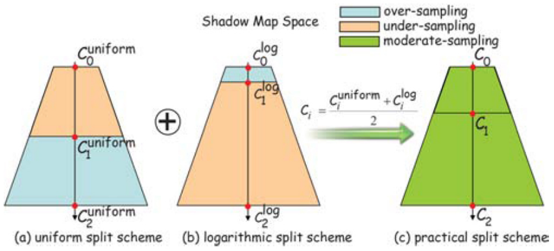


Figure 2.22: Illustration of the split scheme. A combination of a uniform split scheme (a) and a logarithmic split scheme (b) result in a practical split scheme (c) [72].

2.4.4 Parallel-Split Shadow Maps

Parallel-split shadow maps by Zhang et al. [72] also known as *Cascaded shadow maps* are designed to reduce perspective aliasing during shadow mapping in large scale environments. It is based on the idea of splitting the scene into several parts to generate more than one shadow map for the scene. One shadow map is generated for each part. The scene is split in a way that regions nearby the camera are covered by a more dense shadow map than regions in the background. The basic idea is illustrated in Figure 2.21.

Planes parallel to the camera's view plane at depths C_i with i as the index of the plane are used to split the scene along the camera's view frustum. The question is where to locate this split planes. Zhang et al. [72] observed that using a *logarithmic split scheme* would be the theoretically best choice and a *uniform split scheme* the worst case. Since *logarithmic split scheme* is not practicable Zhang et al. [72] suggests using a combination of the *logarithmic* and the *uniform split scheme* and call it the *practical split scheme* (see Figure 2.22).

The split depths C_i^{log} for the *logarithmic split scheme* are computed as:

$$C_i^{log} = n(f/n)^{i/m} \tag{2.11}$$

where n is the near plane of the view frustum, f the far plane and m the number of split parts.

$C_i^{uniform}$ for the *uniform split scheme* are computed as:

$$C_i^{uniform} = n + (f - n)i/m \tag{2.12}$$

Finally the depths C_i of the *practical split scheme* is the combination of C_i^{log} and $C_i^{uniform}$:

$$C_i = \frac{C_i^{log} + C_i^{uniform}}{2} \tag{2.13}$$

The light's frustum is then split into parts where each part focus on one part of the split view frustum. *Parallel split shadow maps* are created by doing one render pass for each part. This results in as many shadow maps T_i as the number of split parts. While rendering the final image shadow determination is done by doing a lookup in the shadow map corresponding to the fragments depth value. That means the lookup for a fragment with depth in the range $[C_{i-1}, C_i]$ is done in shadow map T_i.

Zhang et al. [72] suggest a number of three split parts and compared the visual results of their method with $3 \times 512 \times 512$ shadow maps to other single shadow map methods with a 1024×1024 shadow map. They observed a performance impact caused by the additional render passes but a much better quality of the shadows with even less memory consumption.

Ma et al. [42] improved the split algorithm by removing overlaps of the shadow maps T_i to avoid redundant rendering.

CHAPTER 3

Clustered Deep Shadow Maps for Multiple Volumes and Geometry Using CUDA

In this chapter the main contribution of this master thesis is discussed. Also the polyhedral volume renderer developed by Kainz et al. [29] which is used for this thesis is described. The generation of DSMs and how to use them while rendering are explained. This chapter will also discuss our clustering strategy which is used to increase shadow quality.

3.1 Polyhedral Volume Rendering

Kainz et al. [29] implemented a polyhedral-bounded volume renderer, which organizes objects in a scene graph and treats any scene graph object as a volumetric object with interior and polyhedral boundary. The interior part can either be a 3D texture or a constant color / opacity. This general representation combines the two worlds of the traditional rendering pipeline designed for boundary representations and direct volume rendering, which is typically limited to scenes represented by a single rectilinear grid. It is possible to render multiple intersecting objects at interactive frame rates. Figure 3.1 shows a rendering of two intersecting volumetric objects using the polyhedral volume renderer.

To render multiple volumes and intersect the volumes with geometry it is necessary to know which bounding polygons cover a pixel at which depth. Unlike previous approaches to clip volumes (Weiskopf et al. [65]) or intersect volumes with geometry (Termeer et al. [63] and Borland et al. [8]) the polyhedral volume renderer does not use depth peeling (Everitt [19]). Depth peeling is a technique to sort fragments by their depth using the depth buffer. The drawback of depth peeling is the fact that the rasterization step has to be repeated for the same geometry until all depth layers are found. Conventional shading languages are not flexible enough to do this in one pass. CUDA gives full control at every stage in the rasterization step. This makes it possible

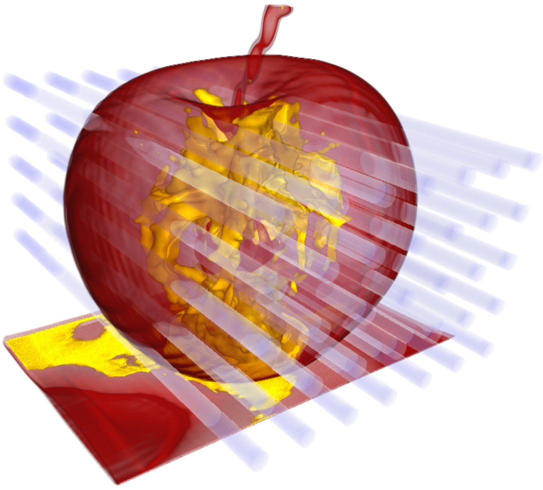


Figure 3.1: Two intersecting volumes rendered with the polyhedral volume renderer by Kainz et al. [29].

to collect information about the different depth layers with just one rasterization step. Depth peeling combined with dynamical shader generation to render multiple volumes is presented by Roessler et al. [53]. Brecheisen et al. [11] and Plate et al. [50] present multi-volume renderers as well.

Since a software implementation in CUDA gives full control over the rasterization and the raycasting stage, the renderer is well suited for the extension presented in this thesis: *Clustered Deep Shadow Maps* for multiple volumes and geometry.

The principle method of operating of the polyhedral volume renderer is illustrated in Figure 3.2. The renderer uses two separate CUDA kernels to generate the final image. The first one transforms the triangles to screen space and rasterizes them. The second kernel processes the rasterized triangles. This includes depth sorting of the rasterized triangles, raycasting and writing the final color to the display buffer.

The following sections give an overview of the CUDA kernels. Implementation details can be found in Chapter 4.

3.1.1 Rasterization in CUDA

The rasterization kernel performs parallel rasterization for each scene triangle, which means that each triangle is processed in a different thread. At first the processed triangle is transformed to screen space. Then the screen is split into tiles with $N \times N$ pixels each (in our case 8×8 pixels). For each tile which is intersected by the bounding box of the triangle a coverage mask [18] is computed. A coverage mask consists of 64 bits. Each bit stores the information whether the triangle covers the corresponding pixel or not.

The results of the triangle rasterization kernel are coverage masks which are stored together with the triangle IDs in global memory to make it available to the fragment kernel. The data

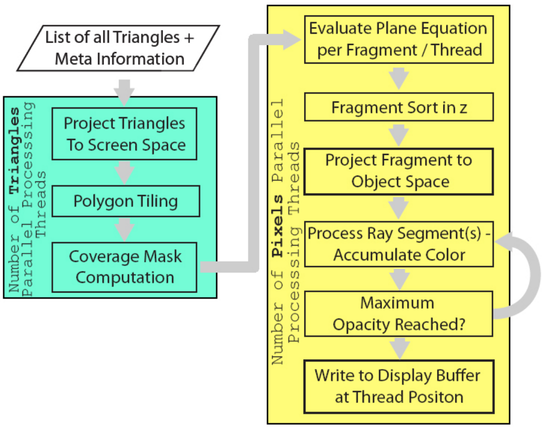

List of all Triangles + Meta Information
Number of Triangles Parallel Processing Threads
Project Triangles To Screen Space
Polygon Tiling
Coverage Mask Computation
Number of Pixels Parallel Processing Threads
Evaluate Plane Equation per Fragment / Thread
Fragment Sort in z
Project Fragment to Object Space
Process Ray Segment(s) - Accumulate Color
Maximum Opacity Reached?
Write to Display Buffer at Thread Positon

Figure 3.2: Illustration of the two kernels of the rendering system [29].

is organized in chunks. A chunk can store up to 64 pairs of coverage mask and triangle ID. A maximum number of 64 chunks is defined for each tile. This implies a total number of 4096 triangles per tile. Each tile stores a list of associated chunk indices which can be accessed from the fragment kernel.

3.1.2 The Fragment Kernel

The first step of the fragment kernel is to sort the triangles covering the given pixel by their depth. Therefore the depth of each fragment has to be calculated. This is done using the plane equation of each triangle. After sorting the fragments from near to far ray casting is performed.

The ray casting step is split into homogeneous segments. A homogeneous segment is the depth interval between two nearby depth sorted fragments. Each object within the processed homogeneous segment is sampled using the same step size. As a step size the minimum step size of each object in the current segment is used to avoid undersampling. If there are only polygonal objects within a homogeneous segment no ray casing is needed.

Depending on the rendering pass the samples are processed differently:

- In the DSM generation pass the opacity values are obtained from a transfer function using the sampled densities. The opacities of each object in the segment are blended. Opacities and depths are then used to generate the depth layers in the DSM.

- In the final rendering pass the sampled densities are also transformed with a transfer function to get colors and opacity values. The contribution of each sample to the final pixel is calculated using the Phong illumination model by Phong [49] including the shadow contribution from the DSM. The composite color is obtained by blending the shaded samples.

3.2 Deep Shadow Maps Generation

For each spotlight one DSM or six DSMs for an omnidirectional light source are generated as shown in Algorithm 3.1. If clusters are used (see Section 3.4) a DSM is generated for each cluster.

Algorithm 3.1 DSM Generation

```
set up the light's camera transformation
rasterize triangles (see Section 3.1.1)
for all pixels do
    (see Section 3.1.2)
    sort fragments by their depth
    for all homogeneous segments do
        build visibility function $V(z)$ and compress it on the fly as described by Lokovic and Veach [40] and Hadwiger et al. [25] (see Section 2.3.3 and 2.3.4)
    end for
end for
```

The first step to generate a DSM is to set up the light's camera transformation. If clusters are used the setup of the camera transformations for the DSMs is described in Section 3.4.2. If no clusters are used the scene is simple rendered from the light's point of view for spotlights. For omnidirectional light sources the scene is rendered from the light's point of view in each direction with a field of view of 90 degree each which results in six separate DSMs. Next the triangles are rasterized as described in Section 3.1.1.

Visibility Function Creation The fragment kernel sorts the rasterized triangles and processes the homogeneous segments. During the process of each homogeneous segment the compressed visibility function $V'(z)$ is created. A visibility function $V'(z)$ consists of several nodes where each node stores an opacity and a depth value. The visibility functions $V(z)$ defines the light attenuation for each depth along the ray from the light source through the scene. The compressed visibility function $V'(z)$ is an approximation of $V(z)$ with a reduced number of nodes. $V'(z)$ is a piecewise linear function where to following equation holds:

$$|V'(z) - V(z)| \leq \epsilon \quad \text{for all } z \tag{3.1}$$

where ϵ is the error tolerance.

The first homogeneous segment creates a node in $V'(z)$ with an opacity of 0 and the nearest fragment's depth. New nodes are created in $V'(z)$ different depending on the types of objects within the homogeneous segment.

If at least one volumetric object is within the current segment, the objects are sampled using ray casting. During the ray casting step, the opacity α_{sample} of each sample is accumulated ($= \alpha_{acc}$) and $length$ is increased by 1 for each sample.

$$\alpha_{acc} = \alpha_{acc} + \alpha_{sample} \cdot (1 - \alpha_{acc}) \tag{3.2}$$

While sampling the segment the compression step is performed. The idea of the compression algorithm (Algorithm 3.2) is to create the compressed function $V'(z)$ using as few line segments as possible without violating the error tolerance constrain.

Algorithm 3.2 Compression

```
take the first sample point as the origin of the current line segment
initialize the lower slope: $m_{lo} \leftarrow -\infty$
initialize the higher slope: $m_{hi} \leftarrow \infty$
repeat
    repeat
        $m_{prev_{lo}} \leftarrow m_{lo}$
        $m_{prev_{hi}} \leftarrow m_{hi}$
        sample next point
        update $\alpha_{acc}$ (Equation 3.2)
        set $m_{lo}$ (Equation 3.3)
        set $m_{hi}$ (Equation 3.4, see Figure 3.3 (b))
        $length \leftarrow length + 1$
    until $\alpha_{sample}$ of the current sample is outside of the slope wedge (Figure 3.3 (c))
    create a new node in $V'(z)$
    make this node the new origin
    set $m_{lo}$ (Equation 3.5)
    set $m_{hi}$ (Equation 3.6)
    $length \leftarrow 1$
until the whole homogeneous segment is sampled (Figure 3.3 (d))
```

At first the two slope values m_{lo} and m_{hi} are initialized with $-\infty$ and ∞. With each sample the previous slopes are stored in $m_{prev_{lo}}$ and $m_{prev_{hi}}$ before m_{lo} and m_{hi} are adjusted:

$$m_{lo} = max\left(m_{lo}, \frac{\alpha_{acc} - \epsilon - \alpha}{length}\right) \tag{3.3}$$

$$m_{hi} = min\left(m_{hi}, \frac{\alpha_{acc} + \epsilon - \alpha}{length}\right) \tag{3.4}$$

If $m_{lo} > m_{hi}$ a new node is created in $V'(z)$ with the depth z of the previous sample and an opacity of $\alpha = \alpha + \frac{(m_{prev_{lo}} + m_{prev_{hi}})}{2} \cdot length$. To find the next node in $V'(z)$ the slopes m_{lo} and m_{hi} are set to:

$$m_{lo} = \frac{\alpha_{acc} - \epsilon - \alpha}{length} \tag{3.5}$$

$$m_{hi} = \frac{\alpha_{acc} + \epsilon - \alpha}{length} \tag{3.6}$$

If there are just polygonal objects within a segment no ray casting has to be performed. Instead two new nodes are created in $V'(z)$. Both nodes have the same depth z which is the

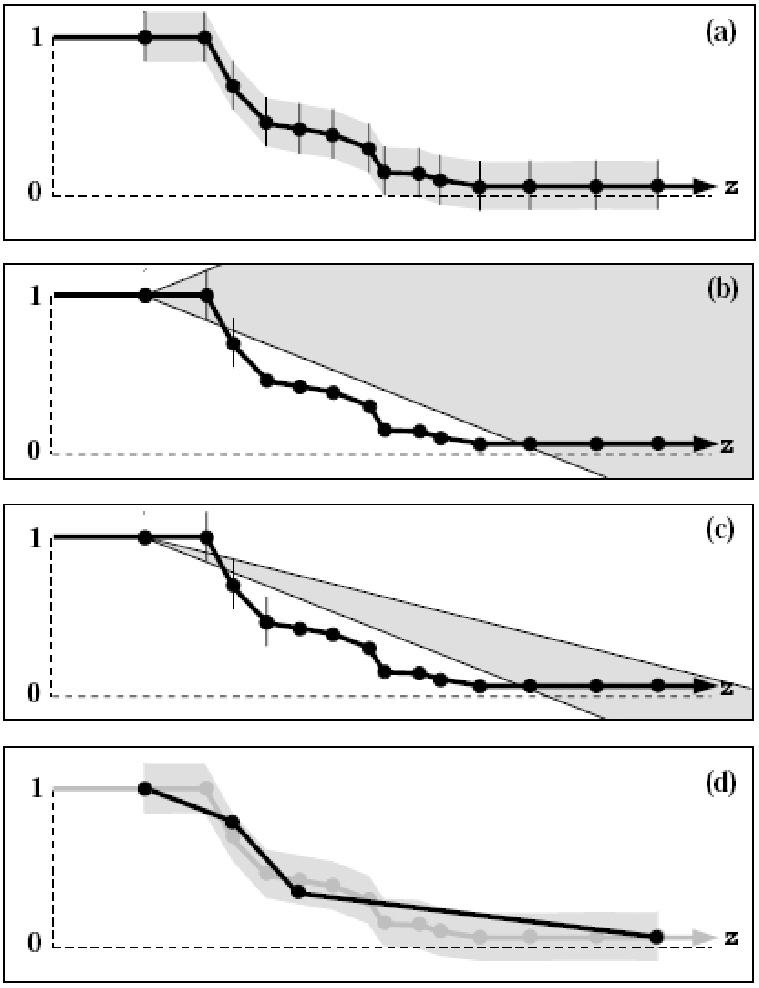


Figure 3.3: Illustration of the compression algorithm [40].

nearest depth of the two fragments describing the homogeneous segment. The first node stores the so far accumulated opacity value without taking the opacity of the polygonal object into account. The opacity of the second node is the accumulated opacity including the opacity of the polygonal object.

Deep Shadow Map Re-Generation In the case of not using clusters a generated DSM can be re-used in further frames until one of the following events occurs:

- The associated light source moves.
- The light source rotates.
- The light's spot cut off changes.

- The scene has changed (transforming objects, adding and removing objects to/from the scene).

If clusters are used (see Section 3.4) the DSM of a cluster may also has to be redrawn if the camera has changed and as a result also the cluster is different compared to the previous frame.

3.3 Rendering Using Deep Shadow Maps

While rendering the scene from the camera's point of view for the final image the generated DSMs are used as shown in Algorithm 3.3 to calculate the amount of light penetrating to the observed point.

Algorithm 3.3 Rendering

```
set up the camera transformation
rasterize triangles (see Section 3.1.1)
for all pixels do
    (see Section 3.1.2)
    sort fragments by their depth
    for all homogeneous segment do
        transform the sampled point to the light space of the light sources to look up light attenuations in the DSMs
        shade the sample point by each light source taking the light's attenuation into account
        sum up the individual shading results to get the final color of the sample which is then composite with the pixel's color
    end for
end for
```

The observed point is transformed into the light space as for standard shadow mapping (see Section 2.2.2). Instead of comparing the point's depth value with the value stored in the shadow map as for standard shadow mapping the depth value is used to find the two depth layers next to the observed point. The opacity values of these two layers are interpolated to obtain the *light attenuation*. If the depth value of the transformed observed point is greater than the depth value of the deepest layer in the DSM there is no interpolation needed. Then the light attenuation is just the opacity value of that deepest layer. On the other hand if the depth value is less than the depth of the first layer in the DSM the light attenuation is 0. The light attenuation value α is:

$$\alpha = \begin{cases} \alpha_1 \cdot \left(1 - \frac{z-z_1}{z_2-z_1}\right) + \alpha_2 \cdot \frac{z-z_1}{z_2-z_1} & \text{if } z > z_f \text{ and } z < z_d \\ \alpha_d & \text{if } z \geq z_d \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

where α_d is the opacity of the deepest layer in the DSM. α_1 and α_2 are the opacities of the two depth layers next to the observed point while z_1 and z_2 are their depth values. z is the depth of the observed point, z_f the depth of the first layer and z_d the depth of the deepest layer in the DSM.

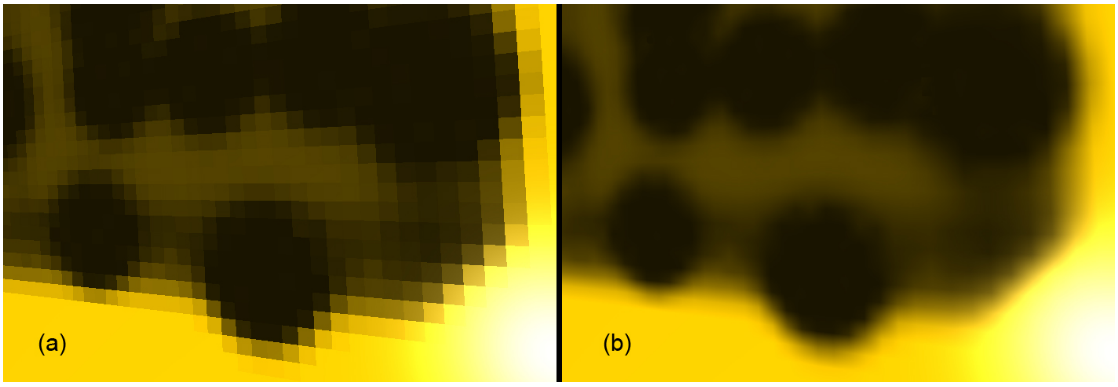


Figure 3.4: Image rendered without (a) filtering the shadow and with (b) filtering.

Filtering Calculating the light attenuation without filtering (=*nearest neighbor*) introduces steplike aliasing artifacts to the final image especially if the DSMs have low resolutions. But even DSMs with high resolution do not avoid these artifacts. Filtering drastically increases shadow quality.

On the one hand DSMs can be prefiltered during DSM generation and on the other hand they can be filtered while doing the lookup during rendering. For this thesis filtering the DSM during the rendering step is implemented.

Instead of doing one single lookup in the DSM the light attenuation is calculated by doing four lookups. The four opacity values of the surrounding DSM texels are taken to compute the weighted sum. This results in a bilinear interpolated light attenuation value. Figure 3.4 shows the difference between filtered and unfiltered shadows.

Shading The observed point p is than shaded using the Phong illumination model [49]. The specular and diffuse term are multiplied by the shadow term which is $1 - \alpha$ where α is the light attenuation value. The color I_p of p is:

$$I_p = k_a i_a + (k_d (L \cdot N) i_d + k_s (R \cdot V)^{n_s} i_s)(1 - \alpha) \tag{3.8}$$

where k_a, k_d and k_s are the ambient, diffuse and specular reflection constants of the material. n_s is the shininess of the material. i_a, i_d and i_s are the ambient, diffuse and specular intensities of the light source. L is the vector to the light source, N the normal vector of the shaded point, R is the reflection vector and V is the viewing vector.

Multiple light sources One aim of this thesis is to allow more than one single light source. It should be possible to have multiple spot lights and multiple omnidirectional light sources inside the scene. For that reason each point is shaded by each light source while the light attenuation is computed for each light source separately. The resulting color I_p is the sum of the contributions of all light sources:

$$I_p = k_a i_a + \sum_{m \,\in\, lights} k_d(L_m \cdot N) i_{d_m} + k_s(R_m \cdot V)^{n_s} i_{s_m})(1 - \alpha_m) \tag{3.9}$$

Sample Accumulation After shading, the samples are accumulated to the final color of the pixel according to Equation 3.10.

$$I_{acc} = I_{acc} + I_p \cdot I_{p_\alpha} \cdot t_{acc} \tag{3.10}$$

I_{p_α} is the opacity of the sample point and t_{acc} is the accumulated transparency:

$$t_{acc} = t_{acc} \cdot (1 - I_{p_\alpha}) \tag{3.11}$$

3.4 Clustering

To reduce perspective aliasing artifacts we create more than one shadow map with equal size for each light source. Each shadow map focuses on a separate part of the scene to optimize shadow map usage.

Similar to *Parallel-split shadow maps* (see Section 2.4.4) we reduce perspective aliasing artifacts and improve the shadow quality using multiple shadow maps. Instead of splitting the scene along the camera's view vector our approach clusters scene objects. For each cluster of objects a separate shadow map is generated while each shadow map has the same resolution. We do not use parameterized shadow maps like *Perspective Shadow Maps* (Section 2.4.1) or *Light Space Perspective Shadow Maps* (Section 2.4.2) because these methods require an update of the shadow map every frame and DSMs are computationally expensive. So we want to avoid a re-generation of the shadow map every frame.

Figure 3.5 illustrates the main idea behind our clustering algorithm. Figure 3.5 (a) shows an overview of the scene, the camera, the light source and the scene seen from the light's point of view. Aside from the fact that it's not a depth map a standard single shadow map would look similar to this view. The red circles in (b) shows two clusters of objects while (c) and (d) illustrates the particular shadow maps of each cluster. Figure 3.5 (e) compares the single shadow map with the shadow maps per cluster. As one can see the multiple shadow map approach decreases the wastage of shadow map texels and therefore increases the shadow quality.

The generation of multiple shadow maps is split into two steps. The first step is clustering, which groups together objects with little shadow contribution to the final rendering into studded clusters and objects with a big shadow impact into sparse clusters. The second step is to focus the shadow maps onto the clusters to optimize the shadow map usage. This is done by adjusting the lightcamera's orientation and the lightcamera's field of view as well as the far and near planes of the lightcamera.

These two steps lead to higher resolutions shadow maps of objects nearby the camera and/or the light source and lower resolutions shadow maps for objects far away from the camera and/or the light source. In addition the depth complexity of the individual shadow maps is higher than

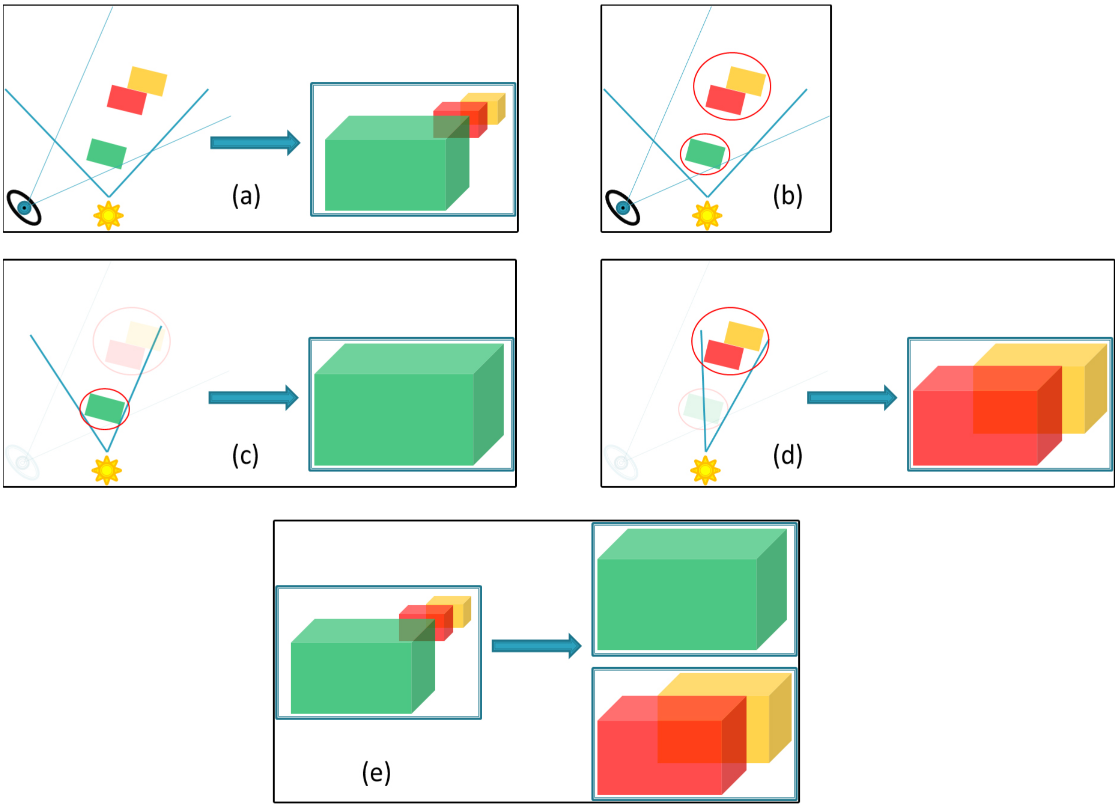


Figure 3.5: Illustration of the basic idea of clustering. (a) Overview of the scene and the corresponding single shadow map. (b) Nearby objects are clustered. (c) Focusing of the shadow map on the first cluster and on the second cluster (c). (d) Single shadow map compared to the two focused shadow maps.

using one single shadow map. An additional advantage of *Clustered Deep Shadow Maps* is the fact that omnidirectional light sources may require less than six shadow maps, depending on the scene configuration.

3.4.1 Building Clusters

In this section the steps to build clusters are explained. At first the bounding boxes of all objects are transformed by a *Light Space Perspective Transformation* and then the distances between these transformed bounding boxes are calculated. Based on the distances and a threshold objects are grouped together into clusters.

Light Space Perspective Transformation

The first step we do is to transform the bounding boxes of each object like we would do for *light space perspective shadow maps*. Therefore we have to compute the *light space perspective projection* as described by Wimmer et al. [70]. See Section 2.4.2 for an overview how to compute the light space perspective projection and Section 4.5 for implementation details.

Since the transformation is a perspective projection it is not sufficient to transform just the minimum and maximum extents of the bounding boxes. We have to transform each of the eight corner points of the boxes separately. Objects which do not intersect with the convex body B (see Section 2.4.2) can be ignored.

Figure 3.6 shows an example of this transformation step. While Figure 3.6 (a) gives an overview of the scene with its bounding boxes of the objects, the camera and the light source, (b) shows the resulting transformed bounding boxes.

Object Distance Calculation

In the next step the distances from each transformed bounding box to each other transformed bounding box is calculated. To calculate the distances between the bounding boxes we identify the two extreme points for each projected bounding box.

The first extreme point is defined by the maximum x and y extents of the projected bounding box and the second extreme point is defined by the minimum x and y extents of the projected bounding box analogously. The distance between two bounding boxes is then defined by the maximum of the distance between the minimum points and the distance between the maximum points.

The result of this step is a set of object pairs with their corresponding distance to each other.

Object Distance Calculation for Omnidirectional Light Sources

An omnidirectional light source is handled like six spot lights. So for omnidirectional light sources the described light space and distance calculations are done six times. One time for each direction.

The results are six distances for each pair of objects. The distance between two objects is then defined as the maximum distance of these six distances.

Cluster Generation

The distances between the projected bounding boxes are the basis for our cluster generation algorithm. One way to generate clusters from this point on would be to group all objects O_i whose bounding box distances are below a given threshold d_0. This would work if we had unlimited memory and computation resources for the clusters. But we want to limit the number of clusters to a defined number m.

So we use the following heuristic to generate clusters:

1. Sort pairs of objects (O_i, O_j) $(i \neq j)$ by their distances d_{ij}.

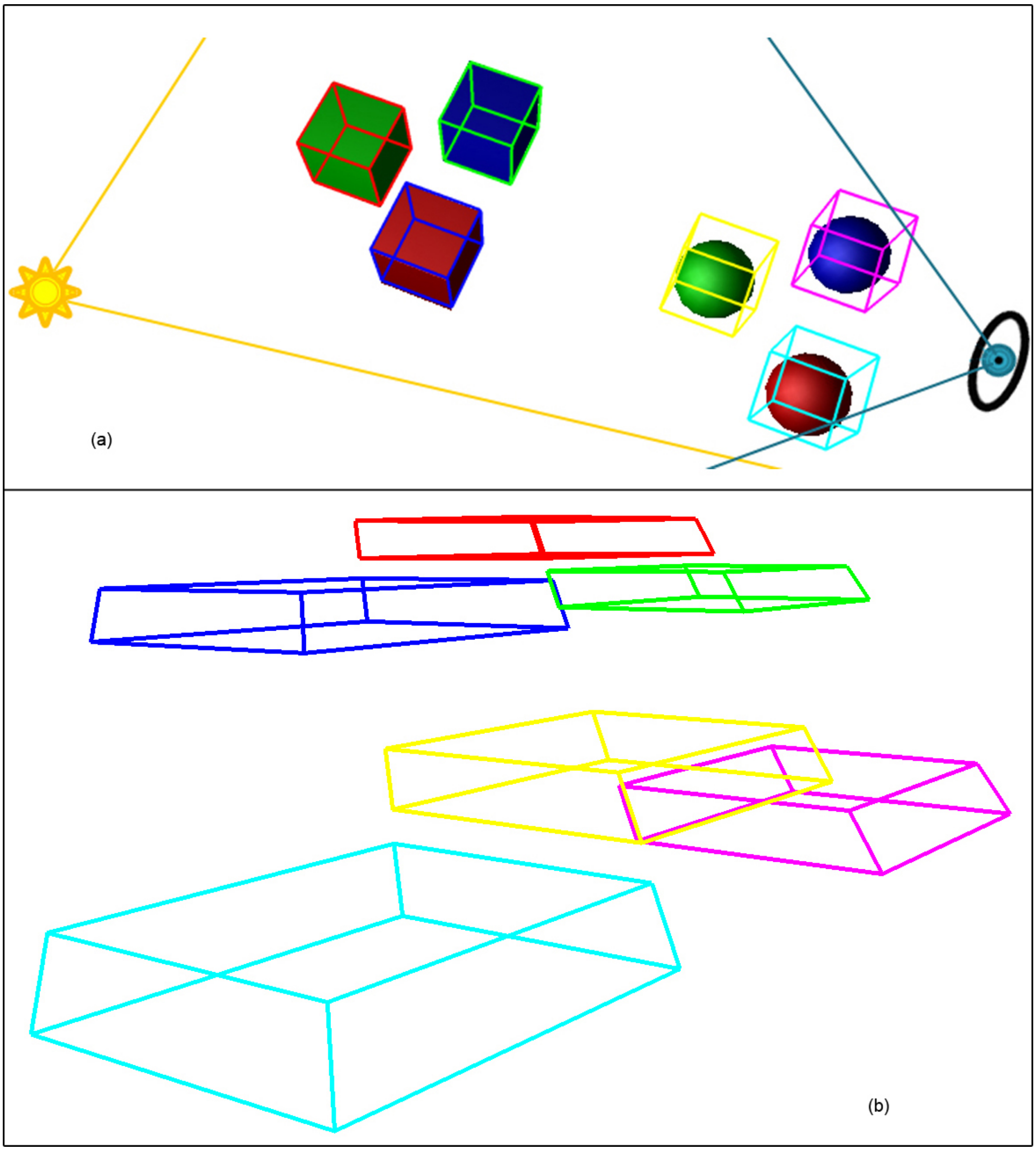


Figure 3.6: (a) Gives an overview of the scene including the bounding boxes of the objects, the position of the light source and the camera. (b) Shows the bounding boxes of the objects after the Light Space Perspective transformation.

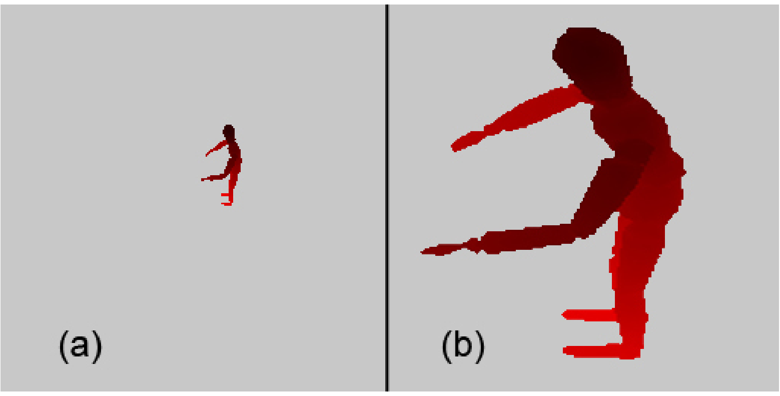


Figure 3.7: Unfocused (a) and focused (b) shadow map.

2. Insert objects O_i sequently into existing cluster C_k where $\forall O_j \in C_k : d_{ij} \leq d_0$.
   OR insert objects O_i sequently into new cluster C_l if $\exists O_j \in C_k : d_{ij} > d_0$ and $l \leq m$.
3. Insert remaining objects O_i into the cluster C_k with $\{max(d_{ij}) \mid O_j \in C_k\} \rightarrow min$.

3.4.2 Adjusting Light's Camera Parameters to Clusters

After the generation of clusters the shadow map should focus on the objects of the cluster to increase the shadow quality. Without focusing the shadow map, the clustering would just be wastage of resources. Figure 3.7 illustrates the difference between unfocused (a) and focused (b) shadow maps.

Focusing is done with the following steps:

1. Intersect the light's view frustum with the bounding box of the cluster if the light source is a spot light.
2. Narrow the frustum's field of view to the smallest possible angle in x- and y-direction where the whole (intersected) bounding box of the cluster is still inside the view frustum.
3. Change the orientation of the light's camera according to the narrowed field of view.
4. Let the near and far planes be at the minimum and maximum extents of the cluster.

An improvement of the focusing step would be the implementation of the algorithm by Low and Ilie [41] to maximize an object's image area. For this thesis their algorithm was not implemented.

3.4.3 Clustered Deep Shadow Map Generation

When rendering a DSM for a cluster only the objects within the cluster are rasterized. It is not necessary to generate *Clustered Deep Shadow Maps* every frame. The shadow map for a cluster has to be rendered again if one of the following cases occurs:

- The light source moves. The clusters have to be rendered from a different position.
- The light source rotates.
- The light's spot cutoff changes.
- The scene has changed (transforming objects, adding and removing objects to/from the scene).
- The cluster has changed which can happen if the camera has moved and the resulting light space perspective transformation leads to a new distribution of the objects in the clusters or because the focus of the shadow map changed.

If just the camera is moving in the scene the shadow maps can stay up to date for a while without re-rendering them. Especially if the camera doesn't move too far.

3.4.4 Use Clustered Deep Shadow Map During Rendering

Rendering the final image from the camera's point of view using clusters is similar to rendering the final image with multiple light sources. To obtain the light attenuation for a fragment the shadow contributions of all clusters of a light source are combined. For each cluster the light attenuation α_{m_c} is computed as described in Section 3.3 and accumulated according to Equation 3.12. This results in one shadow value which can be used the same way as the shadow value of one single DSM would be used. If a cluster does not affect an observed point, which means either the depth of that point is less than the depth value of the first layer in the DSM or the lookup is outside of the DSM, α_{m_c} is 0.

$$\alpha_m = \alpha_m + \alpha_{m_c} \cdot (1 - \alpha_m) \tag{3.12}$$

For spotlights it is additionally necessary to determine if the fragment lies inside the illuminated area of the spot or not. To check this we transform each sample as we would do for a single DSM.

CHAPTER 4

Implementation

This chapter contains a detailed description of the algorithms developed for this thesis. It starts with an explanation of the parts of the rendering engine by Kainz et al. [29] needed for deeper understanding of the extension - *Clustered Deep Shadow Maps*. This chapter also describes different memory management approaches. One method uses 3D textures and the other method uses chunks to store the DSMs. Their advantages and drawbacks are compared. In the sequel implementation details about the generation as well as the usage of DSMs are given. Finally the implementation of the clustering algorithm used for this thesis is described.

4.1 Polyhedral Volume Rendering

This section includes details about the used polyhedral volume renderer developed by Kainz et al. [29]. The implementation of the triangle rasterization step as well as the processing of the fragments is described.

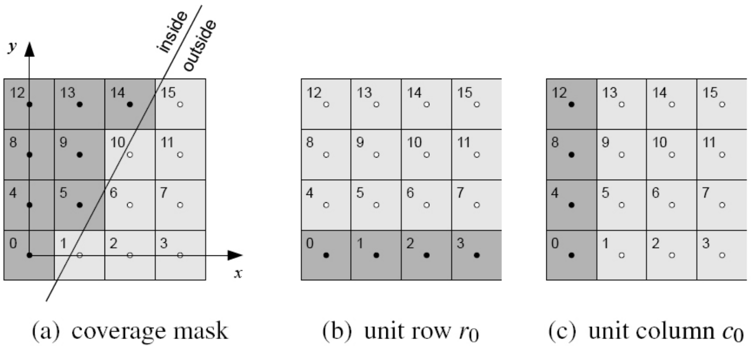
4.1.1 Rasterization in CUDA

In the triangle rasterization step $N \times N$ coverage masks are generated for each triangle. The coverage masks are computed using half planes and bit shift operations. Figure 4.1 illustrates the computation step of a 4×4 coverage mask.

Each edge of a triangle can be represented with a linear equation $ax + by + c = 0$. An edge of a triangle splits the coverage mask by a half plane where the outside of the half plane is defined as $ax + by + c > 0$. If a pixel lies inside the half plane its corresponding bit is set to one in the coverage mask. Otherwise it is set to zero.

Depending on a and b the coverage mask is obtained either by the usage of the unit row $r_0 = 0\text{x}000F$ or the unit column $c_0 = 0\text{x}1111$. If $a > 0$ and $|a| > |b|$ the number of set bits $n_1(y)$ in the row y is:

$$x = -\frac{by + c}{a}, \quad n_1(y) = max(0, min(\lfloor x \rfloor, N)) \tag{4.1}$$



(a) coverage mask (b) unit row r_0 (c) unit column c_0

Figure 4.1: Computation of the coverage mask (a) using shifting operations applied on the unit row r_0 (b) and the unit column c_0 [29].

For each row y the bit mask $r(y)$ is obtained using the two bit shift operators $>>$ (right shift) and $<<$ (left shift):

$$r(y) = (r_0 >> [N - n_1(y)]) << [Ny]) \tag{4.2}$$

The coverage mask m of the tile is the sum of the rows' bit masks:

$$m = \sum_{y=0}^{N-1} r(y) \tag{4.3}$$

If $a > 0$ and $|a| < |b|$ the coverage mask is computed column by column instead. The number of set bits $n_1(x)$ in the column x is:

$$y = -\frac{ax + c}{b}, \quad n_1(x) = max(0, min(\lfloor y \rfloor, N)) \tag{4.4}$$

The bit mask $c(x)$ for each column x is obtained using the two bit shift operators $>>$ (right shift) and $<<$ (left shift) again:

$$c(x) = (c_0 << [(N - n_1(x)) \cdot N]) >> [x]) \tag{4.5}$$

Analogous to the case before the coverage mask m of the tile is the sum of the columns' bit masks:

$$m = \sum_{x=0}^{N-1} c(x) \tag{4.6}$$

If $a \leq 0$ the coverage masks are obtained by symmetry and bit-wise inversion. The coverage masks are stored together with the associated triangle IDs in chunked memory.

4.1.2 The Fragment Kernel

For the fragment kernel the viewport is split into tiles of 8×8 pixels. This results in blocks of 64 pixels. For each pixel a thread is executed which results in 64 parallel running threads within a block. Each thread iterates through the triangles in the chunks generated during the rasterization step that belong to the thread's tile. If the bit of the pixel associated with the thread is set in the coverage mask of a triangle the depth of the triangle is computed using its plane equation. The Triangle ID and its depth is stored in an array of 32-bit values in sorted order. The triangle ID uses 16 bit and the depth also uses 16 bit of such a 32-bit value.

The array is then iterated from the nearest to the farthest triangle by each thread. A pair of two consecutive triangles in the array defines a homogeneous segment which is processed different depending on the rendering pass.

4.2 Memory Management

Memory management to store the depth layers of the *Clustered Deep Shadow Maps* was implemented in two different ways for this thesis. The first method uses 3D textures to store the individual depth layers (see Section 4.2.1). Section 4.2.2 describes the second method where the depth layers are stored using chunked memory.

4.2.1 3D Texture

This memory management method uses one big 3D texture atlas for all light sources. Each DSM is limited to a maximum of 64 depth layers (after compression). As texture format a float2 texture is used to store the depth in the first channel and the opacity in the second channel. Since textures are read-only we write the depth layers to *linear memory* and copy it to *CUDA arrays* which are then bound to textures.

The different DSMs are arranged in 3D grid. Each pair of depth and opacity is stored at the coordinates

x: the x coordinate of the DSM + x_{offset}

y: the y coordinate of the DSM + y_{offset}

z: the depth layer of the DSM + z_{offset}

Memory Management Without Using Clusters

The arrangement of the texture atlas and therefore the offsets x_{offset}, y_{offset} and z_{offset} are implemented in two slightly different ways depending on the application setting if clusters should be used or not. Figure 4.2 illustrates the arrangement of the texture atlas if no clusters are used. LSx are the light sources, while the numbers in brackets are the additional indices of the six views if the light source is omnidirectional.

At first the rows of the texture atlas are filled along the x-axis. If a row is full the next one is used. How many DSMs can be stored in one row depends on the dimensions of the shadow maps

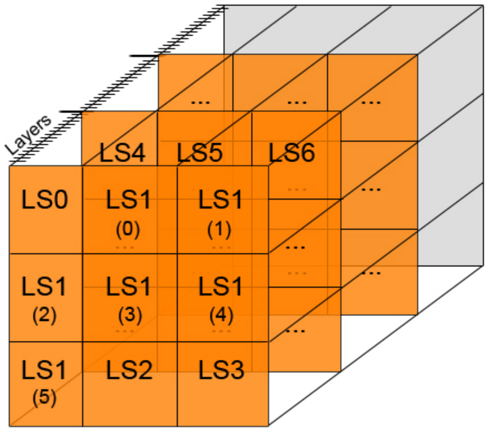


Figure 4.2: Illustration of the 3D texture atlas when no clusters are used. LSx are the light sources while the numbers in brackets are the additional indices of the six views of an omnidirectional light source.

and the maximum dimensions of 3D textures. In CUDA 2.3 3D textures must not be bigger than 2048 × 2048 × 2048. Using DSMs with dimensions of 512 × 512 × 64 implies a maximum of four shadow maps in one row. This also implies a maximum of four rows.

If all rows are full the following DSMs are arranged the same way but in the next z-layer. As mentioned a DSM is limited to a maximum of 64 depth layers in our case. Therefore the first layers of following shadow maps are located at $z = 64$ in the texture atlas.

The offsets for copying linear memory with depth layers into the texture atlas during DSM generation and reading out of it during rendering are computed as

$$\begin{aligned}
x_{offset} &= d_{size} \cdot \left((l_{offset} + c) \mod \frac{t_{max}}{d_{size}} \right) \\
y_{offset} &= d_{size} \cdot \left(\frac{l_{offset} + c}{\frac{t_{max}}{d_{size}}} \mod \frac{t_{max}}{d_{size}} \right) \\
z_{offset} &= d_{layers} \cdot \left(\frac{l_{offset} + c}{\left(\frac{t_{max}}{d_{size}}\right)^2} \mod \left(\frac{t_{max}}{d_{size}}\right)^2 \right)
\end{aligned} \tag{4.7}$$

where d_{size} is the dimension of the shadow maps in x- and y-direction and d_{layers} is maximum number of layers per DSM. t_{max} is the maximum allowed texture size in x-, y- and z-direction. l_{offset} is similar to an index of the light source of the current DSM. While an index usually increases by 1 l_{offset} increases by 1 in the case of a spotlight and by 6 in the case of an omnidirectional light source. Example: The first light source is a spotlight and has an l_{offset} of 0. The second one is an omnidirectional light source and has an l_{offset} of 1 while the third light

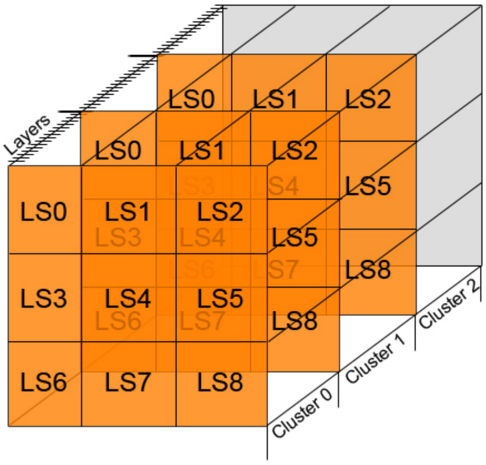


Figure 4.3: Illustration of the 3D texture atlas when clusters are used. LSx are the light sources.

source therefore has an l_{offset} of 7. c is the internal index of the DSM of the light source. For spotlights c can only be 0. For omnidirectional light sources c is between 0 and 5.

A DSM resolution of $512 \times 512 \times 64$ and a maximum texture size limited to $2048 \times 2048 \times 2048$ leads to a maximum number of theoretically 512 DSMs. Because of simplification and performance issues light sources are limited to a maximum of 8 in this thesis. Note that a spotlight requires one DSM slot while an omnidirectional light source requires six DSM slots. The size of allocated texture memory depends on the number of lights and their type. Having l_s spotlights and l_o omnidirectional light sources leads to an allocation of a 3D texture with the size of:

$$\begin{aligned} t_x &= d_{size} \cdot min\left(l_s + 6 \cdot l_o, \frac{t_{max}}{d_{size}}\right) \\ t_y &= d_{size} \cdot min\left(\left\lceil \frac{l_s + 6 \cdot l_o}{\frac{t_{max}}{d_{size}}} \right\rceil, \frac{t_{max}}{d_{size}}\right) \\ t_z &= d_{layers} \cdot min\left(\left\lceil \frac{l_s + 6 \cdot l_o}{\left(\frac{t_{max}}{d_{size}}\right)^2} \right\rceil, \frac{t_{max}}{d_{layers}}\right) \end{aligned} \tag{4.8}$$

Memory Management Using Clusters

Figure 4.3 illustrates the atlas arrangement if the DSMs are generated using clusters.

The texture atlas is filled nearly in the same way as described in Section 4.2.1. The main difference is that the clusters of each light source are arranged along the z-axis of the texture atlas. Therefore number of the maximum number of light sources is reduced because along the z-axis no further light sources can be stored. Another difference is that each light source is handled the same way and it makes no difference if it is a spotlight or an omnidirectional point light source.

The storage and lookup offsets are calculated as

$$\begin{aligned} x_{offset} &= d_{size} \cdot \left(l \mod \frac{t_{max}}{d_{size}} \right) \\ y_{offset} &= d_{size} \cdot \left(\frac{l}{\frac{t_{max}}{d_{size}}} \mod \frac{t_{max}}{d_{size}} \right) \\ z_{offset} &= d_{layers} \cdot c \end{aligned} \tag{4.9}$$

where d_{size} is the dimension of the shadow maps in x- and y-direction. t_{max} is the maximum allowed texture size in x-, y- and z-direction. l is the index of the current light source and c the internal cluster index of the current cluster.

Having a texture size limited to $2048 \times 2048 \times 2048$ and a DSM size of $512 \times 512 \times 64$ theoretically allows a maximum number of 16 light sources with 32 clusters each. Depending on the number of spotlights l_s and omnidirectional light sources l_o the 3D texture is allocated with a size of:

$$\begin{aligned} t_x &= d_{size} \cdot min \left(l_s + \cdot l_o, \frac{t_{max}}{d_{size}} \right) \\ t_y &= d_{size} \cdot min \left(\left\lceil \frac{l_s + \cdot l_o}{\frac{t_{max}}{d_{size}}} \right\rceil, \frac{t_{max}}{d_{size}} \right) \\ t_z &= d_{layers} \cdot c_{max} \end{aligned} \tag{4.10}$$

where c_{max} is the maximum number of clusters per light source.

Advantages and Drawbacks

Using 3D textures to store the depth layers has the following advantages:

- Aligning depth layers in 3D textures is intuitive.
- They allow fast access during rendering because of spatial coherence. Texture cache is utilized.
- Depth layers can be searched using a binary search algorithm.

There are also drawbacks when using 3D textures:

- Memory is wasted if the maximum number of depth layers is not used.
- The number of depth layers is limited.

4.2.2 Chunked Memory

The other memory management method implemented in this thesis uses chunked memory. In a chunk 64 pairs of depth and opacity values can be stored. Each value pair represents a depth layer in the DSM for one texel.

While generating the DSM the viewport is split into tiles of 8 × 8 pixels. This results in blocks of 64 parallel running threads. Each time the first thread of a block wants to store a depth layer a new chunk has to be allocated. A new chunk also has to be allocated if the current chunk is filled up which means 64 depth layers have been written from the same block of threads to the chunk.

Chunk Allocation:

Allocating a chunk means to determine the index of the chunk in c_d which is the array of all chunks. An array $b_l[b]$ counts for each block how many depth layers are already stored for the current block b. This value is incremented by 1 each time a depth layer is stored using the *atomicAdd* function of CUDA. $atomicAdd$ is an atomic function of CUDA which means it is guaranteed to be performed without interference with other threads [47]. More about atomic functions can be found in Section 1.4.2.

Knowing the maximum number of layers in a chunk ($L_C = 64$ in our case) allows calculating the current index b_c of the chunk within the block in which the depth layer should be stored:

$$b_c = \frac{b_l[b]}{L_C} \tag{4.11}$$

Beside b_c the following constants and variables are used to allocate a chunk:

C_B is a constant which defines the maximum number of chunks in a block.

C_{max} is a constant which defines the maximum number of chunks for all blocks.

g_c counts the number of allocated chunks including the number of chunks allocated by other blocks.

$b_g[b][b_c]$ is a two-dimensional array where b is the current block and b_c the index of the chunk within that block. The array stores the global indices to the chunks if they are already allocated.

The index of the chunk in c_d for the chunk b_c in block b is determined as shown in Algorithm 4.1.

The returned index can then be used to write into the associated chunk of the chunk array c_d. Since up to 64 layer can be stored in one chunk also the index of the layer b_{c_l} in this chunk has to be calculated. Therefore the bit-operator & is used:

$$b_{c_l} = b_l[b] \quad \& \quad (L_C - 1) \tag{4.12}$$

Algorithm 4.1 Allocate Chunk

```
if $b_c \geq C_B$ then
    no chunk can be allocated any more for this block, abort the chunk allocation
end if
if $b_g[b][b_c]$ already holds an index then
    return this index as the index of the chunk in $c_d$
else
    call the atomicAdd function of CUDA to increment $g_c$ by 1 and assign the value returned by the atomicAdd function to $chunk\_new$
    if $g_c \geq C_{max}$ then
        the maximum number of chunks for all blocks was reached and no more chunk can be allocated, abort the chunk allocation
    end if
    call the atomicCAS function (atomic Compare And Swap) to check if there is still no chunk index defined in $b_g[b][b_c]$ (could be defined in the meanwhile from another thread of the same block) and set $b_g[b][b_c]$ to $chunk\_new$ if this is the case
    return $b_g[b][b_c]$ as the index of the chunk in $c_d$
end if
```

Chunk Structure

A chunk in the chunk array consists of four float arrays with a size of L_C each:

$\boldsymbol{d}[]$ stores the depth of the depth layers.

$\boldsymbol{\alpha}[]$ stores the opacity of the depth layers.

$\boldsymbol{C_{next}}[]$ stores the indices of the next depth layers' chunks.

$\boldsymbol{L_{next}}[]$ stores the layer indices of the next depth layers within the chunk with the index stored in $C_{next}[]$.

This structure allows *coalesced writes* to the memory (see Section 1.4.1). The two arrays $C_{next}[]$ and $L_{next}[]$ are used to build a linked list of depth layers. Each depth layer has the information where to find the following depth layer without knowing the previous one.

The Chunk Texture

While the presented structure of a chunk allows *coalesced writes* a depth layer cannot be accessed during rendering the final image using a texture lookup with a structure like that. Therefore an additional CUDA kernel rearranges the depth layers in *linear memory* containing four-channel floats:

$[d_0, d_1, d_2, ..., d_{63}]$;
$[\alpha_0, \alpha_1, \alpha_2, ..., \alpha_{63}]$;
$[C_{next_0}, C_{next_1}, C_{next_2}, ..., C_{next_{63}}]$;
$[L_{next_0}, L_{next_1}, L_{next_2}, ..., L_{next_{63}}]$

$\rightarrow$

$[d_0, \alpha_0, C_{next_0}, L_{next_0}]$;
$[d_1, \alpha_1, C_{next_1}, L_{next_1}]$;
$[d_2, \alpha_2, C_{next_2}, L_{next_2}]$;
...
$[d_{63}, \alpha_{63}, C_{next_{63}}, L_{next_{63}}]$

This linear memory is then bound to a texture. This allows getting the depth and opacity values as well as the indices to the next layer with a single lookup for each depth layer. Given a chunk index c and a layer index l the associated depth layer is at the 1D texture coordinate:

$$u = 64 \cdot c + l \tag{4.13}$$

Every light has its own linear chunk memory and texture. If all lights would share the same chunk memory it would be hard to manage the regeneration of DSMs. For some light sources regeneration of the DSMs could be necessary and for others not within one frame. This would lead to fragmented chunked memory.

The Lookup Texture

To make the depth layers in the chunks accessible it is necessary to know where to find the first layer of each texel in the DSM. Therefore an additional lookup texture is used. The lookup texture stores the chunk indices as well as the layer indices of the first depth layers. For one DSM (e.g. for one spotlight) a 2D texture with two integer channels is sufficient. We use a 3D texture to allow more than one light source of different types (spotlights and omnidirectional light sources) and multiple clusters if clusters are used. The arrangement in the lookup texture is similar to the texture atlas arrangement when using 3D textures as the memory management method (Section 4.2.1). The indices to the first depth layer are stored in the 3D texture at the coordinates:

x: the x coordinate of the DSM + x_{offset}

y: the y coordinate of the DSM + y_{offset}

z: z_{offset}

Lookup texture arrangement without using clusters: If no clusters are used the offsets for writing into the lookup texture while generating the DSMs and reading during rendering the final image are:

$$\begin{aligned}
x_{offset} &= d_{size} \cdot \left((l_{offset} + c) \mod \frac{t_{max}}{d_{size}} \right) \\
y_{offset} &= d_{size} \cdot \left(\frac{l_{offset} + c}{\frac{t_{max}}{d_{size}}} \mod \frac{t_{max}}{d_{size}} \right) \\
z_{offset} &= \left(\frac{l_{offset} + c}{\left(\frac{t_{max}}{d_{size}}\right)^2} \mod \left(\frac{t_{max}}{d_{size}}\right)^2 \right)
\end{aligned} \tag{4.14}$$

For the lookup texture a 3D texture with the following dimensions has to be allocated:

$$\begin{aligned}
t_x &= d_{size} \cdot min\left(l_s + 6 \cdot l_o, \frac{t_{max}}{d_{size}} \right) \\
t_y &= d_{size} \cdot min\left(\left\lceil \frac{l_s + 6 \cdot l_o}{\frac{t_{max}}{d_{size}}} \right\rceil, \frac{t_{max}}{d_{size}} \right) \\
t_z &= min\left(\left\lceil \frac{l_s + 6 \cdot l_o}{\left(\frac{t_{max}}{d_{size}}\right)^2} \right\rceil, t_{max} \right)
\end{aligned} \tag{4.15}$$

Lookup texture arrangement when using clusters: In the case of using clusters the offsets are:

$$\begin{aligned}
x_{offset} &= d_{size} \cdot \left(l \mod \frac{t_{max}}{d_{size}} \right) \\
y_{offset} &= d_{size} \cdot \left(\frac{l}{\frac{t_{max}}{d_{size}}} \mod \frac{t_{max}}{d_{size}} \right) \\
z_{offset} &= c
\end{aligned} \tag{4.16}$$

The size of the 3D texture is:

$$\begin{aligned}
t_x &= d_{size} \cdot min\left(l_s + \cdot l_o, \frac{t_{max}}{d_{size}} \right) \\
t_y &= d_{size} \cdot min\left(\left\lceil \frac{l_s + \cdot l_o}{\frac{t_{max}}{d_{size}}} \right\rceil, \frac{t_{max}}{d_{size}} \right) \\
t_z &= c_{max}
\end{aligned} \tag{4.17}$$

Advantages and Drawbacks

The advantages of using chunked memory to store the depth layers are:

- The usage of coalesced writes allows a fast generation of DSMs.
- Since chunks of memory for depth layers are allocated dynamically when they are needed there is no wastage of memory.
- The number of depth layers of a DSM texel is not limited. As long as there is free memory left new chunks for depth layers can be allocated.

Disadvantages of this method are:

- There is no spatial coherence of the depth layers which slows down the lookups caused by cache misses.
- Depth layers cannot be searched using a binary search algorithm. Every lookup requires a linear search.

4.3 Deep Shadow Map Generation

During the DSM generation pass a visibility function is created for each texel in a DSM. The visibility function for a texel in the DSM is created and compressed as described in Section 3.2. Depending on the memory management method (see Section 4.2) the nodes are stored differently.

4.4 Rendering Using Deep Shadow Maps

While rendering the final image the rays are spilt into homogeneous segments as mentioned. For each light source in the scene the samples along the ray through a homogeneous segment are transformed to the light space of the associated DSMs. The next step is to check if the observed sample is within the illuminated area of the current DSM. If this is the case the two adjacent depth layers have to be found in the DSM for each sample as also described in Section 3.3. This two depth layers are then used to calculate the light attenuation for the sample in a further step.

Transform sample to light/cluster space: A sample is transformed to light space (or cluster space if clusters are used) as usual with a matrix. The matrices of all light sources or clusters are stored successively in linear memory.

Check if the sample is within the illuminated area: We do not care if the sample point lies in the shadow or not. We just want to know if the sample would be illuminated if there would be no shadow at all. In the case of omnidirectional light sources all sample points are within the illuminated area. If the light source is a directional spot light source the samples inside of the spot cone are in the illuminated area. Given the lookup coordinates

x, y, z and w in light space the sample is within the illuminated area if $w > 0$ and (x, y) lies within the circular mask of the DSM.

If clusters are used, the sample must be transformed to light space to determine if it is within the illuminated area or not, in addition to the transformation of the sample to cluster space.

Modify the lookup coordinates: If the sample is within the illuminated area the lookup coordinates x, y and z are modified to access the right DSM in the texture atlas. Depending on the memory management method the offsets are calculated either using Equations 4.7, 4.9, 4.14 or using Equations 4.16 to modify the lookup coordinates.

Find the two adjacent depth layers for a sample: Finding the two depth layer where the sample point's depth is in between of these two layers is done different depending on the memory management method:

3D textures If the depth layers are stored in 3D textures spatial coherence is given since one depth layer is stored after each other and additional the neighboring DSM texels are stored directly next to each other. This spatial coherence is used to speed up the search algorithm. When the first sample of a pixel is processed a binary search is used to find the corresponding depth layers in the DSM. All further samples start their search at the result of the last sample. To find the right two depth layers for further samples we search linearly forward and backward through the depth layers. The algorithm assumes that shadow values for nearby samples can be found in similar depth layers which is mostly the case.

Chunked memory If chunked memory is used as the memory management method no spatial coherence is given. It is possible to do a linear search forward through the depth layers and with additional information (bi-directional linked list) it would also be possible to search backwards. But the visibility functions of nearby DSM texels are not necessarily stored next to each other. It is not possible to conclude from a depth layer to another depth layer of a nearby DSM texel. As a result it is not possible to initialize the depth layer lookup using a binary search at the first sample of a pixel as for the 3D texture memory management method. So the two adjacent depth layers have to be searched for each sample separately, which slows down the shadow calculation.

Filtering: To filter the shadow terms we determine the opacities of four neighboring texels in the DSM and build the weighted sum of them. The original lookup coordinates (u, v) are altered as follows to get the four lookup coordinates:

$$\begin{aligned}
u_0 &= \lfloor l_x \rfloor + 0.5 \\
v_0 &= \lfloor l_y \rfloor + 0.5 \\
\\
u_1 &= \lfloor l_x \rfloor + 1.5 \\
v_1 &= \lfloor l_y \rfloor + 0.5 \\
\\
u_2 &= \lfloor l_x \rfloor + 0.5 \\
v_2 &= \lfloor l_y \rfloor + 1.5 \\
\\
u_3 &= \lfloor l_x \rfloor + 1.5 \\
v_3 &= \lfloor l_y \rfloor + 1.5 \qquad (4.18)
\end{aligned}$$

where $l_x = u - 0.5$ and $l_y = v - 0.5$. The corresponding weights are:

$$\begin{aligned}
w_0 &= (1 - l_x + \lfloor l_x \rfloor)) \cdot (1 - l_y + \lfloor l_y \rfloor)) \\
w_1 &= (l_x - \lfloor l_x \rfloor) \cdot (1 - l_y + \lfloor l_y \rfloor)) \\
w_2 &= (1 - l_x + \lfloor l_x \rfloor)) \cdot (l_y - \lfloor l_y \rfloor) \\
w_3 &= (l_x - \lfloor l_x \rfloor) \cdot (l_y - \lfloor l_y \rfloor) \qquad (4.19)
\end{aligned}$$

Figure 4.4 illustrates the lookup coordinates (a) and filter weights (b).

4.5 Clustering

This section describes the implementation of the clustering. Algorithm 4.2 is performed for each light source.

Light Space Perspective Transformation

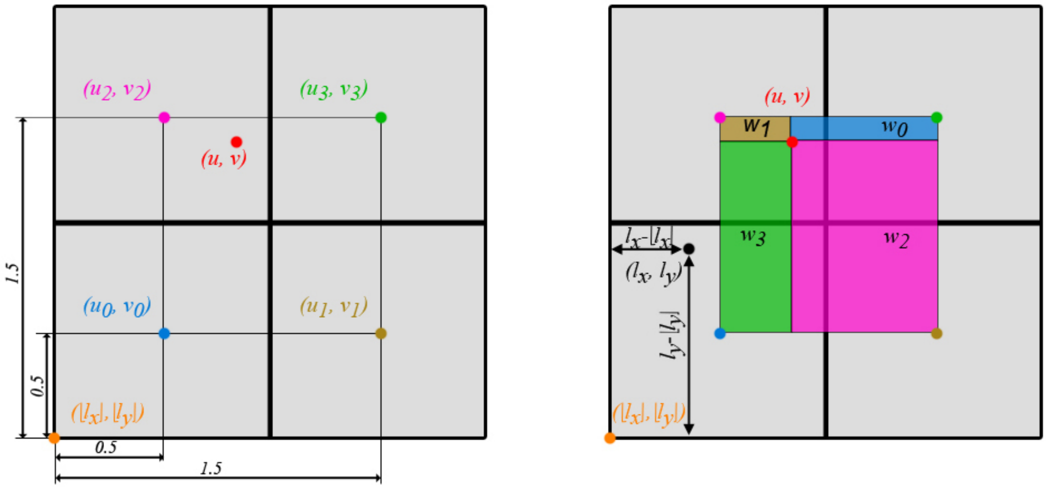
To generate the *Light Space Perspective* transformation at first the convex hull $\mathcal{M}$ has to be found. The convex hull $\mathcal{M}$ includes the camera's view frustum $\mathcal{V}$ and the position of the light source l ($\mathcal{M} = l \cup \mathcal{V}$). $\mathcal{M}$ is then intersected by the light's view frustum $\mathcal{L}$ and the scene's bounding box $\mathcal{S}$ to get the convex body B as described by Wimmer et al. [70] (see Section 2.4.2).

Classes of the library *Coin3D* [62] are used in this thesis to compute the convex body B. Especially the class *SbClip* is used. *SbClip* allows clipping polygons against planes. B is created with the Algorithm 4.3.

Lines 10 and 12 of Algorithm 4.3: Clipping polygons that form a closed body against planes can lead to holes in this body. Therefore a new polygon is added to the list of polygons in such a case to close that hole. The polygon to add is the convex hull of the polygon that includes all vertices resulting at the sectional plane. To create the convex hull of a polygon the *Three Coins*

Algorithm 4.2 Clustering

```
for all directions $dir$ (one for spotlights and six for omnidirectional light sources) do
    set up the light space perspective transformation matrix.
    transform the bounding boxes of each object in the scene which affects the shadow by the matrix created in (1a).
    calculate the distances between each transformed bounding box to each other.
    put the object pairs and their distances into a list if $dir$ is the first direction. If the distance between two objects is bigger than the distance of previous directions overwrite their distance in the list.
end for
{the result of the former steps is a set of object pairs associated with their distance to each other}
sort the list by the distances descending
for all pairs ($A$ and $B$) of nodes in the sorted list do
    for all existing clusters ($C$) do
        if the distance of $A$ to each other node in $C$ is less than a defined threshold $d_0$ then
            put $A$ into $C$
        end if
        if the distance of $B$ to each other node in $C$ is less than a defined threshold $d_0$ then
            put $B$ into $C$
        end if
    end for
    if no cluster is found for node $A$ or node $B$ and the maximum number of cluster has not been reached then
        if the distance between node $A$ and node $B$ is less than the defined threshold $d_0$ and no cluster was found for both then
            create a new cluster and put node $A$ and node $B$ into this new cluster
        else
            if node $A$ has not already found a cluster then
                put node $A$ into a new cluster
            end if
        end if
        if node $B$ has not already found a cluster then
            if the maximum number of clusters has not been reached then
                put node $B$ into a new cluster
            else
                put node $B$ into the nearest cluster
            end if
        end if
    else if no cluster is found for node $A$ and/or node $B$ and the maximum number of cluster has been reached then
        find the nearest cluster for node $A$ and/or $B$ (for the one or both nodes which have not already found a cluster)
    end if
end for
for all clusters do
    if the light source has moved or the cluster is different from the same cluster of the last frame then
        adjust the light's camera parameters to cluster
        generate DSM
    end if
end for
```


(u_2, v_2)
(u_3, v_3)
(u, v)
(u_0, v_0)
(u_1, v_1)
$(\lfloor l_x \rfloor, \lfloor l_y \rfloor)$
1.5
0.5
0.5
1.5
(u, v)
w_1
w_0
w_3
w_2
$l_x - \lfloor l_x \rfloor$
(l_x, l_y)
$l_y - \lfloor l_y \rfloor$
$(\lfloor l_x \rfloor, \lfloor l_y \rfloor)$

(a) (b)

Figure 4.4: Illustration of the lookup coordinates (a) and the computation of the corresponding weights (b) for bilinear filtering.

Algorithm 4.3 Computation of the Convex Body B

```
create a list $c[]$ and fill it with polygons, the polygons are the planes of the camera's view frustum $\mathcal{V}$
if the light source's position $l$ does not lie inside of the camera's view frustum $\mathcal{V}$ then
  for all planes of the camera's view frustum do
    if $l$ is outside of the half plane then
      insert four new triangles into $c[]$ where one vertex of the triangles is always $l$ and the other two vertices are sequentially the corner's of the current view frustum plane
    end if
  end for
end if
create a list $p_{\mathcal{S}}[]$ of planes where the planes are aligned to each side of the scene's bounding box $\mathcal{S}$.
clip all polygons of $c[]$ against all planes of $p_{\mathcal{S}}[]$ (in the clipping step new polygons are added to $c[]$ and others may are removed completely)
create a list $p_{\mathcal{L}}[]$ and fill it with the light's view frustum planes
clip all polygons of $c[]$ against all planes of $p_{\mathcal{L}}[]$ (again in the clipping step new polygons are added to $c[]$ and others may are removed completely)
the remaining polygons in $c[]$ belong to $B$
```

algorithm by Graham [23] is used.

With B the perspective frustum P as well as the light space can be obtained as described in Section 2.4.2.

Transform the Bounding Boxes and Sort Them by the Distances to Each Other

The corners of the bounding boxes of each object in the scene that affects the shadow are transformed by multiplying them with the light space perspective transformation matrix. Objects do not affect the shadow if they do not intersect with the body B. After calculating the distances between the transformed bounding boxes the object pairs are sorted by their distance.

Find the Nearest Cluster

To search for *the nearest cluster* of a node we evaluate for each cluster the maximum distance to each other node in the respective clusters. *The nearest cluster* is then the cluster with the shortest maximum distance to the current node.

Choosing the Distance Threshold d_0 and the Maximum Number of Clusters

The distance threshold d_0 is the maximum distance between each object in a cluster to each other object in the same cluster (as long as the maximum number of clusters is not reached). Choosing the distance threshold is a tradeoff between quality, memory consumption and speed. A lower threshold results in more clusters.

For the maximum number of cluster it is the same as for the threshold d_0. Defining the maximum is a tradeoff between quality, memory consumption and speed. More clusters lead to more DSMs and therefore to more memory consumption and performance impacts.

CHAPTER 5

Results

In this chapter the results of this thesis are presented. Visual improvements as well as memory consumption and performance are compared. The test-device has the following configuration:

CPU: Intel Quad Core i7 920 at 2.66 GHz

RAM: 6GB

GPU: NVIDIA GeForce GTX480

Operating System: Windows Vista 64Bit

All medical datasets used in this chapter are kindly provided by *The Ludwig-Boltzmann Institute - clinical-forensic imaging*.

5.1 Deep Shadow Maps and Conventional Shadow Maps

Figure 5.1 shows an old VW Beetle [15] smoking out of its engine compartment. The scene is illuminated by a spotlight and rendered using conventional shadow maps (Figure 5.1 (a)) as well as with DSMs (Figure 5.1 (b)).

Using a conventional shadow map, as shown on the left image, allows to distinguish between illuminated and unlit areas only. It is impossible to define a region as being partially in the shadow since there is no way to determine the amount of light penetrating to the observed point. As a result the shadow of the smoke looks very unnatural. With DSMs the amount of light penetrating to an observed point can be determined and therefore regions like the one in the shadow of the smoke are more natural on the right image. Furthermore the improvement of DSMs is also noticeable by looking at the shadows of semitransparent objects like the windows in this scene. On the right image the windows produce a brighter shadow compared to the rest of the car. In contrast the shadows of the windows and the shadow of the car are not distinguishable on the left image.

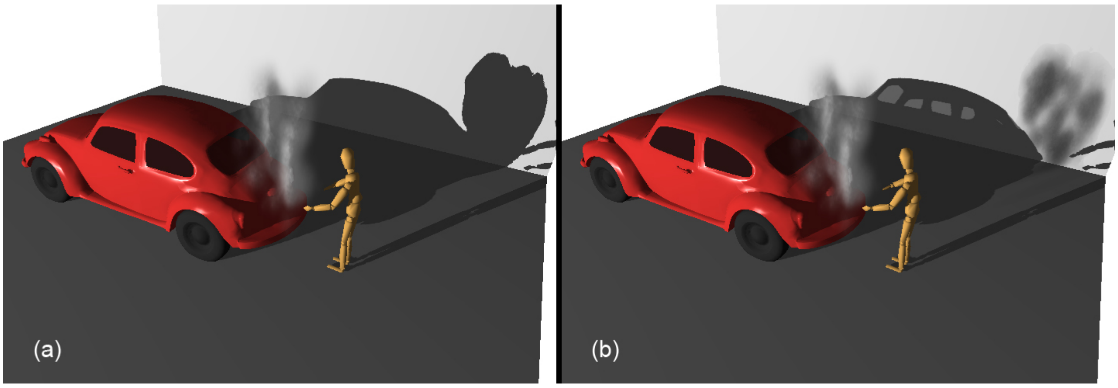


Figure 5.1: A scene rendered with conventional shadow maps (a) and with DSMs (b).

5.2 Shadow Interaction of Volumes and Geometry

An important aim of this thesis is to enable shadow interaction between objects with volumetric character and objects with polygonal geometry. Figure 5.2 shows a scene including fog and an orange car intersecting the fog.

While the fog casts shadows on the car also the shadows casted from the orange car on the fog are clearly visible. Figure 5.2 shows that shadow interaction is even possible for intersecting objects with the implemented method.

5.3 Shadows of Multiple Volumes

Figure 5.3 shows a scene with two VW Beetles. Both are smoking out of their engine compartment which leads to multiple volumes in the scene.

The shadows of the two volumes on the wall overlap at some parts. Considering the shadow of the smoke on the wall the shadow is bright on the left, gets darker in the middle and again bright on the right. The left part is the result of the smoke produced by the blue Beetle while the right part is the result of the smoke produced by the green Beetle. The region with the overlapping shadows appears darker since more light is absorbed of light rays passing both volumes compared to the rays passing only one smoke volume.

5.4 Multiple Spot Lights and Omnidirectional Light Sources

Figure 5.4 shows a scene with a teapot and a teacup. The teapot as well as the teacup is steaming. The scene is illuminated by a spotlight and an omnidirectional light source. The spotlight is placed on the left side and the omnidirectional light source is placed between the teapot and the teacup (represented by the white sphere).

As shown by the image the implementation supports multiple light sources of different types.

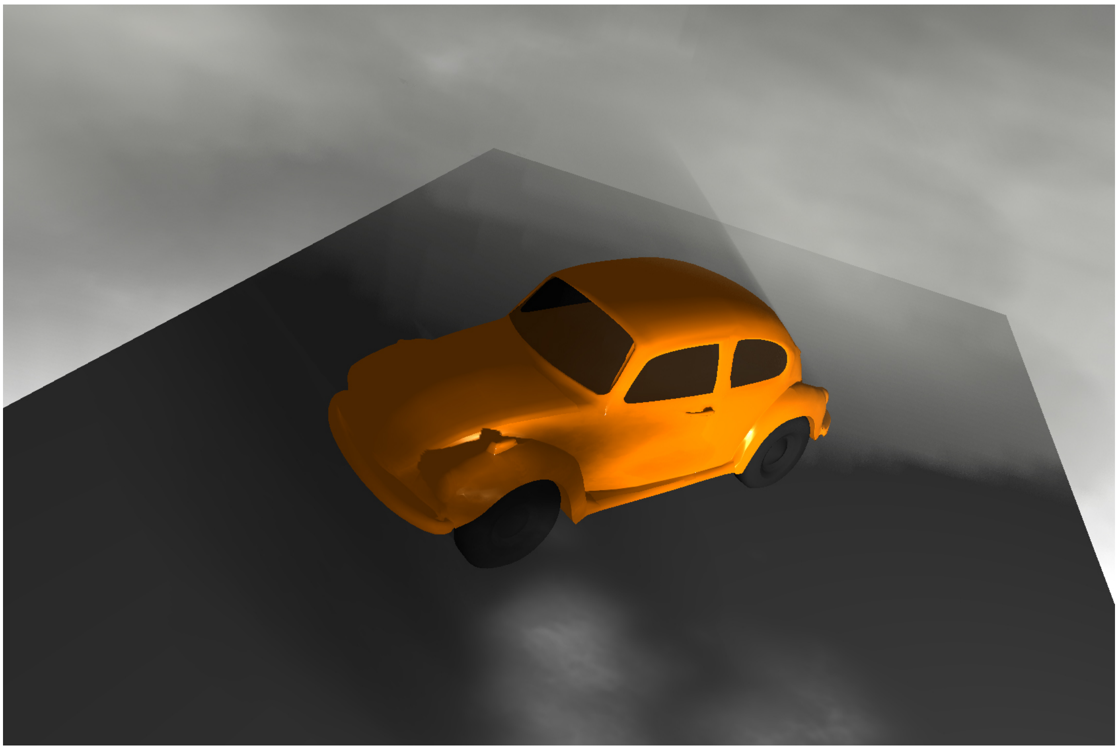


Figure 5.2: An orange car in a foggy scene. You see shadows casted by polygonal geometry on the volume and vice versa.

5.5 Clustering

To improve shadow quality more than one single DSM is generated for a scene. The objects of a scene are clustered depending on the position of the light source and the position of the camera. For each cluster a separate DSM is generated.

Figure 5.5, Figure 5.6 and Figure 5.7 show the same scene from three different points of view. In each of the three figures the left image (a) is the result without using clusters. That means just one single shadow map is used. The right image (b) shows the result when clusters are used. In this case the number of clusters is limited to four. Note that the single DSM has a resolution of 1024×1024 while each of the four separate DSMs has a resolution of 512×512. This makes the results comparable since both methods use the same amount of memory for the shadow computation. Figure 5.8 illustrates how the objects are clustered for (a) the setting of Figure 5.5, (b) the setting of Figure 5.6 and (c) the setting of Figure 5.7. Each object is colored according to its corresponding cluster.

The clustered result of Figure 5.5 includes a lot less stair-stepping artifacts than its analog result without clusters. The same applies to the results shown in Figure 5.6. Comparing Figure 5.8 (a) with 5.8 (b) makes clear how the clustering algorithms depends on the point of view.

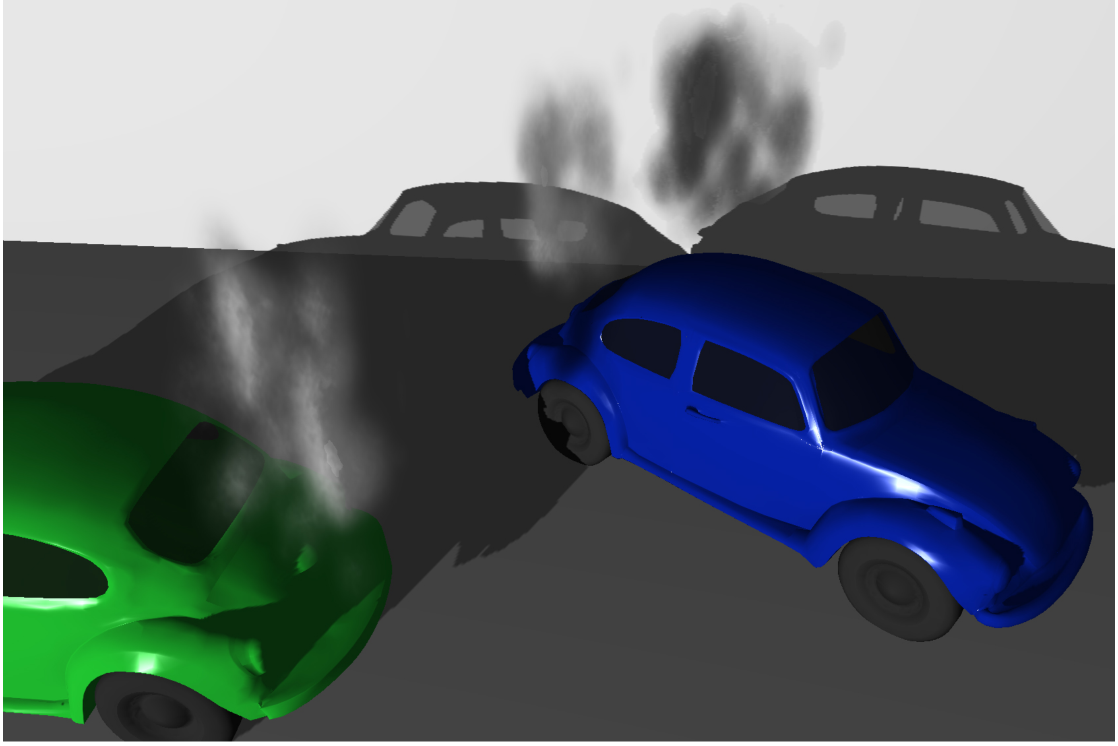


Figure 5.3: Scene including multiple volumes. Overlapping shadows result in darker shadows.

The algorithm tries to keep the clusters near the camera small (the red cluster in (a) and the green cluster in (b)). Between the two results of Figure 5.7 is no noticeable difference.

Using clusters does not only reduce stair-stepping artifacts but also increases depth quantization which leads to less self-shadowing artifacts.

5.6 Filtering

Filtering highly improves shadow quality especially for volumetric objects. Figure 5.9 shows the difference between filtered and unfiltered shadows. The resolution of the DSMs is chosen very small on purpose to enforce the effect of filtering. The filtered shadow of the cup still suffers from stair-stepping artifacts but the shadow of the steam looks much better when it’s filtered.

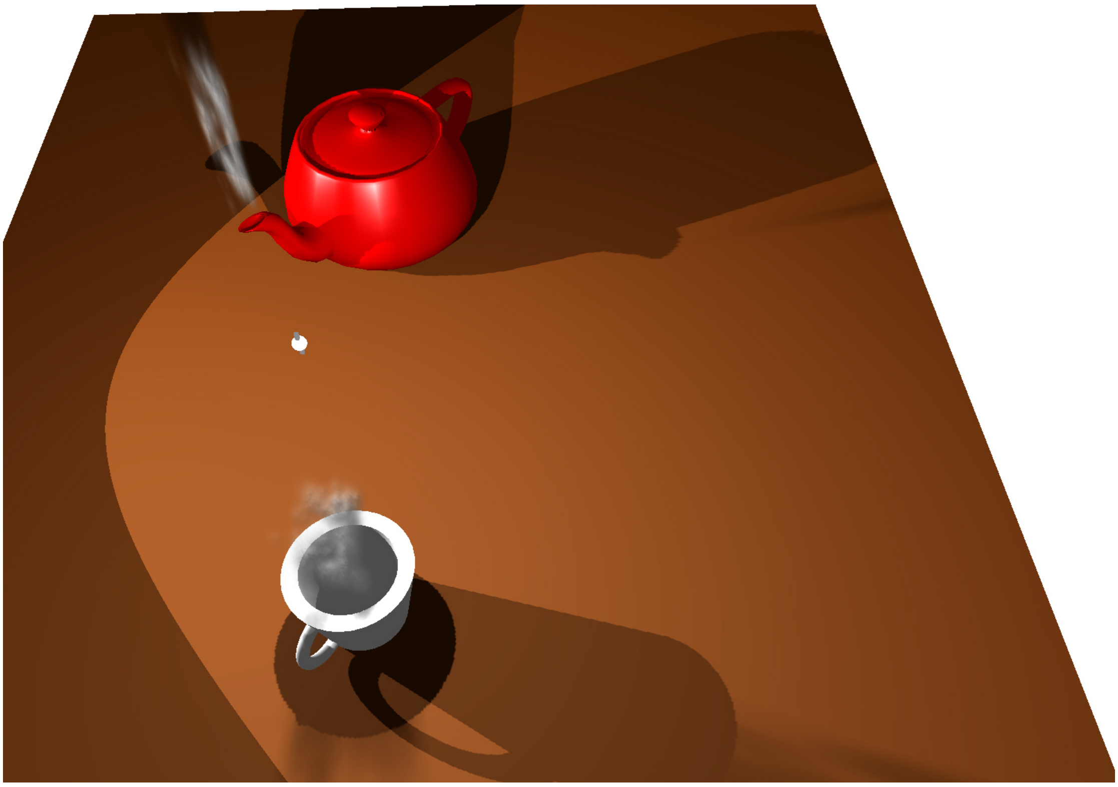


Figure 5.4: Scene is illuminated by one spotlight and one omnidirectional light source.

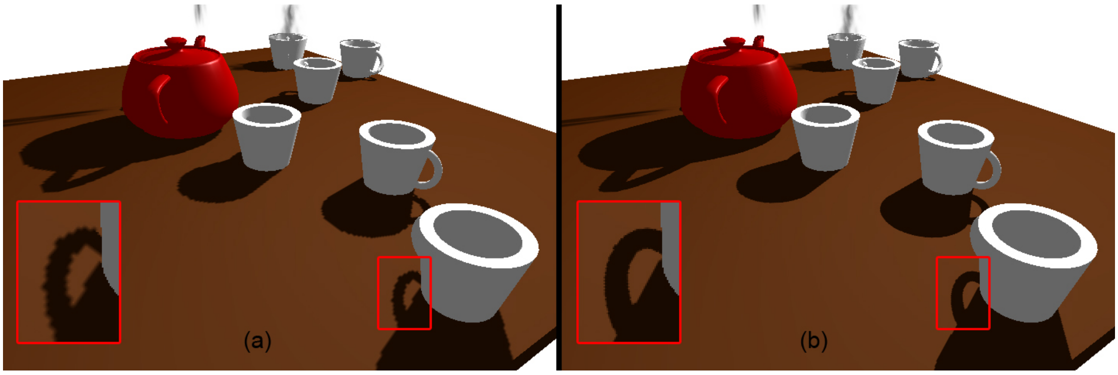


Figure 5.5: Rendering of a scene without clustering (a) and with clustering (b). The red rectangle shows a close-up of the shadow next to the camera.

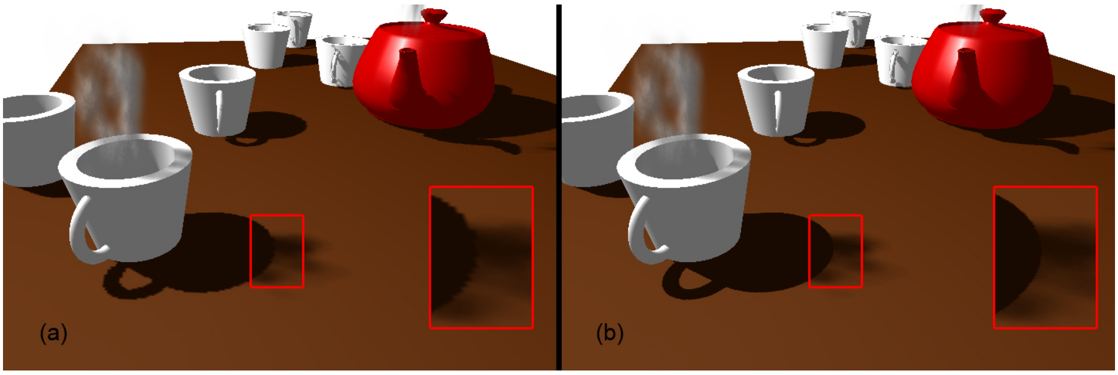


Figure 5.6: Same scene as in Figure 5.5 from another point of view without clustering (a) and with clustering (b). The red rectangle shows a close-up of the shadow next to the camera.

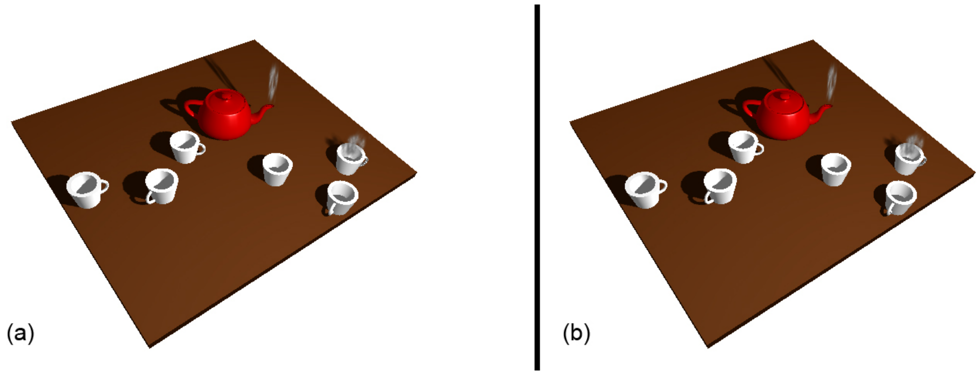


Figure 5.7: Same scene as in Figure 5.5 from another point of view without clustering (a) and with clustering (b).

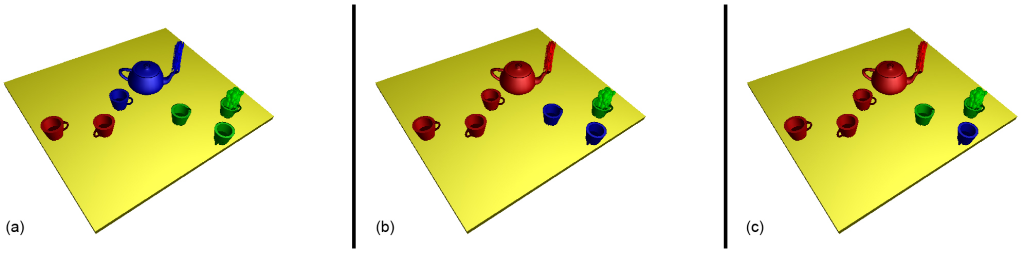


Figure 5.8: Illustrates the results of the clustering algorithm. Objects in the same color belong to the same cluster. (a) Clustering result corresponds to setting of Figure 5.5. (b) Clustering result corresponds to setting of Figure 5.6. (c) Clustering result corresponds to setting of Figure 5.7.

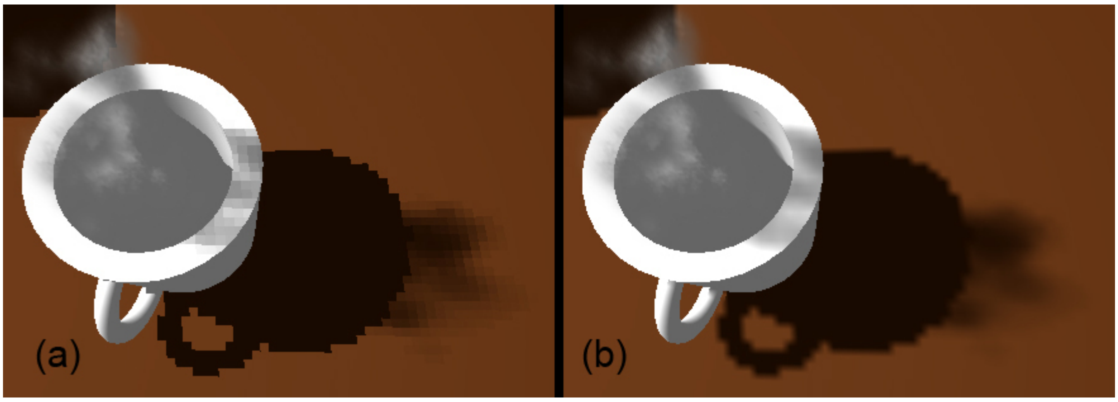


Figure 5.9: Rendering without filtering the shadow (a) and with filtering the shadow (b).

5.7 Performance

To test the performance of the implementation for this thesis the impact of different parameters on computation times is analyzed. The varying parameters are:

- DSM resolution
- volumetric dataset resolution
- error tolerance ϵ
- number of objects

5.7.1 DSM Resolution

The scene shown in Figure 5.6 is rendered with different resolutions of the DSM and the time for generating the DSMs as well as the rendering times (filtered and not filtered) are summarized in Tables 5.1, 5.2, 5.3 and 5.4. The scene consists of one spotlight, two volumetric objects and eight polygonal objects. The scene is rendered with a resolution of 720×480. Completely without shadows the rendering takes about 43ms.

For Table 5.1 3D Textures are used as the memory management method and clustering is disabled. DSM generation times and rendering times (filtered and not filtered) are measured for DSMs with the resolution 256×256, 512×512 and 1024×1024.

The resolution of DSM has a high impact on the DSM generation times and effectively no effect on the rendering times. Filtering slightly increases rendering times.

For Table 5.2 also 3D Textures are used as the memory management method but this time clustering is enabled. Four clusters are used and the DSM resolutions are $4 \times 128 \times 128$ (comparable to the 256×256 single DSM), $4 \times 256 \times 256$ (comparable to the 512×512 single DSM) and $4 \times 512 \times 512$ (comparable to the 1024×1024 single DSM).

[ms]	256×256	512×512	1024×1024
DSM generation	11	31	65
Rendering (not filtered)	46	46	46
Rendering (filtered)	54	53	53

Table 5.1: Computation times in [ms] for varying DSM resolutions. 3D Textures are used as the memory management method. Clustering is deactivated.

[ms]	$4 \times 128 \times 128$	$4 \times 256 \times 256$	$4 \times 512 \times 512$
DSM generation	21	40	80
Rendering (not filtered)	54	55	57
Rendering (filtered)	70	71	72

Table 5.2: Computation times in [ms] for varying DSM resolutions. 3D Textures are used as the memory management method. Clustering is activated.

Comparing the computation times from Table 5.2 with the computation times from Table 5.1 shows that clustering increases DSM generation times as well as rendering times.

The computation times in Table 5.3 reflect the performance when Chunked Memory is used as the memory management method and clusters are disabled. Because no clusters are used the DSM resolution are 256×256, 512×512 and 1024×1024 again.

[ms]	256×256	512×512	1024×1024
DSM generation	7	17	54
Rendering (not filtered)	48	49	49
Rendering (filtered)	56	56	57

Table 5.3: Computation times in [ms] for varying DSM resolutions. Chunked Memory is used as the memory management method. Clustering is deactivated.

The results in Table 5.3 compared to the results in Table 5.1 show that DSMs can be generated faster using Chunked Memory instead of 3D Textures. However the rendering is slightly slower.

For Table 5.4 Chunked Memory is used and clustering is enabled. DSM resolutions are $4 \times 128 \times 128$, $4 \times 256 \times 256$ and $4 \times 512 \times 512$. Clustering again increases computation times for DSM generation and rendering times.

[ms]	$4 \times 128 \times 128$	$4 \times 256 \times 256$	$4 \times 512 \times 512$
DSM generation	17	30	79
Rendering (not filtered)	55	56	57
Rendering (filtered)	69	69	70

Table 5.4: Computation times in [ms] for varying DSM resolutions. Chunked Memory is used as the memory management method. Clustering is activated.

5.7.2 Volumetric Dataset Resolution

To test the impact of the volumetric dataset resolution on DSM generation and rendering times the virtual X-Ray scene shown in Figure 5.10 is used. The scene consists of one volumetric and six polygonal objects and it is rendered with a resolution of 720×480. The resolutions of the volumetric dataset vary from $512 \times 512 \times 96$, $256 \times 256 \times 48$ to $128 \times 128 \times 24$. The test is done for both memory management methods. Table 5.5 contains the results when 3D Textures are used and Table 5.6 summarizes the results for Chunked Memory. In Table 5.7 the average as well as the maximum number of layers per DSM texel depending on the dataset resolution are listed.

[ms]	$512 \times 512 \times 96$	$256 \times 256 \times 48$	$128 \times 128 \times 24$
DSM generation	84	64	52
Rendering (filtered)	570	252	113

Table 5.5: Computation times in [ms] for varying volumetric dataset resolutions. 3D Textures are used as the memory management method ($4 \times 256 \times 256$ DSM).

[ms]	$512 \times 512 \times 96$	$256 \times 256 \times 48$	$128 \times 128 \times 24$
DSM generation	74	55	52
Rendering (filtered)	846	340	130

Table 5.6: Computation times in [ms] for varying volumetric dataset resolutions. Chunked Memory is used as the memory management method ($4 \times 256 \times 256$ DSM).

The resolution of the volumetric dataset has a high impact on rendering times. The influence on DSM generation times is noticeable. As shown in Table 5.7 higher resolutions of the dataset result in more layers per DSM texel.

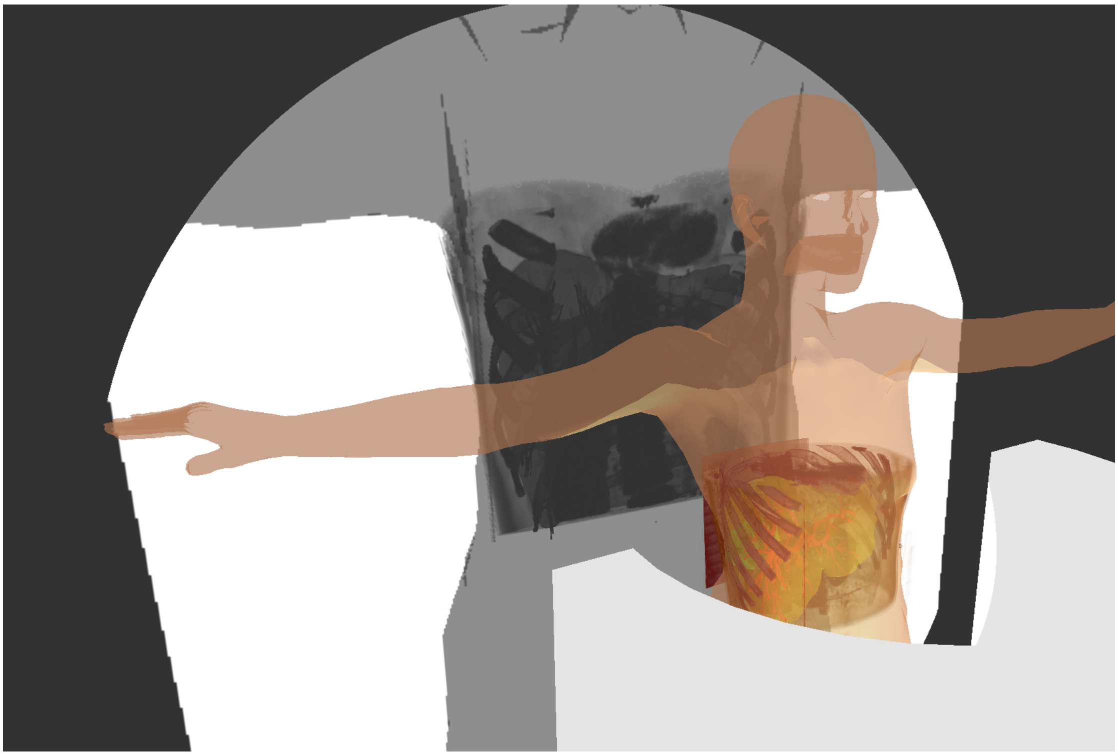


Figure 5.10: A CT scan is projected on the wall in the background.

	$\mathbf{512 \times 512 \times 96}$	$\mathbf{256 \times 256 \times 48}$	$\mathbf{128 \times 128 \times 24}$
Av. # of layers / DSM texel	3.74	3.64	3.64
Max. # of layers / DSM texel	31	27	24

Table 5.7: The number of layers / DSM texel depends on volumetric dataset resolution.

5.7.3 Error Tolerance ϵ

To analyze the performance impact of the error tolerance ϵ on DSM generation and rendering times the scene shown in Figure 5.11 is rendered with different ϵ values and a resolution of 720×480. The test is performed with ϵ values of 0.01, 0.05, 0.1 and 0.2. Table 5.8 shows the results with 3D Textures used as the memory management method. Chunked Memory is used for the results shown in Table 5.9. In Table 5.10 the average as well as the maximum number of layers per DSM texel depending on the error tolerance are listed.

Since the error tolerance ϵ influences the number of nodes in the visibility functions of the DSMs but not the sample rate during DSM generation the rendering is faster with a higher ϵ and the time differences for DSM generation are minimal. Especially the rendering times when

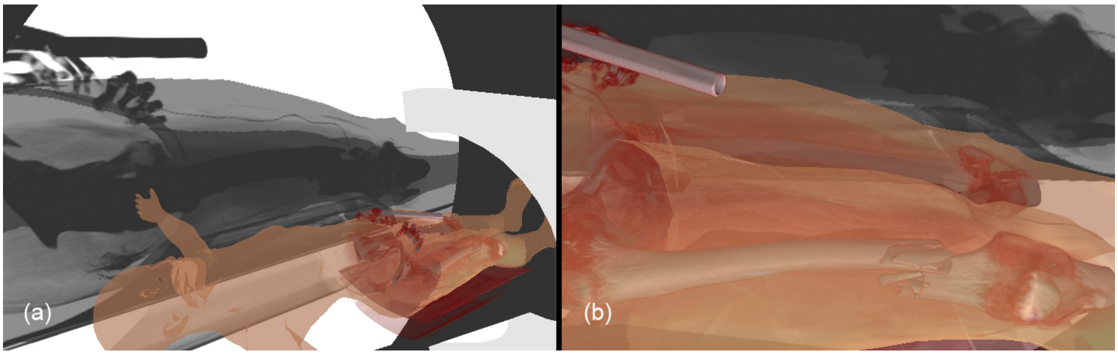


Figure 5.11: (a) Overview of a virtual X-Ray. (b) Close-up of the same scene. The CT-scan is projected on the wall in the background.

[ms]	$\epsilon = 0.01$	$\epsilon = 0.05$	$\epsilon = 0.1$	$\epsilon = 0.2$
DSM generation	70	69	66	65
Rendering (filtered)	166	159	153	149

Table 5.8: Computation times in [ms] for varying error tolerance ϵ. 3D Textures are used as the memory management method ($4 \times 256 \times 256$ DSM).

[ms]	$\epsilon = 0.01$	$\epsilon = 0.05$	$\epsilon = 0.1$	$\epsilon = 0.2$
DSM generation	68	62	65	63
Rendering (filtered)	287	253	231	213

Table 5.9: Computation times in [ms] for varying error tolerance ϵ. Chunked Memory is used as the memory management method ($4 \times 256 \times 256$ DSM).

using Chunked Memory depends on the chosen ϵ because Chunked Memory does not make use of spatial coherence as the 3D Texture memory management method does.

Comparing Table 5.7 with Table 5.10 shows that the average number of layers per DSM texel depends more on the ϵ value than on the resolution of the volumetric dataset. Interesting is the same maximum number of layers per DSM texel for $\epsilon = 0.05$, $\epsilon = 0.1$ and $\epsilon = 0.2$. This maximum number of layers per DSM texel could result from a ray through very inhomogeneous regions of the dataset where ϵ does not make any difference.

	$\epsilon = 0.01$	$\epsilon = 0.05$	$\epsilon = 0.1$	$\epsilon = 0.2$
Av. # of layers / DSM texel	4.01	3.32	3.12	3.00
Max. # of layers / DSM texel	28	21	21	21

Table 5.10: The number of layers / DSM texel depends on the error tolerance ϵ.

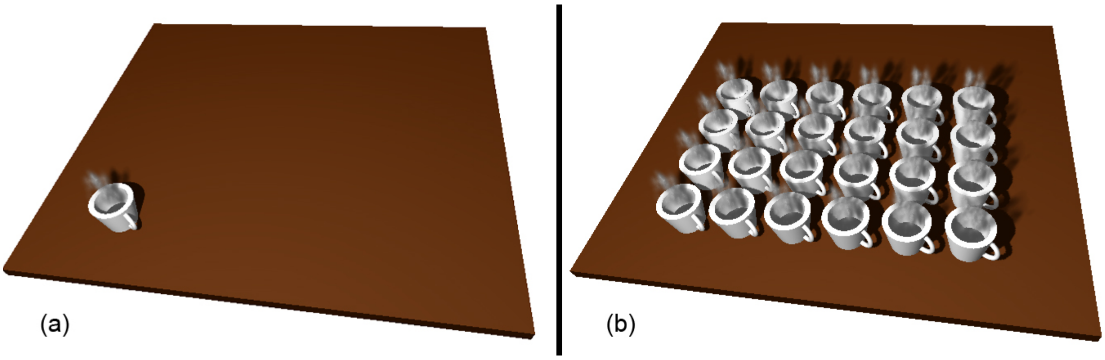


Figure 5.12: This array of teacups is used to analyze the performance depending on the number of objects in a scene. (a) One single teacup and (b) 24 teacups.

5.7.4 Number of Objects

As shown in Figure 5.12 an array of steaming teacups is used to measure the influence of the number of objects in a scene on DSM generation times as well as rendering times. The scene is rendered with a resolution of 720 × 480 again. The number of objects in the scene varies from 1 (Figure 5.12 (a)) to 24 (Figure 5.12 (b)) steaming teacups. The results of this performance test are presented in Figure 5.13.

Clustering, DSM generation and rendering times increase in a very linear manner for both memory management methods. The diagram in Figure 5.13 illustrates the constantly faster DSM generation when Chunked Memory is used. With an increasing number of objects also the difference between rendering times of the two methods increases. Interesting is the fact that in this test the rendering with Chunked Memory is slightly faster than with 3D Textures. Of course the times for clustering are the same for both memory management methods since there is no difference at all.

5.7.5 Summary

To sum up the performance tests, generating DSMs using Chunked Memory is faster than generating DSMs using 3D Textures but especially for volumetric datasets with a high resolution the rendering is slower with Chunked Memory. The test in Section 5.7.4 shows that rendering can also be slightly faster with Chunked Memory than with 3D Textures under some conditions

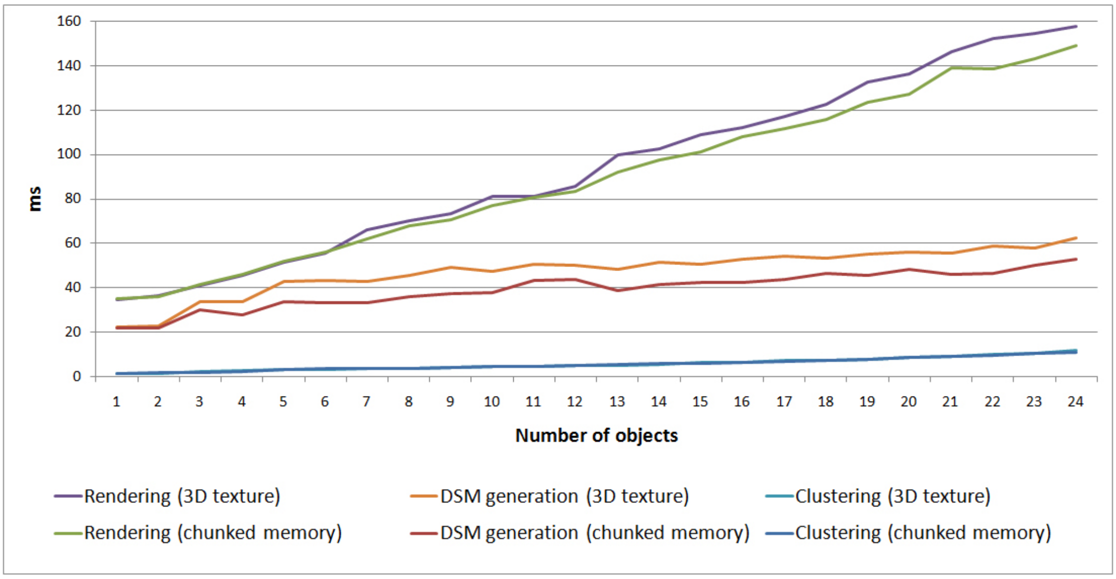


Figure 5.13: This diagram illustrates how a varying number of objects in a scene affects computation times.

(high number of low resolution datasets). Clustering slows down rendering and DSM generation but increases the quality of the shadows. Changing the error tolerance ϵ has just a small impact on rendering times when 3D Textures are used while small error tolerances slow down rendering with Chunked Memory noticeable. Filtering increases rendering times in both cases.

Note that DSMs generation is not necessary every frame but just in the cases mentioned in Section 3.2 and Section 3.4.3. This circumstance increases overall performance. Figure 5.14 shows a gallery of all scenes presented in this chapter including the average frame rates measured during camera movements. All scenes are rendered with $4 \times 256 \times 256$ *Clustered Deep Shadow Maps* for each light source.

5.8 Memory Consumption

The two different memory management methods require different amounts of memory to store the DSMs. The method using 3D Textures requires a constant amount of memory while the Chunked Memory method allocates memory dynamically for each DSM:

3D Texture: $d_{size} \cdot d_{size} \cdot d_{layers}$ (constant) $\cdot 64$ bit

Chunked Memory: $d_{size} \cdot d_{size} \cdot d_{layers}$ (variable) $\cdot 128 + d_{size} \cdot d_{size} \cdot 64$ bit

d_{size} is the dimension of the DSM in x- and y-direction and d_{layers} is the number of layers for the DSM. d_{layers} is constant for the 3D Texture method (e.g. 64) while it is variable for each

8×8 block of DSM texels when using the Chunked Memory method. Additional to the memory which stores the $depth$ and $opacity$ values of a DSM the Chunked Memory method requires a lookup texture.

Usually just a few texels of a DSM need the whole range of possible depth layers. When using 3D Textures the rest of the memory is wasted. This wastage could be reduced by implementing the block layout suggested by Hadwiger et al. [24]. Attempts to allocate memory dynamically with CUDA failed because parallel running threads of the same tile tried to allocate a new block in the DSM at the same time. Mutual exclusion did not work in CUDA.

The same problem occurs when using Chunked Memory. Threads of the same 8×8 block try to allocate chunks at the same time which results in wasted chunks. The wastage can be reduced by using the wrong allocated chunks for further chunks of the block. But it's still not optimal. Table 5.11 lists the average number of used chunks compared to the average number of wasted chunks per 8×8 block in the DSM. For the shown scenes the number of wasted chunks is about 7.8 times the number of used chunks.

Thus the amount of memory used for 3D Textures and Chunked Memory is of the same order of magnitude. For 3D Textures the memory consumption is controllable while it's unpredictable for Chunked Memory.

Scene	**(a)**	**(b)**	**(c)**	**(d)**	**(e)**	**(f)**	**(g)**	**(h)**
Av. # of used chunks / 8×8 block	2.4	2.5	1.9	1.6	1.1	2.0	4.0	3.9
Av. # of wasted chunks / 8×8 block	22.7	21.7	19.0	14.1	10.0	12.5	21.2	17.0

Table 5.11: The number of used chunks / 8×8 block is compared to the number of wasted chunks / 8×8 block for each scene shown in Figure 5.14.

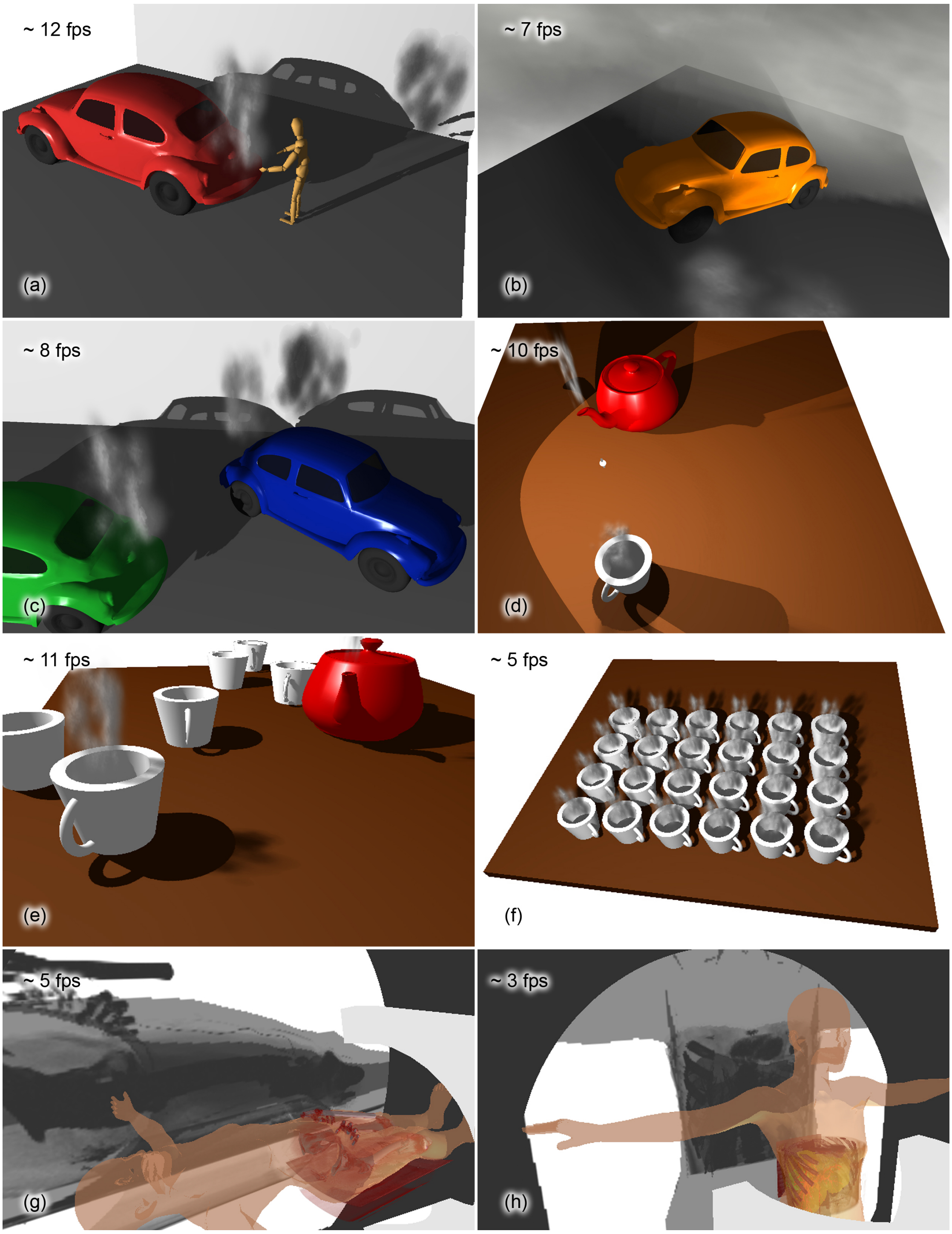


Figure 5.14: An overview of all scenes presented in this chapter including the average frame rates during camera movements.

CHAPTER 6

Conclusion

Chapter 1 introduced this thesis by giving the motivation why shadows are important in computer graphics and how particularly shadows of objects with volumetric characteristic introduce a lot of atmosphere to a scene. Rendering methods are discussed and it is explained what CUDA is and which possibilities it offers to achieve the aims of this thesis.

Chapter 2 gives an overview of work related to this thesis. Methods to render volumes from texture-based volume rendering to splatting and raycasting are discussed. Basic shadow algorithms are explained as well as advanced shadow algorithms designed for calculating shadows of volumes and hair. There is also a section about strategies to reduce aliasing artifacts of shadows.

In Chapter 3 the main contribution of this thesis is discussed. It is explained how CUDA is used to rasterize polygons and how volume raycasting is performed. The generation of *Deep Shadow Maps* as well as the usage of them during rendering is stated. Also the clustering algorithm is introduced.

In Chapter 4 the implementation details are outlined. Two different methods to manage the memory are mentioned and implementation details on *Deep Shadow Map* generation, rendering and clustering are explained.

Finally Chapter 5 presents the visual improvements and compares results with different features turned on and off. Performance tests confirm the possibility to achieve interactive frame rates even if there are multiple volumes within the scene. The chapter also deals with some memory issues.

6.1 Future Work

The *Clustered Deep Shadow Maps* presented in this thesis works well. But there is still some space for improvements. Finding a solution to solve the mentioned memory issues would be one important point to eliminate the memory wastage during *Deep Shadow Map* generation.

The basic idea of the presented clustering algorithm works very well. But it does not always create optimal clusters for all scene/camera/light constellations. Especially very huge objects lead to clustering problems in some cases. Maybe there are better ways to compute the clusters. An idea would be attaching different weights to objects to influence the clustering algorithm manually.

Another issue of *Clustered Deep Shadow Maps* is the discontinuity of the shadow appearance during cluster changes. A solution for this problem would especially improve the visual appearance of dynamic scenes.

List of Figures

1.1 Comparison of the same image without (a) and with shadows (b). Shadows are a visual cue to guess distances between objects. 2
1.2 Shadows reveal objects that cannot be seen from the camera's point of view. 2
1.3 Shadows give information about the receiver's shape. 3
1.4 (a) shows a scene rendered with conventional Shadow Maps. (b) shows the same scene rendered with *Deep Shadow Maps*. 3
1.5 An image of the short movie "Partly Cloudy" by Pixar [60]. 4
1.6 A scene rendered with ray tracing [46]. 6
1.7 Grid of Thread Blocks in CUDA [47]. 7
1.8 (a) well aligned memory access within a 32B, 64B or 128B segment, resulting in one memory transaction. (b) misaligned memory access, resulting in either one transaction with more bytes than necessary or two memory transactions. 8

2.1 Voxels are splatted with a Gaussian distribution on the view plane [26]. 12
2.2 The color cube of the first (a) and second (b) pass [34]. 13
2.3 Illustration of the shadow volume algorithm [55]. 15
2.4 Illustration of the shadow mapping algorithm [55]. 16
2.5 Illustration of the half-angle slicing algorithm [32]. 18
2.6 Examples for visibility functions [40]. 19
2.7 A transmittance function is the product of the surface transmittance function and the volume transmittance function [40]. 20
2.8 The DSM is organized in tiles. An additional low resolution texture references to the corresponding position in the tiled DSM [25]. 21
2.9 Pre-compressed visibility functions (red nodes) lead to a better approximation of the visibility function (dashed line) than regular sampling (green nodes) [25]. 22
2.10 Both images are generated using the same sample rate. (a) Regular sampling without pre-computed visibility functions and with (b) pre-computed visibility functions [25]. 22
2.11 The difference between *Opacity Shadow Maps* and *Deep Opacity Maps* [71]. . . . 23
2.12 Illustration of some texels of an occupancy map, its slabs and the depth-range [58]. 24
2.13 The mapping of a scene into post-perspective space. Top row: world space. Bottom row: post-perspective space. Left: light source in front of the camera. Middle: light source behind the camera. Right: light source in the same plane as the camera [59]. 27

2.14 The left two images show a scene rendered with uniform shadow maps and its corresponding depth map. The right two images show the same scene rendered with Perspective shadow maps and its depth map [59]. 27
2.15 Left: one object is behind the camera but in front of the light source. This object is inverted and behind the infinite plane in post-perspective space. Right: the camera is moved backwards to avoid this problem [59]. 28
2.16 This image illustrates the construction of the body B which includes all points that are important for the shadow map [59]. 29
2.17 The construction of the perspective frustum P [70]. 30
2.18 The view frustum seen from the camera is mapped to trapezoidal space (top) and to the smallest bounding box space (bottom) [44]. 31
2.19 The trapezoidal approximation (a) mapped to trapezoidal space without (b) and with (c) the 80% rule [44]. 32
2.20 Illustration of the side-line computation [44]. 32
2.21 A scene is split into three parts (left). Instead of one single shadow map (right/top) three separate shadow maps are generated for each split part (right/bottom) [72]. . . 33
2.22 Illustration of the split scheme. A combination of a uniform split scheme (a) and a logarithmic split scheme (b) result in a practical split scheme (c) [72]. 33

3.1 Two intersecting volumes rendered with the polyhedral volume renderer by Kainz et al. [29]. 36
3.2 Illustration of the two kernels of the rendering system [29]. 37
3.3 Illustration of the compression algorithm [40]. 40
3.4 Image rendered without (a) filtering the shadow and with (b) filtering. 42
3.5 Illustration of the basic idea of clustering. (a) Overview of the scene and the corresponding single shadow map. (b) Nearby objects are clustered. (c) Focusing of the shadow map on the first cluster and on the second cluster (c). (d) Single shadow map compared to the two focused shadow maps. 44
3.6 (a) Gives an overview of the scene including the bounding boxes of the objects, the position of the light source and the camera. (b) Shows the bounding boxes of the objects after the Light Space Perspective transformation. 46
3.7 Unfocused (a) and focused (b) shadow map. 47

4.1 Computation of the coverage mask (a) using shifting operations applied on the unit row r_0 (b) and the unit column c_0 [29]. 50
4.2 Illustration of the 3D texture atlas when no clusters are used. LSx are the light sources while the numbers in brackets are the additional indices of the six views of an omnidirectional light source. 52
4.3 Illustration of the 3D texture atlas when clusters are used. LSx are the light sources. 53
4.4 Illustration of the lookup coordinates (a) and the computation of the corresponding weights (b) for bilinear filtering. 63

5.1 A scene rendered with conventional shadow maps (a) and with DSMs (b). 66

5.2 An orange car in a foggy scene. You see shadows casted by polygonal geometry on the volume and vice versa. 67
5.3 Scene including multiple volumes. Overlapping shadows result in darker shadows. 68
5.4 Scene is illuminated by one spotlight and one omnidirectional light source. 69
5.5 Rendering of a scene without clustering (a) and with clustering (b). The red rectangle shows a close-up of the shadow next to the camera. 69
5.6 Same scene as in Figure 5.5 from another point of view without clustering (a) and with clustering (b). The red rectangle shows a close-up of the shadow next to the camera. 70
5.7 Same scene as in Figure 5.5 from another point of view without clustering (a) and with clustering (b). 70
5.8 Illustrates the results of the clustering algorithm. Objects in the same color belong to the same cluster. (a) Clustering result corresponds to setting of Figure 5.5. (b) Clustering result corresponds to setting of Figure 5.6. (c) Clustering result corresponds to setting of Figure 5.7. 70
5.9 Rendering without filtering the shadow (a) and with filtering the shadow (b). 71
5.10 A CT scan is projected on the wall in the background. 74
5.11 (a) Overview of a virtual X-Ray. (b) Close-up of the same scene. The CT-scan is projected on the wall in the background. 75
5.12 This array of teacups is used to analyze the performance depending on the number of objects in a scene. (a) One single teacup and (b) 24 teacups. 76
5.13 This diagram illustrates how a varying number of objects in a scene affects computation times. 77
5.14 An overview of all scenes presented in this chapter including the average frame rates during camera movements. 79

List of Tables

5.1 Computation times in [ms] for varying DSM resolutions. 3D Textures are used as the memory management method. Clustering is deactivated. 72
5.2 Computation times in [ms] for varying DSM resolutions. 3D Textures are used as the memory management method. Clustering is activated. 72
5.3 Computation times in [ms] for varying DSM resolutions. Chunked Memory is used as the memory management method. Clustering is deactivated. 72
5.4 Computation times in [ms] for varying DSM resolutions. Chunked Memory is used as the memory management method. Clustering is activated. 73

5.5 Computation times in [ms] for varying volumetric dataset resolutions. 3D Textures are used as the memory management method ($4 \times 256 \times 256$ DSM). 73
5.6 Computation times in [ms] for varying volumetric dataset resolutions. Chunked Memory is used as the memory management method ($4 \times 256 \times 256$ DSM). . . . 73
5.7 The number of layers / DSM texel depends on volumetric dataset resolution. 74
5.8 Computation times in [ms] for varying error tolerance ϵ. 3D Textures are used as the memory management method ($4 \times 256 \times 256$ DSM). 75
5.9 Computation times in [ms] for varying error tolerance ϵ. Chunked Memory is used as the memory management method ($4 \times 256 \times 256$ DSM). 75
5.10 The number of layers / DSM texel depends on the error tolerance ϵ. 76
5.11 The number of used chunks / 8×8 block is compared to the number of wasted chunks / 8×8 block for each scene shown in Figure 5.14. 78

Bibliography

[1] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Rendering Techniques*, pages 161–166, 2004.

[2] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Convolution shadow maps. In *Rendering Techniques 2007: Eurographics Symposium on Rendering*, pages 51–60, Grenoble, France, 2007. Eurographics. ISBN 978-3-905673-52-4.

[3] Thomas Annen, Zhao Dong, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Real-time, all-frequency shadows in dynamic scenes. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–8, 2008. ISBN 978-1-4503-0112-1.

[4] Jukka Arvo. Tiled shadow maps. In *CGI '04: Proceedings of the Computer Graphics International*, pages 240–247, 2004. ISBN 0-7695-2171-1.

[5] Uwe Behrens and Ralf Ratering. Adding shadows to a texture-based volume renderer. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 39–46, 1998. ISBN 1-58113-105-4.

[6] J. Blinn. Me and my (fake) shadow. *IEEE Comput. Graph. Appl.*, 8:82–86, January 1988. ISSN 0272-1716.

[7] Willem H. De Boer. Smooth penumbra transitions with shadow maps. *ACM Journal of Graphics Tools*, 2006, 2006.

[8] David Borland, John P. Clarke B, Julia R. Fielding B, and Russell M. Taylor Ii A. Volumetric depth peeling for medical image display. In *Proceedings of SPIE Visualization and Data Analysis*, pages 1–11, 2006.

[9] Stefan Brabec, Thomas Annen, and Hans peter Seidel. Shadow mapping for hemispherical and omnidirectional light sources. In *In Proc. of Computer Graphics International*, pages 397–408, 2002.

[10] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Practical shadow mapping. *J. Graph. Tools*, 7(4):9–18, 2002. ISSN 1086-7651.

[11] Ralph Brecheisen, Anna Vilanova Bartroli, Bram Platel, and Bart M. ter Haar Romeny. Flexible gpu-based multi-volume ray-casting. In Oliver Deussen, Daniel A. Keim, and Dietmar Saupe, editors, *VMV*, pages 303–312. Aka GmbH, 2008. ISBN 978-3-89838-609-8.

[12] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, 1994. ISBN 0-89791-741-3.

[13] Franklin C. Crow. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248, 1977.

[14] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical report, 1994.

[15] DMI. 2011. URL `http://www.dmi-3d.net`.

[16] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165, 2006. ISBN 1-59593-295-X.

[17] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, 2001. ISBN 1-58113-407-X.

[18] Fiu Eugene, Alain Fournier, and Larry Rudolph. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. *SIGGRAPH Comput. Graph.*, 17: 141–150, July 1983. ISSN 0097-8930.

[19] Cass Everitt. Interactive order-independent transparency. Nvidia technical report, 2001.

[20] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive shadow maps. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 387–390, 2001. ISBN 1-58113-374-X.

[21] Markus Giegl and Michael Wimmer. Queried virtual shadow maps. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 65–72, 2007. ISBN 978-1-59593-628-8.

[22] Naga K. Govindaraju, Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha. Interactive shadow generation in complex environments. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 501–510, 2003. ISBN 1-58113-709-5.

[23] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1(4):132–133, 1972.

[24] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus H. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Comput. Graph. Forum*, 24(3):303–312, 2005.

[25] Markus Hadwiger, Andrea Kratz, Christian Sigg, and Katja Bühler. Gpu-accelerated deep shadow maps for direct volume rendering. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 49–52, 2006. ISBN 3-905673-37-1.

[26] Charles D. Hansen and Chris R. Johnson. *The visualization handbook*. Elsevier, Amsterdam [u.a.], 2005. ISBN 0-12-387582-X.

[27] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '98, pages 39–ff., 1998. ISBN 1-58113-097-X.

[28] Jon Jansen and Louis Bavoil. Fourier opacity mapping. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 165–172, 2010. ISBN 978-1-60558-939-8.

[29] Bernhard Kainz, Markus Grabner, Alexander Bornik, Stefan Hauswiesner, Judith Muehl, and Dieter Schmalstieg. Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore gpus. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–9, 2009. ISBN 978-1-60558-858-2.

[30] Tae-Yong Kim and Ulrich Neumann. Opacity shadow maps. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 177–182, London, UK, 2001. Springer-Verlag. ISBN 3-211-83709-4.

[31] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proceedings of the conference on Visualization '01*, VIS '01, pages 255–262, 2001. ISBN 0-7803-7200-X.

[32] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8:270–285, 2002. ISSN 1077-2626.

[33] Simon Kozlov. Perspective shadow maps: Care and feeding. In *In GPU Gems*, pages 217–244. Addison-Wesley, 2004.

[34] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, 2003. ISBN 0-7695-2030-8.

[35] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 451–458, 1994. ISBN 0-89791-667-0.

[36] Andrew Lauritzen and Michael McCool. Layered variance shadow maps. In *GI '08: Proceedings of graphics interface 2008*, pages 139–146, 2008. ISBN 978-1-56881-423-0.

[37] Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8:29–37, May 1988. ISSN 0272-1716.

[38] D. Brandon Lloyd. *Logarithmic perspective shadow maps*. PhD thesis, Chapel Hill, NC, USA, 2007.

[39] D. Brandon Lloyd, Naga K. Govindaraju, Steven E. Molnar, and Dinesh Manocha. Practical logarithmic rasterization for low-error shadow maps. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 17–24, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-1-59593-625-7.

[40] Tom Lokovic and Eric Veach. Deep shadow maps. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 385–392, 2000. ISBN 1-58113-208-5.

[41] Kok-Lim Low and Adrian Ilie. Computing a view frustum to maximize an object's image area. *journal of graphics, gpu, and game tools*, 8(1):3–15, 2003.

[42] Shang Ma, Xiaohui Liang, Zhuo Yu, and Wei Ren. Light space cascaded shadow maps for large scale dynamic environments. In *MIG '09: Proceedings of the 2nd International Workshop on Motion in Games*, pages 243–255, 2009. ISBN 978-3-642-10346-9.

[43] Tom Malzbender. Fourier volume rendering. *ACM Trans. Graph.*, 12:233–250, July 1993. ISSN 0730-0301.

[44] Tobias Martin and Tiow Seng Tan. Anti-aliasing and continuity with trapezoidal shadow maps. In *Rendering Techniques*, pages 153–160, 2004.

[45] Tom Mertens, Jan Kautz, Philippe Bekaert, and Frank Van Reeth. A self-shadow algorithm for dynamic hair using density clustering. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, page 44, 2004. ISBN 1-59593-896-2.

[46] Mehran Moghtadai. 2010. URL `http://de.wikipedia.org/w/index.php?title=Datei:Glass_is_Liquide.jpg&filetimestamp=20061117013607`.

[47] NVIDIA. Nvidia cuda programming guide 2.3.1. 2009.

[48] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krueger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[49] Bui Tuong Phong. *Illumination for computer-generated images.* PhD thesis, 1973.

[50] John Plate, Thorsten Holtkaemper, and Bernd Froehlich. A flexible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Transactions on Visualization and Computer Graphics*, 13:1584–1591, 2007. ISSN 1077-2626.

[51] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 283–291, 1987. ISBN 0-89791-227-6.

[52] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T-Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000.

[53] Friedemann Roessler, Ralf P. Botchen, and Thomas Ertl. Dynamic shader generation for gpu-based multi-volume ray casting. *IEEE Computer Graphics and Applications*, 28:66–77, 2008. ISSN 0272-1716.

[54] Stefan Roettger and Thomas Ertl. Cell projection of convex polyhedra. In *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, VG '03, pages 103–107, 2003. ISBN 1-58113-745-1.

[55] Daniel Scherzer. Shadow mapping of large environments. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 8 2005.

[56] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Shadow silhouette maps. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 521–526, 2003. ISBN 1-58113-709-5.

[57] Erik Sintorn and Ulf Assarsson. Real-time approximate sorting for self shadowing and transparency in hair rendering. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 157–162, 2008. ISBN 978-1-59593-983-8.

[58] Erik Sintorn and Ulf Assarsson. Hair self shadowing and transparency depth ordering using occupancy maps. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 67–74, 2009. ISBN 978-1-60558-429-4.

[59] Marc Stamminger and George Drettakis. Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 557–562, 2002. ISBN 1-58113-521-1.

[60] Pixar Animation Studios. 2010.

[61] Katsumi Tadamura, Xueying Qin, Guofang Jiao, and Eihachiro Nakamae. Rendering optimal solar shadows using plural sunlight depth buffers. In *CGI '99: Proceedings of the International Conference on Computer Graphics*, page 166, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0185-0.

[62] Kongsberg Oil & Gas Technologies. Coin documentation 3.1.4a. 2010.

[63] Maurice Termeer, Javier Oliván Bescós, and Alexandru Telea. Preserving sharp edges with volume clipping. In *Proceedings Vision, Modeling and Visualization 2006*, November 2006.

[64] Yulan Wang and Steven Molnar. Second-depth shadow mapping. Technical report, Chapel Hill, NC, USA, 1994.

[65] Daniel Weiskopf, Klaus Engel, and Thomas Ertl. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics*, 9:298–312, 2003. ISSN 1077-2626.

[66] Lee Westover. Interactive volume rendering. In *Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, VVS '89, pages 9–16, 1989.

[67] Lee Westover. Footprint evaluation for volume rendering. In *Computer Graphics*, pages 367–376, 1990.

[68] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, 1978.

[69] Orion Wilson, Allen VanGelder, and Jane Wilhelms. Direct volume rendering via 3d textures. Technical report, 1994.

[70] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In Alexander Keller and Henrik W. Jensen, editors, *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, pages 143–151. Eurographics, Eurographics Association, June 2004. ISBN 3-905673-12-6.

[71] Cem Yuksel and John Keyser. Deep opacity maps. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2008)*, 27(2), 2008.

[72] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, pages 311–318, 2006. ISBN 1-59593-324-7.