BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

IMPROVEMENTS OF SHADOW RENDERING

VYLEPŠENÍ VYKRESLOVÁNÍ STÍNŮ

PHD THESIS
DISERTAČNÍ PRÁCE

AUTOR **Ing. JOZEF KOBRTEK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. ADAM HEROUT, Ph.D.**
ŠKOLITEL

BRNO 2021

Abstract

This thesis describes a set of incremental improvements of the shadow volume algorithm. First, a novel robust method of silhouette extraction is detailed, including implementation on several hardware platforms. This technique was later improved, simplified and the whole shadow volume algorithm ported to hardware tessellation. Next, a novel accelerated silhouette extraction algorithm was proposed, based on octree acceleration structure. Finally, the proposed methods were compared with the latest omni-directional techniques casting hard shadows.

Abstrakt

Táto práca sa zaoberá inkrementálnym zlepšením techniky tieňových telies. V práci sa popisuje vylepšenie vykresľovania z pohľadu robustnosti kde bol navrhnutý nový spôsob deterministického výpočtu siluety na rôznych platformách. Táto technika bola v ďalšom kroku zjednodušená a celý algoritmus tieňových telies implementovaný prostredníctvom hardvérovej teselácie. Ďalej bola navrhnutá metóda akcelerovanej extrakcie siluety z modelu pomocou oktálového stromu. Navrhnuté metódy boli v závere porovnané s aktuálnymi modernými algoritmami s tvrdými všesmerovými tieňmi.

Keywords

shadows, shadow volumes, silhouette, tessellation, GPGPU, acceleration structures, octree, shadow maps, visibility, ray tracing, RTX

Klíčová slova

stíny, stínová tělesa, silueta, teselace, GPGPU, akcelerační struktury, oktálový strom, stínové mapy, viditelnost, vrhání paprsků, RTX

Reference

KOBRTEK, Jozef. *Improvements of Shadow Rendering.* Brno, 2021. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Adam Herout, Ph.D.

Improvements of Shadow Rendering

Declaration

Prohlašuji, že jsem tuto dizertační práci vypracoval samostatně pod vedením pana prof. Adama Herouta a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Jozef Kobrtek
September 28, 2021

Acknowledgements

First and foremost, I would like to thank my supervisor prof. Ing. Adam Herout for his support and mentoring throughout my PhD studies. He helped my sort out any problem I came across in a very constructive manner. This thesis could not be completed without my outstanding colleagues, namely Tomáš Milet, Tomáš Starka, Jan Pečiva, Michal Tóth, Michal Kula and Tomáš Lysek; this research would not be possible without their knowledge and creativity. I have learned a lot from them – new ways of solving coding problems, deeper knowledge of design patterns and coding standards and much more. I want to thank Sveťo Nosko, Tomáš Chlubna, Martin a Petr Musil for their joyful and pleasant company at our common lunches, for all the events that we organized together and for letting me use their lab equipment for my DIY projects.

I cannot omit my internship at Škoda Auto. It was a pleasure to cooperate with Antonín Míšek on very interesting projects in VR and ray tracing. I will remember him as one of the best supervisors I've ever had. I also want to thank my current employer, 2K Czech, especially my leader Sobo for letting me combine school and work in the way I needed.

Last but not least, I want to express gratitude to the dearest people in my life - my mother Ida, sister Lenka and girlfriend Ivanka for having patience with me as I was frequently neglecting them. I want to specially dedicate this thesis to my father, who unfortunately had passed away before this thesis was finished.

Contents

1 Introduction **3**

2 Background of Shadow Rendering **5**
2.1 Definition of a Shadow 5
2.2 Importance of Shadows as Visual Clues 6
2.3 Shadows in the Rendering Equation 7
2.4 Overview of Basic Shadowing Methods 10

3 Existing Precise Shadows from Non-Extended Light Sources **20**
3.1 Stencil Shadow Volumes in Depth 20
3.2 Algorithms Using Acceleration Structures 33
3.3 Ray Tracing Methods 38

4 Comparison of Omnidirectional Shadow Mapping Methods **39**
4.1 Cube Mapping and Optimizations 39
4.2 Dual-Paraboloid Shadow Maps 40
4.3 Implementation 42
4.4 Test Results 43
4.5 Evaluation 46
4.6 Extended Quality Comparison 47

5 Robust Silhouette Extraction Improvements for Shadow Volumes **51**
5.1 Robustness Problems of Shadow Volumes 51
5.2 GPU Implementations 53
5.3 Implementation Using AVX Intrinsics and OpenMP 54
5.4 Measurements 55
5.5 Evaluation 56

6 Shadow Volumes Using Tessellation Shaders **57**
6.1 Per-Triangle Tessellation-Based Shadow Volumes 57
6.2 Silhouette-Based Approach 58
6.3 Implementation 62
6.4 Measurements 63
6.5 Evaluation 68
6.6 Comparison on Modern Hardware 68

7 Silhouette Extraction Using Precomputed Potentially Visible Set **72**
7.1 Precomputed Silhouette Extraction Overview 72

7.2 Octree Setup . . . 72
7.3 Octree Build . . . 73
7.4 Octree Traversal . . . 77
7.5 Implementation . . . 77
7.6 Measurements . . . 81
7.7 Improvements . . . 85
7.8 Evaluation . . . 88

8 Comparison of Omnidirectional Precise Shadow Methods 90
8.1 Selected Methods . . . 90
8.2 Implementation and Optimizations . . . 91
8.3 Measurements . . . 93
8.4 Time Complexity . . . 104
8.5 Discussion and Conclusion . . . 104

9 Conclusion 107

Bibliography 109

A List of Publications 120

Chapter 1

Introduction

Shadows are a fundamental part of any modern 3D application. Different areas of the industry put specific criteria on the shadow quality. Games tend to use shadow mapping as preferred shadowing technique even though it suffers from aliasing problems due to the discrete nature of the shadow map, but the general simplicity of the method combined with techniques to mitigate the aliasing problem makes it a usual choice.

On the other hand, there are several areas of CAD/CAM industries where pixel-precise shadows are required, e.g. house visualizations and design, where shadows not only serve as a visual clue for presentations and rendering, but also may indicate defects in the model or scene. Also, being able to render pixel-precise shadows from any arbitrary triangle soup needs to be addressed as model quality may be lower in certain scenarios, e.g. geometry having inconsistent triangle winding or model with edges having more than 2 adjacent triangles.

The focus of this thesis is to improve shadow rendering, particularly shadow volumes, in terms of robustness (rendering arbitrary triangle soups without artifacts) and speed, not only of the rendering process but also silhouette extraction, which is a fundamental step of shadow volumes algorithm. This thesis does not present breakthroughs in the field of shadow rendering, but shows iterative improvements in shadow volumes and concludes by providing an extensive comparison of all known modern precise shadowing techniques.

My research started with comparison of omnidirectional shadow mapping techniques and their optimizations. We compared dual-paraboloid mapping with cube mapping in terms of speed and quality.

The focus then moved towards pixel-precise shadows, to design a precise and robust shadow algorithm to handle virtually any triangle soup for shadow rendering. As the traditional shadow mapping suffers from aliasing problems that are not easy to overcome, we chose stencil shadow volumes as the basis for our research. The algorithm, at that time, suffered from robustness problems, which were sometimes more disturbing than shadow mapping imperfections – blinking triangles caused by numerical issues which could not be addressed by adding bias – that only moved the problem further away. I was a part of the team that improved numerical robustness of the stencil shadow volumes, we got rid of bias in the computations and implemented the method on several platforms. I implemented an optimized CPU version using OpenMP and AVX instruction sets for silhouette extraction, being the fastest CPU-based silhouette extraction algorithm. The resulting methods are still used to this day for shadow rendering in a software by Cadwork company, including their web presentations.

As the hardware evolved, I designed a method to compute shadow volumes using hardware tessellation. Although per-triangle at first, we were able to improve it with Tomáš Milet by partially collapsing the generated geometry from the tessellator to generate shadow volume sides. Tessellator is able to subdivide the input primitive up to hundreds of triangles, which was utilized in the silhouette version of the algorithm.

I continued my research focusing on silhouette extraction, designing an algorithm to speed up silhouette extraction time at the cost of extra memory. I had decided to improve the silhouette extraction from arbitrary triangle soups as it did not get much focus from the research community for some time and the available solutions relied mostly on 2-manifold geometry.

As for my final paper, me and my colleagues have implemented several modern precise shadowing methods and compared them to hardware-accelerated ray tracing. RTX has become a phenomenon recently as it has brought ray-tracing from offline or interactive to real time graphics even on consumer-level hardware. Games have slowly started to utilize this technology to add more effects like precise reflections, but one of the possible applications of RTX is also precise and fast shadow rendering, which will be demonstrated in this thesis.

The thesis is outlined as follows. Chapter 2 introduces the reader into the topic of shadow rendering, providing overview of all base shadowing algorithms and their development over the time. Chapter 3 provides an in-depth analysis of precise shadow methods, putting emphasis on stencil shadow volumes, as they were the most influential for my research. Chapter 4 compares omnidirectional techniques based on shadow mapping in terms of speed and quality. The necessity of having precise omnidirectional shadows is outlined in this chapter as well. Chapter 5 discloses details about the first robust improvement in silhouette computation for shadow volumes, as well as comparison of the algorithm on several hardware platforms. This idea was further expanded in the Chapter 6 where the robust silhouette computation was further optimized. This new approach was combined with a new method of computing shadow volumes using the tessellation pipeline stages. Chapter 7 describes a new method of accelerated silhouette computation using octree storing potentially visible edge sets. This algorithm utilizes a novel compression scheme to reduce its memory consumption and greatly reduces the number of edges that need to be tested when computing the silhouette. Chapter 8 provides an extensive comparison of several modern methods producing precise shadows from omnidirectional light sources. Hardware-accelerated ray tracing is also among the evaluated methods. Finally, Chapter 9 concludes the findings of the thesis.

Chapter 2

Background of Shadow Rendering

This chapter introduces the reader to the problem of shadow rendering, starting with the analysis of shadow importance as visual clues, which is supported by several psychological studies. Shadow rendering techniques evolved over time, starting with ray-traced shadows while rendering images by scan lines and offline, moving to real-time methods such as shadow mapping, shadow volumes, planar projected shadows, and going back to ray traced shadows – but in real time.

2.1 Definition of a Shadow

Several definitions of shadow can be found in the literature. Hasenfratz et al. [41] describes shadow as “region of space for which at least one point of the light source is occluded”. Dictionaries tend to define shadow for example as “a partial or complete darkness, especially produced by a body coming between rays of light and a surface” [1]. These definitions, however, take neither indirect lighting nor transparent occluders into the account. As can be seen in the Fig. 2.1, neither indirect illumination nor scale is considered as the microscopic picture of a surface may have areas resembling shadows, although there is no visible shadow on macro scale. But these micro details are responsible for complex effects under the object’s surface where light is scattered, attenuated or difracted. These interactions are simulated by distribution functions (bidirectional reflectance/transmittance/surface scattering distribution functions) and applied in advanced reflectance models like Cook-Torrance, which uses microfacet distribution to approximate visibility of micro-scale details [30].

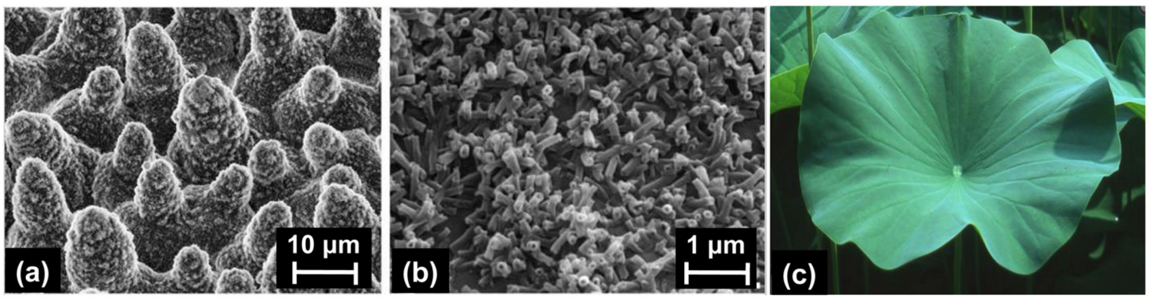


Figure 2.1: Electron microscopy detail of a lotus leaf, in different scales. Structures that cast shadows can be found also on micro scale. But these micro-details affect shading in terms of larger scale, as seen in the subfigure (c).[1]

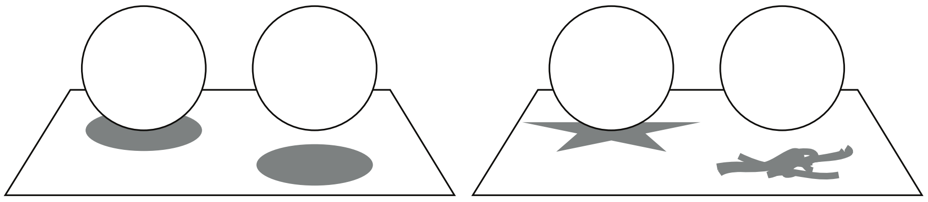


Figure 2.2: Shadow shape does not play important role in object's spatial localization [30].

2.2 Importance of Shadows as Visual Clues

Shadows are something we take for granted in the real world without realizing how important they actually are for aiding our perception due to our natural focus on tangible objects. The importance of shadows as visual clues has been verified by several studies. Psychological studies by Kersten [57, 59, 58] show that shadows have a strong influence on the perceived three-dimensional motion of objects. He was able to induce apparent motion in depth of an object even though the object was stationary. In fact, shadows are such a strong cue that the information provided by the motion of an object's shadow overrides other strong sources of information and perceptual biases, such as assumption of constant object size and general viewpoint. His research also points out that even unnatural light shadows can induce apparent depth motion of an object when the shadows are moved.

Madison et al. [73] experimentally verified that shadows are a valuable clue for determining whether an object was with contact with the ground. Observers provided most accurate judgments when shadows and inter-reflections were drawn. Taya and Miura [104] pointed out that cast shadows are used by our visual system as a cue for predicting the future position of moving objects. Shadow perception was researched even upon infants, Imura et al. [50] experimentally verified that perception of object's trajectory motion from the motion of of cast shadow emerges around 6th month of age as younger infants were unable to determine "up" and "depth" motion events.

Shadows are also a clue to determine object's transparency, as shown by Kawabe [56], who was able to induce transparency by placing shadows inside the object's contours.

The shape of the shadow might not necessarily be important. As illustrated by Ni et al. [87], size matching between casting object and shadow does not necessarily have to correspond. Wagner [108] pointed out that shadows shape does not have appreciable effect on the perception of the object size and position, and even higher order interactions indicate that the shape can be completely ignored, as seen in the Figure 2.2. He also points out that soft shadows may also negatively effect accurate perception of object's shape.

Wagner's findings are applied in the gaming industry. In the early era of computer games, hardware limitations at the time did not allow for implementation of modern real-time shadowing techniques such as shadow mapping, so the game developers relied on simple blob shadows, as seen in the Figure 2.3. This provided sufficient visual clue of the object or character in the scene. Such a technique can be seen even in modern computer or mobile phone games.

[1]Costa, Mafalda; Veigas, Bruno; Jacob, Jorge; et al.: A low cost, safe, disposable, rapid and self-sustainable paper-based platform for diagnostic testing: Lab-on-paper. In *Nanotechnology*. vol. 25. 02 2014: doi:10.1088/0957-4484/25/9/094006.

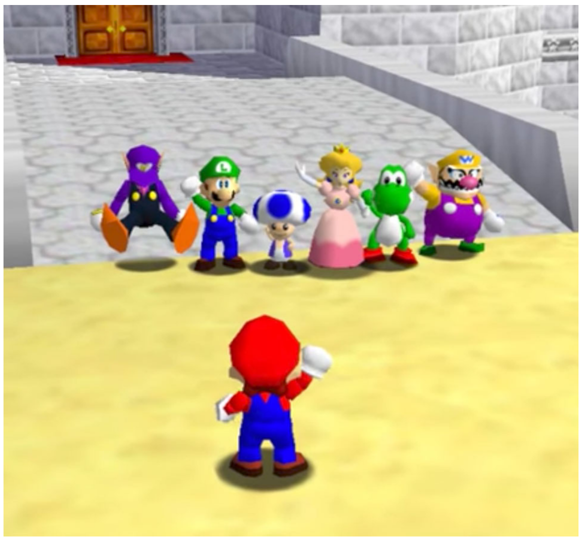


Figure 2.3: Super Mario 64 game (1996) using circular blobs as shadows. Source: technabob.com

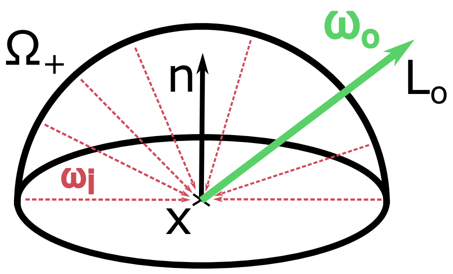


Figure 2.4: Rendering equation, calculating radiance L_o of direction ω_o from point x

2.3 Shadows in the Rendering Equation

In order to define shadow accurately in terms of computer graphics, we need to start with the rendering equation 2.1, formulated by Kajiya [54] and using directional form presented by [49], who omits wavelength dependence as it is not necessary for shadow computation. Scheme can be seen in Figure 2.4.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega_+} f_r(x, \omega_i \to \omega_o) L_i(x, \omega_i) |\cos(\omega_i, n)| d\omega_i \tag{2.1}$$

We can calculate outgoing radiance L_o for point x on the surface and direction ω_o using Equation 2.1, where:

- L_e is emitted radiance from point x and direction ω_0, non-zero for light sources
- the integral, scattering equation, sums the contribution of all reflected incident radiances L_i to point x from half-space Ω_+
- f_r is bidirectional reflectance distribution function (BRDF) describing the ratio between the reflected radiance in direction ω_o and incoming radiance ω_i at point x

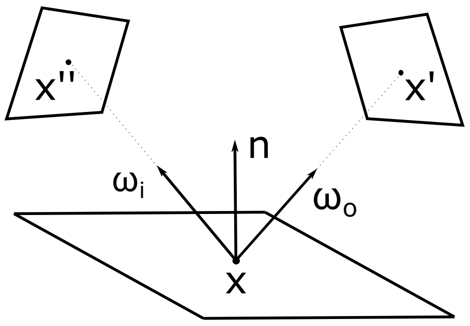


Figure 2.5: Converting rendering equation to surface form [91].

- $|\cos(\omega_i, n)|$ is absolute value of cosine of angle between normal n and incoming direction ω_i

The equation defines balance in terms of energy – the light energy entering the system equals to the energy leaving the system. It also does not take the participating media into the account – radiance is constant along the rays. In such case, we can define a ray-casting function $t(x, \omega)$ that computes the first surface point x' intersected by a ray from x with direction ω, we can write the incident radiance at x using outgoing radiance at x' as in Equation (2.2) [91].

$$L_i(x, \omega_o) = L_o(t(x, \omega), -\omega) \tag{2.2}$$

Substituting to Equation (2.1) yields Equation (2.3).

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega_+} f_r(x, \omega_i \to \omega_o) L_o(t(x, \omega_i), -\omega_i) |\cos(\omega_i, n)| d\omega_i \tag{2.3}$$

Now, the recurrent nature of the rendering equation becomes obvious. In this form, the equation makes geometric relations between objects implicit. To make this behaviour more explicit in the integrand we need to convert the equation into the surface form, as an integral over area instead of directions on the sphere, as seen in the Figure 2.5.

First, the exitant radiance from point x to x' is defined as

$$L(x' \to x) = L(x', \omega)$$

only if x and x' are mutually visible and $\omega = \frac{x'-x}{||x'-x||}$, e.g. normalized direction vector from x to x'. We also need to multiply by Jacobian to get area from solid angle, which is $A = \frac{\Omega}{r^2}$ where Ω is solid angle and r is distance. Combining with original $|\cos(\omega_i, n)|$ term yields geometric coupling term $G(x \leftrightarrow x')$, as in Equation (2.4).

$$G(x \leftrightarrow x') = \frac{|\cos(n, x \to x')||\cos(n_{x'}, x' \to x)|}{||x - x'||^2} \tag{2.4}$$

and substituting to Equation (2.3), using notation from Figure 2.5, results in:

$$L_o(x \to x') = L_e(x \to x') + \int_A f_r(x'' \to x \to x') L_o(x'' \to x) V(x'' \leftrightarrow x) G(x'' \leftrightarrow x) dA(x'') \tag{2.5}$$

where A is the sum of all the surfaces in the scene and $V(x'' \leftrightarrow x)$ is binary visibility function yielding 1 when x and x'' are mutually visible, zero otherwise [91, 30].

This equation can be further simplified for shadow computation. If we take only the direct illumination into account, we can remove the equation's dependency upon itself. Furthermore, the integral becomes non-zero only for points that are located on the light source $\mathcal{L}$. Additivity of the integral also allows us to process the light sources sequentially. We then integrate over all light sources instead of all surfaces of the scene in Equation (2.6), combining surface form from [91] with simplification stated in [30].

$$L_o(x \to x') = \int_{\mathcal{L}} f_r(l \to x \to x') L_e(l \to x) V(l \leftrightarrow x) G(l \leftrightarrow x) d\mathcal{L}(l) \tag{2.6}$$

We can also further assume that BRDF function f_r is mainly diffuse and geometric term G varies little for simple light sources far away from the receiver, which allows for separation of the integral to two – for functions G and L_e. This simplification basically separates shading and shadows in the rendering equation [30].

$$L_o(x \to x') = \underbrace{\int_{\mathcal{L}} f_r(l \to x \to x') G(l \leftrightarrow x) d\mathcal{L}(l)}_{\text{Shading}} \cdot \underbrace{\frac{1}{\mathcal{L}} \int_{\mathcal{L}} L_e(l \to x) V(l \leftrightarrow x) d\mathcal{L}(l)}_{\text{Shadows}} \tag{2.7}$$

In practice, we assume that the light source is homogenous in terms of radiation direction over its surface, which can simplify L_e to a function of position. Provided it is uniformly coloured, it can be omitted from the integral as the constant L_c. The result of this simplification are Equations 2.8a, 2.8b, 2.8c.

$$L_o(x \to x') = dirIllum(x \to x', \mathcal{L}, L_c) \cdot V_{\mathcal{L}}(x) \tag{2.8a}$$

$$V_{\mathcal{L}}(x) = \frac{1}{|\mathcal{L}|} \int_{\mathcal{L}} V(l \leftrightarrow x) d\mathcal{L}(l) \tag{2.8b}$$

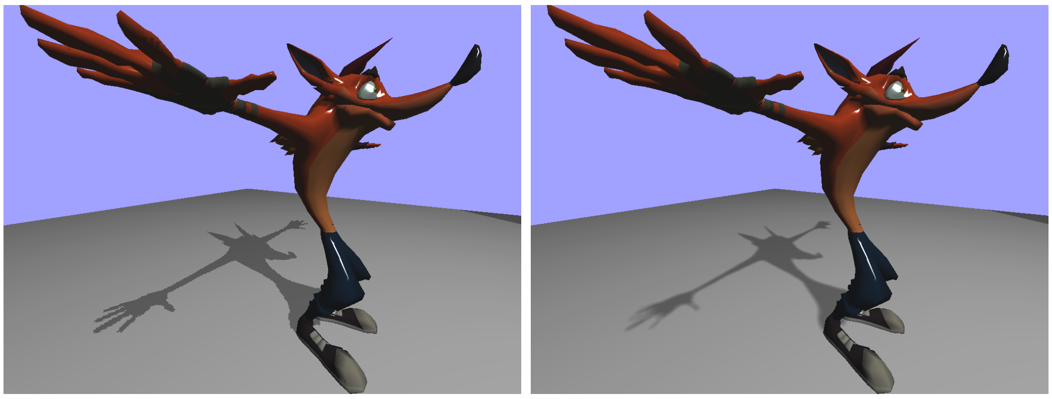
$$directIllum(x \to x', \mathcal{L}, L_c) = L_c \int_{\mathcal{L}} f_r(l \to x \to x') G(l \leftrightarrow x) d\mathcal{L}(l) \tag{2.8c}$$

where $V_{\mathcal{L}}(x)$ is visibility integral and *directillum* is direct illumination from the light source. Equation (2.8b) solves *soft shadows*, as it calculates visibility across the whole surface of the light source. Although light source is described as surface in the integral, usually just a single light sample $\grave{l} \in \mathcal{L}$ for performance reasons. In that case, integral is replaced with $V(x, \grave{l})$ in Equation (2.8b) that returns only binary information (visible/not visible) if point x and light sample $\grave{l}$ are mutually visible. Such shadow consists only from umbra and thus called *hard shadow*.

2.3.1 Shadow Types

The type of a shadows cast is dependant on the type (and configuration) of the light source. We define several categories of light sources – point , directional (which equals to a point light source in the infinity), spotlights, and area (or extended) light sources[92, 115].

Given a point, directional light source or a spotlight, there exists only a single ray that can hit a point x, thus corresponding to a binary information lit/shadowed. This shadow region, which is fully occluded from the light source, is called *umbra* or *hard shadow*.



(a) Hard shadows (b) Soft shadows

Figure 2.6: A game character casting hard and soft shadows from point light source, shadow mapping used. Soft shadows were achieved using percentage closer filtering.

Illustration of these light sources can be seen in the Figures 2.7a and 2.7b. It is also the case when integral Equation (2.8b) becomes $V(x, \grave{l})$ for a single light sample $\grave{l}$.

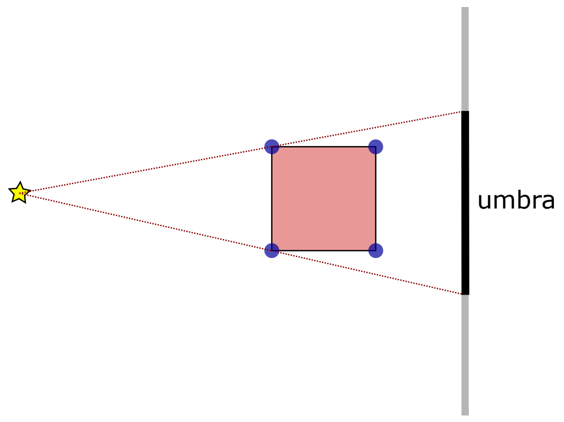
Shadows from area light sources are modeled using Equation (2.8b) and ideally should account for every $\grave{l} \in \mathcal{L}$, thus producing a value in the range 0-100%. Umbra is a fully shadowed region ($V_{\mathcal{L}}(x) = 0$) and *penumbra* or *soft shadow* (where $V_{\mathcal{L}}(x) > 0 \wedge V_{\mathcal{L}}(x) \leq 1$) is a partially shadowed region. It is a common practice to mimic penumbra from point light source using various techniques (e.g. percentage-closer filtering in Shadow Mapping), as can be seen in Figure 2.6.

Although computer graphics distinguishes only 2 types of shadows, there is also a third shadow type called *antumbra*. This type of shadow occurs when the light source is significantly larger than the occluder and spans from the end of umbra and between penumbras. The characteristic of this type of shadow is that when the observer is located inside antumbra, he may observe the occluding object entirely within the disc of the light source (provided the light source is spherical). When put into context of space bodies, for example when the Sun, the Moon and the Earth are aligned in a line and the observer, located in the Moon's antumbra, observes annular eclipse of the Sun, as seen in the Figure 2.8 [32, 106]. Computer graphics generally recognizes this type of shadow as penumbra.

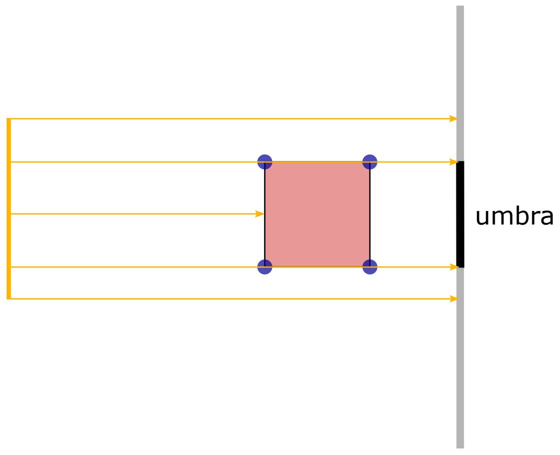
We can also classify shadows based on the receiver configuration. So-called *attached shadow* occurs when the normal of the receiver surface faces away from the light source. *Cast shadows*, on the other hand, are shadows on a surface, normal of which is facing the light source [41]. Example can be seen in the Figure 2.9.

2.4 Overview of Basic Shadowing Methods

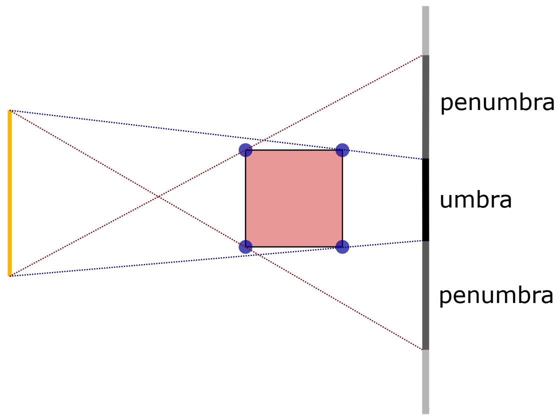
Numerous shadowing techniques have been developed over the course of history. The problem of the shadow calculation is as old as computer graphics itself. One of the first published algorithms for computing shadows was used for printing an object using a plotter [9]. The method resembles raycasting in its nature, using '+' signs of various sizes as shading levels. As Heckbert stated in his list of unsolved graphics problems [43], raytracing was used on the most contemporary images generated by computers and expresses demand for shadowing methods not utilizing raytracing.



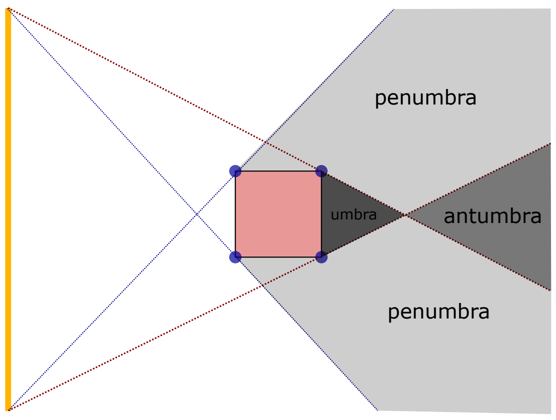
(a) Point light casting umbra



(b) Directional light (umbra)

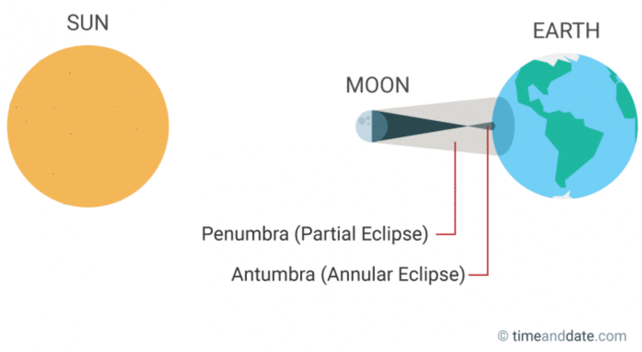


(c) Area light (umbra, penumbra)

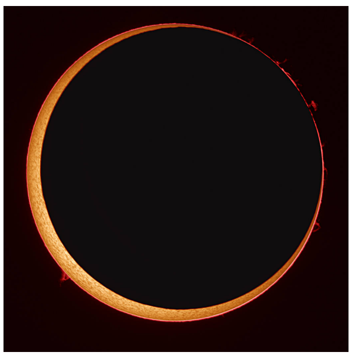


(d) Large area light casting also antumbra

Figure 2.7: Light source and shadow types



(a) Scheme of the Sun eclipse, source: timeanddate.com



(b) Annular eclipse as seen by observer in the Moon's antumbra, source: nasa.gov

Figure 2.8

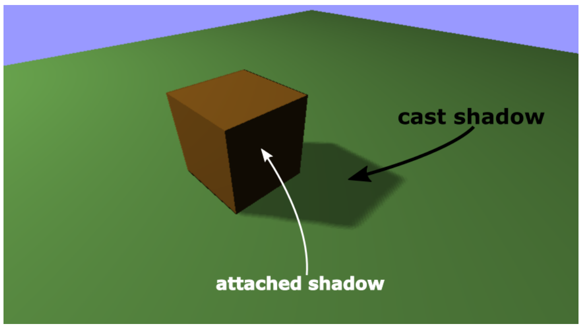


Figure 2.9: Cast and attached shadows

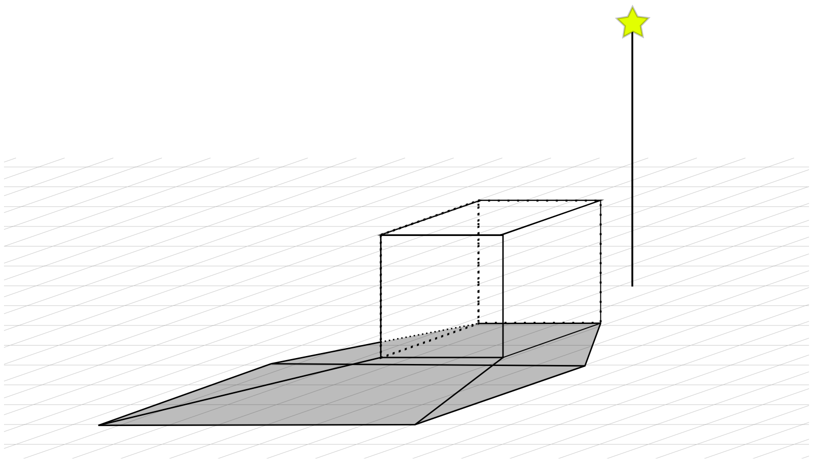


Figure 2.10: Planar projection shadows with geometry squashed on top of the projection surface

2.4.1 Planar Projection Shadows

This shadowing method was introduced by Blinn [14]. The core principle is to project the occluder's geometry to a planar receiver by constructing a custom projection matrix M that squashes object's z-dimension to zero. The occluding object is then drawn in a dark color on the top of the planar surface using matrix M. This method works for point light sources only and produces hard shadows. Illustration of the method can be seen in Figure 2.10.

Although it is relatively simple to construct the projection matrix M, the method has several drawbacks. First of all, the receiver has to be planar which limits the usage of the algorithm. Another issue are z-fighting artifacts – shadow geometry cannot be projected directly on the receiver plane, which would cause blinking artifacts, but the shadow must be slightly levitating above the plane. Shadow must also be clipped to the receiver's boundaries. Another limitation is that the light source might not be positioned between the object and the plane. By doing so, a so-called *anti-shadow* is created on the receiver plane, with vertices projected along the light source, see Figure 2.11. The algorithm does not support self-shadowing.

Making the shadow semi-transparent and blending it with the receiver will enhance the visual quality of such solution (the shadow will no longer be just dark but will also mix with the color of the plane), but double-blending must be resolved as several triangles might be blended to the same location because the occluder might not be convex. This

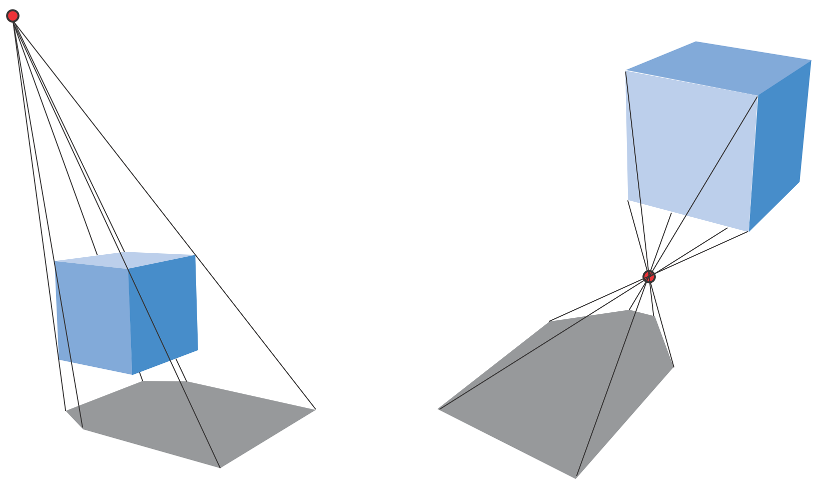


Figure 2.11: Correct shadow on the left (occluder between the plane and the light source), incorrect *antishadow* on the right as the ligh source is below the topmost vertex of the object [7].

can be avoided by implementing the stencil test. In the first pass, the receiver is drawn with stencil operation set to increment on depth pass. Consequently, the shadow object is rendered with depth test disabled and stencil test set to pass only if the stencil value is one and increment on stencil pass with alpha blending turned on. Using stencil buffer also limits the shadows to receiver's area and alleviates the problem with z-fighting [7].

There are several methods that implement soft shadows using planar projection. Heckbert et al. [42] and Herf et al. [45] treat area light source as a grid of point lights. For each of these points a perspective matrix is constructed with frustum having parallelogram as far clipping plane (which is the shadow receiving plane). Subsequently, all occluders are rendered into a texture in dark color. Contributions of all such textures are then averaged in an accumulation texture which is then applied on the receiver plane. The quality of soft shadows is proportional to the amount of the light samples of the extended light source.

Haines [39] improved on the above mentioned method and trades visual quality for speed as his method does not require high amount of render passes in order to achieve soft shadows. First, hard projected shadow is drawn to a texture. The method then finds all silhouette edges and vertices and constructs quadrilaterals and cones respectively with gradient vertex colors ranging from dark at the silhouette edges and white at the projected silhouette edges and cone bases. These objects are then rendered into the shadow texture from top-down perspective with respect to the receiving plane, creating gradient in the shadow texture. The diameter of the cone at the silhouette vertex is based on the vertex height. The resulting texture is then applied on the receiving plane.

When compared to the method by Heckbert et al.[42], Haines' method produces unrealistically big umbra areas, penumbra regions are not physically correct either.

2.4.2 Shadow Texture Techniques

These methods, by design, sit between planar projected shadows and shadows mapping as they try to solve one of the major problems of planar projected shadows – the inability to cast a shadow onto a non-planar receiver. Although this group of methods has no official name, Akenine et al. [7] name them as *shadow texture techniques* or are known as the drop

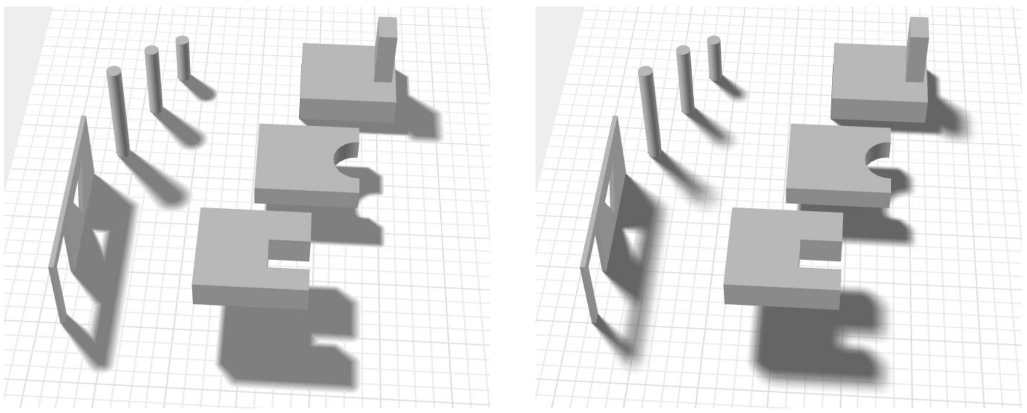
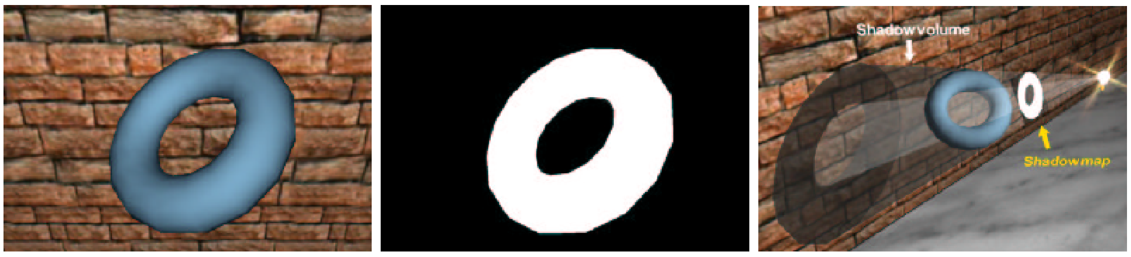


Figure 2.12: Left image – Haines' method, right image – Heckbert and Herf's method calculated with 256 light samples [39]



(a) (b) (c)

Figure 2.13: Nagy's shadow texture method – (a) view from light's perspective, (b) rendered shadow texture, (c) application of the shadow texture [84].

shadows. These methods include e.g. Nguyen's algorithm that uses black-on-white texture [86] or Nagy's method that uses white-on-black texture to decrease lighting intensity on shadow receiving surfaces, as seen in the Figure 2.13 [84].

The technique first renders an occluder into a texture in black while the rest of the texture is initialized to white. The texture does not need to cover the whole area of the receivers as the information in the texture is needed only in places where the shadow is cast. The texture is then applied on the potential receivers. The texturing coordinates are computed using the light's view-projection matrix and computing light-space projection coordinates of a vertex which are subsequently normalized to $< 0, 1 >$ and used to sample the shadow texture.

Some of the drawbacks are shared with the planar projection shadows – the light source may not lie between the occluder and the receiver, otherwise the shadow is cast backwards. Also, the application has to keep track of occluder and receiver objects [7, 30].

These methods, however, provide a simple way to produce soft shadows by simply filtering the shadow texture e.g. by using Gaussian blur. An example of a filteted drop shadow can be seen in the Figure 2.14.

2.4.3 Shadow Mapping

This algorithm, published in 1978 by Williams [111] solves several limitations of the shadow textures. Although published earlier than the shadow textures methods, hardware limita-

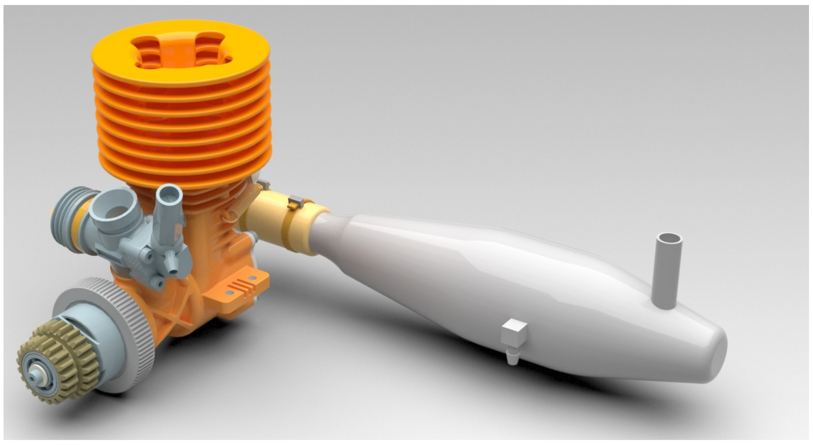


Figure 2.14: Filtered drop shadow rendered in Autodesk Inventor, source: www.caddcentregp.com

tions didn't allow for efficient usage of the method. Instead of rendering just the occluders to a visual texture, shadow mapping renders all visible objects from the light's perspective into a depth texture or so-called *shadow map*, thus the occluders and the receivers are no longer separated. The shadow map stores depth information of all visible surfaces from the light source. Current graphics hardware provides sufficient mechanisms for fast shadow map creation, making it a very popular shadowing algorithm among real-time applications. The scheme of the algorithm can be seen in the Figure 2.15.

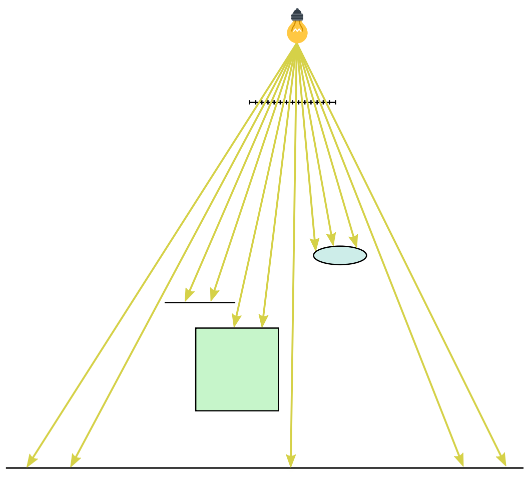
As mentioned above, the first step of the algorithm is creating the shadow map. Scene geometry, visible from the light's perspective, is drawn without shading into the shadow map using light's view (M_{LV}) and projection (M_{LP}) matrices, as seen in the Figure 2.15a. Each texel of the shadow map stores depth the closest object to the light source.

The scene is then rendered from the viewer's perspective and the shadow map is applied to the scene, see Figure 2.15b. The method calculates normalized device coordinates v_L using M_{LV} and M_{LP} for every view sample position v, see Equation (2.9) [30].

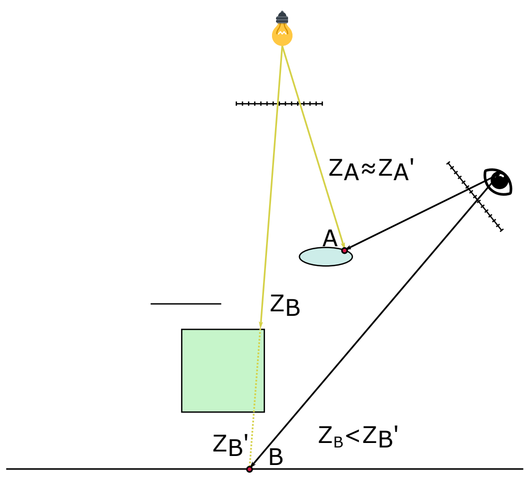
$$
\begin{aligned}
v_L &= M_{LP} \cdot M_{LV} \cdot v \\
v_L &= v_L / v_L.w
\end{aligned}
\tag{2.9}
$$

The texturing coordinates for sampling the shadow map v_S are calculated from $v_L.xy$ by transforming them from range $< -1, 1 >$ to $< 0, 1 >$. By sampling the shadow map using v_S we obtain the depth stored in the shadow map sm_Z. To find out if the view sample v lies in shadow or is lit, we need to compare sm_Z with $v_L.z$ – if the sample's depth $v_L.z$ is larger, it lies in shadow.

One of the downsides of this algorithm is limited shadow map resolution which makes it susceptible to aliasing problems. This problem is evident in the Figure 2.16. A texel in the shadow map covers a certain area in the scene and if the texel orientation (blue line) is not perfectly parallel with the object, the depth value the texel represents is only the value in its centre. In the second step, when the scene is rendered from viewer's perspective and shadow map is sampled, the probability that the view sample will be located exactly in the centre of shadow map's sample is very low, thus the depth obtained from the shadow map (sm_Z) will almost certainly be different from $v_L.z$, causing artifacts – so-called *moiré pattern* or *self-shadow aliasing* as seen in the Figure 2.17a. This problem



(a) Shadow map rendering



(b) Applying shadow map

Figure 2.15: Shadow mapping scheme -scene is first rendered from the light's perspective into a shadow map (left), which is applied during the second pass when the scene is rendered from camera (right).

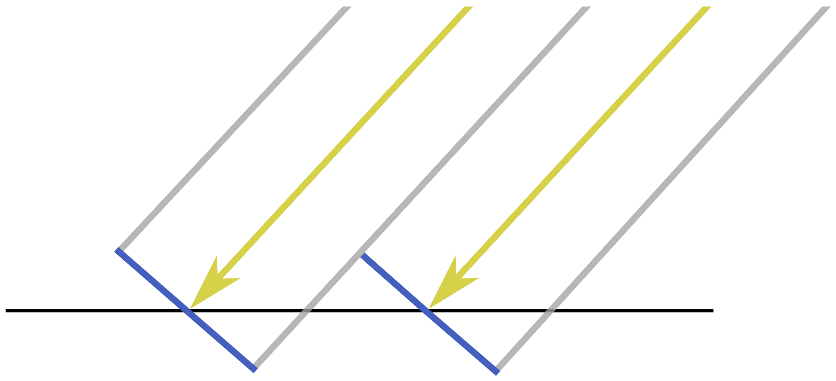


Figure 2.16: Shadow map – samples (blue) contain depth only of its the centre. Grey lines represent sample boundaries. Blue lines are the depth stored at each shadow map sample, yellow lines depict the sampling location. It is clear that a sample stores only a single of many possible depth values (from its centre).

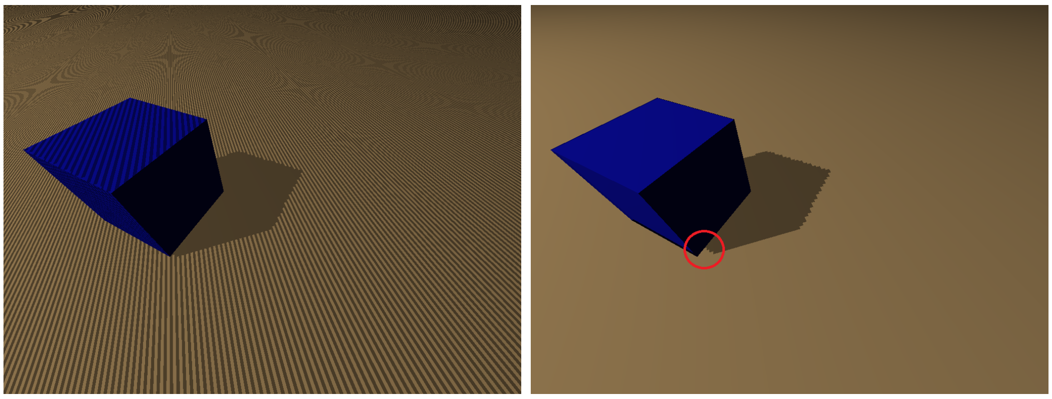

(a) Self-shadowing artifacts (b) Peter Panning due to large bias

Figure 2.17: Shadow mapping problems

is also influenced by numerical limits of the graphics processor which can be improved by making the light's view frustum more tight (e.g. moving the near clipping plane as far as possible and placing the far clipping plane as close as possible) [7, 30]. Milet et al. [82] demonstrated a method to improve shadow map quality by warping using non-orthogonal grid. This approach locally increases resolution for areas which require increased sampling rate, resulting in higher quality shadows on the same shadow map resolution. Similarly to dual paraboloid mapping, it suffers from artifacts caused by linear rasterization where the scene is insufficiently tessellated, dynamic tessellation should be used to deal with this issue.

General solution to help (but not eliminate) self-shadow aliasing is to subtract a value from $v_L.z$ when comparing to sm_Z, called *bias* – to basically move the surface slightly towards the camera [111]. This value might be a constant for the whole scene or computed, mostly based on the surface angle with respect to the light source because the larger the angle between the surface and the light, the larger depth range is covered by a single shadow map texel. Although there are several methods trying to resolve the issue, they require hand-tuning of the parameters [7]. One of the recent methods by Dou et al.[28] tries to compute minimal bias required to eliminate false self-shadowing by tracing a ray R from light through texel's centre point to a tangent plane defined by the fragments position and its normal. The method, however, suffers from noise artifacts when the surface normal is almost perpendicular to R.

When the bias is too large, shadow might become detached and a so-called *light leaking* or *Peter Panning* (based on a cartoon character who was able to detach from his own shadow) can be observed, as in the Figure 2.17b.

The area that a shadow map is able to cover is limited to the light's frustum. Methods based on cascaded shadow maps [31, 114, 68] are able to support large shadowed areas by covering observer's view frustum with several shadow maps, optimizing their placement with respect to the quality of the resulting shadows. These methods are suitable for larger opened scenes with directional light source (sun, moon).

In order to properly model omnidirectional point lights, several frustums have to be arranged and shadow maps rendered since the field-of-view in perspective projection has a limited range. Standard procedure, as proposed by [36], is to create a cube map where each of the six sides is rendered using virtual camera pointing towards one of the directions defined by the axes of the local coordinate system $(\pm x, \pm y, \pm z)$. This approach requires

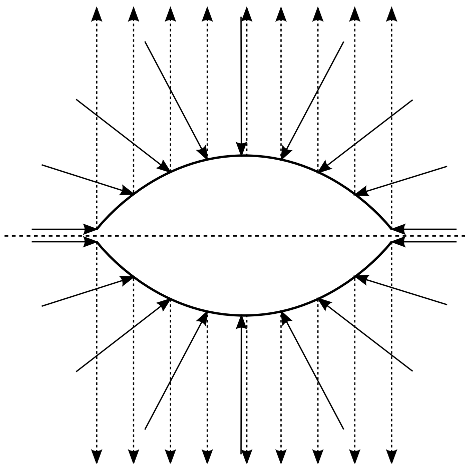


Figure 2.18: Dual-paraboloid shadow mapping. Two paraboloids act as a mirrors which reflect incident rays from the hemisphere into the direction of the paraboloid.

the geometry to be rendered up to six times. Brabec et al.[17] discuss a method based on non-linear projection requiring only two textures instead of six using dual-paraboloid mapping which captures the scene on two paraboloids attached back-to-back, as seen in the Figure 2.18. The downside of this method is that the scene geometry must be highly-tessellated (e.g. triangles should be small), otherwise produces artifacts due to linear interpolation of the rasterization unit. Another approach aimed at reducing the amount of the render passes was proposed by Liao [71]. Instead of using a cube he subdivides the light space using four tetrahedra. This method was further extended and optimized in [27] to support large amount of light sources using tiled depth texture and quad tree to assign tiles of different sizes based on the contribution of the light source in the scene. Although tetrahedron mapping requires less memory than cube mapping, the downside of this method is its higher overhead.

2.4.4 Shadow Volumes

Shadow volumes were first proposed by Crow [24]. The core idea of the algorithm is that shadows are defined by the volume of space they encompass. Unlike shadow mapping which operates in the image space, shadow volumes work in the object space. This shadow volume, or shadow polygon as mentioned in the paper, is constructed from object's contour (or silhouette) edges, extruded in the light direction to the infinity and clipped to the view frustum. Silhouette edges are those edges that have a front and backfacing polygon attached to them, from the light's perspective. The method then sorts the shadow polygons by depth and during the rendering decides whether a fragment lies in the shadow or not by testing the the number of pierced front and backfacing shadow polygons along the ray cast from camera to the scene. If more front-facing polygons are hit during the ray traversal, the resulting fragment is declared as shadowed, lit otherwise. The demonstration of the method can be seen in the Figure 2.19. Crow's method, however, does not count with the case when camera lies inside a shadow volume – the result of the ray test must be reverted otherwise all shadowed regions will be lit and vice versa. As being the core method of pixel-precise shadow rendering, it is analyzed in-depth in the Chapter 3.1.

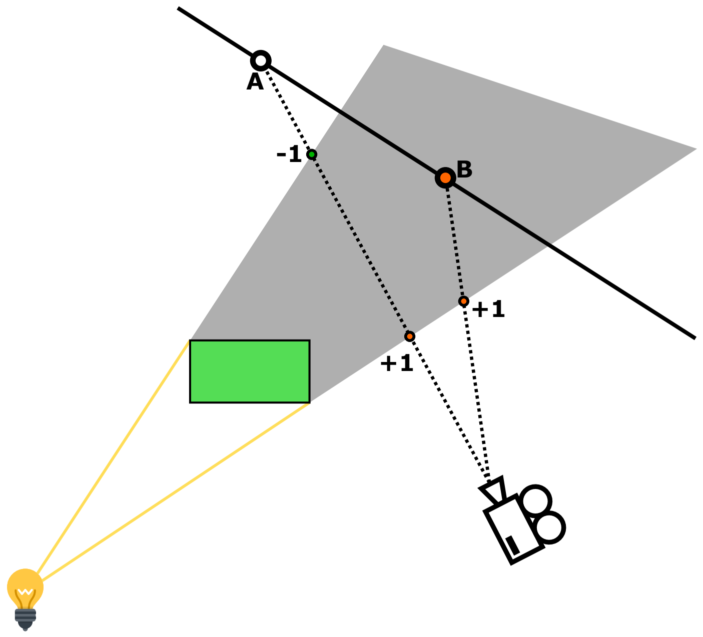


Figure 2.19: Shadow volumes as proposed by Crow[24]. Rays are traced from camera through the scene and shadow polygons. If a front-facing shadow polygon is hit, the value of the ray is incremented, decremented when the shadow polygon is back facing. Sample A is lit as its ray value is zero, sample B is shadowed because the the value of its ray is non-zero, e.g. non-equal amount of front and back-facing shadow polygons were hit.

2.4.5 Ray-Traced Shadows

Ray-tracing was one of the first method for shadow rendering, mentioned in several early works, e.g. [9, 24] and was used primarily for offline rendering. Shadows are also a fundamental part of Whitted-style ray-tracing [110], which is the basis of all modern ray tracing algorithms. The core idea is to cast a ray from the fragment position towards the light source – if the ray hits any occluding geometry, the fragment is considered shadowed. Typically, an acceleration structure (kD tree, BVH) is used to reduce the number of triangles tested by culling parts of scene. With the recent development of acceleration structures and hardware, ray-traced or hybrid approaches became feasible in real-time applications as well. The ray-tracing methods are described in more detail in the Chapter 3.3.

Chapter 3

Existing Precise Shadows from Non-Extended Light Sources

Different areas of the 3D graphics put various requirements on shadow rendering. Gaming industry does not usually require pixel precise shadows, the aim is to provide reasonable quality in certain time budget for a smooth framerate. Imperfections in the shadows can be solved by filtering and variants of shadow mapping are the usual method of choice. Design industry (CAM, CAD), however, often requires precise hard shadows which is difficult to achieve using shadow mapping due to having limited resolution and aliasing problems, thus other methods must be researched to provide an alternative.

3.1 Stencil Shadow Volumes in Depth

Shadow volumes pose as core method in the field of pixel-precise shadow rendering. As mentioned in the Chapter 2.4.4, the method was initially meant for ray-tracing where ray from the camera would increment and decrement its value when entering or leaving a shadow volume.

3.1.1 Construction

In order to create infinite-sized shadow volume geometry, affine coordinates are not sufficient as they cannot efficiently represent point at infinity. Homogeneous coordinates are used instead to extrude from object's edges. Based on Figure 3.1, let's consider two points L and A in homogenous space. Any point A' on the line $\overline{LA}$ can be represented as $\alpha L + \beta L$ [74]:

$$\begin{aligned} A' = \alpha L + \beta A &= (\alpha L_x, \alpha L_y, \alpha L_z, \alpha) + (\beta A_x, \beta A_y, \beta A_z, \beta) \\ &= (\alpha L_x + \beta A_x, \alpha L_y + \beta A_y, \alpha L_z + \beta A_z, \alpha + \beta) \end{aligned} \tag{3.1}$$

Provided we transform the Equation 3.1 to affine space, we divide by $\alpha + \beta$ resulting in:

$$\begin{aligned} &(\frac{\alpha L_x + \beta A_x}{\alpha + \beta}, \frac{\alpha L_y + \beta A_y}{\alpha + \beta}, \frac{\alpha L_z + \beta A_z}{\alpha + \beta}, 1) = \\ &(\frac{\alpha}{\alpha + \beta} L_x + \frac{\beta}{\alpha + \beta} A_x, \frac{\alpha}{\alpha + \beta} L_y + \frac{\beta}{\alpha + \beta} A_y, \frac{\alpha}{\alpha + \beta} L_z + \frac{\beta}{\alpha + \beta} A_z, 1) \end{aligned} \tag{3.2}$$

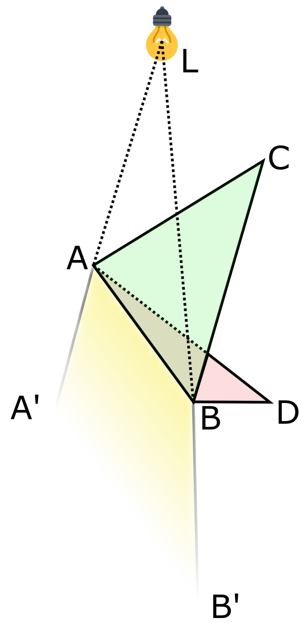


Figure 3.1: Shadow Volume construction. Edge $\overline{AB}$ is common for triangles ABC and ABD. The edge $\overline{AB}$ is a silhouette edge because it has both light front (ABC) and backfacing (ABD) triangles attached. The points A' and B' lie in the infinity and together with points A and B form a so-called *shadow volume side* quad.

Under assumption that:

$$\frac{\alpha+\beta}{\alpha+\beta} = \frac{\alpha}{\alpha+\beta} + \frac{\beta}{\alpha+\beta} = 1 \tag{3.3}$$

we can substitute the following into the Equation 3.2:

$$phi = \frac{\beta}{\alpha+\beta}; \alpha \neq -\beta \tag{3.4}$$

resulting in

$$\begin{aligned} ((1-\phi)L_x + \phi A_x, (1-\phi)L_y + \phi A_y, (1-\phi)L_z + \phi A_z, 1) = \\ (L_x - \phi L_x + \phi A_x, L_y - \phi L_y + \phi A_y, L_z - \phi L_z + \phi A_z, 1) = \\ (Lx + \phi(A_x - L_x), L_y + \phi(A_y - L_y), L_z + \phi(A_z - L_z), 1) = \\ L + \phi(A - L) \end{aligned} \tag{3.5}$$

Equation 3.5 in affine space does not account for all points in homogenous space, because of the condition in Equation 3.4. Points, for which $\alpha = -\beta$, however, exist in homogenous space and substituting this condition to Equation 3.1 yields:

$$A' = -\beta L + \beta A = \beta(A_{xyz} - L_{xyz}, 0) \tag{3.6}$$

Provided that $\alpha = \beta = 1$, the homogeous coordinates of a point lying on the line $\overline{LA}$ in the infinity (A') can be calculated as $A' = (A_{xyz} - L_{xyz}, 0)$.

The shadow volume side quads should also be properly oriented, e.g. having the vertex winding such as the normal points outside of the volume. This is carried out by making a dot product between the light position and the triangle plane and reversing the vertex winding if the test is negative.

A naive approach on shadow volume construction would be to create a shadow volume per every single triangle, e.g. extruding every triangle in the scene to the infinity. Such approach is not suitable for modern hardware due to excessive fillrate but there were architectures such as Pixel Planes that had constant triangle rasterization time, no matter what area did the triangle cover on the screen [34]. Silhouette-based approach, proposed even by Crow in his original paper, is more efficient on modern architectures as not so many shadow volume sides have to be rasterized, thus saving fillrate.

3.1.2 Silhouette Extraction

An edge is considered a silhouette edge if one of the two triangles attached to the edge is light-facing and one is not; or the normal of one of the triangles points to the light and the normal of the latter does not. In order to determine whether an edge is silhouette or not, we first need to construct an equation of the plane for both triangles attached to the edge. Subsequently, both equations are tested against the light's homogenous coordinates, see Equation 3.7, where (a, b, c, d) are the coefficients of the triangle's plane equation and (x, y, z, w) are homogenous coordinates of the light source. Provided the result of the test is positive, the triangle is facing the light source, facing outwards otherwise [33].

$$ax + by + cz + dw \begin{cases} > 0 \\ = 0 \\ < 0 \end{cases} \tag{3.7}$$

The geometry data have to be pre-processed in order to acquire topology information about edge-triangle connectivity. Older implementations of the shadow volume algorithm performed this test on the CPU. Van Waveren [109] described optimal implementation of a silhouette extraction and shadow volume geometry generation using SSE2 instruction set.

With the introduction of programmable graphics pipeline, vertex shader silhouette computation and shadow volume geometry generation was proposed by Brabec et al. [18]. First, the model is pre-processed and topology information is computed. Then, each vertex is given a unique identifier and transformed vertex coordinates are stored in a floating-point RGBA texture. Edges are then rendered as single points with additional vertex data in the vertex attributes. The silhouette test is performed in the fragment shader by fetching the vertex positions from previously created texture. The result of the test, per edge, is the written to another texture. Shadow volumes are created using the information in the vertex texture, the silhouette and the silhouette test data. Vertices of the edges, that are not silhouette, are moved outside the view frustum in the vertex shader. Because the algorithm uses textures to store vertices, the amount of geometry in the scene is limited due to hardware limitations at that time.

When geometry shader was introduced to the graphics pipeline, a new algorithm for computing shadow volume geometry was designed to utilize the new pipeline stage. Stitch et al. [101] extruded shadow volume side geometry and generate the caps in the geometry shader using triangle adjacency mode. The geometry shader then receives 6 vertices on the input for each triangle – 3 vertices of the triangle itself and all opposing vertices for every edge, see Figure 3.2. Because the adjacency information is generated by the hardware, no vertex pre-processing is required. Every edge is processed twice to handle some non-manifold cases like holes (when an edge is connected to a single triangle).

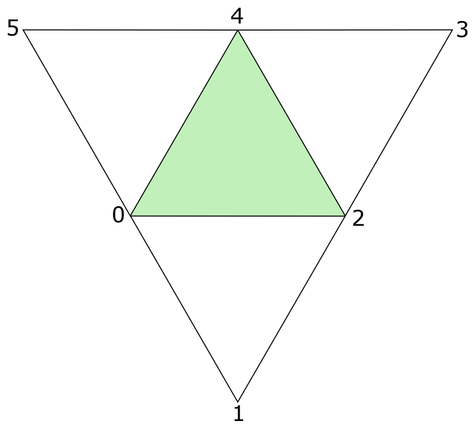


Figure 3.2: Triangle adjacency in geometry shader, vertex order numbered [101]

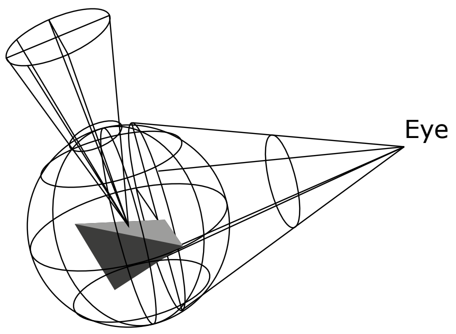


Figure 3.3: Normal cone constructed from a simple object. Traversal consists of creating a cone from the camera position towards the centre of the bounding sphere and testing for intersection with the normal cone [52].

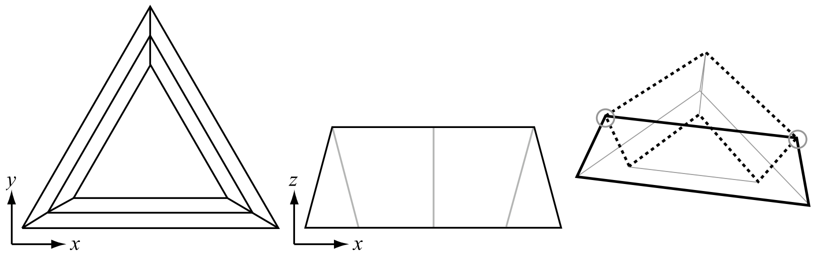


Figure 3.4: A 2-manifold object seen from multiple views, having vertices with degree of 4 (circled). The light source is in the viewer's position [6].

Johnson et al. [52] designed a normal cone hierarchy algorithm to accelerate culling, local minimum distance computation or silhouette extraction. The core idea is to build a binary hierarchical structure where each cell contains a bounding spehere, cone axis and cone semiangle. The cone axis vector is the average of all normals stored in the subtree, the cone angle is the maximum angle between the cone axis and normals. When extracting a silhouette, a view cone is constructed from the camera position towards the bounding sphere of a node, its direction pointing from camera position to the bounding sphere centre. The view cone direction is then tested against the nodes cone for intersection – if intersected, traversal continues through this node. This algorithm is also capable of producing silhouettes of variable precision by stopping the hierarchy traversal once the cone angle reaches a certain threshold. This method, however, is unable to handle non-manifold meshes.

Olson and Zhang [88] designed a silhouette extraction method based on Hough transform of a mesh stored in an octree. Method of Gooch et al.[38] is based on projecting triangle normals onto a Gaussian sphere. Every edge is represented by an arc on the Gaussian sphere. When using orthographic projection, a plane representing the view angle is placed through the origin of the sphere. Every arc intersected by this plane is a silhouette edge. They also proposed a hierarchical version of the algorithm. Pop et al. [93] used dual-space representation of vertices. Silhouette is computed by finding intersection of the viewpoint's dual plane with the duals of the mesh edges. This approach, however, is too performance-expensive [51].

Akenine-Möller et al. [6] explain the degree of a vertex on a silhouette edge as the number of silhouette edges connected to the vertex. A general misconception for 2-manifold objects was that the degree of a vertex on a silhouette edge is always 2. The paper proves that the degree of such edge can be higher than two, but will always be even, as seen in the Figure 3.4. McGuire [77] performed a statistical measurement on 897 2-manifold models trying to generalize silhouette size equation. He placed every model inside a sphere and generated 10 000 points on the sphere as light sources from which he computed the object's silhouette. The result of his finding is $s \approx f^{0.8}$ where s is silhouette size and f is the total number of triangles of the model.

3.1.3 Algorithm Implementations

There are two major implementation of the Shadow Volume algorithm, based on stencil test settings. Method known as *z-pass* or *depth-pass* uses the same principle as designed by Crow – counting visible fragments of the shadow volume along the ray path from the camera. *Z-fail* method reverses this test – counts fragments from the infinity towards the surface.

Z-Pass

Heidmann [44] made a first practical implementation of Crow's algorithm on modern graphics architecture. He based his approach on Jordan curve theorem in 3D space – whenever a volume separates the space to interior and exterior, any curve connecting a point in the interior space with a point in the exterior space will have to intersect a shadow volume uneven number of times. In order to meet Jordan curve criteria, the shadow volume assumes to be watertight [29], although this condition may be lifted – Heidmann's method does not use capped shadow volumes, which also introduces several fail-cases.

Algorithm 1 Heidmann's z-pass shadow volumes, camera perspective, per light source

```
1: Render scene to depth buffer only from camera's perspective
2: Create shadow volume side quads
3: Clear stencil buffer, disable color and depth writes
4: Set stencil operation to increment on depth test pass
5: Enable back face culling
6: Draw shadow volume side quads
7: Set stencil operation to decrement on depth test pass
8: Enable front face culling
9: Draw shadow volume side quads
```

In order to count rays entering and leaving the shadow volumes, Heidmann used the stencil buffer. The algorithm is outlined in Algorithm 1. Drawing front-facing shadow volume side quads and incrementing the stencil volume simulates a ray entering the shadow volume, by drawing back-facing shadow volume side quads and decrementing the stencil value is equal to a ray exiting the volume. The stencil buffer then contains a mask of lit pixels (zero stencil value) and shadowed (non-zero stencil value). This implementation of shadow volume algorithm is also known as *z-pass.*

This approach, however, has several drawbacks. The method produces incorrect results when the near clipping plane intersects any of the shadow volumes side quads, causing holes and breaking the Jordan's theorem. Also, the above-mentioned algorithm does not produce correct results when camera is located inside the shadow, see Figure 3.5.

Both issues can be solved by so-called *capping* of the shadow volume at near clipping plane – either by projecting the scene geometry onto the near clipping plane or initializing the stencil buffer with the correct values. Diefenbach [26] proposed a method in screen space to initialize the stencil buffer with "visible internal pixels of the volume". However, Everitt et al. [33] found several cases where this method fails. McCool's [75] method uses shadow mapping to fix near clipping plane capping. Kilgard's algorithm [61] tries to cap the shadow volumes at the near clipping plane by projecting the scene geometry onto it. Hornus et al. [47] designed a method called *ZP+*. The method constructs a frustum from the light source towards the camera's near plane and shear it so the frustum's far plane matches the camera's near plane. Then, the algorithm uses perspective matrix describing the frustum to render front facing scene geometry and increment stencil on depth pass. This initializes the stencil buffer to the correct values for subsequent z-pass shadow volumes. The numerical problems of this method could be partially alleviated by using depth clamping,

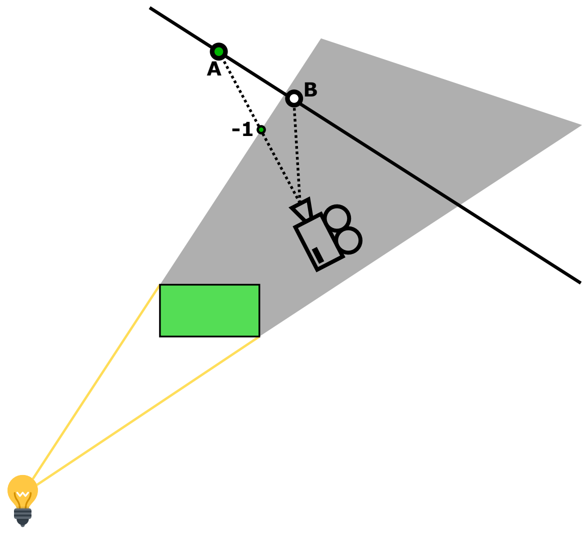


Figure 3.5: Z-pass: inverted shadow test when camera is in shadow

as proposed by Eisemann et al. [30] in method called *++ZP*, but because the projection matrices when rendering from light's and camera's perspective are different, the numerical problems would still occur.

Batagelo et al.[11] designed a hybrid algorithm combining BSP tree and stencil shadows. Every frame, the algorithm constructs a SVBSP (Shadow Volume BSP tree) from a list of sorted shadow casting polygons, discarding those that are inside other polygons. Then, the method computes the set of silhouette edges from the set of visible shadow casting polygons by traversing the BSP tree which are subsequently used in the z-pass. The authors also propose capping the shadow volumes by computing the near clipping plane equation in the world space and clipping every shadow volume polygon against the plane. This process generates a set of intersection vertices which are sorted in polar order and turned into a light cap.

These methods are, however, often complex (Kilgard's method defines several cases that needs to be addressed separately) and suffer from robustness problems that lead to pixel-wide cracks in the shadows. Due to floating point precision, the capping geometry might get clipped away by the front clipping plane, as is the case for ZP+ algorithm.

Z-Fail

Improving robustness of z-pass has been proven to be too problematic, thus a new approach was designed. Bilodeau and Songy [13] and Carmack [19] independently discovered that reversing the whole z-pass stencil test also produces correct results. Instead of incrementing/decrementing the stencil for shadow volumes in front of an object (= when shadow volume side quad fragments pass the depth test), stencil buffer is modified for fragments that fail the depth test, e.g. are behind an object. In other words, rays are now traced from the infinity towards the point of interest. The algorithm is described in Algorithm 2.

Because of this property, the case in the Figure 3.5 will produce valid results for point A because the shadow volume side quad that is penetrated by the ray will not leave any imprint in the stencil buffer, because it would pass the depth test. Similarly, point B would be correctly shadowed because if we prolonged the ray from the camera to the point B, we

Algorithm 2 Z-fail shadow volumes, camera perspective, per light source

```
1: Render scene to depth buffer only from camera's perspective
2: Create shadow volume side quads
3: Clear stencil buffer, disable color and depth writes
4: Set stencil operation to increment on depth test fail
5: Enable front face culling
6: Draw shadow volume side quads
7: Draw shadow volume front and back caps
8: Set stencil operation to decrement on depth test fail
9: Enable back face culling
10: Draw shadow volume side quads
11: Draw shadow volume front and back caps
```

would intersect a shadow volume quad that is located below the surface, thus failing the depth test and modifying the stencil buffer values.

Unlike z-pass, which required shadow volume caps only to handle special cases, z-fail requires the shadow volumes to be watertight, e.g. to be capped on both ends, see Figure 3.6. A so-called *front/light cap* consists of light front-facing occluder geometry, a *back/dark cap* from back-facing geometry projected to the infinity [19]. In case of directional light source, the back cap does not need to be rendered as it consists only from a single vanishing point, the sides are triangles and not quads [112]. Another problem arose as to how to correctly render back caps at the infinity without being clipped by the far clipping plane. Everitt et al. [33] designed a projection matrix P_{inf} that would project such geometry to the far clipping plane, see Equation 3.8, where $Near$ is the near clipping plane distance, T and B are y-axis limits (top, bottom), L and R are x-axis limits (left, right).

$$P_{inf} = \begin{bmatrix} \frac{2\cdot Near}{R-L} & 0 & \frac{R+L}{R-L} & 0 \\ 0 & \frac{2\cdot Near}{T-B} & \frac{T+B}{T-B} & 0 \\ 0 & 0 & -1 & -2\cdot Near \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.8}$$

Modern hardware has a built-in support for clamping depth values at the far clipping plane. Calling `glEnable(GL_DEPTH_CLAMP)` in OpenGL will disable near and far plane clipping during rasterization and will clamp fragment's depth value to $min(z_{near}, z_{far}), max(z_{near}, z_{far})$ where z_{near} and z_{far} define the range of the depth buffer. When using this hardware feature, a conventional projection matrix can be used instead of P_{inf} described in the Equation 3.8. This feature also enables the algorithm to work with orthogonal projection as well [33]. An improved version of this extension was proposed by AMD as `GL_AMD_depth_clamp_separate`, which allows to choose the plane on which the clamping is performed [16].

Another hardware feature that reduces the number of draw calls is called *two-sided stencil test*, in OpenGL as extension `GL_EXT_stencil_two_side` [60] and since OpenGL 2.0 standardized as core function `glStencilOpSeparate`. Instead of rendering the front and back faces separately with different culling and stencil settings twice, shadow volumes can be rendered once by setting stencil operation for both sides and with culling disabled [33]. An outline of the optimized algorithm can be seen in Algorithm 3.

Z-fail, although solving several problems of z-pass, has to work with 2-manifold watertight geometry only. It is also generally slower (up to two times) than z-pass due to rendering of the front and back caps [30].

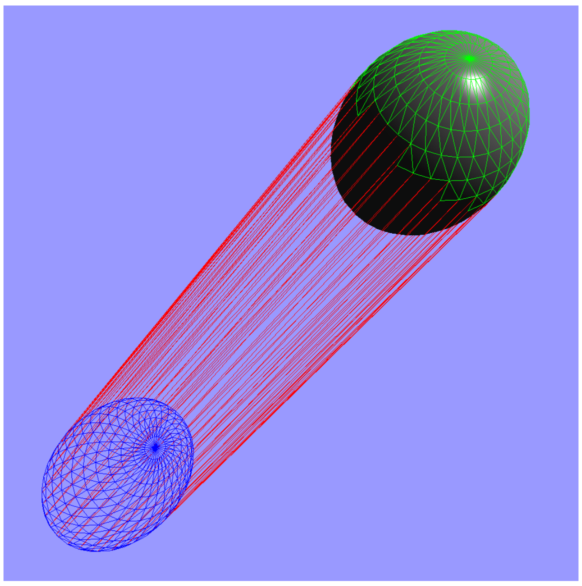


Figure 3.6: Visualized shadow volume of a sphere. Red – shadow volume side geometry (extruded silhouette edges), green – front cap, blue – back cap rendered at far clipping plane. Silhouette of the object can be seen on the boundaries of the front cap. Each triangle's normal points outwards of the volume.

Algorithm 3 Z-fail shadow volumes with two-sided stencil testing

```
1: Render scene to depth buffer only from camera's perspective
2: Create shadow volume side quads
3: Clear stencil buffer, disable color and depth writes
4: Set two-sided stencil operation to increment on depth test fail for back faces
5: Set two-sided stencil operation to decrement on depth test fail for front faces
6: Draw shadow volume side quads
7: Draw shadow volume front and back caps
```

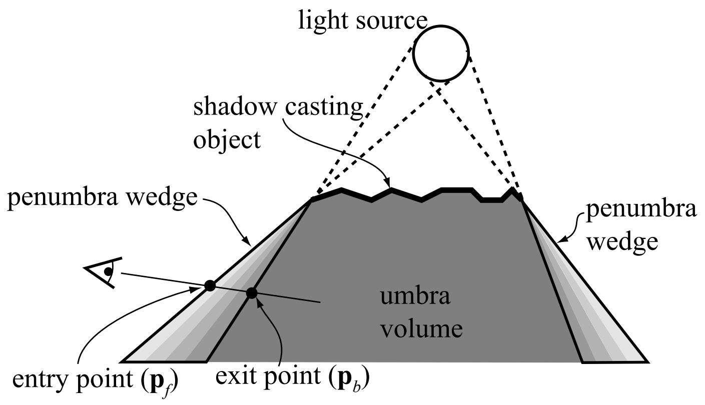


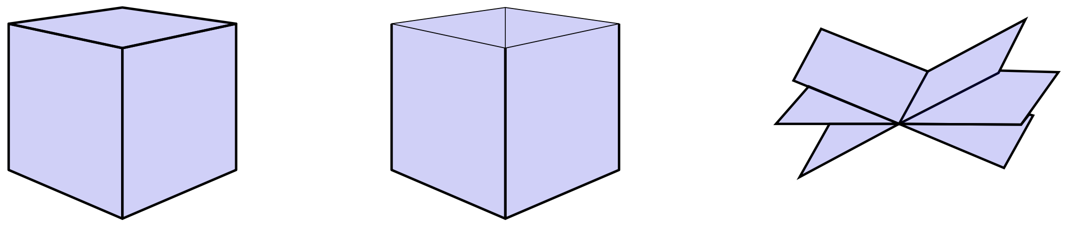
Figure 3.7: Soft shadow volumes generated from penumbra wedges [5]

3.1.4 Soft Shadow Volumes

Although shadow volumes are primarily used when precise shadows are needed, there were attempts to modify the algorithm for soft shadows. Unlike shadow maps, which can be easily filtered, the method proposed by Akenine-Möller et al.[5] renders penumbra wedges similarly to soft projected shadows by Heckbert et al. [42]. The method renders these wedges into 8-16 bit stencil buffer, which is later used to modulate the scene's lighting. The wedges are constructed from silhouette edges and a spherical light source, as seen in the Figure 3.7. One of the issues the algorithm faces is to address the wedge connectivity to avoid artifacts, as well as overlapping wedges. The method is limited to 2-manifold objects.

Assarsson et al. [10] further improved this algorithm. Shadow wedge construction from silhouette edges is now independent on each other. The edge's vertices $e_0, e1$ are sorted by the distance towards the light and provided e_1 is the vertex closer to the light source, vertex e_0 is moved to the same distance from the light as e_1, creating vertex e_0'. The wedge is then created from vertices e_0' and e_1. The shadow is no longer exact, but the above mentioned approach works faster than the original approach. The shadow geometry is then rendered in two passes, first pass projects fragments inside the wedges to the area light source, second pass compensates for overestimated umbra regions from the first pass.

Laine et al. [67] also use two stage algorithm. First step constructs a hemicube acceleration structure from wedges created from extracted silhouette edges. They are conservatively rasterized onto the hemicube, which is centered and oriented according to the light source. After this step, each cell in the hemicube contains a list of wedges whose footprint intersects the cell. Then, the method determines which light samples are visible from point p by acquiring the list of corresponding wedges from the hemicube and computing the light source occlusion. This method was basis for Lehtinen et al. [69], who pointed out on several problems – the method is overly conservative due to two dimensional nature of the hemicube, unpredictable performance and big performance penalties when transforming the light. To address these problems, he rasterizes the wedges into a hierarchical 3D grid, implemented as BSP tree, where each cell contains the list of wedges either intersecting or containing the cell. The hierarchical approach greatly improved the performance as the number of wedges that needed processing during traversal was lowered by an order of magnitude.



(a) 2-manifold　(b) 2-manifold with a boundary　(c) Non-manifold

Figure 3.8: Occluder types, based on geometry topology. A 2-manifold mesh consists of edges having exactly 2 adjacent faces. 2-manifold with edge boundary (mesh with a hole, a plane, a circle, ...) has 1-2 faces per edge. Non-manifold mesh has more than 2 faces per edge [62]

3.1.5 Non-Manifold Meshes

The problem of virtually all previously mentioned shadow volume algorithms is their inability to handle non-manifold meshes or 2-manifold meshes with a boundary, e.g. meshes with holes, trailing edges with only one triangle attached or edges with more than 2 adjacent triangles, as seen in the Figure 3.8. The problem is that non-manifold models may not generate equal number of front-facing and back-facing shadow volume side quads, relative to the camera. This was first observed by Bergeron [12] and proposed a solution – to increment/decrement the ray value by 2 when a shadow volume side quad extruded from a silhouette edge having 2 triangles attached. When the ray hits a quad extruded from an edge that has only a single triangle attached to it, its value is changed by 1. This algorithm is able to correctly render shadow volumes from 2-manifold casters with a boundary.

Aldridge et al. [8] pointed out that geometry with more than 2 triangles per edge also need to be addressed. The paper also notes that triangle winding constraint plays important role in the current algorithms (all triangles are expected to have consistent winding throughout the whole model) when determining if an edge is a silhouette one or not. But winding as a constraint has to be lifted when dealing with arbitrary triangle soups where an edge might be adjacent to either more than 2 triangles or to triangles with inconsistent winding. Instead of storing winding information in the edges (with respect to the triangles), the method stores winding for every triangle with respect to every edge it connects to. He introduced counters for every edge. If a triangle is determined as light-facing, the edge counters for all attached edges are either incremented if the triangle and the edge have the same winding, decremented otherwise. If the edge counter is less than zero, the edge is then extruded with reversed winding, if positive then the edge is extruded with the same winding as its direction. If the counter is zero, the edge is not a silhouette edge. Extruded shadow volume side quads are then rendered edge-counter-times. The algorithm, however, requires the mesh to be orientable, e.g. not for two-sided geometry. Also, he ignores light back-facing geometry (e.g. geometry with reversed winding in general) which may still contribute to object's shadow. The method would produce different results using the scene in the Figure 3.9 if triangle winding is reversed for any of the triangles.

Standard graphics APIs don't allow for arbitrary stencil value to be added, thus when the edge counter in the algorithm of Aldgridge et al. is greater than 1, the shadow volume side quad has to be rendered more than once. McGuire [78] tried to solve this issue by

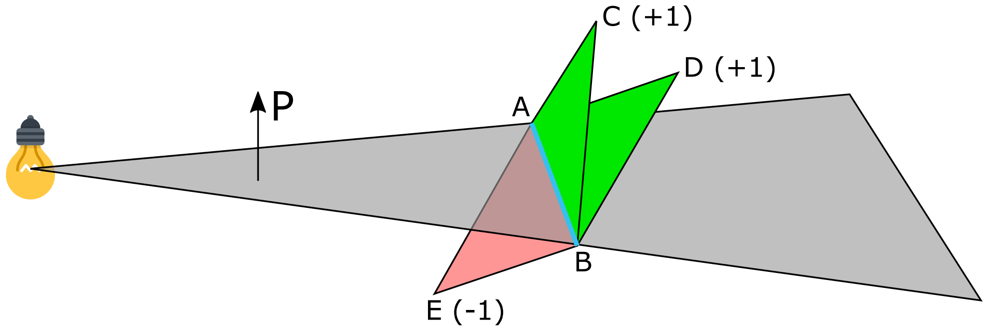


Figure 3.9: Edge multiplicity calculation. Edge AB is extruded along the light direction, creating a light-plane P. Multiplicity is computed as the number of triangles in front of the P minus triangles at the back. The multiplicity of the edge AB is 1.

utilizing additive blending. Adding an arbitrary value to stencil buffer is available on AMD hardware using `GL_AMD_stencil_operation_extended` OpenGL extension [96].

Kim et al. [62] extended the algorithm of Aldridge et al. [8] to non-oriented triangle meshes, being able to render almost any triangle soup, independent of winding. Let's consider an edge AB, as seen in the Figure 3.9. The edge has 3 triangles attached to it (ABC, ABD, ABE). We will call vertices C, D and E as *opposite vertices* with respect to the edge AB. Every opposite vertex is then tested against the light plane's P equation. If the result is positive, the multiplicity of the edge is incremented, if negative – decremented. The absolute of the resulting multiplicity is the number of times the stencil needs to be incremented for the extruded non-manifold edge AB, e.g. the number of times the shadow volume side quad has to be rendered. Positive multiplicity means that the shadow volume side quad, extruded from the edge, will be rendered with the same winding as the edge is stored with, reversed otherwise. Kim has also proven that the resulting stencil mask equals to the number of surfaces that are between the light source and the receiving surface. One of the requirements of the algorithm is consistency – when an edge is extracted from a triangle, it has to be correctly identified no matter the edge direction inside the triangle. Based on this assumption, even if the winding of all triangles adjacent to a silhouette edge changes, the multiplicity does not.

Kim designed his method to work with z-pass, although the extension to support z-fail is trivial. Aldridge et al. were not exact on how to render the caps properly, stating that either light front or backfacing triangles alone may introduce artifacts. Kim proposed to render the front cap with multiplicity $+1$ when camera and light are on the same side, -1 when on opposite sides with respect to the triangle being rendered. The back cap can be rendered with opposite sign as the front cap. This mean that caps are rendered from all of the object's geometry.

Kim's method also supports transparent shadow casters by utilizing a float buffer instead of stencil buffer, called a light map. Every extruded edge will multiply the value in the light map by the transparency value of the caster, per channel. However, only uniformly-coloured transparent shadow casters are supported. His algorithm, in general, is able to render almost any triangle soup, but suffers from determinism problems when a triangle is almost parallel to the light plane.

3.1.6 Optimizations

Performance of the shadow volumes algorithm can be improved in several areas. As shadow volumes rasterize large amount of geometry, several publications were focused on fillrate reduction. One way to reduce fillrate is to omit shadow casters that are already in shadow. Vlachos and Card [107] sort the input polygons by the closest vertex from the light's perspective. For each polygon, a small frustum from light source is created, the far plane being the polygon itself. All polygons that are inside this frustum, are discarded from the input. This method is not suitable for modern large scenes in real-time. CC Shadow Volumes [72] reduce the amount of rasterization by culling away shadow casters that are already in shadow. The method uses shadow map and occlusion queries to create a set of *potential shadow casters* from light's perspective and a set of *potential shadow receivers* from camera perspective. The algorithm is also able to cull shadow casters whose shadows are not visible on the screen by rendering bounding volumes of potential shadow receivers into the stencil buffer when the depth test fails, from light's viewpoint. This method, however, is not suitable for omni-directional light sources.

There is a category of methods that use acceleration structures in order to speed up the rendering process of shadow volumes. Stitch et al. [101] used hierarchical occlusion culling of axis-aligned bounding boxes (AABB) extended to shadow volumes. The method creates a shadow volume from node's AABB and tests it for visibility by an occlusion query. If the node's shadow is not visible, all subnodes can be skipped. Binary Space Partitioning (BSP)-based methods that don't rasterize shadow volumes are discussed in the Chapter 3.2.

Laine [66] attempted to combine z-pass and z-fail methods, detecting when camera is in shadow in order to use faster z-pass more frequently. His method uses a low-resolution shadow map to locally decide whether to use z-pass or z-fail and modified the stencil algorithm to test the depth against a split plane.

McGuire [76] proposed not to extrude the shadow volumes to infinity, but rather to the range of the light source. This might, however, cause the dark cap to be visible rather than being culled. Second proposed optimization is to compute the area on the screen that the light source covers by projecting a sphere representing the light radius on the screen. Then, the scissor test is enabled during the shadow volume rendering with dimensions covering the projected sphere, but such optimization can only be used when the light source is farther away from the camera. Similar optimization can be used for the depth range – objects outside the projected sphere's depth range don't have to be rendered at all in the illumination pass. This can be achieved by setting the depth range of the depth buffer to the range of the projected sphere.

Aila et al. [2] designed a hierarchical depth-stencil buffer method of rendering shadow volumes. Their algorithm subdivides the screen into 8x8 pixel tiles and stores minimum and maximum depth per tile, creating an axis-aligned bounding box. This structure is filled in the depth pre-pass. If the shadow polygon does not intersect the bounding box (tile) during the shadow volume rendering process, all points inside the bounding box are either all lit or all shadowed. When all shadow volumes are rendered, the content of the cells is classified as either being fully lit, fully shadowed or there was an intersection with a shadow volume, thus a shadow boundary is present in the tile. Second pass of the shadow volume rendering processes only the tiles that were intersected.

Several optimizations are discussed in Legyels's GDC'05 talk [70]. Similarly to McGuire, he describes scissor test optimization based on projecting the light boundaries to the screen for point light source, as seen in the Figure 3.10. Another optimization is depth bounds

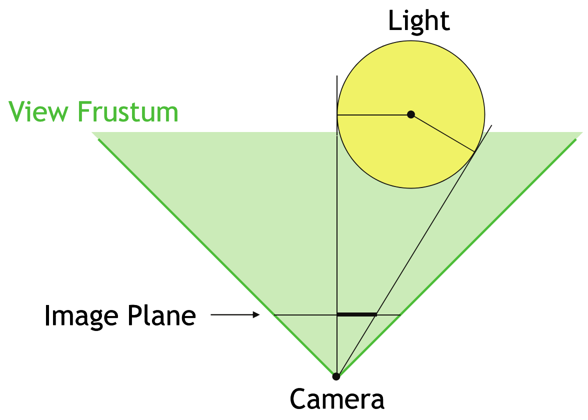


Figure 3.10: Limiting shadow volume rendering of point lights. A light range is projected to the screen and a bounding rectangle is placed around this area, which is ten used for scissor test during shadow volume rendering [70].

test that works similarly to scissor test optimization but in terms of depth – limiting the extension of shadow volumes to the light range. This optimization saves depth and stencil writes for geometry outside the light's range.

Röttger et al. [95] proposed a modification to shadow volumes that uses alpha channel using depth test and blending instead of stencil buffer so it could be used on PlayStation 2. Instead of increment and decrement operations, the method is doubling and halving using `gl_BlendFunc(GL_DST_COLOR, GL_ZERO)` and setting vertex brightness to either 0.5 or 1; the buffer is initialized with a value of 0.25 which is also a value for a lit fragment. Because of color value clamping, even if multiple shadow volumes overlap a fragment, its value will always be either 0.25, 0.5 or 1. The author also proposed to halve the alpha buffer resolution in order to improve shadow volume fill-rate, but this is contrary to the shadow volume nature as a precise method.

McGuire [80] offers several guidelines on implementation of shadow volumes in games. He designed a method to omit the inner model geometry from shadow volume rendering which might introduce visual artifacts under certain scenarios. Another proposed optimization is not to generate light caps of occluding geometry when it is not viewed by the camera as they don't contribute to the shadow count computation. However, modern hardware culling capabilities is able to deal with such geometry easily. He makes use of the scissor test to limit the range of shadow volumes generation and to save fill rate.

3.2 Algorithms Using Acceleration Structures

Although these methods are similar to shadow volumes, they pose a separate category as they neither rasterize shadow volumes geometry into the stencil buffer nor do any ray increment/decrement operation. Instead, these methods traverse an acceleration structure constructed either from the view samples or from the scene's geometry to determine view sample's light visibility.

3.2.1 Methods Building Acceleration Structure from Scene Geometry

Using BSP trees as an acceleration structure was first proposed in SVBSB (Shadow Volume BSP) [20]. Each internal node of the SVBSP is associated with a *shadow plane*, created

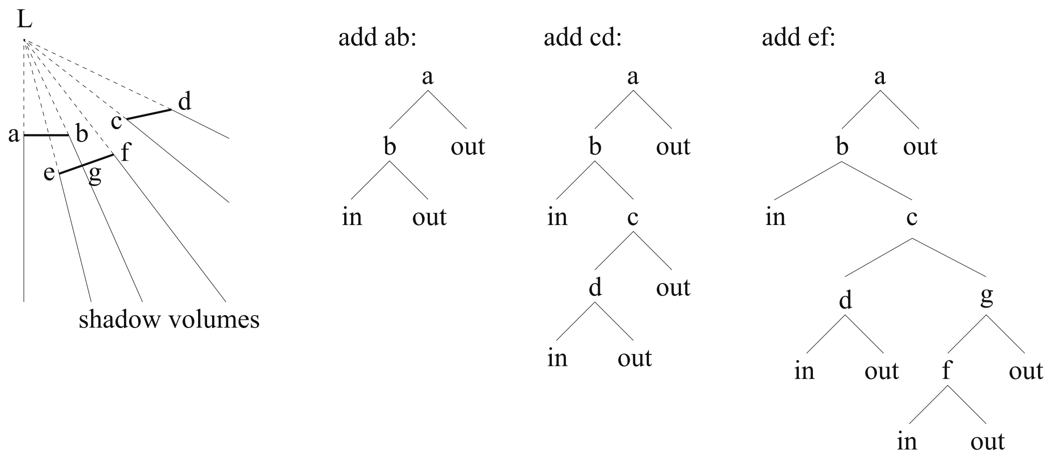


Figure 3.11: 2D case of SVBSP – adding edges to the tree structure. When edge ef is added, it's split to two edges at point g due to intersection with a plane cast through point b [112].

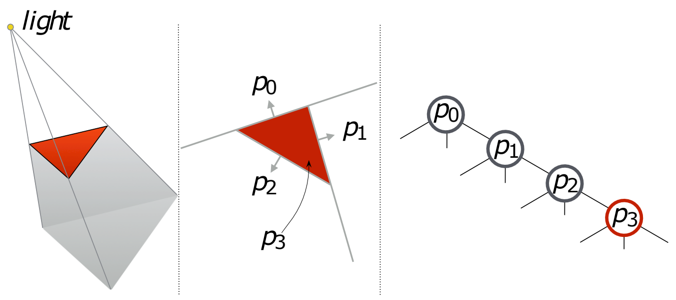


Figure 3.12: Planes created from a triangle frustum: $p0$ - $p2$ shadow planes, $p3$ capping plane. The resulting TOP subtree is displayed on the right. The nodes have positive child node on the right, intersecting in the middle and negative node on the right. [37].

from the light position and an edge. The tree is built incrementally from a front-to-back sorted set of polygons with respect to the light source, thus a triangle being processed is tested only against the planes that are already in the tree (and not future planes), as seen in the Figure 3.11. To test a fragment, one traverses the SVBSP with view sample coordinates and finds out, whether it's located inside a lit or shadowed space of a node, then traverses the respective child node. The structure, however, needs to be rebuilt when the light position changes. Also, polygon clipping during the build process is an expensive operation which introduces numerical errors.

This approach was recently revisited by Partitioned Shadow Volumes (PSV) [37]. Although the acceleration structure needs to be rebuilt when the light source moves, the method no longer clips polygons, thus avoiding robustness issues. Each triangle shadow frustum consists of 4 planes – three shadow planes and one capping plane which is constructed from the triangle itself, as see in the Figure 3.12. Instead of clipping, every tree node representing a shadow plane is not binary but ternary – the third intersection node points to triangles which are intersected by the shadow plane, calling the resulting structure a TOP tree. The triangles are also not sorted but inserted in random order. When inserting a triangle, starting from the root node, all the triangle vertices are tested against a plane. If all vertices lie on the same side of the node's plane, respective child node is cho-

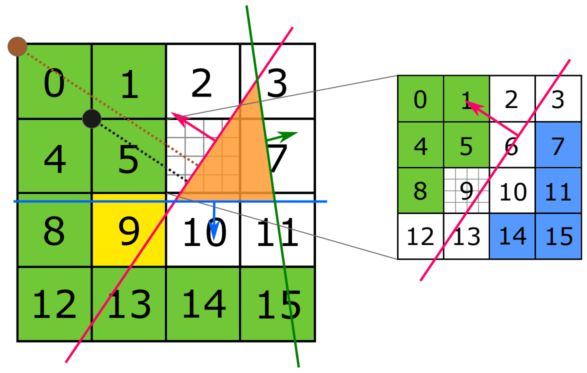


Figure 3.13: Rasterization of triangle frustum into the hierarchical depth buffer. Green tiles are trivially rejected, yellow tile failed trivial test but does not intersect, blue tiles are trivially accepted. Brown and black points of tile 0 are trivial accept and reject test points. The right part of the image shows a close-up of tile 6 [99].

sen (either *positive* or *negative* node). If the plane intersects the triangle, the *intersection* node is chosen. This process repeats until a child node is found, which is then replaced by the shadow frustum of the triangle. Traversal is similar to SVBSB except it ends when the fragment is tested to be inside a capping plane (which is always stored last for every frustum). The algorithm is able to support transparent shadow casters.

PSV was improved by Mora et al. [83]. They identified two major shortcomings of PSV – stack as not being GPU-friendly and lit areas being too costly. The tree depth complexity increases the cost of lit surfaces because in order to find out if a fragment is lit, it must not be contained in any of the frustums encountered during the traversal, thus posing the worst case of the traversal. Mora added depth information to each node, computed as minimum distance between the triangle and the light source. This allowed for skipping node testing if the node lies farther away from the light than the fragment tested. The method proposed a stackless implementation using links inside the structure, trading register pressure for memory bandwidth and a hybrid approach with shorter stack and switching to stackless version when the stack is full. However, the implementation of z-pass stencil shadow volumes that was used during evaluation Mora's method was very ineffective.

Deves et al. [25] improved the method even further. Their contribution was improved scalability with the increasing geometry complexity using clustering. The geometry in the scene is clustered into the groups of 32 elements, encapsulated with either a sphere or a capsule, and stored in a metric tree [105]. The metric of choice is the angular distance, e.g. cone angle created from the light and the bounding sphere or angle between the two support points of the capsule and the light source. The hierarchy is then built by choosing a random pivot node and angular distance such that it divides the remaining elements into possibly two equally-sized sets. The process repeats until all nodes are processed. Traversal is similar to methods mentioned above, except the fragment is first tested against the bounding volume, if located inside then tests the triangles, otherwise traversal continues. This method, however, needs expensive preprocess and is more suitable for scenes with large amount of geometry, other methods are faster for smaller scenes.

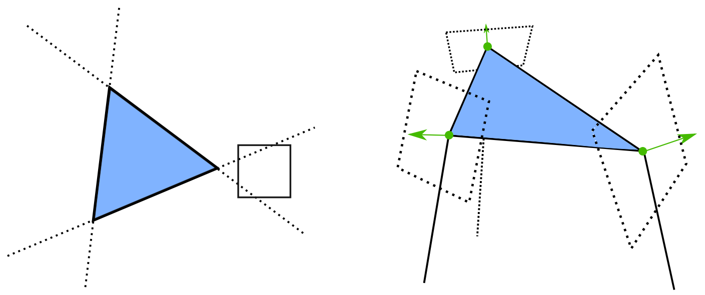


Figure 3.14: Left – false positive tile not lying completely outside any plane. Right image shows the extruded shadow volume of the triangle with extra plane per each vertex as seen in the right image [98].

3.2.2 Methods Building Acceleration Structure from View Samples

Alias-free shadow maps [3] builds a hierarchy of view samples in order to determine their visibility. The method uses a 2D BSP tree to construct a view sample hierarchy, then tests each triangle against the hierarchy for occlusion and traverses down if a node is at least partially occluded. If all samples in a node are determined as shadowed, whole node is marked shadowed and is no longer traversed by subsequent geometry.

A version of alias-free shadow maps by Sintorn et al. [97] creates a list per shadow map texel that stores all view samples which gets projected to this particular shadow map texel. The method then renders scene triangle using light's projection and tests whether the samples, stored in a list in every shadow map texel, are occluded with the currently rendered primitive. Conservative rasterization must be utilized as even marginally covered shadow map texels need to be accounted for. The method does not support omni-directional light sources.

Per-Triangle Shadow Volumes [99] builds a hierarchical depth buffer from the view samples. A single node defines a bounding box in normalized device coordinates plus depth min-max value, called a tile. The upper level covers a 4x4 area of the level below. Then, the method uses software shadow volume rasterizer, written in CUDA, that renders the shadow frustum of every triangle into the hierarchical depth buffer from camera's perspective. CUDA was chosen as no other GPGPU language provided `ballot` intrinsic at that time. When rasterizing the shadow frusta, it can either intersect a tile, in which case the tile has to be subdivided or individual samples tested against the shadow frustum on the lowest level, completely envelop the tile which results in the whole tile to be marked as shadowed, or lying out of the tile, see Figure 3.13. When all triangles are processed, the hierarchy is merged by propagating shadow bitmasks from the upper level to the lower level, where all the view samples receive shadowed/lit flag. Although this method is capable of producing shadows from transparent casters, it is dependant on bias value when testing tiles against the frustum planes. Also cases like tile number 9 on the left side in the Figure 3.13 are prone to producing false positives.

This technique is further extended in Clustered Per-Triangle Shadow Volumes [98] by building an acceleration structure from view samples based on their proximity and testing triangle frusta against the structure. The view samples are grouped into the clusters of 8 by 8. Every cluster then receives a Morton code based on their screen-space coordinates. Then, a full hierarchical tree is build from the clusters with branching factor of 32 and depth of 5. Every node contains a 32-bit bitmask indicating the presence of each of the children

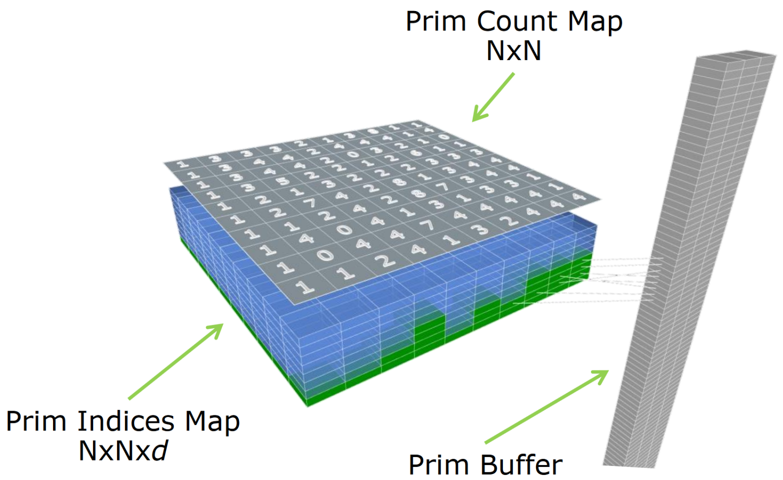


Figure 3.15: Data structures used by Deep Primitive Map [102].

and its AABB for faster culling. The hierarchy is build in a similar way as in [99] but in order to address the false positive problem from the previous method caused by wedges formed of any 2 frustum planes as see in the Figure 3.14, a plane is added per-vertex that contains the vertex and the light source, making the frustum-tile intersection test robust. The traversal is dispatched as collection of jobs, each job processes a single triangle and is assigned to a warp of 32 thread. Each thread of the warp tests the triangle against its particular child, the result is then broadcast to all warp threads and shadow mask for all children is set. If the lowest level is processed, the view samples in the cluster are tested, otherwise the traversal continues for every intersected child.

Story [102] has proposed the Deep Primitive Map (DPM) technique that works similarly to the IZB, but instead of storing view samples, it makes lists of all triangle IDs that cover a particular IZB cell. The data structure seen in Figure 3.15 is sampled as a shadow map to obtain the list of triangle IDs that are ray-tested from the view sample position towards the light. For the method to work correctly, conservative rasterization needs to be used as all triangles touching a particular IZB cell must be stored in its list. As the method stores triangle IDs, the maximum length of the lists has to be experimentally determined for every scene.

Frustum Traced Shadows [113] conservatively renders the scene geometry against the IZB. A shadow volume is created from a triangle and the light source and every IZB cell touched or covered by the triangle then tests its view-space samples against the triangle's frustum (shadow volume).

A common problem of all the IZB-based methods is long lists, which cause low GPU occupancy and slow IZB traversal. In order to match eye and light space sampling, Wyman et al.[113] propose that ideal parametrization for these methods are cascades, opting for the technique described by Lauritzen et al.[68]. Story [103] uses dynamic reprojection of the light space area where the list lengths exceed a selected threshold. A second projection matrix is computed that projects the selected area into the second IZB list head texture.

3.3 Ray Tracing Methods

Ray tracing was the first method used to render shadows in 3D graphics, although an offline method for several decades. Development of ray tracing focused mostly on improving its performance by providing better acceleration structures and optimizing ray traversal by techniques like ray sorting [35] and packeting [94]. The traversal optimizations allow the ray tracer to better utilize the GPU hardware as secondary rays become very divergent in direction and invalidate cache data frequently as neighbouring threads may access completely different parts of the acceleration structure. Also, ray packet saves on the amount of operations. Shadow rays are different as they either all converge to a single point (point light) or go in the same direction (directional light). To make matter even simpler, it is usually sufficient to detect any collision with the scene geometry and not the closest, provided the scene does not have transparent casters.

Soft shadow can be achieved by distributed ray tracing which casts multiple rays towards and area light source and averages their results to a percentage of occlusion [22].

A recent method by Boksanský et al. [15] uses standard forward-rendering rasterization pipeline for shading but recently introduced hardware-accelerated raytracing (nVidia RTX) for shadows. Bounding volume hierarchy (BVH) is used as an acceleration structure generated automatically by the RTX API. In order to make the shadow tracing effective, the method uses penumbra detection and adaptive sampling to lower the number of shadow rays based on the sample visibility in four previous frames. Data from previous frames are also used to determine the sampling rate and the filter kernel sizes. With the introduction of hardware acceleration for ray tracing in modern graphics hardware, ray tracing adaptation as (not only) a shadowing technique for real-time applications will probably increase in the near future.

Chapter 4

Comparison of Omnidirectional Shadow Mapping Methods

The motivation of this research was to compare omnidirectional techniques available for shadow mapping, in terms of performance, quality and ease of implementation.

Omnidirectional shadow maps can be represented by faces of a cube map [36]. In this case, six render passes are needed to fill the data into the cube map faces. Secondly, the Dual–Paraboloid Shadow Mapping (DPSM) technique [17, 89] can be used. It is capable of capturing the whole environment in two render passes using parabolic projection. However, the mapping is not linear and thus not fully supported by contemporary graphics hardware. Recently, different techniques have been introduced [21, 46] that discuss other types of parametrizations.

This method was presented on the WSCG conference [85]; my contribution was designing and implementing the testing framework, including the tests and performing the evaluation.

4.1 Cube Mapping and Optimizations

Cube mapping technique was first proposed by Gerasimov [36] where a set of 6 textures forming a cubemap captures the the depth from the light source in $\pm x$, $\pm y$ and $\pm z$ directions in the same way as standard shadow mapping does. King and Newhall [63] introduced a method to reduce the number of shadow frusta that needs to be rendered. If the light source is outside the view frustum, then we can skip rendering of at least one face of the shadow map, as seen in the Figure 4.1. For our experiments, we used the following technique for efficient culling of cube map faces. A camera view frustum and each cube map frustum are tested for their mutual intersection. Those frusta that do not intersect can be discarded for further rendering because they do not affect the final image. The efficient culling of arbitrary frustum F against the camera view frustum V works as follows. A frustum is defined by 8 boundary points and 12 boundary edges. To determine whether the two frusta intersect, two symmetric tests have to be performed as seen in the Figure 4.2.

Firstly, it should be tested whether a boundary point of one frustum lies inside other frustum. Secondly, boundary edges of one frustum are tested for intersection against one or more clip planes of other frustum. For each face of the cube shadow map, we investigate whether the camera view frustum intersects the shadow face frustum and vice versa. If it is not the case, the shadow face frustum does not affect the scene and we can skip the additional processing. It is also necessary to take into account shadow casters outside the

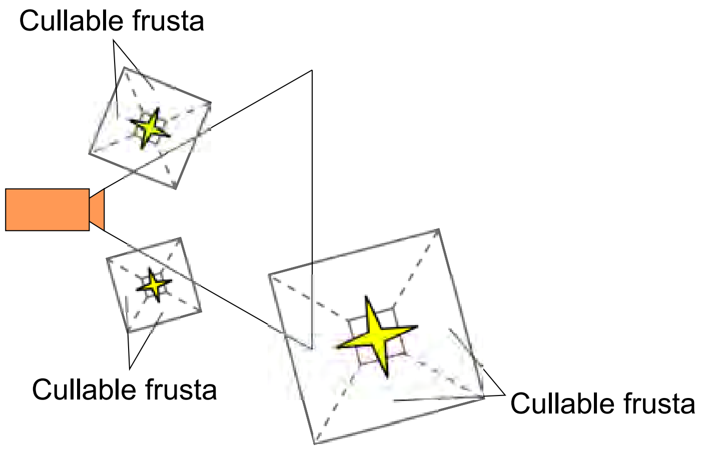
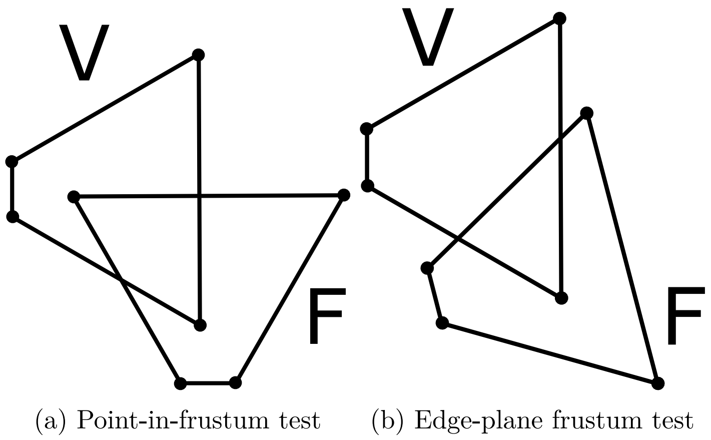


Figure 4.1: Shadow map frusta from omnidirectional point lights sources in 2D. At least on of the frustum is cullable if the light is not inside the camera's view frustum.



(a) Point-in-frustum test (b) Edge-plane frustum test

Figure 4.2: Two-stage frustum test

view frustum. If we cull the shadow caster against the view frustum, the projected shadow may still be visible in the view frustum. On the other hand, culling the shadow caster against the cube map frustum draws invisible shadows as well. King also [63] suggests to use frustum-frustum intersection test described above for the shadow casters as well. Since we use point light sources, rays are emitted from a single point towards all shadow casters. This is analogous to the perspective projections. If the shadow casters are enclosed by bounding objects, frusta representing the projected shadows can be created and then the frustum-frustum test can be applied in this case as well.

4.2 Dual-Paraboloid Shadow Maps

Parabolic projection, proposed by [17], is based on a totally reflective mirror in a shape of a paraboloid that reflects incident rays from a hemisphere to the paraboloid direction. The function of the paraboloid can be seen in the Equation (4.1).

$$f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2), \qquad x^2 + y^2 \leq 1 \tag{4.1}$$

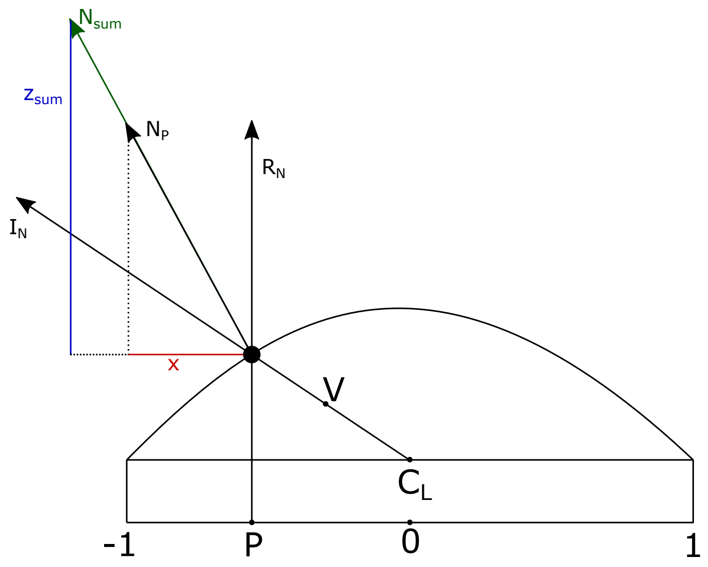


Figure 4.3: Scheme of dual-paraboloid mapping. A vertex V is projected to paraboloid to point P. N_P is the normal vector on the intersection of incident vector I_N with the paraboloid, R_N is its reflected vector along N_P. N_{sum} is the sum of I_N and R_N [1].

The texturing coordinates on the shadow map are then computed from the point on the paraboloid plane (shadow map) where the ray intersects the paraboloid. As seen in the Figure 2.18, the key observation is that all incident rays are reflected in the same direction. A point P on the paraboloid is given as:

$$P = (x, y, f(x, y)) \tag{4.2}$$

which also equals to the texturing coordinates to address the paraboloid map. The scheme of the method can be seen in the Figure 4.3. The normal vector N_P of point P can be obtained as a cross product of partial derivatives (tangents) of the Equation (4.1) with respect to x and y, as seen in the Equation (4.3).

$$\begin{aligned} T_x &= \frac{\delta P}{\delta x} = (1, 0, \frac{\delta f(x, y)}{\delta x}) = (1, 0, -x) \\ T_y &= \frac{\delta P}{\delta y} = (0, 1, \frac{\delta f(x, y)}{\delta y}) = (0, 1, -y) \\ N_P &= T_x \times T_y = (x, y, 1) \end{aligned} \tag{4.3}$$

As seen from the Equation 4.3, the reflection vector is always going to be $(0, 0, \pm 1)$ in paraboloid local coordinate system.

The direction vector of the incident projection ray corresponds to the normalized vertex position in the paraboloid coordinate system. When we add the reflection vector and the direction vector, we get a vector that corresponds to the normal at the point of projection but with different length, N_{sum}, as seen in the Equation (4.4) [48].

[1]http://gamedevelop.eu/en/tutorials/dual-paraboloid-shadow-mapping.htm

$$N_P = (x, y, 1) \iff I_N + R_N = N_{sum} = (x_{sum}, y_{sum}, z_{sum})$$
$$N_P = (x, y, 1) \iff \frac{N_{sum}}{z_{sum}} = (\frac{x_{sum}}{z_{sum}}, \frac{y_{sum}}{z_{sum}}, 1) \tag{4.4}$$

4.3 Implementation

We implemented the testing framework in DirectX 11 and HLSL shading language. Our implementation does not use any hardware-specific features.

4.3.1 Omnidirectional Shadow Mapping

The omnidirectional part was implemented in a standard shadow map fashion. Camera was placed to the position of the light source and rendered the selected shadow frusta. The efficient frustum culling (EFC) is performed on the CPU to determine, which shadow map frusta are needed to be rendered. We used geometry shader to render up to all 6 cube shadow map faces in a single pass, using a bitmask in the geometry shader to indicate which frusta are active. To speed up the rendering process, we also implemented view frustum culling for the objects being redenred to the shadow maps. We used bounding spheres to cull the objects.

4.3.2 Dual-Paraboloid Shadow Mapping

There are several differences from standard shadow mapping when rendering using parabolic projection. Unlike cube mapping, paraboloid mapping requires only up to two passes to cover the whole scene. Appropriate view matrix is constructed based on the orientation of the paraboloid. Because we are doing our own projection, the projection matrix is either unit matrix or can be completely skipped. The vertex shader only poasses the vertices through, we exploited the geometry shader to render both paraboloids in one pass. We used similar culling technique to cull paraboloids – we checked for intersection with the plane separating the paraboloids, if the camera frustum intersects the plane, both paraboloids are rendered.

```
vec4 vertexEyeSpace = ModelViewMatrix * vec4(in_Vertex,1.0);
float Length = length(vertexEyeSpace);
float clipDepth = vertexEyeSpace.z;
vertexEyeSpace.xyz = normalize( vertexEyeSpace.xyz );
vertexEyeSpace.xyz += vec3(0, 0, 1); //adding reflection vector Rn
//division by z_sum, obtaining (x,y) position
vertexEyeSpace.xy /= vertexEyeSpace.z;
vertexEyeSpace.z = (Length - near)/(far - near);
vertexEyeSpace.w = 1.f;
```

Listing 4.1: Vertex Parabolic Projection GLSL code

The vertex transformation core can be seen in the Listing 4.1. The first line transforms the vertex to paraboloid's view space. The vertex position in view space is then normalized to acquire I_N and vector R_N $(0, 0, 1)$ is added which produces the N_{sum} normal. Upon

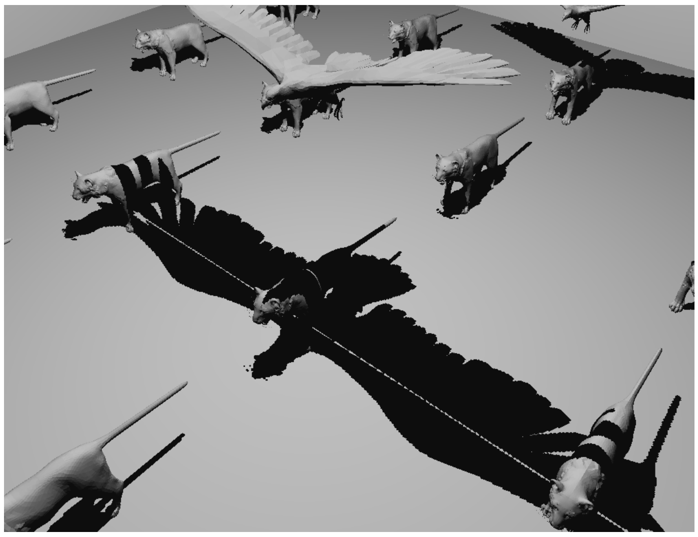


Figure 4.4: An example of an artifact on the boundary of the two paraboloids

dividing the normal with its z coordinate we obtain the position P as noted in the Figure 4.3. The projected vertex then needs z and w coordinates, which are supplied on the lines 7 and 8. The computed depth is then stored to the shadow texture. Custom depth clipping is implemented against the clipping hemisphere in the pixel shader using the *clipDepth* value, discarding only those fragments that are closer than the near clipping hemisphere.

To obtain the depth value from the shadow map, similar steps are required as during the shadow map creation. We first find out if the shaded fragment is in the possitive or negative paraboloid by testing its light's space depth. Then, (x, y) coordinates for accessing the shadow texture are computed, similarly as in the Listing 4.1 and mapped to $< 0, 1 >$. The acquired depth is then compared to the depth of currently processed fragment, similarly as standard shadow mapping algorithm.

The memory footprint of dual paraboloid method is clearly lower than cube shadow mapping, but there are also several disadvantages to this approach, most notably artifacts caused by interpolation during the build step. Large polygons cause visual artifacts due to incorrect interpolation, thus the rendered scene needs to be finely tessellated. Frequent problem is also the seam between the two paraboloids, see Figure 4.4.

4.4 Test Results

The methods were tested on a Core i5 661 processor running at 3.33 GHz and nVidia GeForce GTX 560 Ti graphics card. The rendered images had resolution of 1024×768. The tests we carried out in several variants of both methods – unoptimized (rendering all the geometry in all the frusta), optimized with view frustum culling of rendered objects ("Optim") and finally with efficient cube frustum / paraboloid culling ("EFC" / "PC"). The methods were compared in terms of performance and shadow quality.

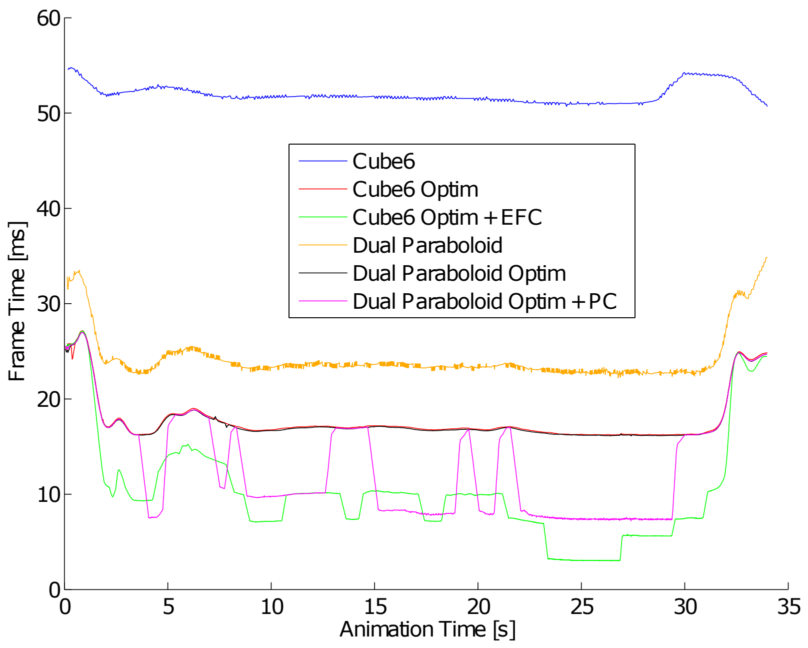


Figure 4.5: Results of a timed flythrough using several levels of optimizations among DPSM and Cube Shadow Mapping, time of the whole frame

4.4.1 Performance Evaluation

The methods were tested on a flythrough with a scene of approximately 3 million vertices and render target and shadow map resolution 1024 × 1024. The results of the flythrough can be seen in the Figure 4.5 and 4.6.

It is clear that the methods greatly benefited from all optimizations. The most optimized cube mapping approach takes up only 20 % of rendering time compared to the naive form. The unoptimized version of cube mapping performed the worst of all methods as it had to render all the scene geometry 6-times. The DPSM was able to cull one paraboloid at max, thus often rendering all the scene geoemtry visible from each paraboloid, it seems that the granularity of cuba mapping poses a performance advantage at times, whereas DPSM provides lower memory consumption. Cube mapping was able to save up to 83 % of its performance which can be seen at around 25th second of the flythrough compared to DPSM, which can only save up to 50 %.

The efficiency of the EFC can be seen in the figure 4.7. As mentioned above, cube mapping was able to render only to a single frustum in around 25th second of the test where the best performance was observed and very rarely used all 6 frusta. DPSM rendered to a single paraboloid most of the time during the test.

Tables 4.1 and 4.2 show the average frame rate among all the tested methods with respect to increasing shadow map resolution on two differently-sized test scenes. As shadow map used was a 32-bit float texture, its size ranged from 24 MB (1024 × 1024) to 384 MB (4096 × 4096) in total for cube mapping, one third of the size for DPSM. If we take the cube shadow map performance at 1024 × 1024 as 100 %, increasing resolution to 2048 × 2048 drops the performance only by 15.56 % in average, increasing to 4096 × 4096 causes drop in performance by 45.4 %. DPSM dropped 11.28 % when switching to 2048 × 2048 and 40.03 % in average when the shadow map resolution was set to 4096 × 4096. The higher sensitivity

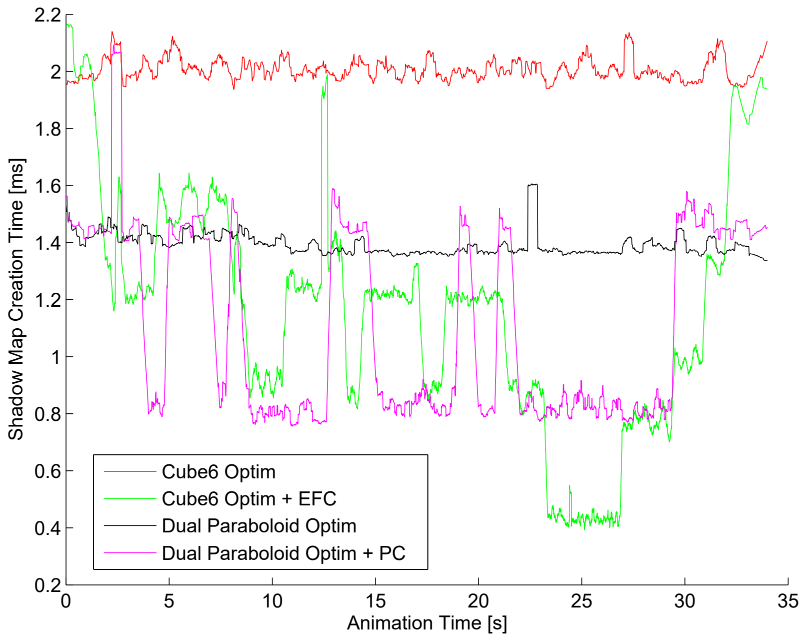


Figure 4.6: Comparing optimizations of DPSM and Cube Shadow Mapping variants, time of shadow map creation only

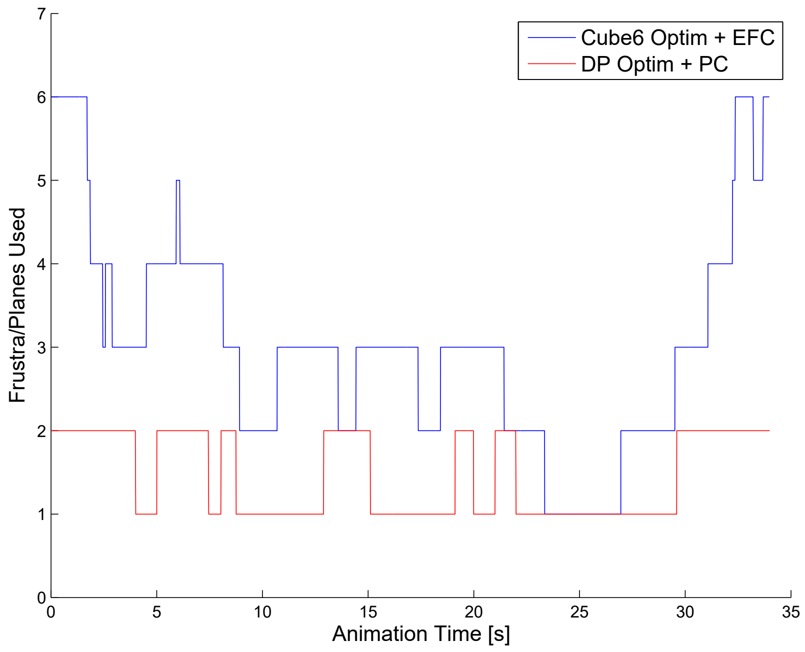


Figure 4.7: The number of active frusta / paraboloids during the flyghthrough

Method	1024	2048	4096
Cube6	75.71	70.04	47.9
Cube6 Optim	150.43	116.76	64.04
Cube6 Optim + EFC	188.71	151.67	89.68
DP	167.95	146.62	97.52
DP Optim	207.24	178.67	109.4
DP Optim + PC	208.15	180.24	110.95

Table 4.1: Averge FPS on a flythrough on a scene with approximately 600 000 vertices

Method	1024	2048	4096
Cube6	19.11	18.38	16.21
Cube6 Optim	57.15	51.23	36.50
Cube6 Optim + EFC	127.47	114.21	83.38
DP	41.50	39.74	33.17
DP Optim	57.47	54.32	42.85
DP Optim + PC	90.56	86.08	69.58

Table 4.2: Averge FPS on a flythrough on a scene with approximately 3 million vertices

to increased resolution among cube mapping is probably due to larger amount of shadow textures that method requires. Although DPSM uses less video memory, the method is not as fast as cube mapping as optimizations provide greater performance benefits in terms of culling.

4.4.2 Quality Evaluation

As discussed in the Chapter 4.2, DPSM suffers from artifacts when the occluding geometry is insufficiently tessellated as well as artifacts on the line where the two paraboloids are separated, as seen in the Figure 4.8. It is apparent that cube mapping produces higher quality shadows than DPSM in every resolution. At 512×512, both methods miss the details of the tiger's whiskers, but DPSM does not produce correct shadow even at 4096×4096, although consumes only a third of the memory compared to the cube mapping.

At 512×512, both methods miss the details of the tiger's whiskers, but DPSM does not produce correct shadow even at 4096×4096, although consumes only a third of the memory compared to cube mapping.

4.5 Evaluation

The initial assumption was that multiple render passes performed by the cube shadow maps technique should be very time consuming process. The result of the measurement is that an unoptimized version of the cube shadow maps exhibits the worst performance of the examined algorithms. When a simple optimization technique was used, it significantly increased the performance, in fact, the best of the examined algorithms. The performance and the visual quality of the cube shadow maps is superior compared to the dual-paraboloid algorithm. The parabolic projection introduced notable artifacts on the boundary of the two paraboloids and in places where the tessellation of the scene is not sufficient. This it due to the linear interpolation performed by the rasterization unit in the graphics hardware

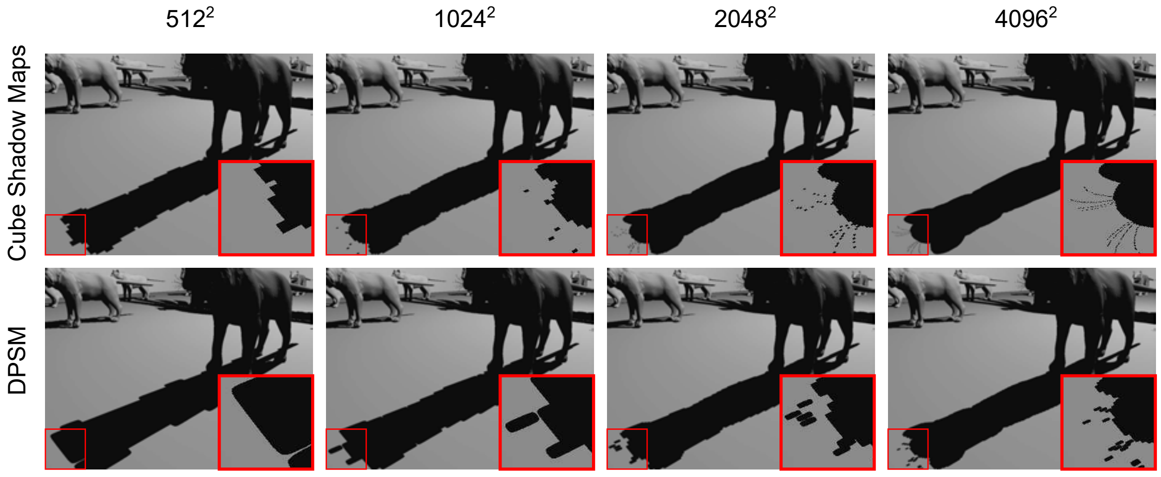


Figure 4.8: Quality comparison between cube shadow mapping and DPSM in various resolutions

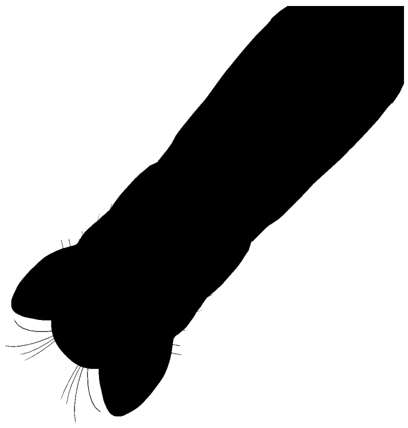


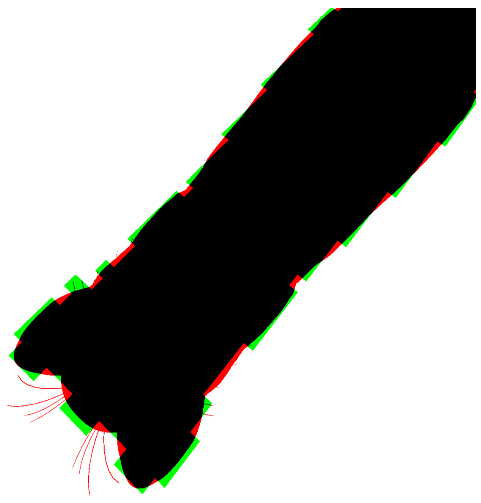
Figure 4.9: Reference image, produced by the stencil shadow volumes.

as the parabolic projection is not linear. Where the memory consumption is concerned and shadow quality is not of such importance, the dual-paraboloid approach is the better solution.

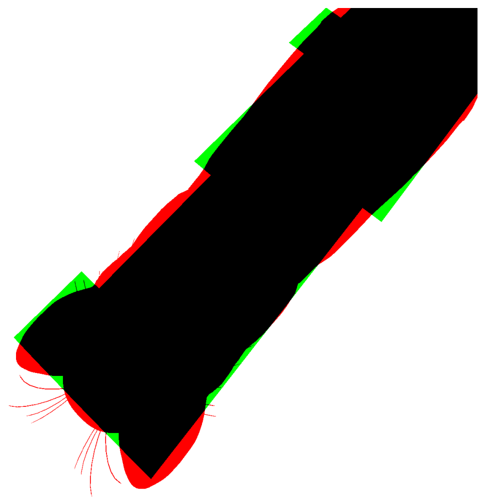
4.6 Extended Quality Comparison

I compared the same methods versus shadow volumes, which served as a pixel-perfect reference. I used the same tiger model as in the original comparison and tested under the same resolutions. The reference shadow rendering using stencil shadow volumes can be seen in the Figure 4.9. The Figures 4.10 and 4.11 contain the differences versus the reference image. The red areas are missing the shadow where is should be, the green areas should not be shadowed compared to the reference image. The shadow that each method produced is combination of black and green areas.

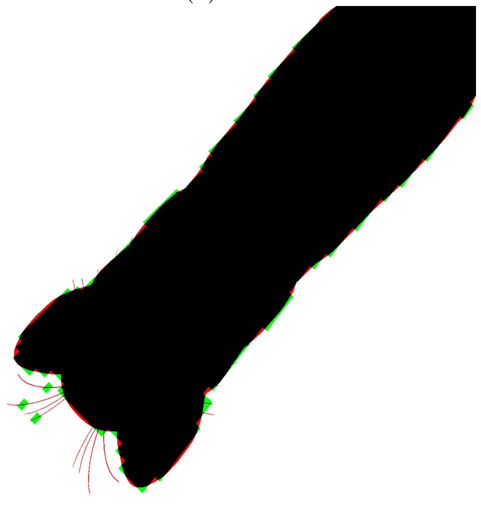
In order to measure qualitative differences between the methods, I used similar metric as in [64] to measure dark/lit area coverage, comparing each tested method and resolution to the shadow volumes. Both the reference and the investigated method image were blurred,



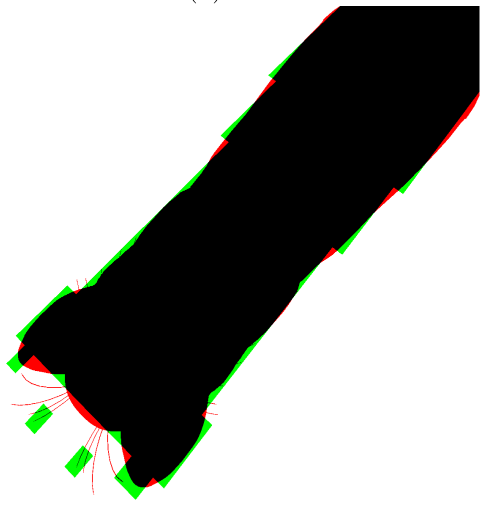
(a)



(b)

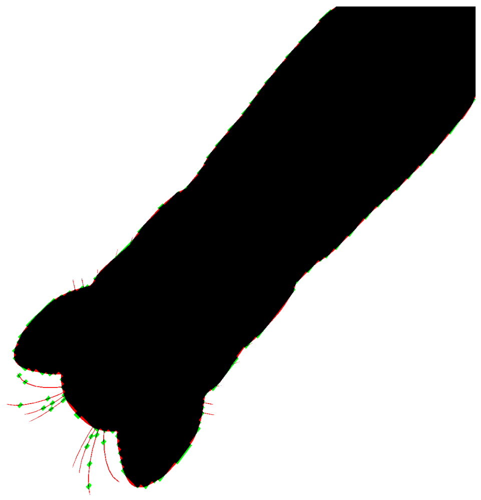


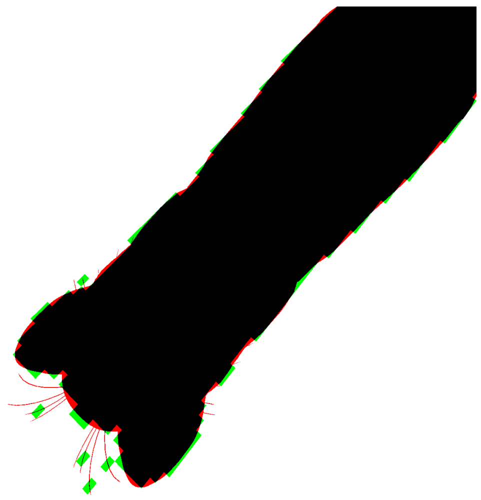
(c)

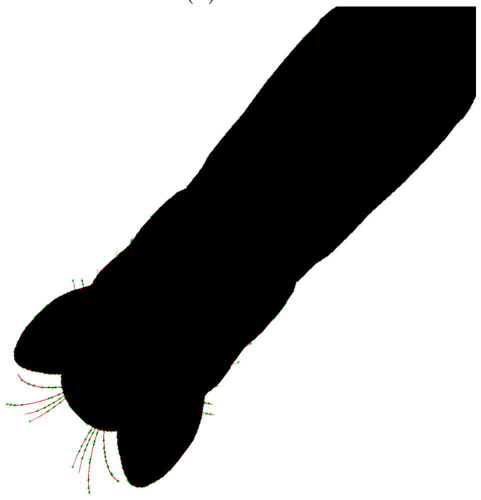


(d)

Figure 4.10: Comparison of cube shadow mapping (left) and dual-paraboloid shadow mapping (right) against shadow volumes, resolutions of 512 (top row) 1024 (bottom row), per texture side. The red areas are missing in the evaluated method's shadow, compared to the shadow volumes; green area indicate a shadowed region that was lit in the precise method.


(a)


(b)


(c)

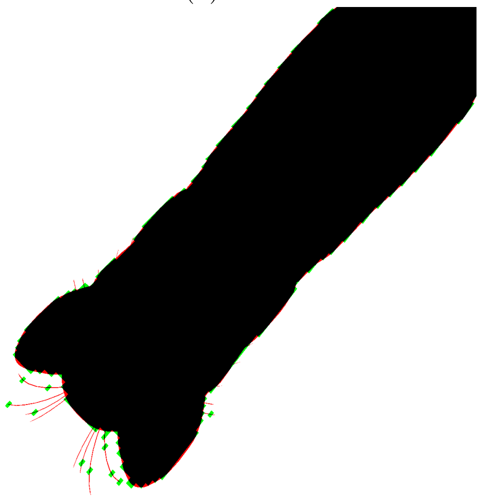

(d)

Figure 4.11: Comparison of cube shadow mapping (left) and dual-paraboloid shadow mapping (right) against shadow volumes, resolutions 2048 (top row) and 4096 (bottom row). The description is identical with Figure 4.10.

		lit coverage	dark coverage
512	Cube	0.785	0.666
	DP	0.807	0.153
1024	Cube	0.909	0.846
	DP	0.705	0.749
2048	Cube	0.951	0.914
	DP	0.860	0.769
4096	Cube	0.968	0.968
	DP	0.932	0.907

Table 4.3: Comparison based on lit/dark area coverage based on [64]. The higher the score, the closer to the precise method (shadow volumes), which would achieve the score of 1.

then the coverage of the shadowed and lit areas was computed for both images. Finally, each pixel in the particular region (lit, shadowed) in the image of the precise method was tested against the corresponding pixel from the evaluated method, whether it is also lit or shadowed. The unmarked pixels were weighted by a number of 15 to increase the score diversity as the shadow contains a lot of solid area. The resulting score is computed according to the Equation 4.5, where S is the coverage score, W weight, N_U the amount of unmarked pixels and N_R the size of the region. The results can be seen in the Table 4.3.

$$S = 1 - \frac{W \cdot N_U}{N_R} \tag{4.5}$$

Cube Shadow Mapping is better in the dark coverage than Dual-Paraboloid method in every test case. The situation is similar in lit coverage, despite the situation at 512×512 where DP has better lit area coverage but the shadow coverage was mediocre, see Figure 4.10b. Cube mapping scores -3 to 22 % more than Dual-Paraboloid on lit coverage, 6.3-77 % better on shadowed, averaging to 8.23 % and 27.6 % respectively. Both methods are very close at 4096×4096, the difference is 3.7 % in lit area score and 6.3 % in shadowed area score.

Chapter 5

Robust Silhouette Extraction Improvements for Shadow Volumes

CAD/CAM environment requires a precise and robust omni-directional shadow algorithm for displaying arbitrary triangle soups as models produced by the design software can contain triangles with incorrect winding, holes, degenerate triangles or edges with more than two triangles adjacent to them. To address these needs, we chose shadow volumes as they are more robust and easier to implement than other methods available at that time. However, shadow volumes, although producing precise shadows from omni-directional light sources, suffer from numerical instabilities, which we addressed in the following paper.

These findings were published in a paper presented on GraphiCon'2013 conference [90]. I was responsible for CPU implementation using AVX instruction set and acceleration using OpenMP to distribute the workload among more threads.

5.1 Robustness Problems of Shadow Volumes

Kim [62] designed a method to handle non-manifold shadow casters, which was used as the basis for our approach and was described in detail in the Chapter 3.1.5. He introduced the concept of edge *multiplicity* – a light plane is created from the edge and the light source. All the opposing vertices are then tested against the edge and the edge multiplicity is computed as the difference between the amount of opposing vertices above and below the plane, as seen in the Figure 3.9. The steps of the algorithm are outlined in the Algorithm 4.

The problem of not only this method but all other stencil shadow volume algorithms is when OV is very close to the LP. This situation is depicted in the Figure 5.1. The proposed algorithm resolves the above issue connected with the inconsistency of triangle edges multiplicity evaluation. The main idea of the improvement is that the triangles, where the inconsistency can occur, are removed from the silhouette calculation. Because these triangles are (almost) parallel with the LP (their shadow volume would be zero), they cannot affect the shape of the resulting shadow. In fact, the removal of the triangles is equivalent to evaluation of its edges multiplicity to 0 which would occur in triangles parallel to the LP if the precision was not limited.

One way to solve it in the "per-edge" scenario would be to process also the other two edges of a triangle when dealing with an edge and a single OV. This evaluation also has to be consistent, meaning that the vertex order must be preserved for each edge. We do this by implementing a comparison operator for the vertices and propagating a flag for the

Algorithm 4 Robust Shadow Volumes

```
1: Convert triangle to model to edge representation. Each edge is represented only once,
   with the list of all its opposing vertices (OV).
2: Every frame, an oriented light plane (LP) is cast from the edge vertices and the light
   source.
3: Edge multiplicity is initialized to 0
4: Each OV of the edge is tested against the LP. If it lies above the plane, increments
   the multiplicity of the edge, if below – decrements. Then the OV lies on the plane,
   multiplicity value does not change.
5: The set of edges with non-zero multiplicity form the model's silhouette.
6: Each silhouette edge is rendered extruded as many times as multiplicity with winding
   based on the multiplicity sign.
7: if z-fail then
8:     Render caps from all scene geometry
9: end if
```

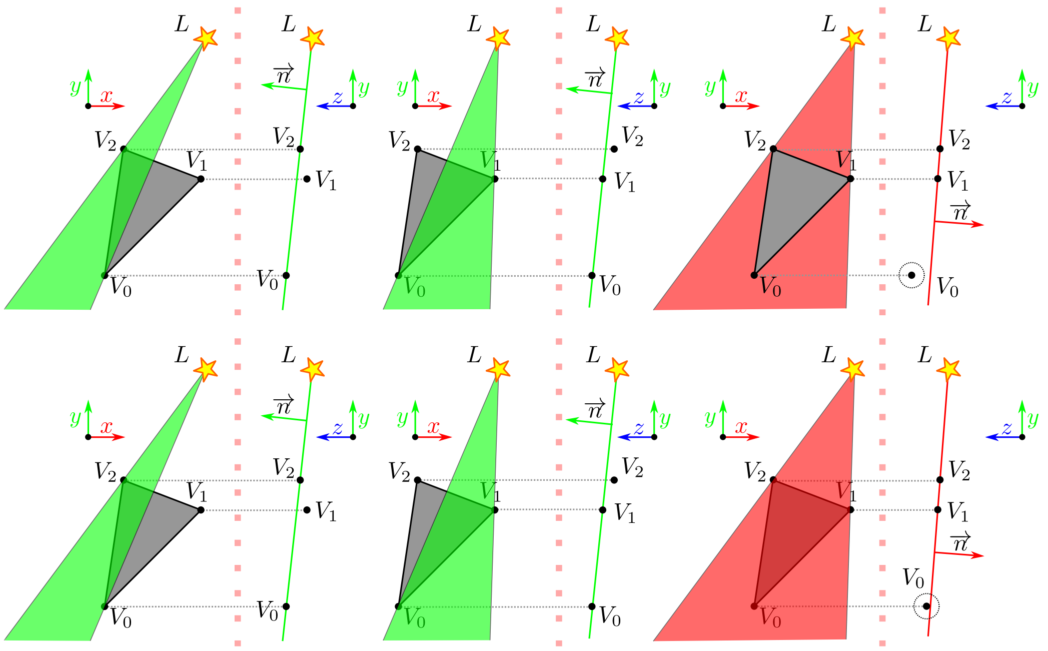


Figure 5.1: An example of correct (top row) and incorrect (bottom row) multiplicity computation of a grey triangle. Green and red triangle represent a light plane cast from the light source towards the edge, each color having different orientation. Each situation is supplemented with a side view with the light plane as a reference. In the top right situation, vertex V_0 is an OV for edge V_1, V_2 and is very close to the light plane. The circle around the projection of V_0 shows the numerical error margin. We see that in the top case, V_0 is correctly identified as being behind the light plane. However, when the light slightly moves, the numerical precision would cause the vertex V_0 to be evaluated incorrectly as being in front of the light plane.

edge, if it is in-order of the triangle's winding. Thus, we modified step 4 in the Algorithm 4 as follows:

4. Create a triangle from the OV and the edge. Compute multiplicity of all 3 edges as $+1$ or -1 by testing the remaining vertex as OV of the respective edge. If the vertex lies exactly on the plane, its multiplicity is 0. If the multiplicity is not consistent, set the multiplicity of the current OV to 0. The final edge multiplicity is the sum of all OV multiplicities.

Algorithm 5 Robust Multiplicity Computation

```
Function: CalcEdgeSingleOvMultiplicity
Input: Edge ($V_1$, $V_2$), OV
Output: Multiplicity $m$
 1: $isSwaped \leftarrow$ false
 2: if $V_1 > V_2$ then
 3:     Swap($V_1$, $V_2$)
 4:     $isSwaped \leftarrow$ true
 5: end if
 6: $V \leftarrow L - V_1 L_w$
 7: $N \leftarrow normalize((V_1 - V_2) \times V)$
 8: $LP \leftarrow (N_x, N_y, N_z, -N \cdot V_1)$
 9: $m \leftarrow sign(LP \cdot OV)$
10: if $isSwaped =$ true then
11:     $m \leftarrow -1 \cdot m$
12: end if
13: return $m$
```

The actual multiplicity calculation of an edge and its single OV is outlined in the Algorithm 5. This calculation is evaluated for all 3 edges of a triangle, consisting of an edge and an one OV, to determine multiplicity consistency. The absolute value of the resulting multiplicity of an edge, which is the sum of all OV multiplicities, is the number of times the extruded edge needs to be drawn.

5.2 GPU Implementations

The method was implemented on both CPU and GPU. GPU implementation was carried out using geometry shaders and via OpenCL kernel using OpenGL – OpenCL interoperability as compute shaders were not available at that time. Both OpenCL and geometry shader versions process edges in parallel, iterate over all its adjacent triangles. However instead of testing the opposite vertex against the light plane, the methods computes multiplicity by constructing a triangle plane for edge vertices and the opposite vertex and test the light source against it. To discard triangles having inconsistent multiplicity computation, the triangle plane is constructed 3-times differently, each triangle vertex being as pivot. When the triangle facing is inconsistent, the multiplicity of the edge is set to zero. When the multiplicity is non-zero, both methods cast the processed edge multiplicity-times. The OpenCL implementation writes all the generated geometry to a vertex buffer object, which is subsequently drawn using just a vertex shader into the stencil buffer. The caps geometry is generated in the OpenCL kernel as well, geometry shader version uses a separate shader to render front and back caps of the shadow volumes.

5.3 Implementation Using AVX Intrinsics and OpenMP

The implementation was based on SSE implementation proposed by Van Waveren [109]. The AVX intrinsics were used to load the edge data and opposite vertex, compute the light planes and compute the multiplicity of the edge.

AVX instruction set provides 256-bit long registers, which allowed us to pack two 4-component vectors into a single registers, thus processing two edges at once, unlike SSE with 128-bit long registers (single vector). Registers are represented by a special data type `__m256`. We processed 4 edges at once by a single thread, but the results needed to be updated to their respective triangles, creating a critical section as the whole process was parallelized by OpenMP. The outline of the computation can be found in the Algorithm 6.

Algorithm 6 AVX OpenMP Multiplicity Computation

```
1:  $PrepareMaskRegisters()$
2:  $nofIters \leftarrow nofEdges/4$
3:  for $nofIters$ in parallel do
4:      $edges \leftarrow Load4EdgesAVX()$
5:      $planes \leftarrow ConstructLightPlanesAVX()$
6:      $OVs \leftarrow LoadNofOVsAVX()$
7:      for $max(OVs)$ do
8:          $mults \leftarrow ComputeMultsAvx(edges, planes, OVs)$
9:          $UpdateTriangleStructure(edges, mults)$
10:     end for
11: end for
12: $ProcessLeftoverEdges()$
```

The *For* loop on the line 3 ran in parallel using OpenMP, utilizing all cores in the system. It took more instructions to properly load the data as they were not optimally stored in the testing framework, using masked load operations in the step 4. The computation itself (lines 5, 6, 8) processed data of two edges at once per register. Although the data is processed per-edge, the results are written to a data structure holding triangle information. As a triangle is formed by 3 edges, this data has to be updated in a serialized way, we used atomic operations to update the triangle data. This data is then used to generate the shadow volume sides.

In order to determine inconsistency in multiplicity calculation, the method marks each triangle as light front or back facing during the multiplicity evaluation of an adjacent edge, using the information about the in-order flag of the edge and the sign of the multiplicity test of the particular opposite vertex. In the final step of the method, a triangle that has been marked to be both front and back facing, with respect to the light, is discarded from further processing.

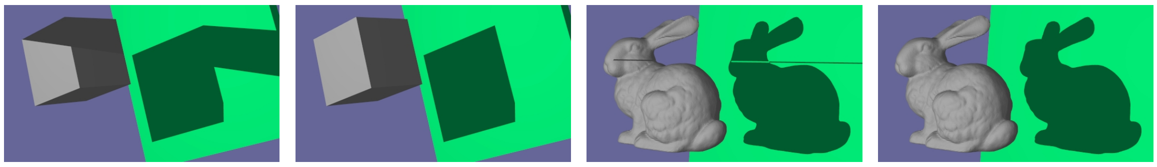


Figure 5.2: This image shows the difference between the original algorithm and our robust algorithm. The right image of each couple shows the result of our robust algorithm. The first couple of images shows a very simple model, where artefacts are most visible - the light source is exactly above the edge of the cube. The second couple shows artefacts on more complex model, which could appear in real applications.

	GF650Ti	AMD7950	HD3000	HD4000
CPU	4.9 / 5.0	12.9 / 13.9	9.9 /11.2	6.3 / 5.9
AVX+OMP	4.9 / 5.1	11.8 / 13.5	10.0 / 12.4	6.6 / 8.6
GS	30 / 40	92 / 98	n/a	4.4 / 8.7
OpenCL	42 / 41	83 /80	n/a	7.6 / 7.7

Table 5.1: Performance comparison of new robust method (left) versus the original silhouette method (right). Results in average frames per second (FPS).

5.4 Measurements

We tested the approach on two CPU platforms supporting AVX instruction set – Intel Core i7-2600K of the "Sandy Bridge" generation (first to introduce the AVX instructions) and Core i5-3570K of the "Ivy Bridge" generation. Table 5.1 shows the slowdown induced by the extra computation and memory access required by the robust method. The slow down, on average, was less than 10 %. The first 3 platforms were evaluated on the i7-2600K CPU, the last platform on i5-3570K. Apart from CPUs, the methods were tested on AMD Radeon HD 7950 and nVidia GeForce GTX 650 Ti.

Table 5.2 shows the test results when comparing simple and more complexly shaped models. All scenes were similar in the amount of triangles, around 65 000. Each scene also consisted of a variant where all objects were baked into a single one. The bold values indicate the best result for a particular test scene on a certain hardware platform.

The second test was focused on the behaviour of the methods with increasing amount of geometry, from 10^5 to approximately 10^6 triangles. The results can be seen in the Table 5.3, top row of each scenario. It can be seen that the amount of geometry has significant impact on the performance, as the amount of geometry increased by an order of a magnitude, so did increase the frame time.

The bottom row of the results in each cell in the Table 5.3 uses about the same amount of geometry, but instead of a single detailed sphere, a grid of 10 x 10 spheres which consists of approximately the same amount of triangles as a respective scene with a single sphere. The test shows that the number of isolated objects does not have impact on the results, probably due to synchronization or data transfers.

The proposed solution also managed to alleviate the problem with triangles almost parallel with the light, as can be seen in the Figure 5.2. The robust test is able to remove such triangles from computation and during our evaluation we experienced no visual artifacts on none of the test scenes and platforms.

	GF650Ti		AMD7950		HD3000		HD4000	
CPU	4.9	5.0	12.9	7.8	9.9	**11.0**	6.3	5.4
	5.5	5.5	13.3	9.6	12.2	**10**	9.6	5.5
AVX+OMP	4.9	5.1	12.9	4.3	9.9	6.2	6.3	6.0
	5.5	5.5	12.8	5.2	**12.8**	5.1	**11.9**	**6.2**
GS	30	30	**92**	95	n/a	n/a	4.4	n/a
	34	34	**156**	124	n/a	n/a	6.0	n/a
OpenCL	**42**	**52**	83	**96**	n/a	n/a	**8.1**	**8.2**
	54	**67**	142	**147**	n/a	n/a	11.3	6.0

Table 5.2: Several implementations compared on multiple test scenes. Each scenario consists of 4 results for 4 different scenes – 10 individual bunnies (top left), 10 baked bunnies (top right), 10 individual spheres (bottom left), and 10 baked spheres (bottom right). The results are average FPS.

	GF650Ti		AMD7950		HD3000		HD4000	
CPU	3.9	35	5.1	81	**5.3**	**34**	**37**	20
	3.4	27	8.4	61	6.5	32	4.4	18.5
AVX+OMP	3.9	35	5.7	81	4.9	32	4.2	**23**
	3.4	28	9.3	63	**6.7**	**34**	**5.0**	**24**
GS	23	165	85	**650**	n/a	n/a	3.7	15
	18.7	**100**	**84**	**482**	n/a	n/a	3.1	19.3
OpenCL	**43**	**192**	**91**	430	n/a	n/a	**6.0**	19.3
	13.4	26	22	54	n/a	n/a	4.4	9.1

Table 5.3: Test cases: one sphere with 10^6 triangles (top left), one sphere with 10^5 triangles (top right), 10x10 spheres each with 10^4 (bottom left), and 10x10 spheres with 10^3 triangles (bottom right). The results are average FPS.

5.5 Evaluation

The proposed approach proved to be working well and producing quality shadows with no visual artifacts. At the same time, it exhibits high performance in variety of hardware platforms and can be efficiently implemented in CPU both using the traditional instructions and the SIMD instructions as well as in GPU using Geometry Shaders as well as using OpenCL.

The OpenCL implementation suffers from synchronization between OpenCL and OpenGL, which is a necessity when using interoperability between these interfaces. This occurred on the scenes with many objects as the synchronization had to be performed with each rendered object. This method favored scenes with up to ∼100 objects. Geometry shader performs second best and as it is not bound with any synchronization, it's the fastest method for scenes with large amount of objects.

It is clear that benefit from the AVX-OpenMP implementation on 2600K was minimal or in some cases (scenes "10 baked bunnies" and "10 baked spheres") even slower. Sandy Bridge CPU benefited more from the implementation, increasing performance up to 30 %; modern architecture was able to benefit more from the algorithm. One of the reasons the speedup is not higher could be the critical section for triangle data update which results in partial serialization. Also, the input data structure format did not favor AVX execution but other implemented methods benefited from its structure; a more suitable data structure could have been designed.

Chapter 6

Shadow Volumes Using Tessellation Shaders

With the advancing development of the graphics pipeline and the introduction of the tessellation pipeline, new ways of geometry processing became possible. We saw opportunity in this new rendering pipeline stages and first produced a prototype of a two-pass per-triangle shadow volume method utilizing the tessellation pipeline. Second iteration of the algorithm produced a single-pass method, which then became the basis of a silhouette-based approach. The following chapter will also discuss the improvements in robust silhouette computation. These findings were presented on WSCG conference [81]. I am the author of the initial per-triangle version, then cooperated on the silhouette version and have written most of the paper.

6.1 Per-Triangle Tessellation-Based Shadow Volumes

The tessellation pipeline consists of 3 stages – *control shader* whose task is to compute the tessellation factors to the second stage, the *tessellation unit*, which is a fixed-function part of the pipeline that performs the subdivision of the primitive based on the tessellation factors. The generated vertices are then processed in the *evaluation shader* [23].

The first prototype of the method was designed around the scheme in the Figure 6.1. We start with a triangle for which we set the inner tessellation factor to 3 and outer to

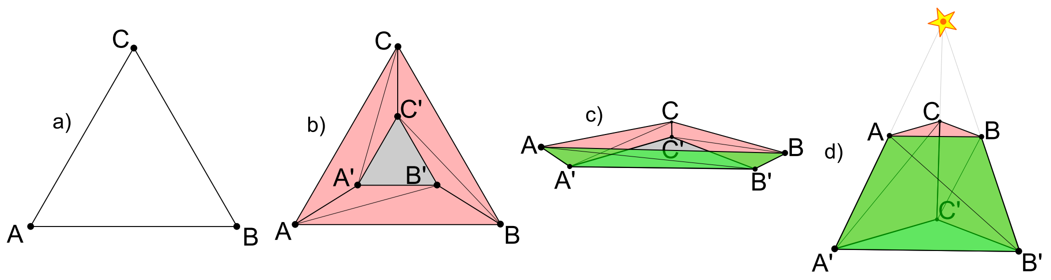


Figure 6.1: Creating semi-enclosed shadow volume from a triangle. Initial triangle in *a*) is tessellated using outer factors (1, 1, 1) and inner (3) *b*). Points A', B', C' are given positions of the points A, B, C *c*) and then pushed to infinity to form a volume with back cap *d*). Front cap is absent and must be rendered in the second pass.

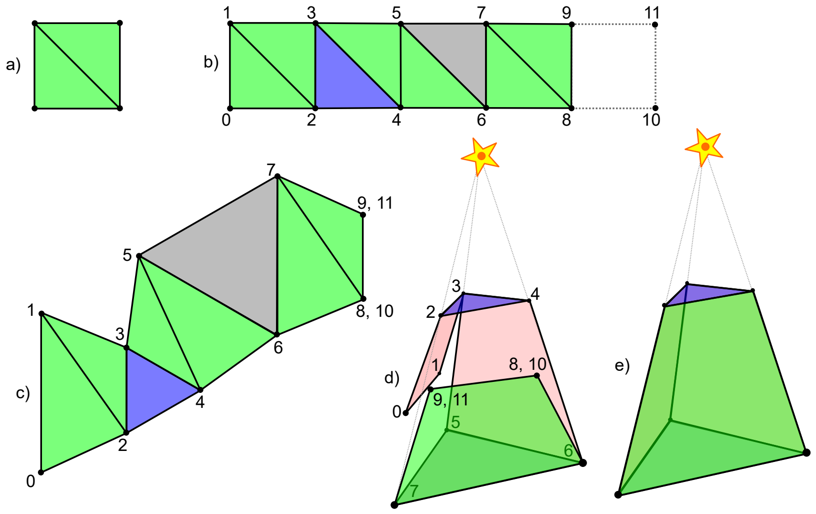


Figure 6.2: Single-pass per triangle method, a full shadow volume is created in a single pass. One point is added to the triangle in order to form a quad *a*) which is then tessellated using outer factors (1, 5, 1, 5) and inner (5, 1) *b*). Points 10 and 11 are merged with 8, 9. Light cap is visualized as blue, dark cap grey *c*). Then we join points 0 – 7, 1 – 5, 2 – 9, 4 – 8 and push points 5, 6, 7 to infinity *d*) to make the enclosed shadow volume *e*).

1, equal spacing and reversed triangle winding, resulting in the shape in the Figure 6.1*b*. The red color denotes the inner part of the volume and green outer. The evaluation shader moves the points A', B' and C' first to the positions of A, B and C and the projects them to the infinity, forming the dark cap. The reason of the reversed vertex ordering is for the cap to face outwards and not inside the shadow volume. Vertices A, B and C retain their positions. As the volume lacks the light cap, the scene geometry must be rendered again to seal the volumes from the top as light caps when z-fail version of shadow volumes is needed.

The second iteration of the per-triangle approach is able to produce enclosed shadow volumes in a single pass. Unlike the initial method, we start with a quad rather than a triangle. The schematic can be seen in the Figure 6.2. The input quad patch is tessellated using outer factors (1, 5, 1, 5) and inner (5, 1). The evaluation shader merges vertices 10-8 and 11-9 as they are superfluous. In order to connect the sides of the shadow volume together, we merge vertices 0 – 7, 1 – 5, 2 – 9 and 4 – 10 to enclose the volume. Vertices 5, 6 and 7 (and their merged counterparts 0 and 1) form the dark cap of the shadow volume. Vertices 2, 3, 4 and merged 8, 9, 10 for the light cap. As the per-triangle shadow volumes are slow, we investigated the triangle collapsing even more to design a silhouette-based approach.

6.2 Silhouette-Based Approach

The silhouette-based approach is based on the findings of collapsible geometry from the single-pass per-triangle method. In order to extend it to a silhouette approach, we need edge data instead of the standard geometry as the input for the tessellation stage. The control shader then computes the edge multiplicity and calculates the tessellation factors. Evaluation shader bends the tessellated shape to create a set of overlapping quads that form the shadow volume side geometry.

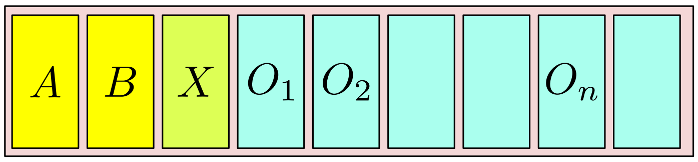


Figure 6.3: The input patch of the tessellation control shader. A and B are the edge vertices, X is the number of opposite vertices and $O_1 - O_n$ are the opposite vertices.

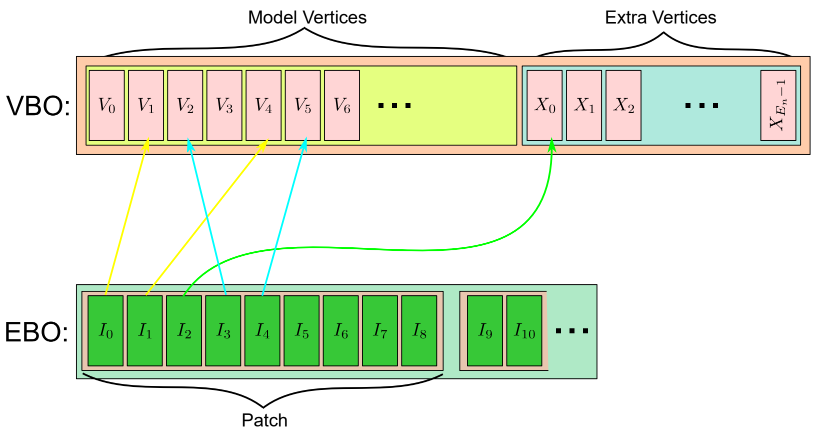


Figure 6.4: Combining patch data using Vertex Buffer Object (VBO) and Element Buffer Object (EBO). The VBO is extended by E_n positions, each storing the number of opposite vertices of the respective edge.

6.2.1 Input Patch

The input patch can be seen in the Figure 6.3. As vertices may have different amount of opposite vertices, some of the positions are left empty. In order to reduce the memory requirements (not to have the scene stored twice), we used slightly altered existing geometry data, seen in the Figure 6.4. The vertex buffer has to be extended only by E_n slots, which is the number of edges in the model. These positions store the number of adjacent triangles to the edge. Although the vertex data is a 4-component vector, we store only a single value, leaving 3 values unused. This data structure is then addressed by element buffer (EBO) to create the patch. The length of the patch is $maxMultiplicity + 3$.

6.2.2 Tessellation Control Shader

A single instance of the control shader computes the edge's multiplicity from the data in the input patch. One instance can read the data of the whole patch, thus able to read all the opposite vertices and compute the multiplicity. For multiplicity computation, we implemented a faster approach than described in the Chapter 5.1. Instead of testing the multiplicity coherence among all 3 edges of a triangle, we designed a new method that requires only a single edge to be tested and is described in the following section. The final multiplicity is then used to compute the inner and outer tessellation factors as seen in the Algorithm 7. We used a quad as the tessellated primitive, fractional odd spacing and clockwise triangle orientation.

Algorithm 7 Tessellation Factors Computation

```
1: $absMult \leftarrow abs(Multiplicity)$
2: $tessFactor \leftarrow 2 \cdot absMult - 1$
3: gl_TessLevelOuter$[0] \leftarrow (absMult > 0)$
4: gl_TessLevelOuter$[1] \leftarrow tessFactor$
5: gl_TessLevelOuter$[2] \leftarrow 1$
6: gl_TessLevelOuter$[3] \leftarrow tessFactor$
7: gl_TessLevelInner$[0] \leftarrow tessFactor$
8: gl_TessLevelInner$[1] \leftarrow 1$
```

Improved Multiplicity Computation

We simplified the robust multiplicity computation described in the Chapter 5.1. Our previous method always computed multiplicity across all triangle edges to determine if the triangle should be discarded or not. We now compute the multiplicity only once from each opposite vertex using a so-called *reference edge* of a triangle.

The choice of a reference edge needs to be the same for all occurrences of the triangle, which we also promoted to be winding-independent. For this, we implemented vertex ordering that would guarantee that the reference edge is always constructed from the 2 smallest vertices of the triangle, based on the metric of choice. The ordering can be seen in the Equation (6.1) and Algorithm 8. The edge vertices A, B on the input are already sorted in the pre-process step, simplifying the sorting process in the shader. The metric is calculated as a sign of a dot product of weights and sign coordinate difference between the two vertices. The whole process of multiplicity computation can be seen in the Algorithms 9 (multiplicity of a single OV) and 10 (multiplicity of all OVs). Generally speaking, we moved the computation from per-triangle, as described in the Chapter 5, to per-edge.

$$\begin{aligned} A < B &\iff GreaterVec(A,B) < 0 \\ A = B &\iff GreaterVec(A,B) = 0 \\ A > B &\iff GreaterVec(A,B) > 0 \end{aligned} \tag{6.1}$$

Algorithm 8 Vertex Comparison Function

```
Function: GreaterVec
Input: Vertices $V_1$, $V_2$
Output: -1 $(V_1 < V_2)$, 0 $(V_1 = V_2)$, 1 $(V_1 > V_2)$
1: $signs \leftarrow \text{sign}(V_1 - V_2)$
2: $weights \leftarrow (4, 2, 1)$
3: $r \leftarrow \text{dot}(signs, weights)$
4: return $\text{sign}(r)$
```

6.2.3 Tessellation Evaluation Shader

In order to draw the shadow volume quad multiplicity-times, the resulting shape coming from the tessellator needs to be bent to create overlapping quads, as seen in the Figure 6.5. Each invocation of the evaluation shader needs to assign proper coordinates of the generated

Algorithm 9 Function to compute the multiplicity of an edge and one OV

```
Function: ComputeMult
Input: Vertices $A, B, C$, $A < B$, light position $L$
Output: Multiplicity $m$
 1: $X \leftarrow C - A$
 2: $Y \leftarrow L - AL_w$
 3: $N \leftarrow X \times Y$
 4: $m \leftarrow \text{sign}(N \cdot (B - A))$
 5: return $m$
```

Algorithm 10 Function to compute the multiplicity of an edge and all of its OVs

```
Function: ComputeEdgeMult
Input: Edge vertices $A, B$, $A < B$, set of OVs $\Omega$, light position $L$
Output: Multiplicity $m$
 1: $m \leftarrow 0$
 2: for $O_i \in \Omega$ do
 3:   if $A > O_i$ then
 4:     $m \leftarrow m + \text{ComputeMult}(O_i, A, B, L)$
 5:   else
 6:     if $B > O_i$ then
 7:       $m \leftarrow m - \text{ComputeMult}(A, O_i, B, L)$
 8:     else
 9:       $m \leftarrow m + \text{ComputeMult}(A, B, O_i, L)$
10:     end if
11:   end if
12: end for
13: return $m$
```

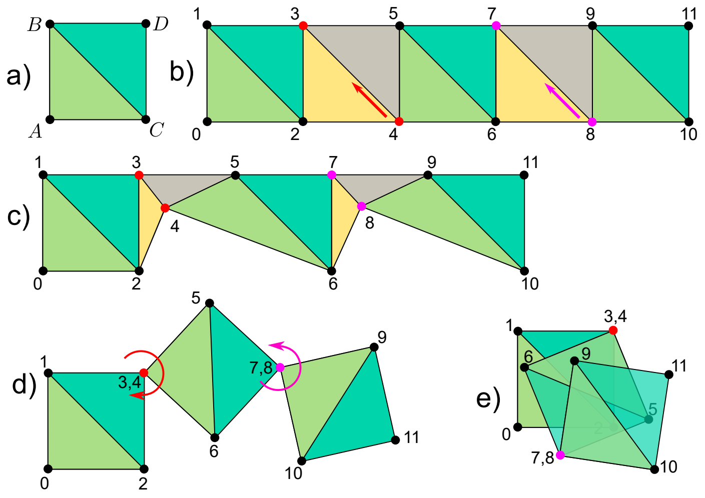


Figure 6.5: Transformation of a quad through the tessellation. *a*) an input quad gets tessellated to the shape in *b*) with multiplicity of 3. Only the green-toned triangles will be rendered, yellow and grey will be degenerated. Transition from *b*) through *c*) to *d*) shows the process of degeneration by merging vertices 3-4 (red) and 7-8 (purple). Transition from *d*) to *e*) shows rotation around the red and purple vertices to for the resulting overlapping series of quads.

Algorithm 11 Evaluation Shader

Input: Quad vertices A, B, C, D, tessellation coordinates $x, y \in [0, 1]$, multiplicity m
Output: Vertex V in world-space coordinates

```
1: $P \leftarrow array(A, B, C, D)$
2: $a \leftarrow round(x \cdot m)$
3: $b \leftarrow round(y)$
4: $id \leftarrow 2a + b$
5: $t \leftarrow (id \bmod 2) \oplus (\lfloor id/4 \rfloor \bmod 2)$
6: $l \leftarrow \lfloor (id + 2)/4 \rfloor \bmod 2$
7: $n \leftarrow t + 2l$
8: $V \leftarrow P(n)$
9: **return** $V$
```

vertex, based on its tessellation coordinates. The process is outlined in the Algorithm 11 where $\oplus$ denotes logical XOR operation. Vertices A, B, C, D are edge vertices and their extrusions to the infinity along the light direction, as in the Equation (6.2) where V_1 and V_2 are the edge vertices and L is the light source.

First, the ID of the vertex is calculated from multiplicity and the tessellation coordinates. The computed ID is then used to select one of the vertices that form the shadow volume quad.

$$\begin{aligned} A &= (V_1, 1)^T \\ B &= (V_2, 1)^T \\ C &= (V_1 - L, 0)^T \\ D &= (V_2 - L, 0)^T \end{aligned} \tag{6.2}$$

6.3 Implementation

The above mentioned methods were implemented in an open-source application using OpenGL. One of the first observations was that the single-pass per-triangle method suffers from inconsistent rasterization of two identical triangles at the same depth but with reversed winding – depth of the fragments from both triangles differs which resulted in z-fighting artifacts. We had to manually push the front cap's fragments into depth of 1.0f, so they would fail the depth test, otherwise we observed self-shadowing artifacts. By assigning the depth manually in the fragment shader the early depth test during rasterization did not occur, resulting in performance loss over two-pass method.

As we used z-fail method of shadow volumes, caps needed to be drawn as well. We drew the caps using the same multiplicity paradigm described above, to determine the triangle's orientation towards the light source. It was also necessary to maintain consistency among all the calculations, including the caps.

As the maximum tessellation factors are limited, this also poses a limit to maximum multiplicity the algorithm can handle. Drivers at the time allowed for a maximum of 64 as a tessellation factor, which according to the Algorithm 7 line 2, yields maximum multiplicity of 32. However, we have found a case where the multiplicity of 32 is insufficient – the popular "Power Plant" model has several edges that have multiplicity of 128. Without the

Sponza	R280				G680			
Resolution	TS		GS		TS		GS	
800x600	112	130	51	49	175	178	62	61
1024x768	116	124	49	50	177	178	61	60
1366x768	107	116	48	48	159	160	60	61
1920x1080	95	101	47	46	122	126	60	59

Table 6.1: Performance evaluation of both GS and TS robust methods on Sponza scene under several resolutions, average frames per second. The left column of each method contains values of the original 3-edge testing method, second column results of the new *reference edge* approach.

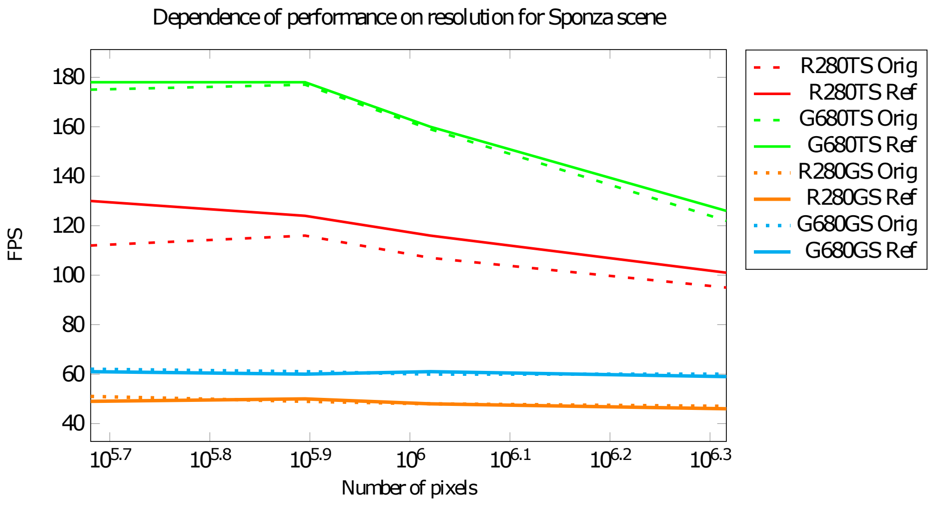


Figure 6.6: Dependence of resolution on performance (FPS), measured on Sponza scene.

loss of generality, such edge can be split into multiple instances, each having 32 opposite vertices.

6.4 Measurements

We performed extensive tests of both per-triangle and silhouette approach. Our new silhouette approach was also compared to Sintorn's Alias-Free Shadow Maps (AFSM) [100] and shadow mapping with 8K $\times$ 8K resolution. We also put our new method against a similar geometry shader implementation. The algorithms were tested on nVidia GeForce GTX 680 and AMD Radeon R9 280X and on two scene – flythroughs on Sponza scene that took 16 seconds and through a synthetic scene with a grid of 10 $\times$ 10 spheres with configurable amount of triangles, taking 40 seconds.

The first conducted test was on the Sponza scene, comparing tessellation shader (TS) and geometry shader (GS) implementations, as well as the old and new robust computation. The results can be seen in the Table 6.1. The dependency on resolution is outlined in the Figure 6.6. As can be seen, the tessellation implementation outperforms the geometry shader variant, although the latter has more stable FPS with the increasing resolution. The new deterministic method is faster in most cases, except for some cases in the GS implementation.

Spheres10x10	R280				G680			
Triangles	TS		GS		TS		GS	
32400	984	**995**	490	484	739	**825**	542	540
67600	921	**963**	488	487	624	**667**	494	513
102400	615	**729**	484	479	491	**555**	372	402
360000	203	233	270	**272**	218	**228**	131	135
1081600	72	88	104	**110**	82	**94**	46	49
1440000	56	72	84	**91**	67	**81**	36	39
1960000	34	41	59	**62**	49	**58**	26	28

Table 6.2: Comparison of two determinism methods implemented in GS and TS on scene with 10×10 spheres with increasing amount of triangles, in FPS. Values in bold represent performance of the fastest method on particular platform and amount of triangles. Left column represents old determinism method, right column the new reference edge. Results in FPS.

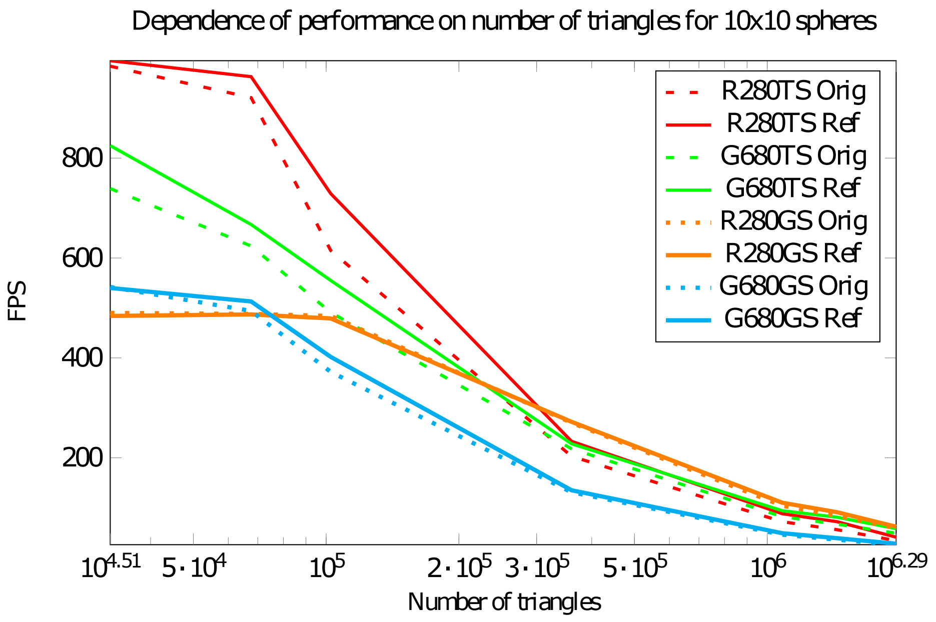


Figure 6.7: Dependence of scene complexity on performance (FPS), measured on a scene with 10×10 spheres with increasing amount of triangles.

Spheres 1M	R280				G680			
Objects	TS		GS		TS		GS	
1	73	92	111	**120**	106	**134**	55	60
4	74	94	113	**121**	101	**126**	53	58
25	64	76	97	**101**	76	**88**	46	50
64	68	76	**90**	89	61	**66**	40	43
100	64	70	**84**	82	58	**62**	39	42
240	58	55	**70**	64	**50**	49	35	36
399	53	48	**61**	54	**36**	36	27	28
625	43	38	**53**	46	**29**	27	22	22
851	40	44	46	**50**	24	**25**	19	20
1250	35	**37**	28	31	**19**	19	16	16
2500	**23**	19	15.1	15.4	**12**	11	10.8	10.1
3116	21.2	**21.5**	12.8	12.5	11.1	**11.2**	9.1	9.2
3920	**15.7**	14	10.1	10.12	**9.1**	8.7	7.7	7.5
5100	**14.8**	14.2	7.8	7.75	**8.2**	8.2	6.7	6.8
15600	**7.45**	6.45	3.07	3.14	**10.5**	9.1	3.6	3.6

Table 6.3: Test scene having approximately 1 million triangles but increasing amount of spheres. Results in FPS.

As the sphere scene provided with more flexibility with respect to the amount of objects and the total geometry, we concluded more tests on this scene. The first scenario was keeping constant amount of spheres (10×10) but increase their tessellation, from approximately 32 000 to almost 2 million triangles, results can be found in the Table 6.2 and the Figure 6.7. It can be observed that tessellation variant of the algorithm is faster only on scenes with up to 100 000 triangles on AMD platform. There is a notable drop in performance when increasing the amount of triangles to 360 000 on AMD platform as the tessellation version was initially twice as fast as GS. TS implementation is favored on nVidia platform, its reference edge implementation being the fastest in all of the test cases.

The following test kept constant amount of triangles in the scene to around 1 million (with maximum deviation of 2 %) but the number of spheres in the scene was increasing. No hardware instancing was used, every object was drawn using a separate draw call. Results can be found in the Table 6.3. This test was biased by the CPU overhead due to increasing amount of draw calls. The results are at times contrary to our previous measurements, where the reference edge approach is actually slower in the most scenarios. Also, tessellation approach takes over GS on AMD platform with the increased amount of draw calls. TS was the fastest method on nVidia platform as in previous tests, but the optimized reference edge approach being on-par or slightly slower at times, by around 1-3 FPS. This test demonstrates how different platforms deal with the increased amount of draw calls combined with our rendering algorithms.

We did not directly compare our approach with Sintorn's AFSM [100], but he claimed that his algorithm is 3-times slower than 8K shadow mapping (SM). We performed similar test on a scene with 10×10 spheres and increased the amount of tessellation of the spheres. The results of the test can be seen in the Table 6.4 and graph in the Figure 6.8. Shadow mapping performance drops only by 51 % when comparing the cases with the least and most triangles in the Table 6.4, which is substantially less than shadow volumes which lose 96 % of the performance, but we were able to outperform shadow mapping on scenes with up to 400K triangles. When evaluating results at almost 2M triangles, the R9 280X dropped to

Spheres10x10	R280		G680	
Triangles	TS	SM	TS	SM
32400	**995**	252	**825**	245
67600	**963**	250	**667**	237
102400	**729**	244	**555**	225
360000	**233**	219	**228**	190
1081600	88	**168**	94	**135**
1440000	72	**155**	81	**115**
1960000	41	**120**	58	**103**

Table 6.4: Comparison of 8k shadow mapping with reference edge TS approach on scene with 10×10 spheres, results in FPS.

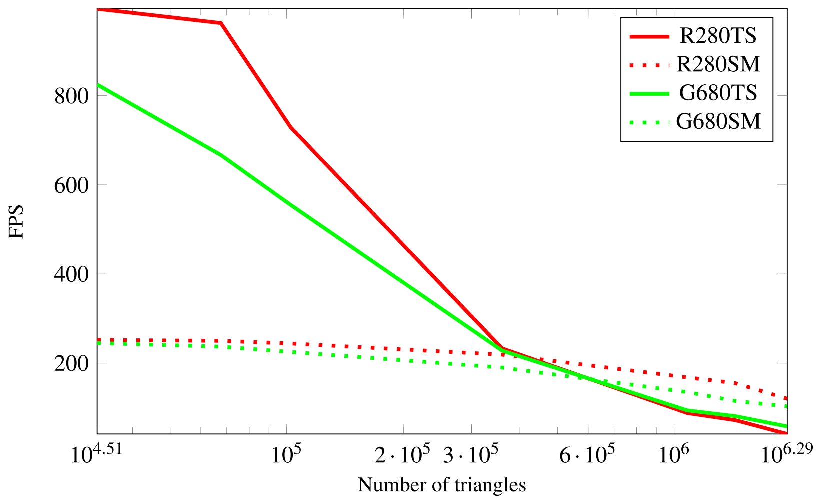


Figure 6.8: Graph showing performance dependence of TS and SM on the amount of triangles in a scene with 10×10 spheres.

Sponza	R280				G680			
Variant	TS		GS		TS		GS	
Spheres1x1	1.24	1.3	1.36	1.4	1.19	1.25	1.13	1.11
Spheres50x50	1.65	1.98	1.77	2.07	3.81	5.04	3.16	3.35

Table 6.5: Performance ratio between a camera overlooking the whole scene and when looking away. Left column of each implementation contains values for older robust method, right for reference edge.

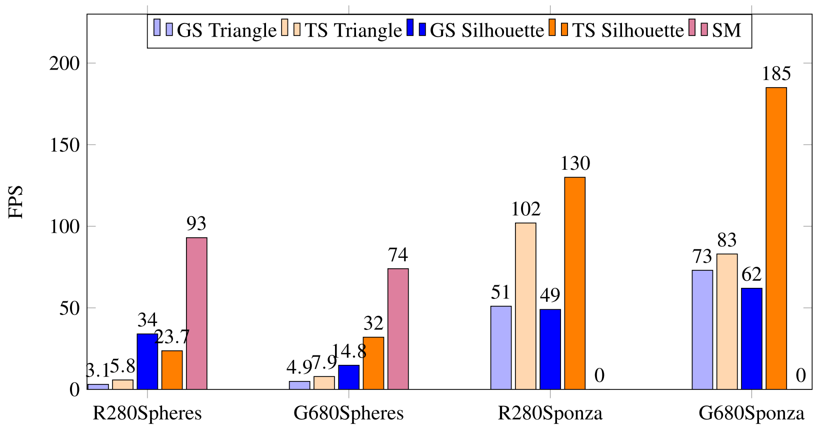


Table 6.6: Comparison of per-triangle and silhouette methods on spheres scene

34 % of SM performance whereas GTX 680 was only 44 % slower than SM. This makes our method on-par or faster than Sintorn's AFSM, depending on the platform used.

We have noticed a different behaviour among both of the tested graphics cards when the camera was not facing the scene. Although OpenSceneGraph, which we used in our test application, culls non-visible geometry, it does not cull shadow volumes from the rendering process. We set up two camera views, one looking towards the scene and the second looking away (an empty view), from the same position. The test was conducted on two scenes, with a single sphere and a grid of 50 × 50 spheres. Both cards seem to deal with such situation differently, as can be observed in the Table 6.5, which shows the FPS ratios between the empty view and scene view. GTX 680 was more efficient in culling the non-visible shadow volume geometry, being up to 5-times faster than the full scene overview. AMD hardware, at that time, was not very efficient, providing only up to 98 % increase in speed.

The last test conducted was comparison of per-triangle versus the the silhouette methods and shadow mapping (SM only on the sphere scene as it was not omnidirectional). The sphere scene had around 4 million triangles and the results are presented in the Table 6.6. To our surprise, the per-triangle TS method was actually faster than both GS methods on Sponza scene on both tested platforms, although the difference between per-triangle and silhouette method is 122 % on nVidia graphics card and only 27 % on AMD. The sphere test scene, having almost 8-times more triangles, favors silhouette methods. Compared to shadow mapping, AMD card drops below the 1/3 performance ratio but GTX680 was able to maintain 44 % of SM performance.

6.5 Evaluation

The two-pass per-triangle tessellation method is, in some cases, faster than silhouette algorithm implemented in geometry shader, but loses performance as geometry amount in the scene increases. Compared to geometry shader per-triangle implementation, the tessellation method proved to be faster in all our tests, no matter the amount of scene complexity. The silhouette method is more efficient and as we have proven in our measurements, mostly in the scenes with higher amount of geometry. GeForce GTX680 benefited greatly from this algorithm, being faster than geometry shader silhouette method in every case. As for Radeon R9 280X, geometry shader method is more suitable. Tessellation method on Radeon was faster on Sponza scene, but the synthetic tests on the scene with configurable amount of spheres and level of detail showed that it's performance is dominant only up to 300K of triangles when having multiple objects in the scene, or only up to 15K triangles when only a single detailed object was drawn. was able to outperform nVidia-based card on the less detailed scenes, but only up to aforementioned 300K triangles.

The robust algorithm was sped up by using a novel method of multiplicity computation, which was able to provide up to 31 % performance gain in tessellation method (13.5 % in average), maximum speedup in geometry shader was 10.7 % with average of 3.4 %.

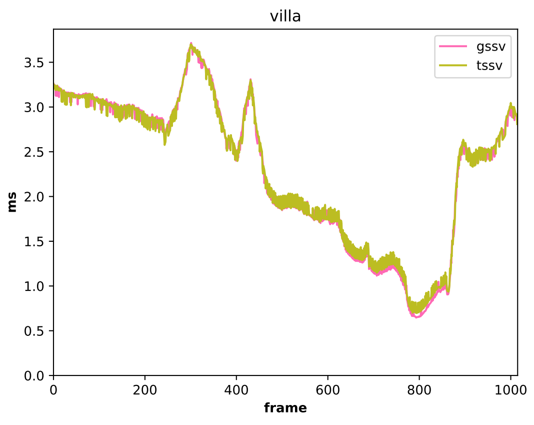
In comparison to standard SM and Sintorn's Alias-Free Shadow Maps (AFSM), our tessellation method provides better performance than 8K shadow maps up to 400K triangles and then falls to 43 % performance of shadow mapping at 4M triangles on GeForce, 34 % on Radeon, which is on par or better than AFSM (which is 3-times slower than 8K SM) and is also simpler to implement.

6.6 Comparison on Modern Hardware

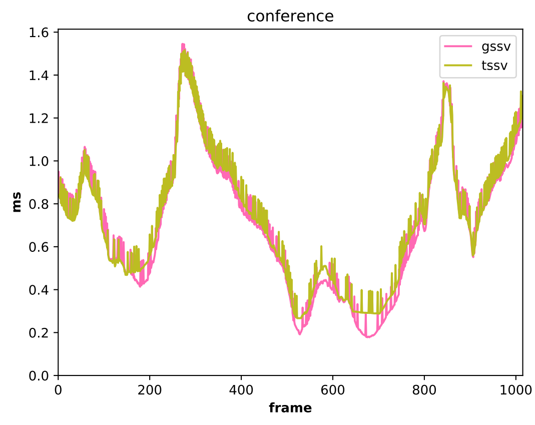
I re-measured both geometry shader and tessellation shader implementations on newer hardware to test the difference between them over the course of several GPU generations. The tests were conducted on GeForce GTX 2080Ti and AMD Radeon 5700XT running on AMD ThreadRipper 1920X with 32GB of memory and Windows 10. Several popular test scenes were selected - Sintorn's "Villa", "Conference Room", "Crytek Sponza", "Citadel", "Buddha" and "Hairball". The scene properties are described in the Table 8.1. The algorithms were tested on flythroughs having 1000 frames, each frame rendered 5-times and an average of these times used in the evaluation. Two resolutions were tested: 1920×1080 and 3840×2160. An average time was computed from the whole flythough. The results can be seen in the Table 6.7, frame times of 2080Ti in the Figure 6.9 and Figure 6.10 shows the performance of the Radeon 5700XT.

Based on the data in the Table 6.7, the geometry shader implementation on RTX 2080Ti is on average 5.36 % faster than the tesselation at 1920×1080, the gaps narrows to 1.5 % in 4K, the biggest difference on the "Buddha" test scene where the geometry shader was 7 % faster. Tessellation was marginally faster in two cases in 4K. I assume that the higher amount of rasterization in the higher resolution diminished the computational difference between the methods.

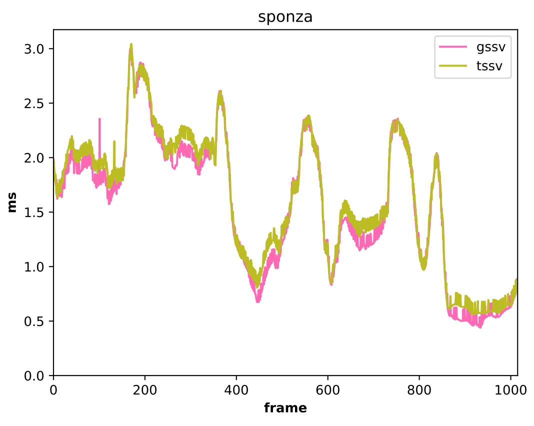
The AMD card shows more decisive results - geometry shader implementation is faster in every scenario, at full-HD by 8 % in average, at 4K by 2.91 % in average. To compare both cards, RTX 2080Ti in combination with GSSV is in average 12 % faster in fullHD, TSSV about 14.7 % faster. The difference between the cards is smaller in 4K - only 1.49 % in average in GSSV test and 2.88 % in TSSV evaluation.



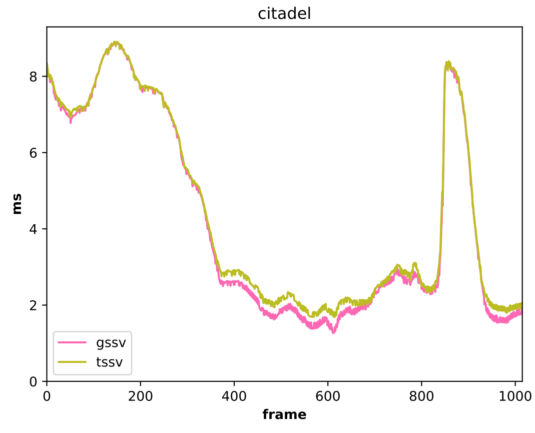
(a)



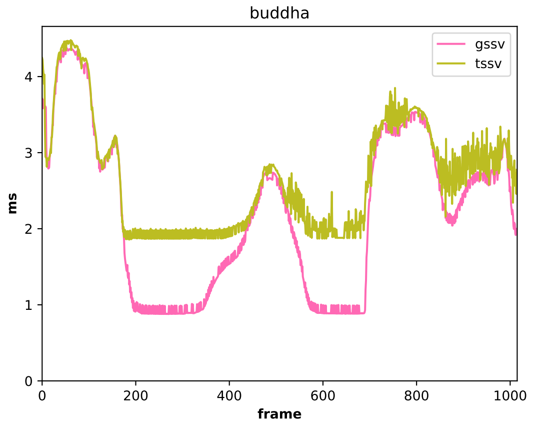
(b)



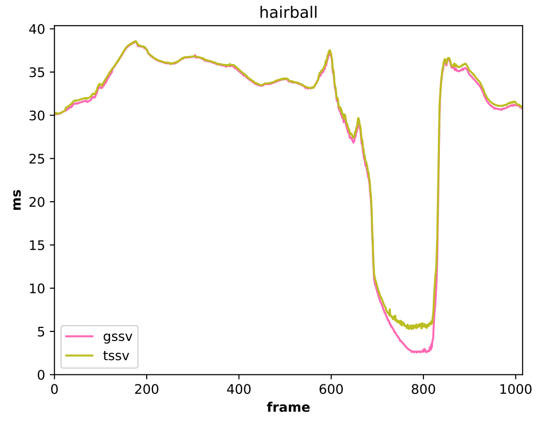
(c)



(d)

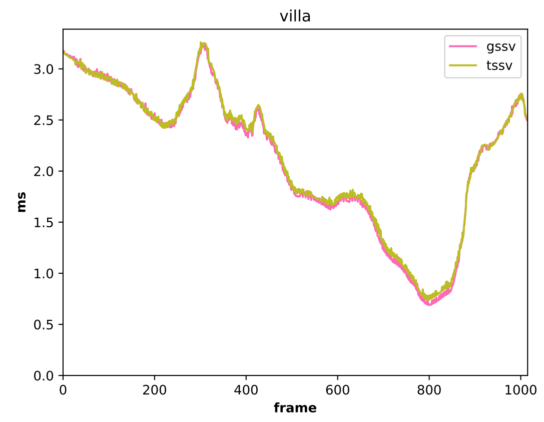


(e)

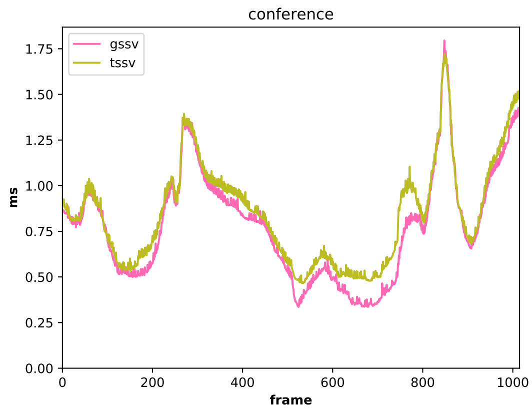


(f)

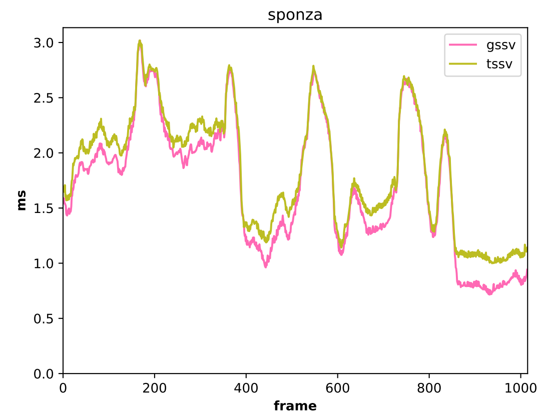
Figure 6.9: Comparison of geometry shader (GSSV) and tessellation (TSSV) implementation of the robust shadow volume algorithms at 1920×1080 on GeForce RTX 2080Ti.



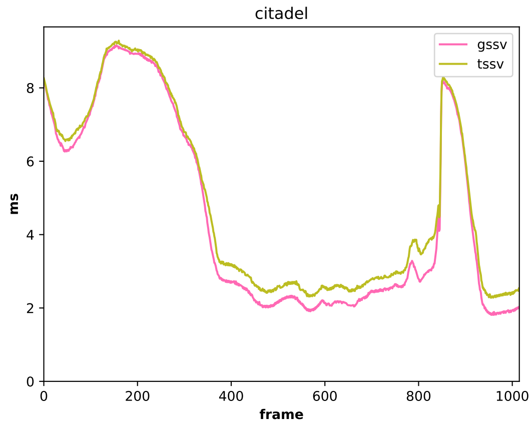
(a)



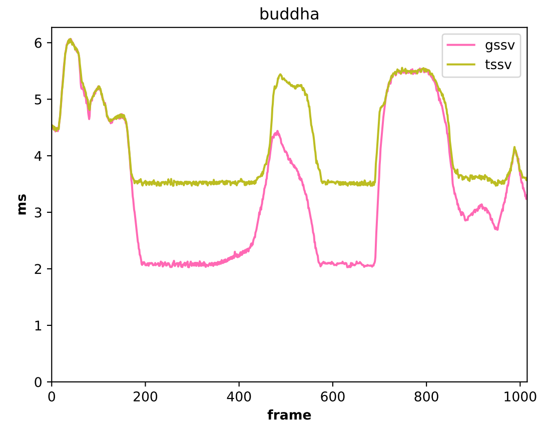
(b)



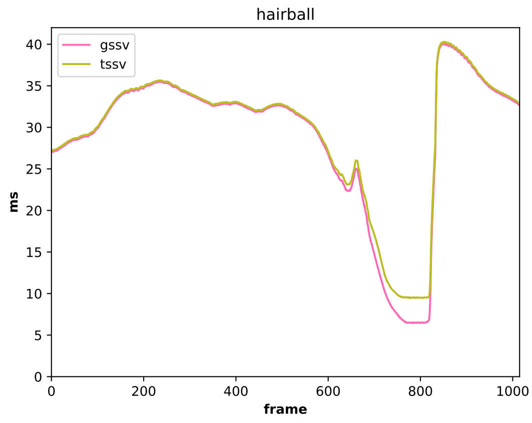
(c)



(d)



(e)



(f)

Figure 6.10: Comparison of geometry shader (GSSV) and tessellation (TSSV) implementation of the robust shadow volume algorithms at 1920×1080 on Radeon 5700XT.

	RTX 2080Ti				RX 5700XT			
	1920x1080		3840x2160		1920x1080		3840x2160	
	GSSV	TSSV	GSSV	TSSV	GSSV	TSSV	GSSV	TSSV
Villa	**2.31**	2.32	5.56	**5.52**	**2.13**	2.15	**4.84**	4.87
Conference	**0.71**	0.73	**1.82**	1.83	**0.78**	0.84	**1.70**	1.74
Sponza	**1.56**	1.62	4.32	**4.31**	**1.71**	1.84	**4.42**	4.53
Citadel	**4.24**	4.38	**9.80**	9.93	**4.52**	4.85	**8.97**	9.18
Buddha	**2.22**	2.67	**3.90**	4.20	**3.46**	4.23	**5.46**	5.93
Hairball	**29.69**	30.15	**64.12**	64.41	**28.91**	29.48	**60.68**	61.35

Table 6.7: Average shadow mask creation times in *ms* on both platforms and both algorithms. The bold values mark the faster of the two algorithms, within one platform, scene and resolution.

The biggest difference was measured on the “Buddha” scene - GeForce was 57 % faster than Radeon at fullHD, 40.6 % faster at 4K. This scene has higher amount of geometry but simple silhouette, which means that raw computational power could be more exploited than raw rasterization performance as the RTX 2080Ti is generally faster than its AMD counterpart. Compared to previous observations, the tessellation method is no longer faster than the geometry shader implementation on both platforms.

Chapter 7

Silhouette Extraction Using Precomputed Potentially Visible Set

Silhouette extraction is a fundamental part of stencil shadow volumes as per-triangle approach has been proven to be slow. The research around the acceleration of silhouette extraction was focused mostly on 2-manifold meshes. As the shadow volume algorithms presented in the previous chapter deal with non-manifold casters, I have decided to design a method that would improve silhouette extraction speed on such casters.

The paper that describes this new method was presented on WSCG conference [65]. I am the author of the method, its implementation and wrote the text of the paper.

7.1 Precomputed Silhouette Extraction Overview

Several methods have been designed to improve silhouette extraction of 3D models, as documented in the Chapter 3.1.2. Most of the methods, however, deal only with 2-manifold casters and as I stated earlier in the thesis, our focus were arbitrary triangle soups. The method described in this chapter was based on a method by Airey et al. [4], using brute-force approach to precompute the sets of potentially silhouette edges and stores them in a hierarchical data structure. Octree was chosen for this matter. As the resulting acceleration structure tends to grow in size notably, we designed a compression scheme to reduce the size of the resulting octree. The whole process is outlined in the Figure 7.1.

7.2 Octree Setup

The octree itself represents the volume where the point light source can be located. This marks the area where the silhouette computation is accelerated and represents the area with the highest probability of where the light source will be located. In the actual implementation, we allow the user to configure the size of the top level of the octree based on the scene's AABB and a multiplication factor that scales the size of the AABB. The scaling factor could be set to lower values (around 1) for closed scenes such as Sponza where the light source can be located inside the scene's AABB or higher values (5-10 or more) for simple models such as Happy Buddha. Figure 7.1b shows very tight fitting of the octree

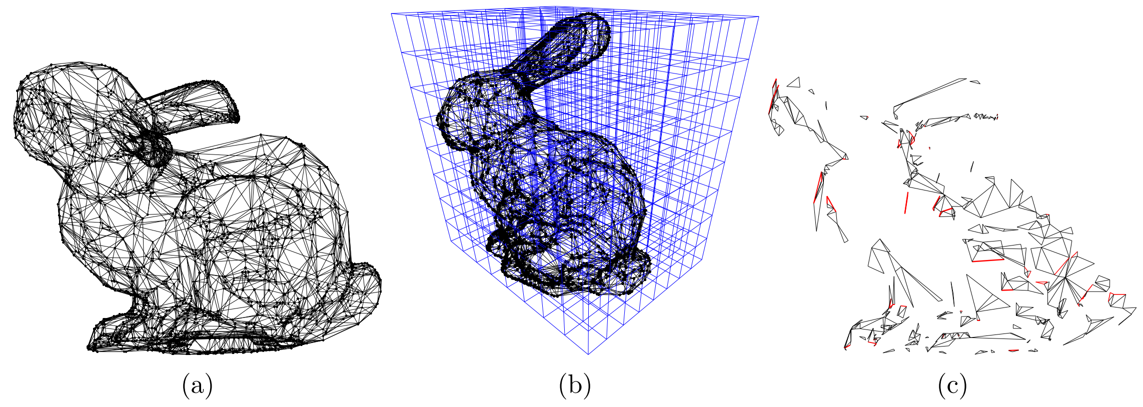


Figure 7.1: Mechanics of the proposed algorithm *a*) A model of a bunny having 5000 edges. *b*) We enclose the model by user-defined axis-aligned bounding box which acts as a top level of the octree and is subsequently subdivided and filled with precomputed sets of silhouette and potentially silhouette edges. *c*) During traversal, only a small amount of edges needs to be tested (black) and end up as silhouette edges (red).

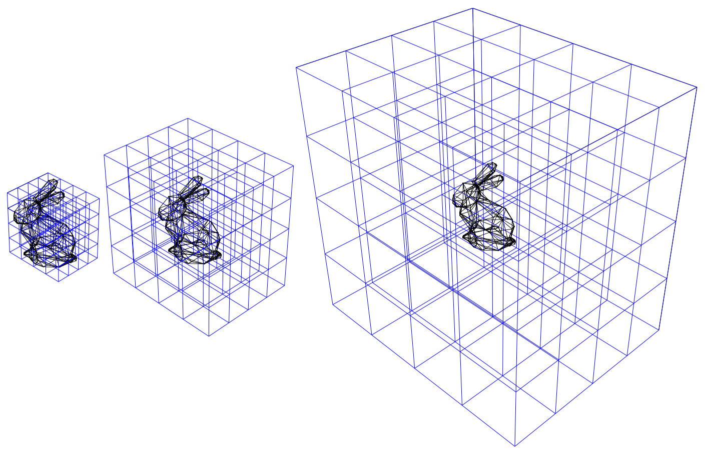


Figure 7.2: Custom scale of the model's AABB to cover larger volume for accelerated silhouette extraction. Image shows two-level octree with same voxelization ($4 \times 4 \times 4$).

around the model, in practice the octree would have to cover much larger space in this scenario, as depicted in the Figure 7.2.

The depth of the octree can also be customized. We have experimentally found out that the depth of $3 - 5$ is sufficient in the most scenarios, leading to total of $512 - 32\,768$ cells on the lowest level, as seen in the Figure 7.3. Larger scales consume too much memory, as will be described below, because each octree level increases the memory consumption of the method by a factor of 4.

7.3 Octree Build

The build process fills the octree with sets of potentially silhouette edges (PE) and edges that are guaranteed to be silhouette (SE) for a particular voxel. At first, the set of all edges and their opposite vertices is extracted. In order to test an octree voxel (as AABB) against an edge, a plane is constructed from both the edge vertices and the opposite vertex (OV).

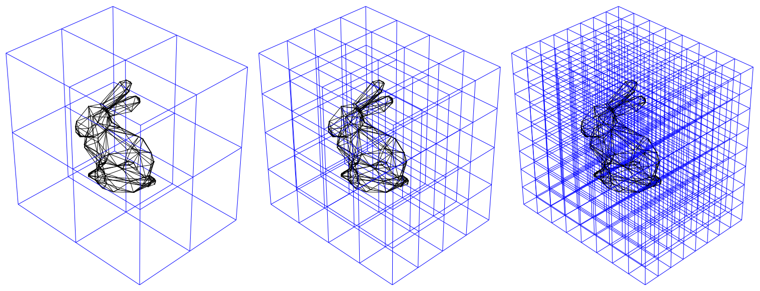


Figure 7.3: Custom levels of octree subdivision, from level 1 (8 bottom level voxels) on the left to 3 (512 voxels) on the right.

This plane is then tested for intersection with the AABB. If any of the planes intersect the AABB, the edge is stored among the PE set of the particular voxel because if the light source is located inside the volume the multiplicity of the edge from any point inside the volume is not consistent and further subdivision is required to store such edge as silhouette. If the volume is not intersected, two scenarios are possible – either the edge is a SE or is not silhouette at all. To determine this, multiplicity of the edge is calculated from an arbitrary point of the voxel, based on the algorithm described in the Chapter 6.2.2. We chose the minimum point of the voxel's AABB. The whole process is outlined in the Algorithm 12.

Algorithm 12 Function to compute edge – voxel intersection

```
Function: GetEdgeVoxelState
Input: Edge vertices $A, B$, set of OVs $\Omega$, voxel $AABB$
Output: Status PE/SE/NSE $s$, multiplicity $m$
 1: $\Pi \leftarrow Points(AABB)$
 2: $NofPAABB \leftarrow 8$
 3: $m \leftarrow 0$
 4: $s \leftarrow PE$
 5: for $O_i \in \Omega$ do
 6:     $EP \leftarrow ComputePlane(A, B, O_i)$
 7:     $C \leftarrow 0$
 8:     for $P_i \in \Pi$ do
 9:         $C \leftarrow C + TestPointPlane(EP, P_i)$
10:     end for
11:     if $abs(C) == NofPAABB$ then
12:         $m \leftarrow ComputeEdgeMult(A, B, \Omega, GetMinPoint(AABB))$
13:         if $m == 0$ then
14:             $s \leftarrow NSE$
15:         else
16:             $s \leftarrow SE$
17:         end if
18:     end if
19: end for
20: return $(s, m)$
```

ComputeEdgeMult is the same function as in Algorithm 9. Function *ComputePlane* uses similar determinism mechanism as *ComputeEdgeMult* – sorts the points before com-

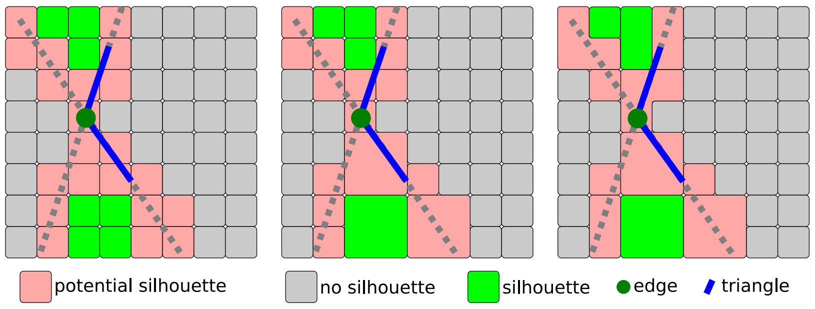


Figure 7.4: The image shows voxels for one edge. The left side of the images shows the first step of the voxel building algorithm. Voxels are classified into 3 categories -– non silhouette, silhouette and potentially silhouette. The next step is to propagate this classification into higher levels (middle image). The right image shows the improvement of compression stage of building algorithm. The octree is transformed into a tree with nodes containing many different subsets of edges defined by bitmasks.

puting the plane, *NSE* as a status stands for *non-silhouette edge*. 2D schematics of the test can be seen in the Figure 7.4 left.

We chose to build the octree in bottom-up fashion, similarly as Karras et al.[55] as their method first builds the acceleration structure and then applies post-processing to further refine it. The proposed method first subdivides the octree's volume (top-level AABB based on scene's AABB) into the smallest voxels based on octree depth. Each voxel is then processed and tested against every edge in the scene, filling its PE and SE sets of edges. We store only edge ID's (and multiplicities for SE) as storing the whole edge would be superfluous. Then, the PE and SE edges are propagated up the hierarchy. If an edge is found common in all 8 siblings (voxels that share the same parent), the edge is stored in the parent instead and removed from all the children. The situation can be observed in the Figure 7.4 middle. This step already decreases the memory consumption significantly.

Provided we store each edge ID as a 32-bit integer and testing on Sponza model with 279 163 triangles, 431 246 edges and octree deepest level of 5 (32 768 voxels) and AABB scale factor of 1, this would yield a total of 25 GB of memory. According to the silhouette estimation proposed by Akenine-Möller and Assarsson [6] which estimates the silhouette size to $f^{0.8}$ where f is the number of triangles, the silhouette would be only 22 734 edges. Provided we store this number of edges in the same octree as mentioned above, we would end up with around 2.9 GB of memory. These cases are without the edge propagation up the tree. When real tests was conducted on Sponza, we measured 4644 MB when not propagated, 4111 MB upon propagation to the upper levels.

7.3.1 Compression Scheme

We decided to address the memory consumption and designed a compression scheme to push more edges up the hierarchy. As mentioned above, we first propagated edges up the octree only if all the eight siblings of one parent share the same edge.

But we can propagate the edges even if not all the sibling share the edge. In such case, we create a bitmask representing the presence of the edge among the parent's children and store the edge under such bitmask. In context of an octree, the bitmask is 8-bit long. This means that it is viable to propagate an edge to parent if it's included in at least 2 of its children as the edge would be stored once (32-bit) plus bitmask (8-bit) instead of being

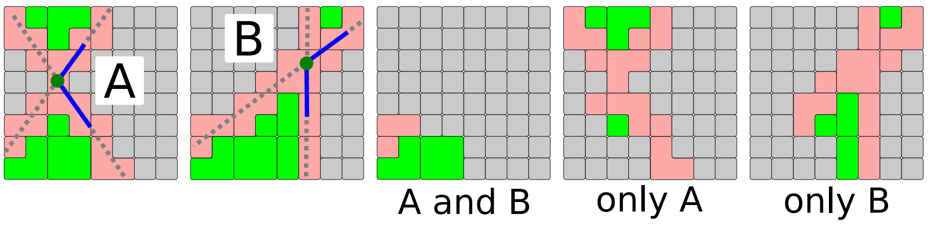


Figure 7.5: The first two images show two edges – A and B. Each edge partitions all voxels into voxel shapes for silhouette case and of potential silhouette case. If some voxel shapes are the same for both edges, the edge subsets of those voxel shapes contain both edges (middle image). Otherwise, voxel shapes contain only one edge.

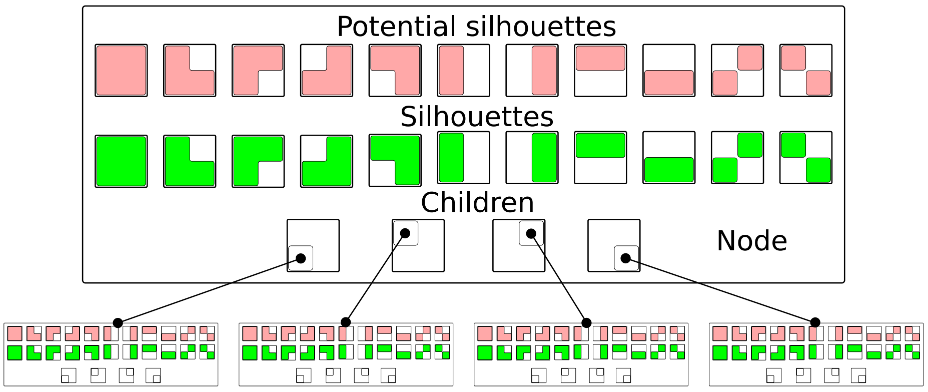


Figure 7.6: Node data in 2D space for the 8-bit bitmask. One node contains sets of PE and SE edges, each addressed by its bitmask. If a set shape does not intersect the triangle planes of an edge, the edge is stored into the set. The largest set shape is chosen if multiple set shapes do not intersect the triangle planes. A node also contains pointers to child nodes.

stored twice (64-bit), which saves around 38 % of memory for the worst case and up to 91 % when 7 siblings share the edge. The approach can be seen in the Figure 7.4 on the right as grouping of neighbouring 4 × 4 voxels into a larger chunks. Figure 7.5 demonstrates a case of 2 edges and a whole level of an octree. Common and disjunct parts are visible in the images, common parts propagated to the upper levels.

Figure 7.6 shows data representation of a node in a 2D quadtree. Each node, representing a voxel, contains two sets of edges – PE and SE, each group of edges represented by a bitmask. This bitmask can be displayed as a shape covering child voxels. If we used the same test scenario as above, the octree size reduced from 4111 MB to 1359 MB which is 33 % of the original size.

In the first version of the algorithm we used this compressed edge propagation only from the lowest to the second lowest level of the octree. As we only propagated between two levels, we designed a similar approach that propagated edges from the lowest level up by two levels and used 64-bit mask instead of 8. Although it further reduced the size of an octree (the test case – from 1359 to 498 MB), the potential amount of bitmask per edge set rose rapidly, e.g. Sponza's octree with scale of 1 and maximum level of 5 can have up to 44 643 bitmasks in a single voxel for a one set of edges. This amount of bitmasks also has negative impact on traversal, which will be described in the following sections.

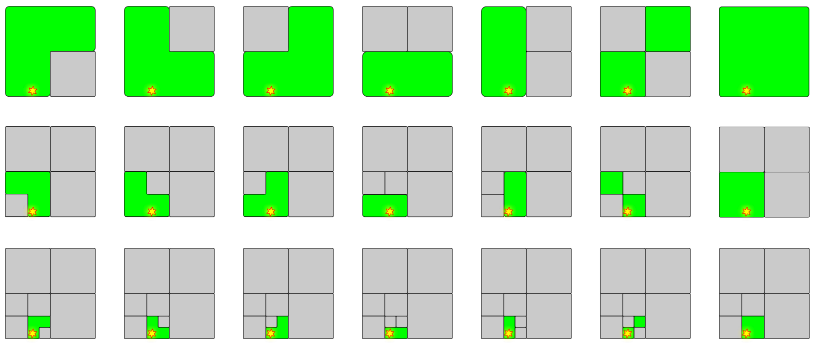


Figure 7.7: Traversal process in 2D for a particular light position through 3 levels of hierarchy. The union of all subsets forms the set of all precomputed silhouette edges. Similar subsets are selected for all potentially silhouette edges. Note that some subsets could be empty. A single edge is contained only in one of the subsets (the largest possible).

7.4 Octree Traversal

The traversal process has to acquire PE and SE subsets from the octree to two buffers. The light source is first mapped into the space of the octree, which can be traversed either top-down or bottom-up. The process acquires all edges from the node where all bits of a bitmask are set and then from all those bitmasks where the the bit of child node on the traversal path is set. Not all bitmasks are necessarily populated.

When traversing top-down, the method needs to select the appropriate child based on AABB-point containment test. On the other hand, traversing the octree bottom-up seems easier because at first, the lowest level voxel index is calculated from the light's position, as seen in the Algorithm 13. As the octree is not sparse and stored linearly, it is easy to compute parent node's ID as well as the index within parent (ChildID), see Equation (7.2) and 7.3.

$$l_x \in \lfloor A_x, B_x), \wedge l_y \in \lfloor A_y, B_y), \wedge l_z \in \lfloor A_z, B_z) \tag{7.1}$$

If a light source lies on a boundary between two or more voxels, only one voxel is selected according to the Equation (7.1) where l is the light position and A, B are minimal and maximal points of the voxel.

$$Parent = \lfloor \frac{NodeID - 1}{8} \rfloor \tag{7.2}$$

$$ChildID = NodeID - (8 * NodeID + 1) \tag{7.3}$$

7.5 Implementation

I have created two versions of the algorithm – the first, presented in the paper, only compressed only one level of the octree, either using 8-bit or 64-bit bitmask. This version was published in the paper [65] as it was the limit of the 64-bit compression scheme. I later reimplemented the method to support 8-bit bitmasks on all of its levels and tested against compute shader implementation of shadow volumes.

Algorithm 13 Get lowest level voxel index from light position

```
Function: GetLowestLevelIndex
Input: Light position $L$, Octree $O$
Output: Voxel index on the lowest level $I$
 1: $BBOX \leftarrow O$.getNodeVolume(0)
 2: if IsOutside($L$, $BBOX$) then
 3:     return $-1$
 4: end if
 5: $DL \leftarrow O$.getDeepestLevel()
 6: $T \leftarrow 2^{DL}$
 7: $SingleUnitSize \leftarrow BBOX$.getMin()$/(T, T, T)$
 8: $PosUI \leftarrow \lfloor (L - BBOX$.getMin()$)/SingleUnitSize \rfloor$
 9: $I = 0$
10: for $i \in [0, DL - 1]$ do
11:     $Q \leftarrow 8^i$
12:     $I \leftarrow I + (((PosUI.x >> i)\&1) << 0) \cdot Q$
13:     $I \leftarrow I + (((PosUI.y >> i)\&1) << 1) \cdot Q$
14:     $I \leftarrow I + (((PosUI.z >> i)\&1) << 2) \cdot Q$
15: end for
16: $I \leftarrow I + O$getNofNodesInPreviousLevels(DL)
17: return $I$
```

7.5.1 Build Implementation – CPU

The method presented in the paper used bottom-up build process. First, the lowest level voxels are filled with data. Each voxel goes through all the edges and tests them for PE/SE. If an 8-bit bitmask is selected, the algorithm tests 8 siblings at once. As such task is easily parallelizable, I utilized OpenMP to run a single thread per a group of 8 sibling voxels using voxel-edge test in the Algorithm 12. Potentially silhouette edges are stored without encoded multiplicity, the silhouette edges use encoding scheme disclosed in the Algorithm 14.

Algorithm 14 Encode edge ID and multipolicity into a single value

```
Function: EncodeEdgeIdMultiplicity
Input: Edge ID $E$, Multiplicity $M$, Number of Bits per Multiplicity $BPM$
Output: Encoded edge ID and multiplicity $EE$
 1: $BitMask \leftarrow (1 << BPM) - 1$
 2: $EE \leftarrow (E << BPM) \,|\, (M \,\&\, BitMask)$
 3: return $EE$
```

After the lowest level of the octree is filled (or two lowest levels if 8-bit bitmask is chosen), the edges are propagated up the hierarchy, initially only the bitmasks where all bits are set (e.g. the all siblings share the edge). Every set of 8 siblings is processed by a single thread. The resulting structure of a voxel contains two unordered maps, each addressed by a bitmask and containing vectors of edge IDs or encoded IDs with multiplicity.

The 64-bit bitmask compression scheme was implemented as a post-process of the octree, constructed without the 8-bit compression. Although the compression could have been integrated to the voxel-edge testing process, the compression scheme did not prove to be interesting by its performance and thus we kept the compression as a separate CPU postprocess. The postprocess creates a set of all edges stored in 64 siblings and creates bitmasks for all edges by testing their presence among all siblings. Each set of 64 siblings is processed by one thread. As the amount of 64-bit bitmasks could be very large, its GPU implementation would be somewhat complicated.

7.5.2 Build Implementation – GPU

In order to speed up the build process, I designed a GPU algorithm to cope with the issue. The biggest concern was the memory – size and optimal layout. As described above, the octree can potentially be very large and in order to bring its construction to GPU, buffers need to be allocated. As it is not known how much data will be stored in each voxel, we must assume the worst case – that all the edges may fall into a single bitmask. As the intermediate octree may not necessarily fit into the GPU memory, I implemented an iterative approach which processes a batch of voxels against all the scene edges and the results are then stored back to the CPU memory.

The amount of voxels in a single batch is calculated based on the maximum size of the buffer we want to allocate for the lowest level voxels (they only store 1 bitmask) divided by the number of edges. In order to speed up the process, I used speculative buffer size on the lowest level per mask as ratio of the full number of edges. This optimization could speed up the process by 20 % by allowing more voxels to be processed at one but when the ratio was too low, the resulting structure was corrupted. The parent nodes need to allocate space for all of its bitmasks based on the speculative ratio as well, in total $2 \cdot 255 \cdot speculativeRatio \cdot nofEdges$ for a parent voxel.

The compute shader then tests a group of 8 siblings against an edge, processed by 8 threads in a subgroup (subgroup being the smallest unit of execution on the GPU). In total, one subgroup of threads (32 on nVidia, 64 on AMD hardware) processes 4 (8 on AMD) siblings. Based on the resulting bitmask, the edge is assigned either to the siblings (not enough siblings tested to contain) or parent. Ballot intrinsic (extension `GL_ARB_shader_ballot`) was used for propagating the result of edge – voxel intersection test among all the siblings. As the subgroup processes 4-8 groups of siblings, the appropriate result for the particular group of 8 threads is extracted using bit shifts and masking. To speed up the process, each workgroup (consisting of several subgroups) first loads a set of edges to shared memory as these edges will be used by all the threads in the workgroup until they are tested by all workgroup threads. Loading them only once from the global memory (which is performance demanding) increased the performance of the solution. Using profiler, the optimal settings found for the kernel execution were 64 threads (2 subgroups on nVidia hardware) and 31 744 bytes of shared memory per workgroup.

The CPU waits for the kernel to finish execution, the edge data are then copied to the CPU memory and next batch of lowest level voxels is prepared for processing. The outline of the algorithm can be seen in the Algorithm 15.

When the lowest two levels are processed, the edges are then propagated up the hierarchy by another compute shader, which operates similarly as the previous shader. Before the shader is launched, all edge IDs in all voxels and bitmasks are sorted on the CPU. The threads also cooperate in sibling groups of 8. Each sibling group of 8 threads iterates over the edges of the first thread in the group, which skips edges not present in the first but other threads. This approach produces slightly larger octree (around 1-2 %), but otherwise all the edges would have to be put into a single set and tested for containment in each

Algorithm 15 Adding edges to the lowest and 2nd lowest level of the octree

```
1: $SZ \leftarrow$ CalcBatchSize()
2: $NofBatches \leftarrow \lceil$GetNofVoxelsLowestLevel()$/SZ\rceil$
3: $LoadedVoxels \leftarrow 0$
4: for $i \in NofBatches$ do
5:     LoadVoxels(min($SZ$, GetNofVoxelsLowestLevel() $- LoadedVoxels$))
6:     DispatchCompute()
7:     SyncGpuCpu()
8:     AcquireVoxelData()
9: end for
```

sibling or more testing among the sibling threads. The first thread reads the following edge from its buffer (pivot edge) and all other threads iterate in their buffers (corresponding to full bitmask), skipping edges that are less than the pivot edge to find the first edge that is equal or higher than the pivot. Then, a ballot is called to determine if the pivot edge is present among all the siblings and stored to parent if all threads report it as present. When the kernel finishes the computation, its results are written back to the main memory. This process iterates bottom-up, starting with the 2nd lowest level upwards.

7.5.3 GPU Traversal

I primarily focused on providing the fastest possible traversal of the octree. Bottom-up traversal was chosen as it is easy to compute the lowest level voxel ID where the light source is detected, as well as parent ID and child ID within the parent. Serialized octree is loaded to the GPU memory and if necessary, split into chunks of up to 2 GB. However, evaluation has shown that very large octrees (more than 2 GB) tend to be twice as slow as those that fit inside the 2 GB; it is thus necessary to adjust the octree parameters to fit it into the size of 2 GB.

Initially, the process required 3 compute shaders to traverse an octree with 8-bit bitmasks on the second lowest level. The first traverses the path bottom-top, selecting all the bitmasks that are valid for processing, gets their sizes and stores them as a prefix scan of their sizes and starting offsets. This kernel spawns only a single workgroup and each thread processes one bitmask in node, traversing bottom-up. In order to calculate prefix scan and its storing index in the global memory using only a single atomic addition, we split a 32-bit integer into 8 bits for the storing position and 24 for the actual prefix scan. This limits the amount of edges to 2^{24} which is roughly 16.7 million. The updated implementation described in the Chapter 7.7.1 does not pose this limitation. We can use 8-bit for storing the index – the total amount of 8-bit numbers having exactly i-th bit set is 128 and not all of them are valid (single bit set), plus the $octreeDepth - 1$ more masks for levels other than 2nd lowest is still less than 2^8 as it is improbable that the octree would be so deep that the sum would overflow 8-bit limit. As stated above, the octree depth in practice is no more than 6 levels $(0 - 5)$.

The computed prefix sum goes to a kernel performing copy operation – copies edge IDs for processing into potential and silhouette buffer. Each thread processes a single edge. First the thread finds its index by binary searching the prefix sum and choosing potentially silhouette or silhouette buffer, then performs the copy operation.

The last kernel takes the edge IDs and processes them – potentially silhouette edges are tested against the light source and cast, data stored in the silhouette buffer are first decoded (edge ID, multiplicity) and then cast. To reduce global memory writes, shared memory is used to perform per-subgroup atomic operations, the result of the subgroup is then atomically added to global memory variable. This technique is used to allocate index for storing the cast edges in the global memory. The shadow volume geometry is then drawn using only a vertex shader.

The octree with 64-bit bitmasks requires more complicated prefix scan implementation as the amount of buffers can go to thousands. Prefix scan is carried out for potential and silhouette edges separately, two kernels when the expected amount of bitmasks is less than 1024, 3 kernels if more. 1024 is typical limit on the amount of threads per workgroup. If more workgroups need to cooperate on the results, an extra kernel is necessary to create a prefix sum from the last elements of the summed data per workgroup. Finally the third kernel adds these sums to the respective elements. This approach is based on Harris's CUDA algorithm [40].

7.6 Measurements

The tests were designed to evaluate build and traversal process of the octree. The method was implemented in a test application using OpenGL, compiled in Visual Studio 2015 x64 and tested on Core i5-6500 system with 16 GB of RAM using GeForce RTX 2080Ti graphics card.

7.6.1 Built Tests

The aim of these tests was to evaluate the building time and octree sizes on several popular models – Šibenik cathedral, Conference room and Sponza. All models come from McGuire's archive [79].

The scenes were evaluated under several different octree settings as seen in the Tables 7.1, 7.2 and 7.3. We set the octree depth to 3–5 as these values present the most usable cases. We have also tested the impact of the increasing voxelized space around the model by scaling the scene's AABB. The tests cover 3 types of edge propagation – propagating only those edges that are common for all 8 siblings up the hierarchy, the 8-bit bitmask utilized to propagate between bottom and 2nd lowest level and 64-bit bitmask compressing between lowest and 3rd lowest octree level.

One of the first notable things is that the build scheme utilizing 8-bit bitmasks is actually faster than the scheme without any bitmasks. This is due to fact that GPU compression occurs in the very first stage of the algorithm thus the following stages have to process a smaller amount of data. However, the 64-bit bitmask propagation is performed as a postprocessing step and it happens on the CPU, thus being very slow, even though the algorithm was written using OpenMP. We tested the 64-bit compression also on the AMD ThreadRipper system with 24 cores, which improved the 64-bit compression build time by around 60 %, but other two methods performed significantly slower, probably due to different architectures of the two processors.

The amount of memory required to store the octree increases roughly by a factor of 4 with every level. On the other hand, the amount of SE extracted from the octree increases by about 10 % and the amount of PE is halved with every level of the octree. This could be observed on all test scenes.

Octree Depth	Scale	Size (MB)	Build (s)	Size c8 (MB)	Build c8 (s)	Size c64 (MB)	Build c64 (s)	Pot Avg	Sil Avg	Pot % Avg	Sil % Avg
3	1	52	0.62	16	0.58	5	11.39	19668	16105	16.76	72.20
	2	57	0.60	18	0.57	5	14.69	21586	15649	18.40	69.91
	4	58	0.61	18	0.57	5	17.12	22088	15514	18.82	69.25
	8	58	0.61	18	0.57	5	17.98	22179	15494	18.90	69.26
	16	58	0.61	18	0.57	6	18.35	22147	15498	18.87	69.19
4	1	235	1.77	77	1.54	27	21.37	10291	18801	8.77	84.28
	2	256	1.75	84	1.55	29	29.13	11359	18531	9.68	82.85
	4	262	1.75	86	1.54	30	32.38	11610	18469	9.89	82.51
	8	263	1.75	86	1.53	30	35.10	11688	18444	9.96	82.37
	16	263	1.74	86	1.53	30	35.32	11690	18448	9.96	82.42
5	1	1022	7.67	341	6.82	127	62.17	5304	20405	4.52	91.52
	2	1122	7.71	376	6.78	139	76.00	5876	20266	5.01	90.62
	4	1147	7.78	385	6.78	141	81.60	6008	20246	5.12	90.45
	8	1155	7.69	388	6.79	143	83.72	6038	20236	5.15	90.41
	16	1155	7.75	388	6.77	142	86.69	6051	20237	5.16	90.39

Table 7.1: Build test of Sibenik scene, consisting of 117 342 edges. We evaluated the build times and resulting octree size under various voxel sizes and scales. The 3rd and 4th columns contain results for octree that only propagates edges common for all 8 siblings up the hierarchy. Columns tagged "c8" and "c64" show build times and sizes when using 8-bit or 64-bit bitmasks. The numbers in "Pot Avg" and "Sil Avg" columns show the average number of PE and SE acquired during octree traversal, tested from each lowest level voxel. The second column from the last tells the average amount of edges from the full edge count that needs to be tested, the last column describes the average amount of SE acquired from octree as the percentage of all silhouette edges observed from light position in the middle of each lowest level voxel.

Octree Depth	Scale	Size (MB)	Build (s)	Size c8 (MB)	Build c8 (s)	Size c64 (MB)	Build c64 (s)	Pot Avg	Sil Avg	Pot % Avg	Sil % Avg
3	1	80	0.84	25	0.79	8	34.95	34338	19426	17.61	65.87
	2	93	0.84	29	0.78	9	50.55	39952	18481	20.49	62.37
	4	96	0.84	30	0.79	10	59.29	41323	18288	21.19	61.56
	8	97	0.84	31	0.78	10	62.89	41656	18265	21.36	61.46
	16	97	0.84	31	0.78	11	64.99	41794	18243	21.43	61.32
4	1	379	2.55	121	2.54	42	69.81	18675	23055	9.58	78.19
	2	431	2.64	139	2.53	48	106.65	21857	22317	11.21	75.35
	4	443	2.62	144	2.52	50	124.19	22595	22187	11.59	74.71
	8	446	2.64	145	2.53	51	134.23	22783	22154	11.68	74.54
	16	447	2.65	146	2.54	52	136.04	22832	22149	11.71	74.51
5	1	1786	10.66	581	8.80	209	150.24	9894	25738	5.07	87.29
	2	2044	11.49	668	8.92	238	222.80	11698	25234	6.00	85.18
	4	2102	11.12	687	8.61	245	265.10	12123	25151	6.22	84.69
	8	2120	11.23	694	8.81	248	284.60	12219	25141	6.27	84.58
	16	2121	11.28	694	8.96	249	291.80	12241	25135	6.28	84.56

Table 7.2: Build test of Conference scene, consisting of 195 019 edges. See Table 7.1 for column description.

Octree Depth	Scale	Size (MB)	Build (s)	Size c8 (MB)	Build c8 (s)	Size c64 (MB)	Build c64 (s)	Pot Avg	Sil Avg	Pot % Avg	Sil % Avg
3	1	192	1.56	60	1.40	19	270.59	83074	31640	19.26	58.98
	2	202	1.57	63	1.38	20	332.78	86744	30845	20.11	57.30
	4	205	1.59	65	1.37	20	361.90	87829	30661	20.37	57.08
	8	206	1.62	65	1.37	21	373.43	88157	30644	20.44	57.08
	16	206	1.60	66	1.38	21	379.24	88382	30618	20.49	57.02
4	1	893	5.31	289	5.10	96	586.34	44817	39567	10.39	73.72
	2	930	5.36	302	4.62	101	705.98	47044	39029	10.91	72.58
	4	943	5.40	306	4.62	102	783.30	47414	38933	10.99	72.34
	8	946	5.42	306	4.64	103	819.03	47559	38906	11.03	72.34
	16	948	5.41	307	4.61	103	837.29	47636	38899	11.05	72.34
5	1	4111	21.17	1359	15.45	498	1210.22	23895	45123	5.54	84.18
	2	4305	21.37	1425	15.90	504	1517.91	25094	44867	5.82	83.47
	4	4345	21.20	1438	16.18	522	1635.41	25321	44782	5.87	83.34
	8	4351	21.25	1441	16.36	511	1735.49	25374	44819	5.88	83.30
	16	4357	21.13	1443	16.26	519	1789.33	25376	44782	5.88	83.31

Table 7.3: Build test of Sponza scene, consisting of 431 246 edges. Check Table 7.1 for column description.

Tables 7.1, 7.2 and 7.3 also show the algorithm's biggest weakness – the memory consumption, dependant on the algorithm settings. For practical standpoint, the use of 8-bit bitmasks seems to be the best choice, in terms of both build time and octree size. The memory consumption of the method can be expressed by Equation (7.4) where S is an approximation of the resulting size of the octree structure in MB, e is the number of edges in millions, d is octree depth and c is compression ratio. The compression ratio was computed as an average ratio between the compressed octree and non-compressed version of the octree. The ratios are 1 for octree without compression (propagating only edges in all siblings), 0.32 when utilizing 8-bit bitmasks and 0.11 than using 64-bit bitmasks.

$$S(e,d,c) = e \cdot 8^d \cdot V_d \cdot c \tag{7.4}$$

Values V_d define the approximate size of a single voxel per 1 million edges. These values were calculated as $Vd(d,e) = S_m/8^d/e$, where S_m is the measured size of non-compressed octree. Values obtained by this equation are $V_3 = 0.93$, $V_4 = 0.53$ and $V_5 = 0.30$. The average relative deviation of Equation (7.4) is 6 %.

7.6.2 Silhouette Extraction Tests

In order to test the silhouette extraction, we implemented a brute-force silhouette extraction method based on OpenCL implementation described in the previous paper [90] but using faster robustness computation described in the Chapter 6.2.2. The method was implemented in OpenGL's compute shader and tested every edge for silhouettness. Both versions of the method outputted edge ID and its multiplicity, shadow volume rendering was not accounted in the resulting time.

We compiled a set of 26 test scenes consisting of several popular test models (Bunny, Šibenik, Conference, Sponza, Gallery, Buddha) [79] as well as two types of synthetic test scenes. The first type were scenes consisting of increasing amount of spheres, arranged in a uniform grid. These scenes had 33 750 to 1 574 640 edges. The second type of synthetic

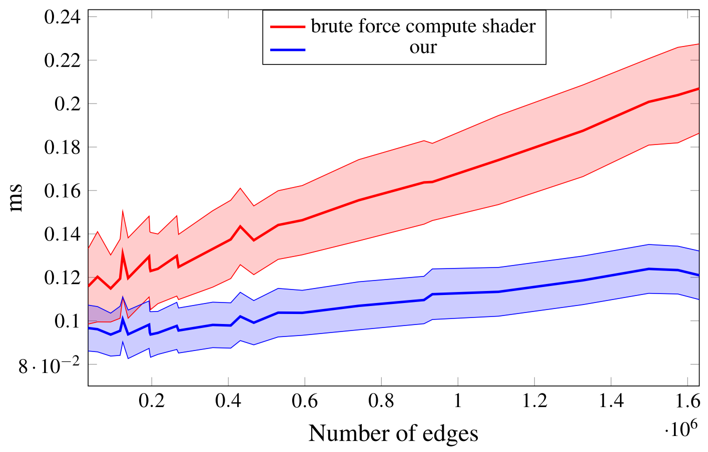


Figure 7.8: Average extraction times for compilation of 26 scenes (sorted by the number of edges). Red line represents brute force compute shader method, blue line represents our new proposed method. The area around the lines represents (±) mean absolute deviation.

Compression	**Average (ms)**	**Max Abs Deviation (ms)**
basic	0.098	0.011
8-bit	0.102	0.012
64-bit	0.134	0.013

Table 7.4: Comparison between compression levels on Sponza scene. Average octree traversal time calculated from 1000 different light positions in the scene and maximum absolute deviation from the average. Row “basic” accounts for propagating edges inly in all siblings.

scenes had randomly positioned spheres, having 124 200 to 933 120 edges. The method was evaluated on octree depths of 3 and 5, posing and the best and worst case scenario, and AABB scales of 1, 2, 4, 8 and 16. In a single test run, we moved the light source throughout the octree volume in a 10×10 grid, both for the hierarchical and brute-force method. For each light source location, we evaluated the extraction time 5-times, resulting in a total of 75 000 measurements in each scene – 50 000 for our approach and 25 000 for the brute force method. The hierarchical method was evaluated for 2 octree depths, that's the reason for having twice as much evaluations as the brute force method.

We computed average value and mean absolute deviation from each scene, the results are shown in the Figure 7.8. It can be observed that the proposed method reduced the sensitivity of silhouette extraction to the number of edges, compared to the brute force approach. The octree-based method performs better on scenes having more than 200 000 edges. Its average absolute variance is almost half of the brute force approach.

Performance of non-compression and 64-bit schemes were evaluated as well, results can be seen in the Table 7.4 which shows an average silhouette extraction time. It can be seen that 64-bit bitmask scheme is about 30 % slower than the other methods; algorithm using 8-bit bitmask seems as the most viable option as it greatly reduces the memory consumption at minimal performance cost of around 3 %.

We can estimate the extraction time for brute force approach as $t_b = E \cdot K$ where E is the number of edges and K the extraction complexity. The designed octree-traversal algorithm yields $t_t = P \cdot E \cdot K + T$ where T is the traversal cost and P is the ratio of potentially silhouette edges, which can be seen in the second last column of Tables 7.1 – 7.3. Based on the build test results, we estimated the P to be 0.2, 0.1 and 0.05 for octree levels 3, 4 and 5. T was measured to be 0.075 ms in average.

7.7 Improvements

I further improved the method after being published. The initial version was focused on two compression schemes – using 8 and 64-bit bitmasks that were propagated only between two levels of the octree. I tried to implement the method more according to the original design – to have a bitmask (8-bit) on all levels of the octree and to make the traversal much faster.

7.7.1 Octree Build

The build process now constructs the octree in a top-bottom fashion in two passes. The idea is that when an edge traverses octree, it is initially marked as PE until the spatial voxelization is fine enough that the edge is tested to be silhouette at certain level or the lowest level is reached. During the first pass, at level i and current voxel in the traversal path V_i, an edge is tested against the children $V_{i+1,x}$, $(x \in 1, 8)$ of V_i in the level $i+1$ where each of the children reports the test status as being PE or SE with multiplicity M, each identified with a bitmask indicating the children that share the same test result. When a child is tested as PE, that voxel $V_{i+1,x}$ is placed on the stack for further processing if level i is less than $maxLevel - 1$ or stored to either V_i if the bitmask has more than one bit set or stored to a child $V_{i+1,x}$ under its full bitmask. All sets of children that are SE and have the same multiplicity M for the tested edge are processed an stored based on the bitmask result, similarly to PE. When the first pass is complete, all SE are processed and stored in the respective voxels, PE are stored in the lowest two levels.

The second pass propagates PE up the hierarchy from $maxLevel - 1$ upwards. First, on every level, each voxel's edge IDs are sorted. Then, a set of all edges from all 8 siblings is constructed and edges from this set are tested against the edges of the sibling voxels. Based on the test result, the tested edge is stored either to the sibling's parent or to a particular sibling (if only a single sibling tested the edge positively).

OpenMP parallelization was utilized in order to speed up the build process. The first pass runs with one edge per thread, but a thread-safe set insertion had to be implemented to prevent data inconsistency. The second pass runs one thread per 8 siblings in a level and no mutual exclusion is required this time as groups of siblings are independent within a level.

I have observed that the build process produces a lot of bitmasks whose number of containing edges is rather small. The optional third pass would move the edges from bitmasks having less than a user-defined threshold of edges from parent to its children. This reduces the amount of bitmasks during the traversal process, but the performance benefit was 2–5 %.

7.7.2 Traversal

The most effort was put to optimize the traversal, which now requires only two kernels in total. The first kernel prepares a so-called *edge ranges* buffer which stores a starting offset to the edge buffer that stores all voxel edges linearly and the number of edges to be read from that particular position, e.g. the number of edges stored under a particular bitmask that starts at a certain offset in a large buffer. A flag stating if the edge range information regards to PE or SE is encoded to the number of edges. In order to distribute work more efficiently, the amount of edges under a single bitmask is divided into several ranges based on the workgroup size of the second kernel that processes the edges.

The edge ranges are processed in the second kernel. Each workgroup processes a single edge range; if a workgroup processes silhouette edge range, each thread reads its respective encoded edge and writes it to the output buffer. When a range of PE is processed, the thread reads the edge vertex data, computes the multiplicity of the edge. If the multiplicity is non-zero, the multiplicity and the edge ID is encoded into a single unsigned integer and stored. The storage index is obtained by using a combination of local and global atomic operations – each subgroup of threads first increments a counter in its shared memory and then only a single thread increments the global counter, reducing global memory access. The value obtained from the global counter is the distributed using shuffle intrinsics.

This new approach writes considerably less data between the kernels compared to the original version of the algorithm, instead of writing all the edge IDs and multiplicities, only the edge ranges are written. There is also no need for computing the prefix scan from the edge sizes. These optimizations increased the performance of the method, as shown below.

7.7.3 Measurements

The new implementation was similarly evaluated in terms of octree size and traversal speed. The results of the octree size test are presented in the Table 7.5. The same models as in the Tables 7.1 – 7.3 were used. As can be seen, the size of the octree reduced by 21.14 % in average compared to the original implementation, peaking at 24.74 % for the smallest scene. The efficiency slightly increases with octree depth but differs only slightly with the changing scale of the octree volume.

The main focus was put to improve traversal. I compared the the method again against a brute force compute shader method. It used the same output format from the compute stage (linear array of encoded edge ID and multiplicity in one 32-bit integer) as the proposed implementation. The brute force algorithm consists from a single compute shader that goes through an array of extracted edges and tests each of them for silhouettness. This algorithm too underwent optimizations – first of all, the multiplicity computation changed slightly – instead of computing the triangle plane in the shader (see Algorithm 10), we precompute these triangle planes and store them instead of the opposite vertices, saving on the amount of instructions during the multiplicity testing. The same optimization was used in the hierarchical implementation. We chose popular test scenes similar to the previous evaluation – Šibenik, Villa, Conference, Sponza, Epic Citadel and Buddha. The amount of triangles and edges of these test scene can be seen in the Table 7.6. The traversal test results can be seen in the Figure 7.9. All scenes were tested with octree maximum depth of 5 (depth of 0 is the top level).

It can be observed that both methods benefit from the optimizations. Compared to the graph in the Figure 7.8, new approach is now able to outperform brute-force method even on smaller scenes. The brute-force approach improved its performance by about 17 % compared to the previous evaluation. Hierarchical approach gained around 20 % on the Buddha scene. Considering the opposite side of spectrum, the scenes having the least amount of edges (Šibenik and Villa), both methods reduced the silhouette extraction time by approximately 70 %.

		Šibenik			**Conference**			**Sponza**		
Octree Depth	**Scale**	**Old**	**New**	**Diff (%)**	**Old**	**New**	**Diff (%)**	**Old**	**New**	**Diff (%)**
3	1	16	13	18.75	25	20	20.00	60	49	18.33
	2	18	14	22.22	29	24	17.24	63	52	17.46
	4	18	14	22.22	30	24	20.00	65	53	18.46
	8	18	14	22.22	31	25	19.35	65	53	18.46
	16	18	14	22.22	31	25	19.35	66	53	19.70
4	1	77	59	23.38	121	98	19.01	289	231	20.07
	2	84	65	22.62	139	112	19.42	302	241	20.20
	4	86	66	23.26	144	115	20.14	306	244	20.26
	8	86	66	23.26	145	116	20.00	306	245	19.93
	16	86	66	23.26	146	116	20.55	307	246	19.87
5	1	341	258	24.34	581	457	21.34	1359	1059	22.08
	2	376	284	24.47	668	525	21.41	1425	1110	22.11
	4	385	290	24.68	687	541	21.25	1438	1119	22.18
	8	388	292	24.74	694	545	21.47	1441	1123	22.07
	16	388	292	24.74	694	546	21.33	1443	1124	22.11

Table 7.5: Comparison of the original and improved implementation in terms of memory consumption on 3 testing scenes. The “Old” column refers to the previous 8-bit compression scheme, “New” refers to the improved implementation. The “Diff” column shows the size difference between the two implementations with the original approach being 100 %. The average size decrease in size is 21.14 %.

	Nof Triangles	**Nof Edges**
Šibenik	75 283	117 342
Villa	88 870	136 663
Conference	124 619	195 019
Sponza	279 163	431 246
Epic Citadel	613 567	921 555
Buddha	1 087 476	1 630 522

Table 7.6: Parameters of the test scenes used during traversal evaluation.

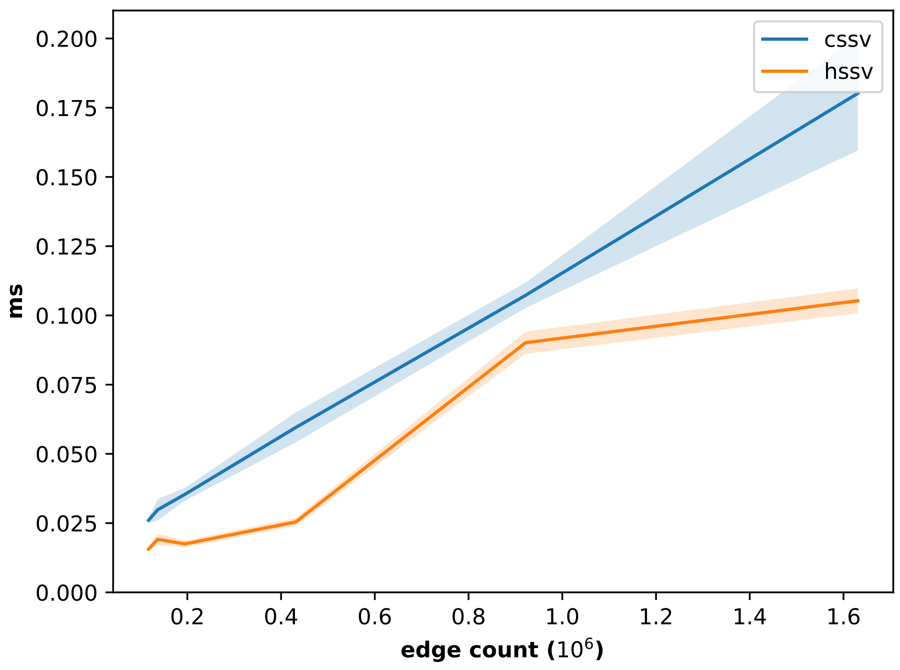


Figure 7.9: Average silhouette extraction times and mean absolute deviations for a set of popular test scenes (Šibenik, Villa, Conference, Sponza, Epic Citadel and Buddha) with up to 1.6 million edges. CSSV is the brute-force extraction method, HSSV new optimized hierarchical implementation.

7.8 Evaluation

At first, the hierarchical method was implemented having two compression schemes using 8-bit or 64-bit bitmasks, but worked only between two levels of the octree. Although the 64-bit scheme produced smaller octrees, the traversal was costlier as the method produced a lot more bitmasks – tens of thousands compared to theoretical maximum of 247 for 8-bit scheme (256 minus zero and all 1-bit numbers). The resulting algorithm is less sensitive to the number of edges and its absolute deviation is less than the brute-force compute shader method.

The improved implementation of the hierarchical method compresses all levels of the octree using 8-bit bitmasks. The traversal was greatly optimized by removing the prefix scan and greatly reducing the amount of memory written and read during the process. The amount of kernels was reduced to just two. Both hierarchical and brute-force methods were optimized by precomputing triangle planes and storing them instead of the opposite vertices and using encoded multiplicity with edge ID to reduce the amount of data written in the output.

The method, however, carries several disadvantages as well. The biggest one is memory consumption as the size of the octree can go to gigabytes, especially when the octree depth is 4 or 5, depending on the tested scene. Although the improved implementation reduced the amount of memory by about 21 %, its size still exceeds practical expectations. Besides, the method cannot handle elastic or deforming geometry as this would require the acceleration structure to recompute, which cannot be achieved in real time.

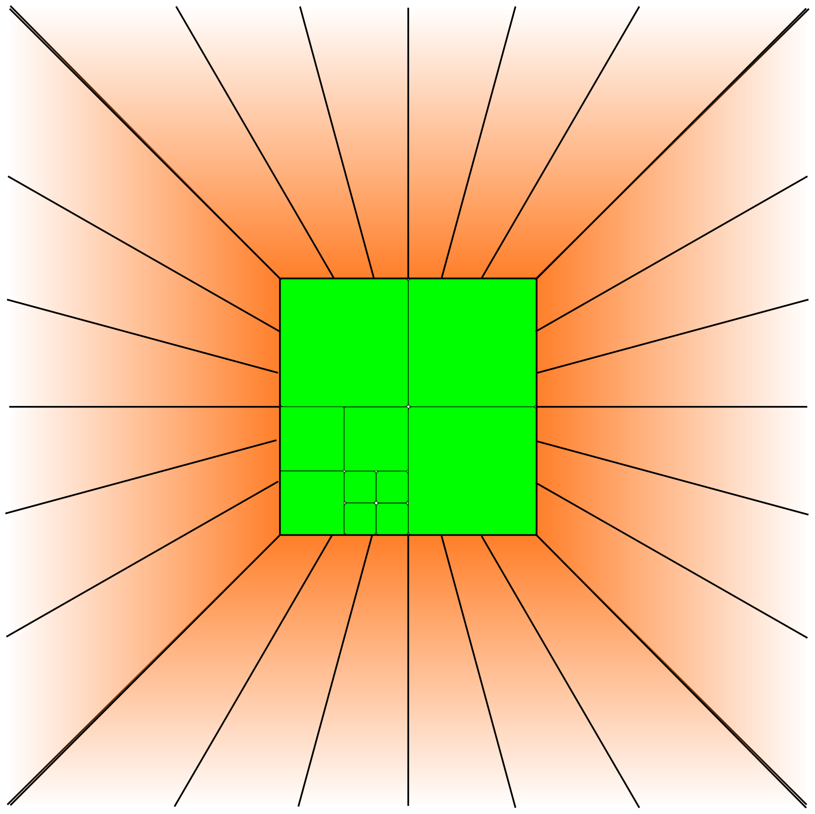


Figure 7.10: Instead of using one hierarchical structure, the algorithm would use two — one for the close vicinity of the model and one for all the other space around. The green part shows the hierarchical structure as presented, the orange part is the second hierarchical structure that uses spatial angles instead of voxels.

Future research could also evaluate usage of homogeneous coordinates, which may create hierarchy with unlimited spatial span based on spatial angles, see Figure 7.10. This would remove the limitation on the space where a light can be positioned. Directional lights could be used with this hierarchy as well - the cells on the bottom level would represent specific spatial angle of possible incoming light direction.

Chapter 8

Comparison of Omnidirectional Precise Shadow Methods

Several interesting shadowing techniques rendering shadows from omni-directional light sources have emerged recently. We categorized these methods into 4 categories – stencil shadow volumes, methods using traversal of an acceleration structure from view samples or scene geometry, methods based on Irregular Z-Buffer and ray tracing. Recent development of graphics hardware has brought hardware support for ray tracing into consumer graphics hardware, at the time of writing by nVidia as its RTX API available to DirectX 12 and Vulkan as extensions. We further optimized some of the tested methods in terms of robustness and performance. All methods are evaluated on several popular test scenes and in different resolutions. The whole testing framework, including source codes of the tested algorithms, have been made publicly available.

These findings have been recently accepted to the Computer Graphics Forum. I am the author of the text, implemented Omnidirectional Frustum-Traced Shadows, Deep Partitioned Shadow Volumes and coordinated the testing process.

8.1 Selected Methods

We categorized the omnidirectional shadowing techniques into several categories – stencil shadows, methods using acceleration structures built from view samples, acceleration structures built from scene geometry, methods using irregular Z-buffer (IZB) [53] and ray tracing. We tested and evaluated several methods representing each category.

Stencil shadows are represented by an implementation in the compute shader (CSSV) based on the algorithm proposed by Milet [81].

The second category, acceleration structures built from the view samples, will be represented by the Per-Triangle Shadow Volumes (PTSV [99]) and Clustered PTSV (CPTSV [98]).

We chose Deep Partitioned Shadow Volumes from the category of methods building acceleration structures from scene geometry as its shader sources are publicly available and the scale of the evaluated models should suit the method.

Although IZB-based methods are not explicitly omnidirectional, we have implemented the Frustum-Traced Shadows[113] using omnidirectional parameterization, similar to omnidirectional shadow mapping (Omnidirectional Frustum-Traced Shadows, OFTS).

As we used a single point light source in our test scenes, we implemented a hardware-accelerated ray tracer casting one shadow ray per fragment using NVIDIA RTX. The source codes for both testing programs are freely available on GitHub [1] as a reference for other researchers and as a public benchmark.

All methods except for the ray tracing were implemented in a multi-platform framework using OpenGL 4.5 core. Ray tracing was implemented in Vulkan using `VK_NV_ray_tracing` extension. Both test programs use the same mechanics – we utilized a deferred pipeline to first render the G-buffer (position, depth, normal, color, triangle ID) which serves as the input for the shadowing method. As all these methods use different ways of applying shadows to the scene (stencil mask, acceleration structure traversal, shadow map), we decided to unify the tested algorithms in terms of output. Each method is supposed to fill a *shadow mask* texture that is then used in the final rendering pass to apply shadows to the view samples.

The following subsections describe our implementations for all these methods in more detail.

8.2 Implementation and Optimizations

We based our stencil shadow volumes on z-fail using techniques from [81] and [62], as it addresses the robustness issues of shadow volumes. The connectivity information is extracted on the CPU and without loss of generality, we set the maximum possible multiplicity to 2 in our tests – edges having more than 2 adjacent triangles were split into several instances. The compute shader then tests every edge for silhouetteness. In order to minimize the amount of data written to the global memory, the compute shader only outputs one encoded integer per silhouette edge consisting of edge ID and its multiplicity. Then, the geometry shader receives this encoded information as a vertex attribute and casts the edge side as many times as its multiplicity with correct triangle winding based on the multiplicity sign. To speed up the computation process even more, we replace storing opposite vertices with the edges with pre-computed triangle planes that would otherwise have to be calculated every time in the silhouette testing shader from the edge vertices and from the opposite vertex. The shadow volume caps are rendered similarly using the geometry shader, we compute multiplicity from the triangle and the light source, the sign of the multiplicity determining the winding of the triangle.

8.2.1 PTSV and CPTSV

Both PTSV and CPTSV aim to reduce the shadow volume rasterization times as it is the most demanding step. They do not use traditional rasterization of shadow volumes as the stencil methods, instead they hierarchically rasterize the shadow volumes into an acceleration structure. PTSV uses a hierarchical depth buffer and a shadow mask buffer. CPTSV uses a 3D tree of view sample clusters, as seen in Figure 8.1. All hierarchical structures have a branching factor equal to 32 (SIMD size) in the original design. Threads in each warp cooperate in rasterization of one shadow frustum.

We re-implemented both methods in OpenGL according to the original papers and available source codes. The authors provided us with the implementation of PTSV in CUDA. We modified the implementation to run on different hardware (AMD) and to support different

[1]https://github.com/dormon/Shadows, https://github.com/neithy/NeiGinPublic

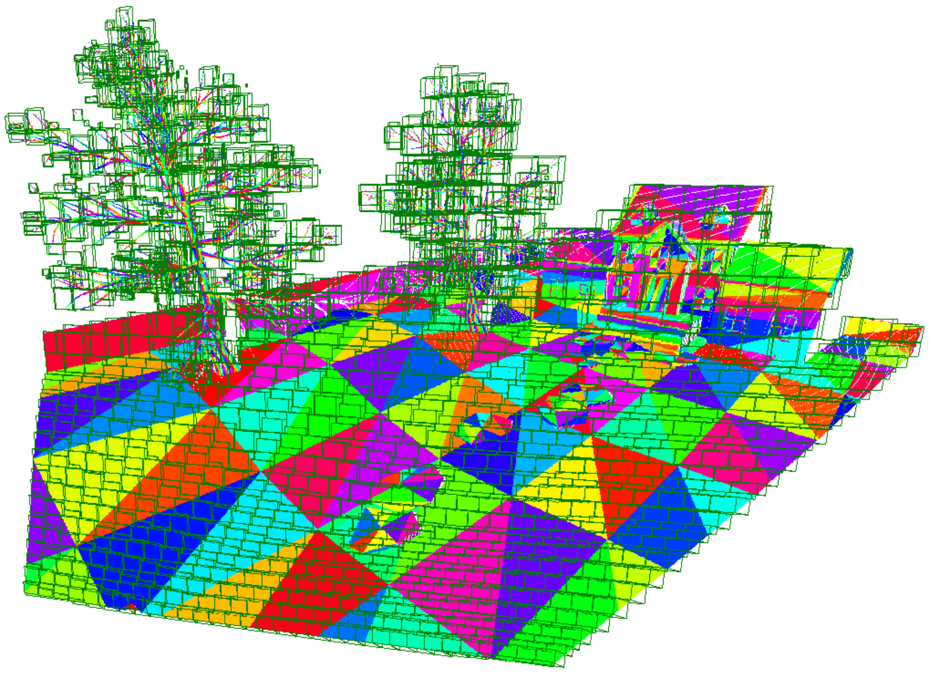


Figure 8.1: The image shows view-samples (colored triangles) and clusters (green boxes) on one level of the CPTSV's hierarchy.

resolutions and fixed some visual artifacts. We tested different memory storage types (textures and buffers) for intermediate results. In the end, we chose textures as they performed slightly better.

CPTSV was implemented from scratch with all the extensions mentioned in the original paper, except for load balancing. We implemented the traversal shader using a small stack since template recursion is not possible in OpenGL.

We further optimized the construction of upper parts of the 3D cluster tree by storing the IDs of active nodes of the previous level and launching only the appropriate number of threads. We also tried to minimize the number of operations required to compute Morton codes for x, y, and z components with different bit lengths. Furthermore, we optimized the traversal step of the algorithm by reducing the number of registers required for the stack using local memory.

According to our measurements, we are convinced that our implementation is on a par with, or faster, than the original implementation.

8.2.2 Deep Partitioned Shadow Volumes

The shaders for DPSV [83] have been made publicly available by the authors; we provided all the necessary inputs and outputs for the method. We implemented a deterministic shadow plane construction based on [81], which helped with blinking artifacts we encountered with this method. After testing all 3 variants of the method's TOP tree traversal (stack, stackless, hybrid), we opted for the hybrid variant, combining both approaches, as it was the fastest of all 3 versions.

8.2.3 Omnidirectional Frustum-Traced Shadows

This method was implemented using all optimizations mentioned in the original paper[113] (tight-fitting projection, discarding back-facing view samples, removal of already shadowed view samples, depth buffer initialization) including the reprojection of areas with very long lists using off-center projection [103]. Our implementation works similarly to omnidirectional shadow mapping, utilizing 6 light frusta to cover all directions. We cull those frusta in two stages – first on the CPU by computing collisions of light frusta with camera frustum and producing a bit mask, which speeds up the heat map construction. The second frustum culling occurs when computing new projection matrices, as some of the visible light frusta may not contain any view samples. The whole method runs in a single pass using compute shaders to compute the heat map (list lengths per light-space texel), new projection matrices and IZB; the depth buffer optimization and the IZB traversal utilize geometry shader and layered rendering to draw to several textures simultaneously. The IZB traversal pass requires conservative rasterization; we use NVIDIA's OpenGL extension `GL_CONSERVATIVE_RASTERIZATION_NV`.

8.2.4 Ray Tracing

Shadow rendering by ray tracing was implemented separately in a Vulkan test program using NVIDIA's `VK_NV_ray_tracing` extension. Similarly to other implemented methods, it used geometry information provided by the G-buffer to create a per-fragment shadow mask. A simple ray generation shader casts a ray from the fragment position towards the light and the corresponding miss shader marks the fragment as lit. As a precise triangle intersection is not required, we enabled an optimization in the form of `TerminateOnFirstHit` flag, i.e., terminating BVH traversal upon hitting any triangle towards the light source.

To use the ray tracing extension, a two-level bounding volume hierarchy is required. The bottom level consists of elements of the geometry, and the top level is built from bounding volumes of object instances. In the case of static geometry, the acceleration structure is created once and never updated. For a scene with moving objects, only the top level needs to be updated with transformations. The bottom level structure needs to be updated only when the geometry changes (elastic simulation, skinning, etc.). In order to simulate several possible scenarios, we included measurements without BVH updates as the best-case scenario and a full per-frame BVH rebuild as the worst case. Building is done on the GPU; therefore, all memory needs to be allocated in advance. The API provides an upper estimate of the required memory amount based on the provided geometry, but the final size of the BVH is usually around 50 % of the estimate.

8.3 Measurements

The measurements were carried out on a set of popular scenes; details can be seen in Table 8.1. Although "Villa" is a small scene compared to modern standards, it was used in [98], and we will demonstrate that this method was designed specifically for scenes of this type - high depth complexity and falls behind on other scene types. Even "Hairball", with its 2.8 million triangles, poses a challenge for the tested algorithms. All scenes were tested with a single point light source. The "Buddha" and "Hairball" scenes were slightly modified – we positioned both models on a plane, acting as a shadow receiver. Each scene was tested using a camera flythrough that took 1 000 frames. Every frame was rendered 5

Table 8.1: Test scenes used for method evaluation.

Scene	Triangle Count	Edge Count
Villa	88 870	136 663
Conference	124 619	195 019
Sponza	279 163	431 246
Closed Citadel	613 567	921 555
Buddha	1 087 476	1 630 522
Hairball	2 880 002	4 290 005

Table 8.2: Memory consumption of all tested algorithms on all scenes. The sizes for the ray tracer are only for the BVH structure. As we only cast one secondary ray per fragment, they should fit into the GPU registers. The memory footprint of Sintorn's methods (PTSV, CPTSV) depends on the resolution and other factors, so the table only shows the size of shadow frusta buffers. All sizes are in MB.

Scene	RTX	CSSV	DPSV	PTSV	CPTSV
Villa	5.63	7.82	10.85	8.13	9.49
Conference	7.88	11.16	15.21	17.85	20.83
Sponza	17.50	24.68	34.08	39.48	46.06
Citadel	38.56	52.73	74.90	84.37	98.43
Buddha	68.40	93.30	132.75	149.27	174.16
Hairball	181.02	245.48	351.56	392.76	458.22

times and the average time of the shadow mask creation was written to a `.csv` log file. The tests are focused on resolutions of 1920×1080 and 3840×2160, but we have also tested on other resolutions, from $1K \times 1K$ to $4K \times 4K$, when evaluating the resolution dependency. The memory consumption of all methods is analyzed as well.

All tests were carried out on an AMD ThreadRipper 1920X system with 32 GB of RAM and a GeForce RTX 2080 Ti graphics card with hardware ray tracing support. Some of the test were run on a GeForce GTX 1080Ti that supports RTX API in fallback mode using compute shaders. The system runs on Windows 10, and both test applications were compiled using Visual Studio 2019.

8.3.1 Memory Consumption

We measured the amount of memory each method requires for its acceleration structures; the results can be seen in Table 8.2. The memory footprint of OFTS is not affected by the scene geometry, as it depends only on the method parameters and the screen resolution. We used two sets of parameters based on profiling – lower resolutions (up to 1920×1080) used 1024 by 1024 for one slice of the IZB's head texture, and higher resolutions used 2048×2048. The reason for this was to cope with the performance drops caused by insufficient reprojection of the densest areas of the heat map. The heat map had a resolution of 512×512 for all screen resolutions. This resulted in memory consumption of up to 98 MB for lower resolutions (up to 1920×1080), and up to 403 MB at the resolution of 4000×4000. The reason for such a high amount of memory at the higher resolutions is that the IZB's

Table 8.3: Average shadow mask creation times across all the test scenes at 1920×1080. The bold values represent the fastest algorithm (except ray tracing). "RTX_AVG" represents an average of "RTX" and "RTX rebuild". The "AVG" column contains the average shadow mask creation time across all frames on all scenes. All values are in milliseconds.

	Villa	**Conf.**	**Sponza**	**Citad.**	**Buddha**	**Hairb.**	**AVG**
CPTSV	**1.30**	8.51	3.08	4.93	7.04	33.55	9.74
CSSV	2.23	**0.68**	**1.57**	4.15	**2.24**	29.90	6.80
DPSV	1.87	3.11	2.91	5.16	8.47	120.88	23.74
OFTS	2.05	2.05	3.70	**3.30**	5.22	**14.32**	5.08
PTSV	12.73	12.73	10.25	10.66	352.64	97.90	81.42
RTX rebuild	1.18	1.47	2.33	4.33	7.23	16.50	5.50
RTX	0.24	0.28	0.37	0.26	0.08	0.54	0.29
RTX AVG	0.71	0.88	1.35	2.30	3.66	8.52	2.90

head texture has 12 layers (6 sides with reprojection), the same for the depth texture for the z-buffer optimization pass. At lower resolutions, the benefit of being independent on the scene geometry prevails on larger models; however, at higher resolutions (3840 × 2160 and more), the method is the second most demanding, even at the "Hairball" scene.

From the rest of the tested methods, of which the memory footprint is dependent on the amount of scene geometry, ray tracing reports the lowest amount of video RAM (based on the upper estimate reported by the RTX API), followed by CSSV. The stencil method only needs one buffer to store the edge information and another to write the resulting encoded multiplicity and the edge ID. DPSV stores the TOP tree nodes in a single linear buffer and its memory consumption is around 40 % higher than CSSV.

The memory requirements of PTSV and CPTSV depend on two factors: the number of triangles and the resolution. Both algorithms need to allocate a shadow frusta buffer, the size of which can be seen in Table 8.2.

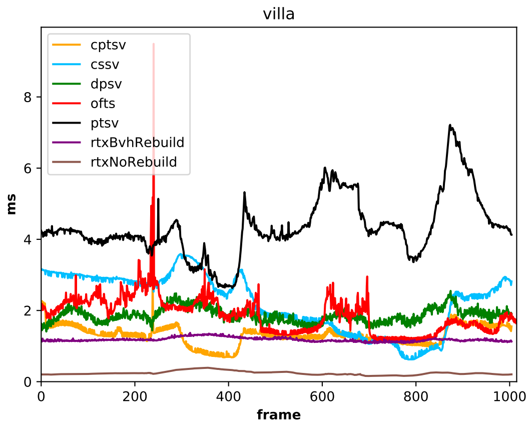
Apart from the shadow frusta buffer, PTSV uses two acceleration structures – one hierarchically stores the depth ranges per tile, the other contains the actual shadowed/lit information. The depth range is represented by two float values; the number of tiles on every level can be expressed by Equation (8.1), where T is the number of tiles, R the resolution, N the number of levels, B the branching factor (workgroup size) and i the index of the level. The size of these hierarchical structures is 5 MB at 4K resolution.

$$T = \frac{R}{b^{N-i}} \tag{8.1}$$

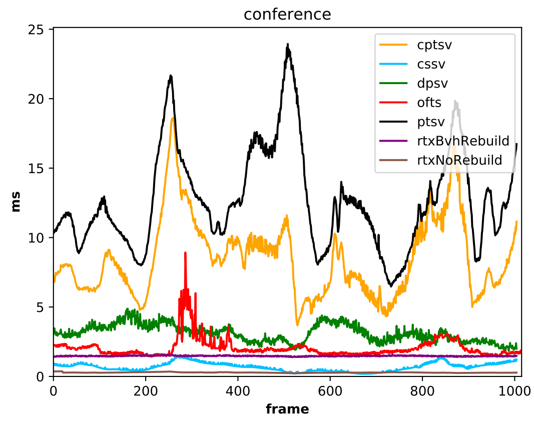
The memory footprint of CPTSV's hierarchical structures is much larger than PTSV. The actual size of the structures depends greatly on resolution, branching factor, and z' bits in the cluster key; the amount of memory can go up to 6 GB at $4K \times 4K$ (buffer containing the AABBs of the clusters). We have also implemented the memory reduction scheme as mentioned in the original paper, reducing the amount of memory approximately 6.5-times at a cost of about 5 % of the performance.

8.3.2 Evaluation at 1920×1080

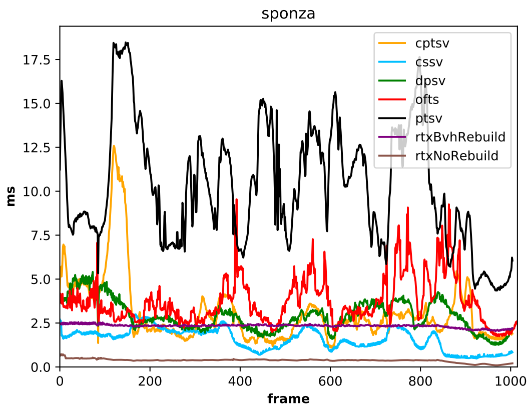
The results of all methods and scenes at 1920 × 1080 can be seen in Figure 8.2 and the average shadow mask creation times in Table 8.3. PTSV was removed from the "Buddha" graph (Figure 8.2e) for clarity, as its average performance was 352 ms.



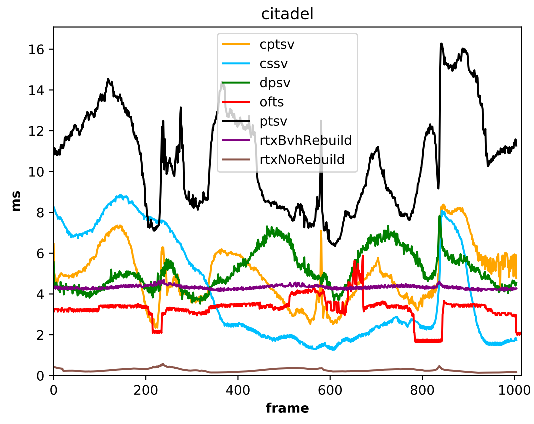
(a)



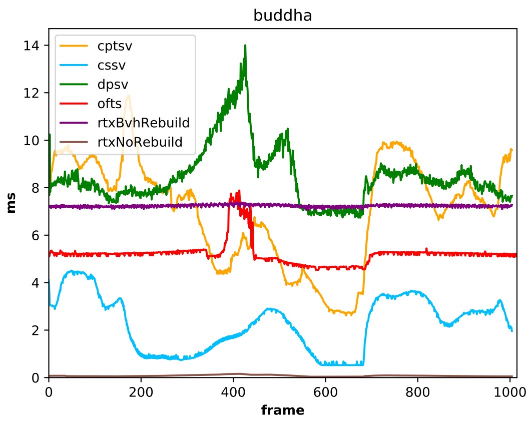
(b)



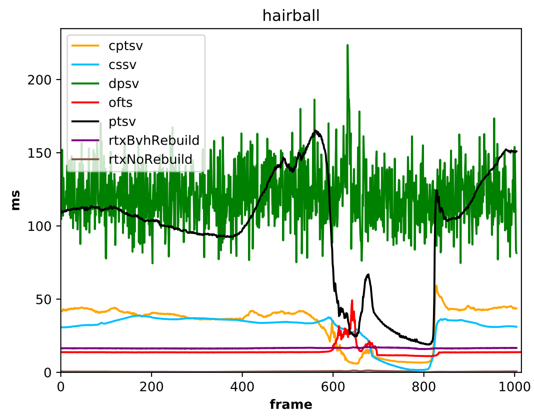
(c)



(d)



(e)



(f)

Figure 8.2: Comparison of all the methods using flythroughs on various test scenes, resolution 1920 × 1080, sorted by the amount of triangles. Each graph represents one tested scene with results from methods tested. Ray tracing is measured twice – without any modification to the BVH (as "rtxNoRebuild") being the best possible scenario and with full BVH rebuilt every frame ("rtxBvhRebuild") for the worst case. The "Buddha" scene does not include the PTSV method, as its average performance was 352 ms and was removed for clarity.

Although RTX without BVH rebuild is the fastest method of all, it is also an ideal condition that would probably not be achieved in a real scenario. “RTX AVG” as an average is probably closer to a practical case, but even then it was the fastest in the most cases. Ray tracing was also the most stable method tested.

CSSV’s average time was spoiled by the “Hairball” scene, where the silhouette is not simple – the model is comprised chiefly of thin geometry, producing a complex silhouette that generates a lot of shadow volumes requiring rasterization. Otherwise, this method would be second to ray tracing in this scenario, even surpassing the “RTX AVG” time twice. Compared to OFTS, stencil shadow volumes perform better on enclosed scenes (“Conference”, “Sponza”) and scenes with a relatively simple silhouette (“Buddha”, as well as “Conference”, has a relatively simple silhouette). Tree branches on “Villa” cast shadows that cause a high fill rate, which leads to lower performance of the stencil method compared to other techniques.

PTSV is the slowest method on most of the scenes. The “Buddha” scene seems to pose a non-trivial problem for PTSV as the model contains a high number of small triangles positioned mostly in the middle of the viewport. This triangle distribution causes an imbalanced GPU load when traversing the acceleration structure of the view samples, as only a handful of the view samples are affected by the vast majority of the scene geometry, causing the method to perform very poorly in this test case.

CPTSV excels on the “Villa”, as the scene was specifically designed for this method.Both PTSV and CPTSV perform unusually on the “Conference” scene, where the average time is slower than on “Buddha”, despite having just 11 % of its geometry. This is probably caused by missing load-balancing optimization, as there are larger triangles in the scene (table, floor, etc.). A single triangle is processed by one warp, which causes improper load balancing when the triangle covers a large portion of the screen, as a lot of nodes need to be processed. Apart from these two cases, the method can be considered an average one – not the fastest, not the slowest.

DPSV randomly builds its TOP tree every frame, meaning that the quality of the acceleration structure differs from frame to frame. It can be observed as fluctuations mostly on the “Hairball” scene. The complex and concentrated geometry in this scene poses a challenge for the build phase of the algorithm, resulting in huge variation of the frame times. The method was faster than CPTSV up to “Sponza”, but does not scale as well as other methods with the increasing amount of geometry.

OFTS had to be tuned for smaller (up to 1920×1080) and larger resolutions separately, as the parameters greatly affected sudden performance drops caused by the reprojection area being too large and the most exposed lists did not get properly redistributed. This happens most often when the camera is very close to the geometry or when the reprojection itself cannot benefit from the shape of the reprojection area (e.g., when the longest lists are in opposite corners of the heat map). Such spikes can be seen on the “Villa” scene. Its performance also depends on the number of active light frusta, which can be seen e.g. on the “Sponza” scene where the light is positioned in a way so that all 6 frusta are facing some of the scene’s geometry, thus there are always multiple frusta active during the flythrough; compared to e.g. “Buddha”, where most of the scene geometry will be concentrated in a single frustum. OFTS can handle complex geometry, like “Hairball”, better than other conventional methods. This method suffers from the same problem as omnidirectional shadow mapping – seams between the cubemap faces, exhibiting as an occasional line of lit fragments.

Table 8.4: Average shadow mask creation times across all the test scenes at 3840 × 2160. The table description is identical to Table 8.3.

	Villa	**Conf.**	**Sponza**	**Citad.**	**Buddha**	**Hairb.**	**AVG**
CPTSV	**2.16**	18.51	7.52	12.61	5.96	29.95	12.79
CSSV	5.42	**1.74**	**4.27**	9.68	**3.97**	64.29	14.90
DPSV	4.95	8.52	7.01	10.62	10.61	129.97	28.61
OFTS	4.25	4.78	8.18	**4.43**	6.21	**17.52**	7.57
PTSV	5.40	41.62	23.52	13.19	316.87	91.77	82.06
RTX rebuild	1.85	2.23	3.36	5.0	7.31	17.66	6.24
RTX	0.90	1.06	1.40	0.99	0.25	1.93	1.09
RTX AVG	1.38	1.65	2.38	3.0	3.78	9.80	3.67

8.3.3 Evaluation at 3840×2160

The results of the 4K flythroughs are presented in Figure 8.3 and Table 8.4; the results of the "Buddha" scene in Figure 8.3e are again missing the PTSV graph, as the method performed very slowly – 316 ms on average. We were surprised by the results of the stencil method on the 4K resolution, as we estimated that the rasterization of the shadow volumes geometry would cause CSSV to perform as one of the slowest, but the results seem to follow a similar trend as at 1920 × 1080. In terms of the average across all scenes, OFTS was again second-fastest to ray tracing, followed by CSSV (mainly because of poor performance on the "Hairball" scene) and CPTSV . Interestingly, the PTSV was able to outperform CPTSV on the "Hairball" scene compared to the full-HD test; the reasons will be disclosed below. OFTS had to be tuned for higher resolution, as we often experienced spikes in frame times; for example, there were 270 ms spikes on both "Sponza" and "Hairball". We had to adjust the resolution and reprojection threshold to cope with them, but they are still visible. Ray tracing is again the fastest solution.

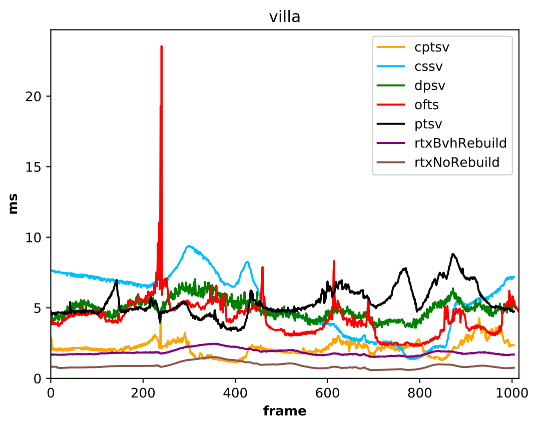
8.3.4 Frame Time Decomposition

The timings of the particular components of each method can be seen in Figure 8.4, measured on the "Sponza" scene at 1920 × 1080 resolution. Sponza was chosen because the omni directional light source can demonstrated be very well in its enclosed atrium.

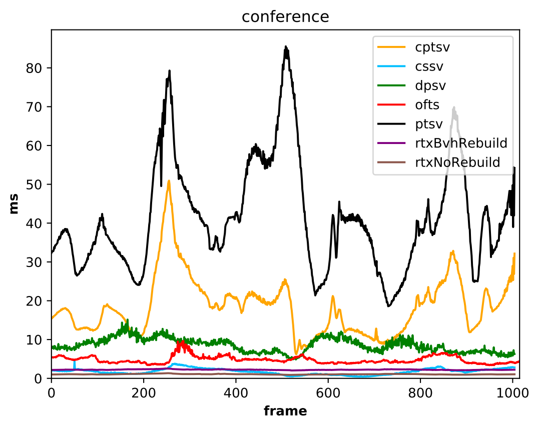
CSSV can be broken down into two parts – silhouette computation and shadow volume rasterization. We used the *z-fail*, thus we rendered both the front and back caps for each shadow volume. The silhouette extraction takes only around 0.05 ms ("compute"). The extruded sides take the longest to render, as they consume a lot of fillrate. Drawing caps takes on average 9.5 % of the total shadow computation time (0.15 ms in average).

The time to build the hierarchical tree from the view samples in CPTSV is significantly faster than in PTSV, 0.44 vs 3.17 ms on average. Wedge optimizations for faster tile culling sped up the rasterization of the shadow volumes against the hierarchical tree structure.

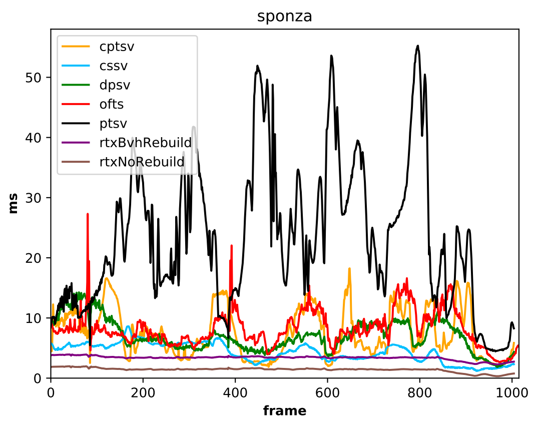
All stages of OFTS except traversal take about 1 ms combined. The traversal itself takes 2.72 ms on average on this scene, about 73 % of the shadow compute time. "Sponza" as an enclosed scene provides a good example, as the number of active frusta frequently changes in the course of the test as well as projected area in each frustum. Sudden spikes are caused by IZB lists being long.



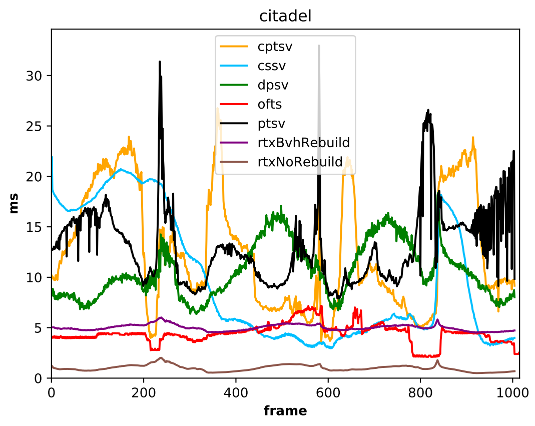
(a)



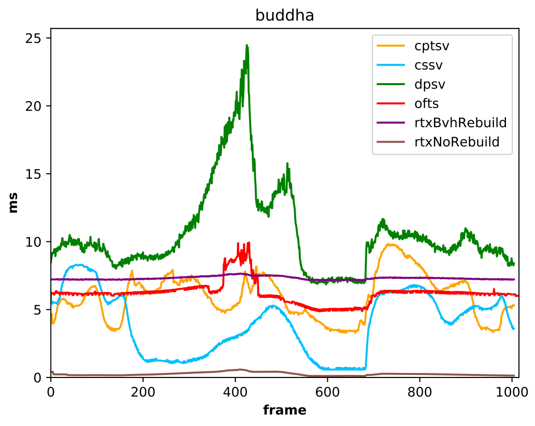
(b)



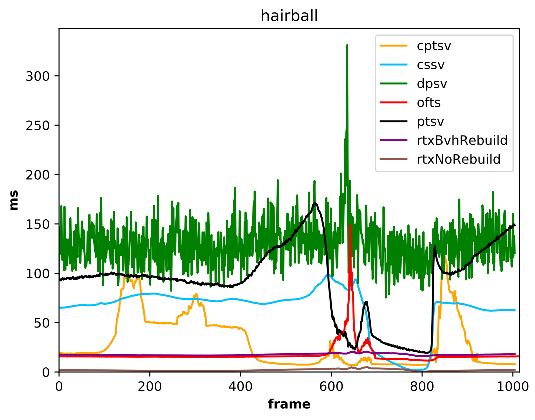
(c)



(d)

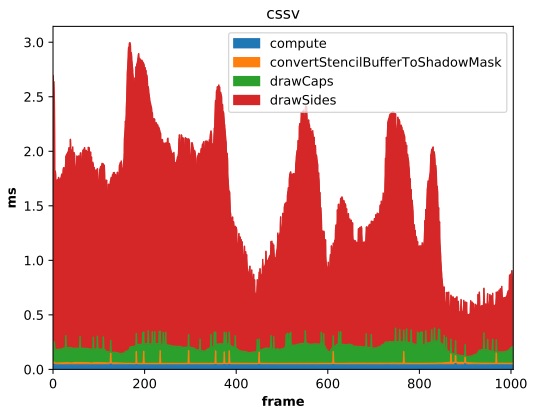


(e)

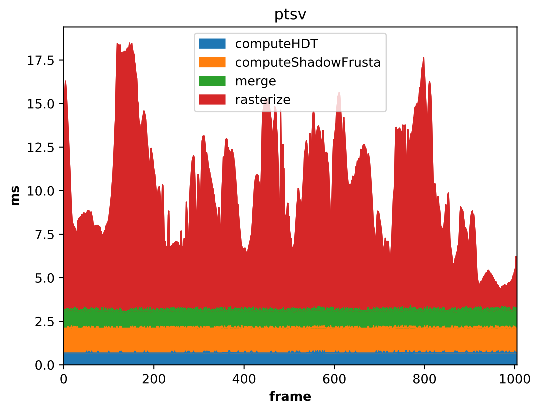


(f)

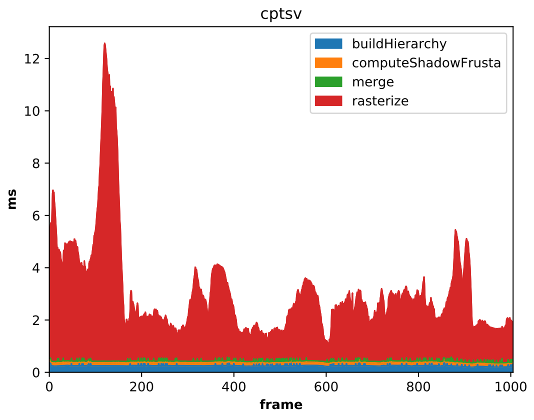
Figure 8.3: Comparison of all the methods using flythroughs on various test scenes, resolution 3840×2160. The methods are labeled the same way as in Figure 8.2. The test on the "Buddha" scene again does not include the PTSV method, as its average performance was 316 ms.



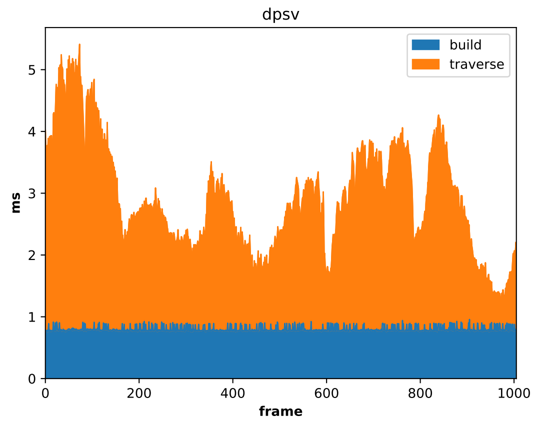
(a)



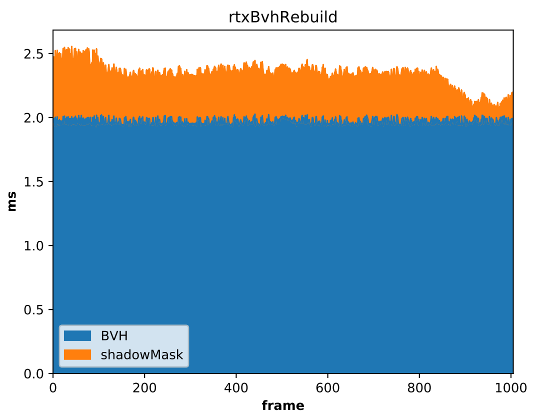
(b)



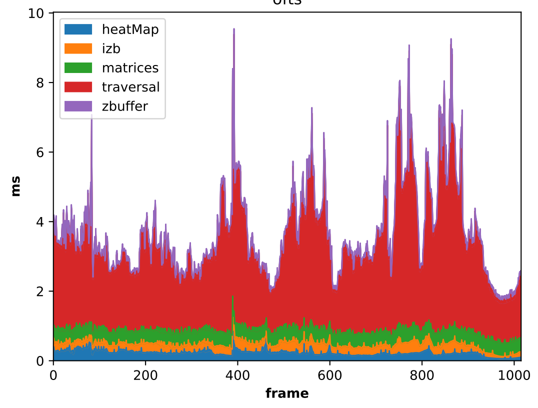
(c)



(d)



(e)



(f)

Figure 8.4: Frame time decomposition of all methods on the Sponza scene at 1920 × 1080. Ray tracing is represented by its worst-case scenario (full BVH rebuild every frame).

Table 8.5: Average shadow mask creation times across all the test scenes at 1920 × 1080 running on GeForce GTX 1080Ti. The table description is identical to Table 8.3.

	Villa	**Conf.**	**Sponza**	**Citad.**	**Buddha**	**Hairb.**	**AVG**
CPTSV	2.14	10.30	4.36	7.05	11.22	49.48	14.09
CSSV	2.31	**0.74**	**1.79**	4.51	**2.90**	32.01	7.38
DPSV	2.93	4.63	4.19	6.87	9.86	137.08	27.59
OFTS	**1.87**	2.22	3.97	**3.66**	5.94	**16.39**	5.67
RTX rebuild	3.62	3.22	5.12	7.26	8.53	26.28	9.01
RTX	2.39	1.65	2.74	2.45	0.42	7.66	2.89
RTX AVG	3.01	2.44	3.93	4.86	4.47	16.97	5.95

The ray tracer spent most of the frame time building the BVH structure; the tracing itself is only around 20 % of the total time. We have also found out that the initial BVH build takes up 10 ms more than all subsequent rebuilds, probably due to memory allocation. The traversal part is very fast, also because all the rays are coherent and converge to a single point.

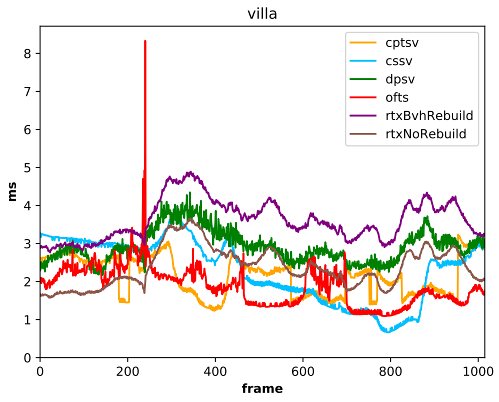
8.3.5 Evaluation on GeForce GTX 1080Ti

We repeated the measurements using GeForce GTX 1080Ti, which supports the RTX API in fallback mode, representing a well-optimized software ray tracing solution. The results can be seen in Figure 8.5 and Table 8.5. As PTSV was the slowest method of all, we excluded it from this measurement.

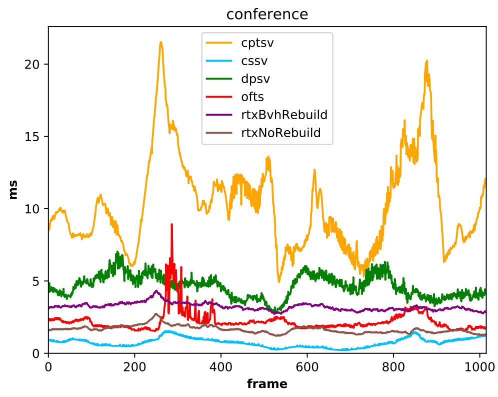
Ray tracing, on average, performs 2.1-times slower than on the RTX 2080Ti; rebuild 1.6-times and the traversal-only scenario 10-times. The acceleration structure traversal benefits mostly from the hardware acceleration. Compared to other methods on this platform, ray tracing without rebuild is not the fastest method until the Citadel scene. Although a combined average time of ray tracing was second to pure traversal on the 2080Ti, it was surpassed by OFTS on the 1080Ti. CSSV is also faster than RTX with rebuild on the legacy platform. Conventional methods were, on average, 15.5 % (8-30 %) slower than on the 2080Ti.

8.3.6 Dependency on Triangle Count

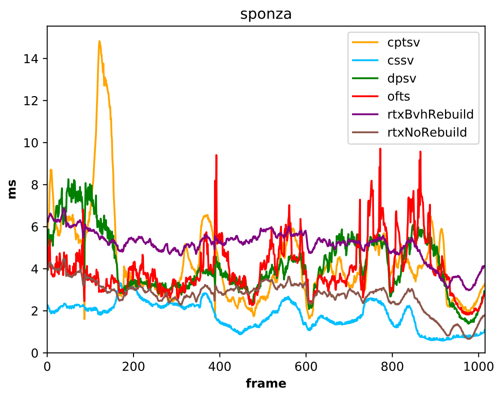
Figure 8.6 shows performance dependency on triangle count across all of the tested scenes and methods at 1920×1080. It was calculated as the average and mean absolute deviation from all the frame times of the flythrough on a particular scene. Due to PTSV's behavior on the "Buddha" scene (described above), the method was excluded from the graph. It also has the highest dependency on the triangle configuration of all the tested methods; its mean absolute deviation at 4000×4000 was 30 ms. It can be seen that ray tracing without rebuild does not put a lot of stress on the RT cores of the GPU; we are tracing 2 megarays at 1920 × 1080 shadow rays per frame, where the hardware is, in theory, capable of 8 gigarays per second. OFTS shares similar triangle dependency with ray tracing with full rebuild, having the lowest average and mean absolute deviation of the conventional methods. CSSV is something of a surprise, as the method needs to rasterize a lot of infinite shadow volume geometry. Although having lower triangle dependency than CPTSV in the tested scenarios, the tide would change for larger scenes as the average curves of both methods



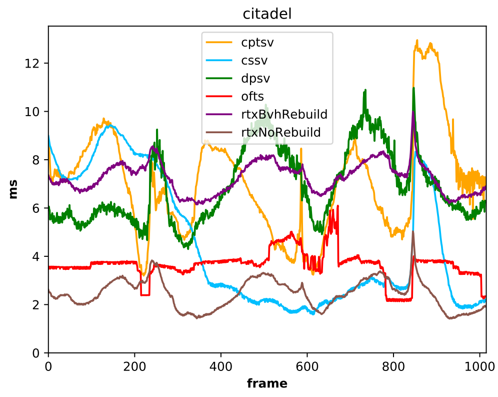
(a)



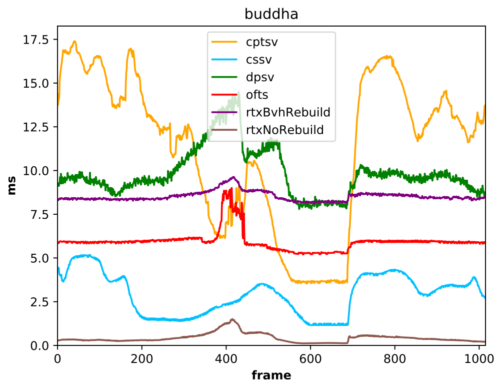
(b)



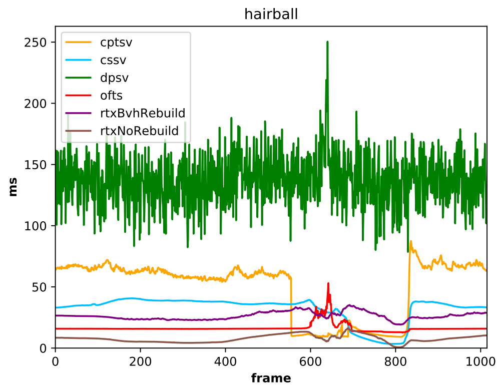
(c)



(d)



(e)



(f)

Figure 8.5: Evaluation at 1920 × 1080 using GeForce GTX 1080Ti, having only software support for ray-tracing. The methods are labeled the same way as in Figure 8.2. PTSV was excluded from the measurements for clarity.

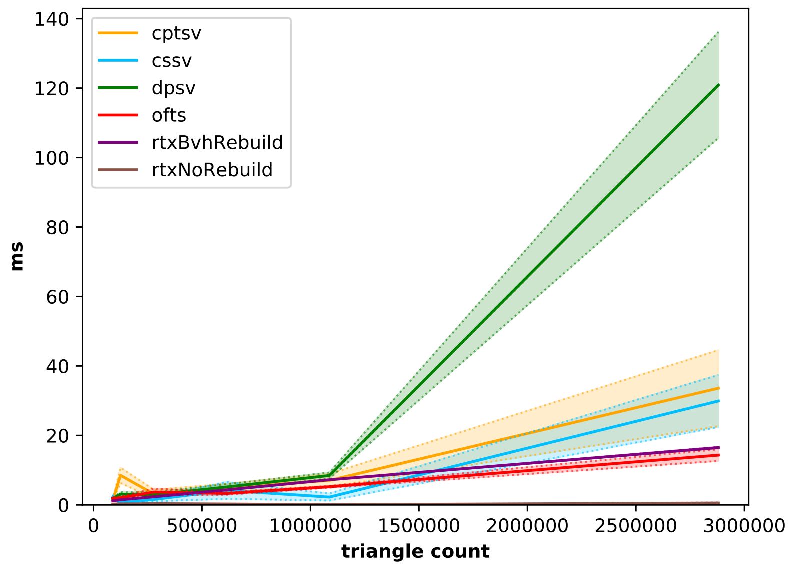


Figure 8.6: Dependency on the triangle count across all tested scenes at 1920 × 1080. PTSV was excluded for clarity.

converge. With the increasing number of triangles, DPSV was gradually outperformed by other methods, and its curve steep final segment is the result of the "Hairball" scene.

8.3.7 Dependency on Screen Resolution

We produced a graph in a similar fashion for dependency on the resolution as well; see 8.7.

CPTSV's average shadow computation time increased 7.5-times at 4096×4096 compared to $4K$ resolution despite the fact the acceleration structures have the same size in both cases. This occurred on every test scene. We found out that it is caused by incorrect view sample depth clustering. The majority of the view samples in all screen tiles fell into the same cluster even if their depth was different. The reason is the exponential division of the view frustum's depth. The amount of bits allocated for encoding depth using the Morton code depends, apart from other factors, on the vertical screen resolution and on the distance of the near clipping plane. To encode the depth at 4096 × 4096, the bit count would exceed the allocated 10 bits, causing the z-part of the Morton key to overflow and thus storing all view samples into the same furthest cluster. This resulted in a very long AABB of the cluster (along the z axis) which causes very slow traversal, as most of the clusters have to be visited by the majority of the shadow frusta. One of the possible fixes would be to allocate more bits (13 or more) for the depth, but in that case, a 32-bit integer would be insufficient and an arbitrary bit array would slow down the method. Using 64-bit keys would double the already high memory consumption. Another possibility would be to divide the frustum using a different scheme.

Provided CPTSV would not be affected by the problem mentioned above, we extrapolated a hypothetical average frame time between 3.8 – 5.1 ms (using quadratic regression and power curve) for $4K \times 4K$ resolution. In such a case, the method would have had one of the lowest resolution dependencies of all the tested methods.

Although PTSV and CPTSV have a lower resolution dependency thanks to the hierarchy they build, they have larger initial overhead and don't scale well with the increasing amount

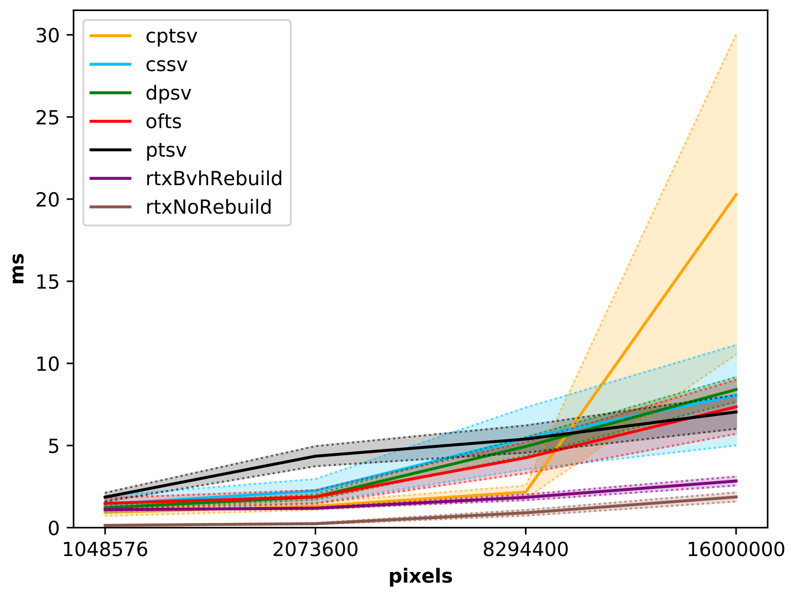


Figure 8.7: Computational times as they depend on the screen resolution on the "Villa" scene.

of triangles. As expected, CSSV is more sensitive to resolution than other methods; its curve is similar to OFTS but absolute deviation is the widest (except for $4k \times 4k$). DPSV follows CSSV in this scenario, but ends up slightly above. Hardware-accelerated ray tracing has the lowest sensitivity on all the test scenes.

8.4 Time Complexity

Although the authors of the original papers do not state the complexity of their algorithms, we tried to estimate the average time complexity based on increasing scene complexity, screen resolution and multiple light sources for every major part of the tested algorithms. These findings can be seen in Table 8.6. In terms of resolution, most of the algorithms scale linearly with the increasing amount of pixels except for CPTSV. When multiple lights are used, ray tracing seems again to be the best possible method as it does not require a BVH rebuild, unlike all the other evaluated methods.

8.5 Discussion and Conclusion

We were able to compare several modern omni directional shadowing methods with precise hard shadows. These methods were tested on several popular test scenes under multiple resolutions.

Hardware-accelerated ray tracing is the clear winner in terms of speed, implementation difficulty and even memory consumption in the most cases. It is clear that having dedicated hardware units can reduce the BVH traversal time by a factor of 10. Ray tracing (traversal only) on legacy hardware was able to outperform all the methods on scenes having more than 600 000 triangles. Ray tracing also has the lowest triangle and resolution dependency in our tests. Unlike all other tested algorithms, support for multiple light sources does not

Table 8.6: Time and memory complexities of the tested algorithms, broken down into stages. An empty cell means the stage of a particular algorithm is not dependent on the number of triangles or screen pixels.

Method	**Resolution**	**Geometry**	**Multiple lights**
CSSV (adjacency)	O(1)	O(N log(N))	O(1)
CSSV (silhouette)	O(1)	O(N)	O(N)
CSSV (rasterize)	O(N)	O(N)	O(N)
PTSV (shadow frusta)	O(1)	O(N)	O(N)
PTSV (depth stencil)	O(N)	O(1)	O(N)
PTSV (traversal)	O(N)	O(N)	O(N)
CPTSV (shadow frusta)	O(1)	O(N)	O(N)
CPTSV (view cluster hierarchy)	O(N)	O(1)	O(1)
CPTSV (traversal)	O(log(N))	O(N)	O(N)
OFTS (shadow frusta)	O(1)	O(N)	O(N)
OFTS (IZB build)	O(N)	O(1)	O(N)
OFTS (traversal)	O(N)	O(N)	O(N)
RTX (build)	O(1)	O(N log(N))	O(1)
RTX (trace)	O(N)	O(log(N))	O(N)
DPSV (build)	O(1)	O(N log(N))	O(N)
DPSV (traverse)	O(N)	O(log(N))	O(N)

require a BVH rebuild, as the structure can be reused, since it is independent on the light position. Transparent casters and sub-pixel precision are also supported.

Although the authors of Frustum-Traced Shadows did not design the method primarily for omnidirectional parametrization, the method works very well with this configuration. It has one of the lowest triangle dependencies, a predictable memory footprint and was able to handle the “Hairball” scene second best to ray tracing. Its implementation is straightforward and does not require any preprocessing. The downsides are higher memory consumption on higher resolutions, and performance drops due to long lists, which need to be addressed using parameters which are scene-dependent. We noticed a light-leaking artifact on the seams between the frusta, probably caused by different projections. Although CSSV was faster on the enclosed test scenes, OFTS has better geometry dependency and, in most cases, better resolution dependency. It was the only method to outperform the average of combined RTX time on the 1080Ti.

The CSSV algorithm was a surprise, as almost all previous papers presented stencil-based shadow volumes as being too slow for larger resolutions. Our implementation was able to compute an object’s silhouette in 0.03 to 0.1 ms across the test scenes, thus the majority of the method’s time was spent on the rasterization of shadow volumes. Although based on z-fail, CSSV was the fastest conventional method on “Buddha”, “Conference”, and “Sponza” in both 1920×1080 and 3840×2560. These scenes have relatively a simple silhouette, mostly observable on “Buddha”, where the method is on average 48 % faster than OFTS behind it. As no bias was used during any of its stages, the method has the most accurate shadows of the tested algorithms. We think the advance in the graphics hardware and increased rasterization performance can also be credited for the performance of the stencil shadow volumes. The method’s implementation is among the easier ones;

its memory consumption is second to ray tracing. The downsides are edge extraction as a preprocess step and unpredictable performance due to shadow volume rasterization. If not for the “Hairball” scene, which was the worst-case for this method, we would declare CSSV to be the second-fastest algorithm.

DPSV was improved by deterministic shadow plane calculations, which helped with robustness of the method but we still experienced blinking triangles in the “Buddha” scene, as the model consists of very small triangles. The randomness of the TOP tree build quality between consecutive frames caused the method to perform less stably locally, most notably on the “Hairball” scene. Our measurements have shown that this method is more suitable for scenes with up to 1 million triangles. The method is easy to implement, as source codes for both its shaders are already available and requires no geometry preprocessing. The memory consumption was average compared to other methods. Overall, the method ends up 4th in our comparison, as it did not excel in any of the observed parameters.

We succeeded in porting PTSV and CPTSV algorithms from CUDA to OpenGL, making them available to other hardware platforms. These methods, particularly CPTSV, are difficult to implement and their memory consumption is the highest of all tested methods, as it depends not only on the geometry, but also on the screen resolution. CPTSV’s acceleration structure, containing AABBs for view sample clusters, can take up to 6 GB at 4096×4096 if the memory optimization is not used, but can still take around 1 GB when optimized. The method also suffers from an incorrect acceleration structure build when the screen resolution is very high, which negatively affects its performance. If not for this issue, this method would have had one of the lowest sensitivities to screen resolution. PTSV was the slowest algorithm in the test. In general, we don’t recommend either of these methods for practical use, as there are faster and easier-to-implement methods that also consume less memory.

Except for ray tracing, there is no universal method that would be suitable for all scenarios; all methods have their best and worst cases. If ray tracing is not available, OFTS is suitable for more opened scenes, whereas CSSV handles closed or scenes with simple silhouette. Our tests also conclude that more complex code does not necessarily yield faster frame times.

In the future, the ray tracing implementation could be optimized for other hardware platforms, including the current generation of game consoles. It would be interesting to see even bigger scenes, although “Hairball” was already challenging for the most of the algorithms as the fastest time behind ray tracing was 14ms at 1920×1080. Multiple light sources is also an issue that is not frequently evaluated on these methods.

Chapter 9

Conclusion

The goal of this thesis was to improve rendering of precise shadows from omnidirectional light sources. I did not present a major breakthrough in the field of shadow rendering, instead a series of incremental performance and robustness improvements to stencil shadow volumes as well as silhouette extraction. Although many researches consider stencil shadow volumes to be a dead algorithm, the measurements in this thesis, notably in Chapter 8, show that with a proper implementation of silhouette extraction and utilization of modern features of the graphics hardware, this algorithm still keeps up with modern methods or even outperforms them.

First, I was a part of the team that designed a robust algorithm for silhouette extraction to deal with triangles almost parallel to the light direction. This completely eliminated shadow artifacts and allowed the usage of arbitrary triangle soup as input for the method. The determinism is based on assumption that the whole triangle may not have equal amount of front an back facing opposite vertices with respect to the light plane constructed from each triangle edge and the light source. I optimized the CPU implementation of the algorithm utilizing AVX instruction set which has proven beneficial, mostly on modern CPU architectures.

I then utilized tessellation to compute shadow volumes. At first, the approach was per-triangle and required two passes, thus rather slow. But we were able to first optimize the approach to a single pass utilizing geometry collapsing and later use the collapsible geometry approach to design a fully silhouette approach. The created method has been proven to be faster on certain platforms than methods using the geometry shader. Moreover, we have simplified the robust multiplicity calculation which no longer has to be evaluated for all 3 triangle edges. Instead, a new approach using a so-called *reference edge* was designed that sorted triangle vertices (formed from the edge and one opposite vertex) and instead of computing the multiplicity from a plane constructed from an edge and the light source, the plane was constructed from the triangle itself and light source was tested to be lying in front or behind the triangle plane. This allowed the algorithm to break the edge-triangle dependency and sped up the computation.

Then, silhouette extraction acceleration was proposed. Instead of calculating the silhouette in a brute-force manner, an octree is constructed with precomputed sets of potentially silhouette edges that need further testing and edges guaranteed to be silhouette for a particular volume of space where a light can be located. To conserve memory, a bitmask-based compression scheme was utilized that propagated edges up the octree even in case when not all siblings shared the edge. The method reduced the difficulty of the edge computation, but at the cost of relatively high memory consumption. The light source also had to be restricted to a certain area of the scene in order to use the accelerated silhouette extraction. The problem with space restriction could be solved by using a spatial angle subdivision. Although not by much (up to around 5 %), this method is the fastest stencil shadow algorithm to date, although it has its limitations.

The first survey paper focused on shadow-mapping-based omnidirectional shadow techniques, comparing cube mapping and parabolic projection (dual-paraboloid shadow mapping). Although the dual-paraboloid approach was faster on smaller scenes, cube mapping technique handled more complex scenes better. The parabolic projection also suffers from visual artifacts when the scene tessellation is low.

We gathered several modern methods rendering omnidirectional shadows from point light sources and tested them on a set of popular scenes and several resolutions. These methods, along with stencil shadows, were put against new hardware-accelerated ray tracer using RTX API in Vulkan. Although the ray tracer was the fastest method of the test, stencil shadows performed very well compared to other more modern methods, competing with omnidirectional frustum-traced shadows. The source codes of the testing software have been made publicly available, which can be easily extended by other researchers with their shadow techniques.

Bibliography

[1] Definition of shadow in English. June 2019.
Retrieved from: https://www.lexico.com/en/definition/shadow

[2] Aila, T.; Akenine-Möller, T.: A Hierarchical Shadow Volume Algorithm. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '04. New York, NY, USA: Association for Computing Machinery. 2004. ISBN 3905673150. page 15–23.
Retrieved from: https://doi.org/10.1145/1058129.1058132

[3] Aila, T.; Laine, S.: Alias-free Shadow Maps. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*. EGSR'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. 2004. ISBN 3-905673-12-6. pp. 161–166.
Retrieved from: http://dx.doi.org/10.2312/EGWR/EGSR04/161-166

[4] Airey, J. M.; Rohlf, J. H.; Brooks, F. P.: Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *SIGGRAPH Comput. Graph.*. vol. 24, no. 2. February 1990: page 41–50. ISSN 0097-8930.
Retrieved from: https://doi.org/10.1145/91394.91416

[5] Akenine-Möller, T.; Assarsson, U.: Approximate Soft Shadows on Arbitrary Surfaces Using Penumbra Wedges. In *Proceedings of the 13th Eurographics Workshop on Rendering*. EGRW '02. Goslar, DEU: Eurographics Association. 2002. ISBN 1581135343. page 297–306.

[6] Akenine-Möller, T.; Assarsson, U.: On the Degree of Vertices in a Shadow Volume Silhouette. *Journal of Graphics Tools*. vol. 8, no. 4. 2003: pp. 21–24.
Retrieved from: https://doi.org/10.1080/10867651.2003.10487591

[7] Akenine-Möller, T.; Haines, E.; Hoffman, N.: *Real-Time Rendering, Fourth Edition*. Natick, MA, USA: A. K. Peters, Ltd.. fourth edition. 2018. ISBN 0134997832, 9781138627000.

[8] Aldridge, G.; Woods, E.: Robust, Geometry-independent Shadow Volumes. In *Proceedings of the 2Nd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. GRAPHITE '04. New York, NY, USA: ACM. 2004. ISBN 1-58113-883-0. pp. 250–253.
Retrieved from: http://doi.acm.org/10.1145/988834.988877

[9] Appel, A.: Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*.

AFIPS '68 (Spring). New York, NY, USA: ACM. 1968. pp. 37–45.
Retrieved from: http://doi.acm.org/10.1145/1468075.1468082

[10] Assarsson, U.; Akenine-Möller, T.: A Geometry-Based Soft Shadow Volume Algorithm Using Graphics Hardware. *ACM Trans. Graph.*. vol. 22, no. 3. July 2003: page 511–520. ISSN 0730-0301.
Retrieved from: https://doi.org/10.1145/882262.882300

[11] Batagelo, H. C.; Júnior, I. C.: Real-Time Shadow Generation Using BSP Trees and Stencil Buffers. In *Proceedings of the XII Brazilian Symposium on Computer Graphics and Image Processing*. SIBGRAPI '99. USA: IEEE Computer Society. 1999. ISBN 0769504817. page 93.

[12] Bergeron, P.: A General Version of Crow's Shadow Volumes. *IEEE Computer Graphics and Applications*. vol. 6, no. 9. September 1986: pp. 17–28. doi:10.1109/MCG.1986.276543.

[13] Bilodeau, W.; Songy, M.: Real Time Shadows. Creativity '99 Creative Labs Inc. sponsored game conference. 1999.

[14] Blinn, J.: Me and My (Fake) Shadow. *IEEE Computer Graphics and Applications*. vol. 8, no. 01. jan 1988: pp. 82–86. ISSN 1558-1756. doi:10.1109/MCG.1988.10001.

[15] Boksanský, J.; Wimmer, M.; Bittner, J.: Ray Traced Shadows: Maintaining Real-Time Frame Rates. In *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, edited by E. Haines; T. Akenine-Möller. Berkeley, CA: Apress. 2019. ISBN 978-1-4842-4427-2. pp. 159–182. doi:10.1007/978-1-4842-4427-2_13.
Retrieved from: https://doi.org/10.1007/978-1-4842-4427-2_13

[16] Boudier, P.; Sellers, G.: GL_AMD_depth_clamp_separate. online. 15th September 2010. openGL extensions manual.
Retrieved from: https://www.khronos.org/registry/OpenGL/extensions/AMD/AMD_depth_clamp_separate.txt

[17] Brabec, S.; Annen, T.; Seidel, H.-P.; et al.: Shadow Mapping for Hemispherical and Omnidirectional Light Sources. *Advances in Modelling, Animation and Rendering (Proceedings Computer Graphics International 2002), Springer, 397-408 (2002)*. June 2002: pp. 397–408. doi:10.1007/978-1-4471-0103-1_25.

[18] Brabec, S.; Seidel, H.-P.: Shadow Volumes on Programmable Graphics Hardware. *Computer Graphics Forum*. vol. 22. 09 2003: pp. 433–440. doi:10.1111/1467-8659.00691.

[19] Carmack, J.: Email communication with Mark Kilgard. online. 26th May 2000.
Retrieved from: https://fabiensanglard.net/doom3_documentation/CarmackOnShadowVolumes.txt

[20] Chin, N.; Feiner, S.: Near Real-time Shadow Generation Using BSP Trees. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '89. New York, NY, USA: ACM. 1989. ISBN 0-89791-312-4. pp. 99–106.
Retrieved from: http://doi.acm.org/10.1145/74333.74343

[21] Contreras, M. S.; Valadez, A. J. R.; Martínez, A. J.: Dual Sphere-Unfolding Method for Single Pass Omni-Directional Shadow Mapping. In *ACM SIGGRAPH 2011 Posters*. SIGGRAPH '11. New York, NY, USA: Association for Computing Machinery. 2011. ISBN 9781450309714.
Retrieved from: https://doi.org/10.1145/2037715.2037793

[22] Cook, R. L.; Porter, T.; Carpenter, L.: Distributed Ray Tracing. *SIGGRAPH Comput. Graph.*. vol. 18, no. 3. January 1984: page 137–145. ISSN 0097-8930.
Retrieved from: https://doi.org/10.1145/964965.808590

[23] Cozzi, P.; Riccio, C.: *OpenGL Insights*. CRC Press. July 2012. ISBN 978-1439893760.

[24] Crow, F. C.: Shadow Algorithms for Computer Graphics. *SIGGRAPH Comput. Graph.*. vol. 11, no. 2. July 1977: pp. 242–248. ISSN 0097-8930.
Retrieved from: http://doi.acm.org/10.1145/965141.563901

[25] Deves, F.; Mora, F.; Aveneau, L.; et al.: Scalable Real-time Shadows Using Clustering and Metric Trees. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations*. SR '18. Goslar Germany, Germany: Eurographics Association. 2018. pp. 83–93.
Retrieved from: https://doi.org/10.2312/sre.20181175

[26] Diefenbach, P. J.: *Pipeline Rendering: Interaction and Realism Through Hardware-based Multi-pass Rendering*. PhD. Thesis. Philadelphia, PA, USA. 1996.

[27] Doghramachi, H.: Tile-based Omnidirectional Shadows. In *GPU Pro 360: Guide to Shadows*, edited by W. Engel. chapter 14. Natick, MA, USA: A. K. Peters, Ltd.. first edition. 2018. ISBN 0815382472, 9780815382478. pp. 193–217.

[28] Dou, H.; Yan, Y.; Kerzner, E.; et al.: Adaptive Depth Bias for Shadow Maps. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '14. New York, NY, USA: ACM. 2014. ISBN 978-1-4503-2717-6. pp. 97–102.
Retrieved from: http://doi.acm.org/10.1145/2556700.2556706

[29] Eisemann, E.; Assarsson, U.; Schwarz, M.; et al.: Shadow Algorithms for Real-time Rendering. In *Eurographics*. 2010.

[30] Eisemann, E.; Schwarz, M.; Assarsson, U.; et al.: *Real-Time Shadows*. Natick, MA, USA: A. K. Peters, Ltd.. first edition. 2011. ISBN 1568814380, 9781568814384.

[31] Engel, W.: Cascaded Shadow Maps. In *Shader X5: Advanced Rendering Techniques*, edited by W. Engel. chapter 10. Rockland, MA, USA: Charles River Media Inc.. 2006. ISBN 1584504994. pp. 197–206.

[32] Espenak, F.: Glossary of Solar Eclipse Terms. September 2019.
Retrieved from: https://eclipse.gsfc.nasa.gov/SEhelp/SEglossary.html

[33] Everitt, C.; Kilgard, M. J.: Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. Technical report. nVidia Corporation. 2002.

[34] Fuchs, H.; Goldfeather, J.; Hultquist, J. P.; et al.: Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. In *Advances in Computer Graphics I*, edited by G. Enderle; M. Grave; F. Lillehagen. Berlin, Heidelberg: Springer Berlin Heidelberg. 1986. ISBN 978-3-642-46514-7. pp. 169–187.

[35] Garanzha, K.; Loop, C.: Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Comput. Graph. Forum*. vol. 29. 05 2010: pp. 289–298. doi:10.1111/j.1467-8659.2009.01598.x.

[36] Gerasimov, P.: Omnidirectional Shadow Mapping. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, edited by F. Randima. chapter 12. Addison-Wesley Professional. 2004. ISBN 0321228324. pp. 193–203.

[37] Gerhards, J.; Mora, F.; Aveneau, L.; et al.: Partitioned Shadow Volumes. *Comput. Graph. Forum*. vol. 34, no. 2. May 2015: pp. 549–559. ISSN 0167-7055.
Retrieved from: https://doi.org/10.1111/cgf.12583

[38] Gooch, B.; Sloan, P.-P. J.; Gooch, A.; et al.: Interactive Technical Illustration. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*. I3D '99. New York, NY, USA: ACM. 1999. ISBN 1-58113-082-1. pp. 31–38.
Retrieved from: http://doi.acm.org/10.1145/300523.300526

[39] Haines, E.: Soft Planar Shadows Using Plateaus. *J. Graph. Tools*. vol. 6, no. 1. January 2002: pp. 19–27. ISSN 1086-7651.
Retrieved from: http://dx.doi.org/10.1080/10867651.2001.10487534

[40] Harris, M.; Sengupta, S.; Owens, J. D.: Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, edited by H. Nguyen. chapter 39. Addison-Wesley Professional. first edition. 2007. ISBN 9780321545428.

[41] Hasenfratz, J.-M.; Lapierre, M.; Holzschuch, N.; et al.: A survey of Real-Time Soft Shadows Algorithms. *Computer Graphics Forum*. vol. 22, no. 4. December 2003: pp. 753–774.
Retrieved from: http://maverick.inria.fr/Publications/2003/HLHS03a

[42] Heckbert, P.; Herf, M.: Simulating Soft Shadows with Graphics Hardware. *Tech. rep.* May 1997.

[43] Heckbert, P. S.: Ten Unsolved Problems in Rendering. In *Workshop on Rendering Algorithms and Systems, Graphics Interface '87*. April 1987.

[44] Heidmann, T.: Real shadows real time. *IRIS Universe*. vol. 18. November 1991: pp. 28–31.

[45] Herf, M.; Heckbert, P. S.: Fast Soft Shadows. In *Visual Proceedings, SIGGRAPH '96*. 1996. page 145.

[46] Ho, T.-Y.; Wan, L.; Leung, C.-S.; et al.: Unicube for Dynamic Environment Mapping. *IEEE Transactions on Visualization and Computer Graphics*. vol. 17, no. 1. January 2011: page 51–63. ISSN 1077-2626.
Retrieved from: https://doi.org/10.1109/TVCG.2009.205

[47] Hornus, S.; Hoberock, J.; Lefebvre, S.; et al.: ZP+: Correct Z-pass Stencil Shadows. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. New York, NY, USA: ACM. 2005. ISBN 1-59593-013-2. pp. 195–202.
Retrieved from: http://doi.acm.org/10.1145/1053427.1053459

[48] Imagination Technologies Limited: Dual Paraboloid Environment Mapping Whitepaper. Technical report. 17th April 2017.

[49] Immel, D. S.; Cohen, M. F.; Greenberg, D. P.: A Radiosity Method for Non-diffuse Environments. *SIGGRAPH Comput. Graph.*. vol. 20, no. 4. August 1986: pp. 133–142. ISSN 0097-8930.
Retrieved from: http://doi.acm.org/10.1145/15886.15901

[50] Imura, T.; Yamaguchi, M. K.; Kanazawa, S.; et al.: Perception of motion trajectory of object from the moving cast shadow in infants. *Vision Research*. vol. 46, no. 5. 2006: pp. 652 – 657. ISSN 0042-6989.
doi:https://doi.org/10.1016/j.visres.2005.07.028.

[51] Isenberg, T.; Freudenberg, B.; Halper, N.; et al.: A Developer's Guide to Silhouette Algorithms for Polygonal Models. *IEEE Comput. Graph. Appl.*. vol. 23, no. 4. July 2003: pp. 28–37. ISSN 0272-1716.
Retrieved from: https://doi.org/10.1109/MCG.2003.1210862

[52] Johnson, D. E.; Cohen, E.: Spatialized Normal Come Hierarchies. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*. I3D '01. New York, NY, USA: ACM. 2001. ISBN 1-58113-292-1. pp. 129–134.
Retrieved from: http://doi.acm.org/10.1145/364338.364380

[53] Johnson, G. S.; Lee, J.; Burns, C. A.; et al.: The Irregular Z-Buffer: Hardware Acceleration for Irregular Data Structures. *ACM Trans. Graph.*. vol. 24, no. 4. October 2005: page 1462–1482. ISSN 0730-0301. doi:10.1145/1095878.1095889.

[54] Kajiya, J. T.: The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM. 1986. ISBN 0-89791-196-2. pp. 143–150.
Retrieved from: http://doi.acm.org/10.1145/15922.15902

[55] Karras, T.; Aila, T.: Fast Parallel Construction of High-Quality Bounding Volume Hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. New York, NY, USA: Association for Computing Machinery. 2013. ISBN 9781450321358. page 89–99.
Retrieved from: https://doi.org/10.1145/2492045.2492055

[56] Kawabe, T.: Spatially Augmented Depth and Transparency in Paper Materials. In *SIGGRAPH Asia 2018 Emerging Technologies*. SA '18. New York, NY, USA: ACM. 2018. ISBN 978-1-4503-6027-2. pp. 12:1–12:2.
Retrieved from: http://doi.acm.org/10.1145/3275476.3275482

[57] Kersten, D.; Knill, D. C.; Mamassian, P.; et al.: Illusory motion from shadows. *Nature*. vol. 379. 1996: page 31.

[58] Kersten, D.; Mamassian, P.: Cast Shadow Illusions. In *The Oxford Compendium of Visual Illusions*, edited by G. Shapiro; D. Todorović. chapter 20. Oxford University Press. 2017. pp. 214–220.

[59] Kersten, D.; Mamassian, P.; C Knill, D.: Moving cast shadows induce apparent motion in depth. *Perception*. vol. 26. 02 1997: pp. 171–92. doi:10.1068/p260171.

[60] Kilgard, M.: GL_EXT_stencil_two_side. online. 15th September 2005. openGL extension manual.
Retrieved from: https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_stencil_two_side.txt

[61] Kilgard, M. J.: Robust Stencil Shadow Volumes. CEDEC 2001 presentation. 2001.
Retrieved from: http://developer.download.nvidia.com/assets/gamedev/docs/StencilShadows_CEDEC_E.pdf

[62] Kim, B.; Kim, K.; Turk, G.: A Shadow-Volume Algorithm for Opaque and Transparent Nonmanifold Casters. *Journal of Graphics, GPU and Game Tools*. vol. 13. 2008: pp. 1–14.

[63] King, G.; Newhall, W.: Efficient omni-directional shadow maps. In *ShaderX3: Advanced Rendering with DirectX and OpenGL*, edited by W. Engel. Hing-ham, MA, USA: Charles River Media Inc.. 2005. ISBN 1584503572. pp. 435–448.

[64] Kluczek, K.: Quality Metric for Shadow Rendering. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2016. pp. 791–796.

[65] Kobrtek, J.; Milet, T.; Herout, A.: Silhouette Extraction for Shadow Volumes Using Potentially Visible Sets. In *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*. Union Agency. 2019. ISBN 978-80-86943-37-4. pp. 9–16.
Retrieved from: https://www.fit.vut.cz/research/publication/11975

[66] Laine, S.: Split-plane Shadow Volumes. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '05. New York, NY, USA: ACM. 2005. ISBN 1-59593-086-8. pp. 23–32.
Retrieved from: http://doi.acm.org/10.1145/1071866.1071870

[67] Laine, S.; Aila, T.; Assarsson, U.; et al.: Soft Shadow Volumes for Ray Tracing. *ACM Trans. Graph.*. vol. 24, no. 3. July 2005: page 1156–1165. ISSN 0730-0301.
Retrieved from: https://doi.org/10.1145/1073204.1073327

[68] Lauritzen, A.; Salvi, M.; Lefohn, A.: Sample Distribution Shadow Maps. In *Symposium on Interactive 3D Graphics and Games*. I3D '11. New York, NY, USA: ACM. 2011. ISBN 978-1-4503-0565-5. pp. 97–102.
Retrieved from: http://doi.acm.org/10.1145/1944745.1944761

[69] Lehtinen, J.; Laine, S.; Aila, T.: An Improved Physically-Based Soft Shadow Volume Algorithm. *Comput. Graph. Forum*. vol. 25. 09 2006: pp. 303–312.

[70] Lengyel, E.: Advanced Stencil Shadow Advanced Stencil Shadow and Penumbral Wedge Rendering Penumbral Wedge Rendering. *Game Developers Conference*.

March 2005.
Retrieved from: http://www.terathon.com/gdc05_lengyel.pdf

[71] Liao, H.-C.: Shadow Mapping for Omnidirectional Light Using Tetrahedron Mapping. In *GPU Pro: Advanced Rendering Techniques*, edited by W. Engel. chapter 3. Natick, MA, USA: A. K. Peters, Ltd.. first edition. 2010. ISBN 1568814720. pp. 455–475.

[72] Lloyd, D. B.; Wendt, J.; Govindaraju, N. K.; et al.: CC Shadow Volumes. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*. EGSR'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. 2004. ISBN 3-905673-12-6. pp. 197–205.
Retrieved from: http://dx.doi.org/10.2312/EGWR/EGSR04/197-205

[73] Madison, C.; Thompson, W.; Kersten, D.; et al.: Use of interreflection and shadow for surface contact. *Perception & Psychophysics*. vol. 63, no. 2. Feb 2001: pp. 187–194. ISSN 1532-5962. doi:10.3758/BF03194461.
Retrieved from: https://doi.org/10.3758/BF03194461

[74] Marsh, D.; Marshall, D. L.: *Applied Geometry for Computer Graphics*. Berlin, Heidelberg: Springer-Verlag. first edition. 1999. ISBN 1852330805.

[75] McCool, M. D.: Shadow Volume Reconstruction from Depth Maps. *ACM Trans. Graph.*. vol. 19, no. 1. January 2000: pp. 1–26. ISSN 0730-0301.
Retrieved from: http://doi.acm.org/10.1145/343002.343006

[76] McGuire, M.: Efficient Shadow Volume Rendering. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, edited by F. Randima. chapter 9. Addison-Wesley Professional. 2004. ISBN 0321228324. pp. 137–166.

[77] McGuire, M.: Observations on Silhouette Sizes. *Journal of Graphics Tools*. vol. 9, no. 1. 2004: pp. 1–12. jgt.
Retrieved from:
https://casual-effects.com/research/McGuire2004Silhouette/index.html

[78] McGuire, M.: Single-pass Shadow Volumes for Arbitrary Meshes. In *ACM SIGGRAPH 2007 Posters*. SIGGRAPH '07. New York, NY, USA: ACM. 2007. ISBN 978-1-4503-1828-0.
Retrieved from: http://doi.acm.org/10.1145/1280720.1280912

[79] McGuire, M.: Computer Graphics Archive. July 2017.
Retrieved from: https://casual-effects.com/data

[80] McGuire, M.; Hughes, J. F.; Egan, K.; et al.: Fast, Practical and Robust Shadows. Technical report. NVIDIA Corporation. Austin, TX. Nov 2003.
Retrieved from:
http://developer.nvidia.com/object/fast_shadow_volumes.html

[81] Milet, T.; Kobrtek, J.; Zemčík, P.; et al.: Fast and Robust Tessellation-Based Silhouette Shadows. In *WSCG 2014 - Poster papers proceedings*. University of West Bohemia in Pilsen. 2014. ISBN 978-80-86943-72-5. pp. 33–38.
Retrieved from: https://www.fit.vut.cz/research/publication/10587

[82] Milet, T.; Navrátil, J.; Zemčík, P.: An Improved Non-Orthogonal Texture Warping for Better Shadow Rendering. In *WSCG 2015 - Full Papers Proceedings*. Union Agency. 2015. ISBN 978-80-86943-65-7. pp. 99–107.
Retrieved from: https://www.fit.vut.cz/research/publication/10889

[83] Mora, F.; Gerhards, J.; Aveneau, L.; et al.: Deep Partitioned Shadow Volumes Using Stackless and Hybrid Traversals. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations*. EGSR '16. Goslar, Germany: Eurographics Association. 2016. ISBN 978-3-03868-019-2. pp. 73–83.
Retrieved from: https://doi.org/10.2312/sre.20161212

[84] Nagy, G.: Real-Time Shadows on Complex Objects. In *Game Programming Gems*, edited by M. DeLoura. Charles River Media. 2000. pp. 567–580.

[85] Navrátil, J.; Kobrtek, J.; Zemčík, P.: A Survey on Methods for Omnidirectional Shadow Rendering. *Journal of WSCG*. vol. 20, no. 2. 2012: pp. 89–96. ISSN 1213-6972.
Retrieved from: https://www.fit.vut.cz/research/publication/9936

[86] Nguyen, H.: Casting Shadows on Volumes. *Game Developer*. vol. 6. March 1999: pp. 44–53.

[87] Ni, R.; Braunstein, M.; J Andersen, G.: Perception of scene layout from optical contact, shadows, and motion. *Perception*. vol. 33. 02 2004: pp. 1305–18. doi:10.1068/p5288.

[88] Olson, M.; Zhang, H.: Silhouette Extraction in Hough Space. *Computer Graphics Forum*. vol. 25. 09 2006: pp. 273–282. doi:10.1111/j.1467-8659.2006.00946.x.

[89] Osman, B.; Bukowski, M.; McEvoy, C.: Practical Implementation of Dual Paraboloid Shadow Maps. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*. Sandbox '06. New York, NY, USA: Association for Computing Machinery. 2006. ISBN 1595933867. page 103–106.
Retrieved from: https://doi.org/10.1145/1183316.1183331

[90] Pečiva, J.; Starka, T.; Milet, T.; et al.: Robust Silhouette Shadow Volumes on Contemporary Hardware. In *Conference Proceedings of GraphiCon'2013*. GraphiCon Scientific Society. 2013. ISBN 978-5-8044-1402-4. pp. 56–59.
Retrieved from: https://www.fit.vut.cz/research/publication/10408

[91] Pharr, M.; Jakob, W.; Humphreys, G.: *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.. third edition. 2016. ISBN 0128006455, 9780128006450.

[92] Planinšič, G.; Viennot, L.: *Shadows: stories of light*. Physics Education Division of the European Physical Society. 2010.
Retrieved from: https://cdn.ymaws.com/www.eps.org/resource/collection/016775D4-8888-474D-887F-3E33AEA5E6D0/EPSPED_MUSE_SHWS_sl.pdf

[93] Pop, M.; Duncan, C.; Barequet, G.; et al.: Efficient Perspective-accurate Silhouette Computation and Applications. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry*. SCG '01. New York, NY, USA: ACM.

2001. ISBN 1-58113-357-X. pp. 60–68.
Retrieved from: http://doi.acm.org/10.1145/378583.378618

[94] Reshetov, A.; Soupikov, A.; Hurley, J.: Multi-Level Ray Tracing Algorithm. In *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. New York, NY, USA: Association for Computing Machinery. 2005. ISBN 9781450378253. page 1176–1185.
Retrieved from: https://doi.org/10.1145/1186822.1073329

[95] Röttger, S.; Irion, A.; Ertl, T.: Shadow Volumes Revisited. UNION Agency. 01 2002. pp. 373–380.

[96] Sellers, G.: GL_AMD_stencil_operation_extended. online. 11th January 2012. openGL extension manual.
Retrieved from: https://www.khronos.org/registry/OpenGL/extensions/AMD/AMD_stencil_operation_extended.txt

[97] Sintorn, E.; Eisemann, E.; Assarsson, U.: Sample Based Visibility for Soft Shadows Using Alias-free Shadow Maps. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*. EGSR '08. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. 2008. pp. 1285–1292.
Retrieved from: http://dx.doi.org/10.1111/j.1467-8659.2008.01267.x

[98] Sintorn, E.; Kämpe, V.; Olsson, O.; et al.: Per-triangle Shadow Volumes Using a View-sample Cluster Hierarchy. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '14. New York, NY, USA: ACM. 2014. ISBN 978-1-4503-2717-6. pp. 111–118.
Retrieved from: http://doi.acm.org/10.1145/2556700.2556716

[99] Sintorn, E.; Olsson, O.; Assarsson, U.: An Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects Using Per-triangle Shadow Volumes. In *Proceedings of the 2011 SIGGRAPH Asia Conference*. SA '11. New York, NY, USA: ACM. 2011. ISBN 978-1-4503-0807-6. pp. 153:1–153:10.
Retrieved from: http://doi.acm.org/10.1145/2024156.2024187

[100] Sintorn, E.; Olsson, O.; Assarsson, U.: An Efficient Alias-Free Shadow Algorithm for Opaque and Transparent Objects Using per-Triangle Shadow Volumes. *ACM Trans. Graph.*. vol. 30, no. 6. December 2011: page 1–10. ISSN 0730-0301.
Retrieved from: https://doi.org/10.1145/2070781.2024187

[101] Stitch, M.; Wächter, C.; Keller, A.: Efficient and Robust Shadow Volumes Using Hierarchical Occlusion Culling and Geometry Shaders. In *GPU Gems 3*, edited by H. Nguyen. chapter 11. Addison-Wesley Professional. first edition. 2007. ISBN 9780321545428. pp. 239–256.

[102] Story, J.: Hybrid Ray-Traced Shadows. *Game Developers Conference*. March 2015.

[103] Story, J.: Advanced Geometrically Correct Shadows for Modern Game Engines. *Game Developers Conference*. March 2016.
Retrieved from: http://developer.download.nvidia.com/gameworks/events/GDC2016/jstory_hfts.pdf

[104] Taya, S.; Miura, K.: Cast shadow can modulate the judged final position of a moving target. *Attention, perception psychophysics*. vol. 72. 10 2010: pp. 1930–7. doi:10.3758/APP.72.7.1930.

[105] Uhlmann, J. K.: Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*. vol. 40, no. 4. 1991: pp. 175 – 179. ISSN 0020-0190. Retrieved from: http://www.sciencedirect.com/science/article/pii/002001909190074R

[106] Veras, D.; Breedt, E.: Eclipse, transit and occultation geometry of planetary systems at exo-syzygy. *Monthly Notices of the Royal Astronomical Society*. vol. 468, no. 3. 03 2017: pp. 2672–2683. ISSN 0035-8711. http://oup.prod.sis.lan/mnras/article-pdf/468/3/2672/13146452/stx614.pdf. Retrieved from: https://doi.org/10.1093/mnras/stx614

[107] Vlachos, A.; Card, D.: Computing Optimized Shadow Volumes for Complex Data Sets. In *Game Programming Gems 3*, edited by D. Treglia. Charles River Media. 2002. pp. 367–371.

[108] Wanger, L.: The Effect of Shadow Quality on the Perception of Spatial Relationships in Computer Generated Imagery. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*. I3D '92. New York, NY, USA: ACM. 1992. ISBN 0-89791-467-8. pp. 39–42.
Retrieved from: http://doi.acm.org/10.1145/147156.147161

[109] van Waveren, J. M. P.: Shadow Volume Construction. online. 8th April 2005. Retrieved from:
http://fabiensanglard.net/doom3_documentation/37730-293752.pdf

[110] Whitted, T.: An Improved Illumination Model for Shaded Display. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '79. New York, NY, USA: Association for Computing Machinery. 1979. ISBN 0897910044. page 14.
Retrieved from: https://doi.org/10.1145/800249.807419

[111] Williams, L.: Casting Curved Shadows on Curved Surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '78. New York, NY, USA: ACM. 1978. pp. 270–274.
Retrieved from: http://doi.acm.org/10.1145/800248.807402

[112] Woo, A.: *Shadow Algorithms Data Miner*. Hoboken, NJ: CRC Press. 2012. ISBN 9781439880234.

[113] Wyman, C.; Hoetzlein, R.; Lefohn, A.: Frustum-traced Raster Shadows: Revisiting Irregular Z-buffers. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*. i3D '15. New York, NY, USA: ACM. 2015. ISBN 978-1-4503-3392-4. pp. 15–23.

[114] Zhang, F.; Sun, H.; Xu, L.; et al.: Parallel-split Shadow Maps for Large-scale Virtual Environments. In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications*. VRCIA '06. New York, NY, USA:

ACM. 2006. ISBN 1-59593-324-7. pp. 311–318. doi:10.1145/1128923.1128975. Retrieved from: http://doi.acm.org/10.1145/1128923.1128975

[115] Žára, J.; Beneš, B.; Sochor, J.; et al.: *Moderní počítačová grafika*. Praha: Computer Press. second edition. 2005. ISBN 80-251-0454-0.

Appendix A

List of Publications

[1] Navrátil, J.; Kobrtek, J.; Zemčík, P.: A Survey on Methods for Omnidirectional Shadow Rendering. In *Journal of WSCG*. vol. 20, no. 2. 2012. pp. 89-96. ISSN 1213-6972.

[2] Pečiva, J.; Starka, T.; Milet, T.; Kobrtek, J.; et al.: Robust Silhouette Shadow Volumes on Contemporary Hardware. In *Conference Proceedings of GraphiCon'2013*. Vladivostok: GraphiCon Scientific Society. 2013. pp. 56-59. ISBN 978-5-8044-1402-4.

[3] Klampár, M.; Spohner, M,; Škarvada P.; Sobola, D.; Kobrtek, J.; et al.: Dielectric Properties of Epoxy Resins with Oxide Nanofillers and Their Accelerated Ageing. In *IEEE Catalog Number CFP13EEI-USB*. Ottawa, Ontario, CA. 2013. pp. 159-164. ISBN 978-1-4673-4738-9.

[4] Milet, T.; Kobrtek, J.; Zemčík P.: Fast and Robust Tessellation-Based Silhouette Shadows. In *WSCG 2014 - Poster papers proceedings*. Plzeň: University of West Bohemia in Pilsen. 2014. pp. 33-38. ISBN 978-80-86943-72-5.

[5] Milet, T.; Tóth, M.; Pečiva, J.; Starka, T.; Kobrtek, J.; et al.: Fast robust and precise shadow algorithm for WebGL 1.0 platform. In *ICAT-EGVE 2015 - International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments*. Kyoto: Eurographics Association. 2015. pp. 85-92. ISBN 978-3-905674-84-2.

[6] Kobrtek, J.; Milet, T.; Herout, A.: Silhouette Extraction for Shadow Volumes Using Potentially Visible Sets. In *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*. Plzeň: Union Agency. 2019. pp. 9-16. ISBN 978-80-86943-37-4.