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

SHADOWING AND LIGHTING ACCELERATION
AKCELERACE VRHÁNÍ STÍNŮ A OSVĚTLOVÁNÍ

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR Ing. TOMÁŠ MILET
AUTOR PRÁCE

SUPERVISOR prof. Ing. ADAM HEROUT, PhD.
ŠKOLITEL

BRNO 2020

Abstract

The goal of the thesis is to present methods for acceleration of shadows and lighting. The main focus of the thesis is an acceleration of per-sample precise shadows using shadow volumes on different platforms. Other parts of this thesis also describe methods for increasing the precision of shadow map based methods and lighting.

Abstrakt

Cílem této práce je prezentovat metody pro akceleraci výpočtů stínů a osvětlení. Práce se zabývá akcelerací na vzorek přesných stínů pomocí stínových těles na různých platformách. Obsahem práce je také zvýšení přesnosti stínových map a zvýšení přesnosti osvětlování scény s mnoha světly.

Keywords

shadows, shadow volumes, shadow maps, silhouettes, robustness, lighting, warping, potentially visible set, many lights

Klíčová slova

stíny, stínová tělesa, stínové mapy, siluety, robustnost, osvětlování, deformace, potenciálně viditelné množiny, mnoho světel

Reference

MILET, TOMÁŠ. *Shadowing and Lighting Acceleration.* Brno, 2020. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Adam Herout, PhD.

Shadowing and Lighting Acceleration

Declaration

I declare that this thesis is my original work. I cited all sources and literature. I have written this thesis under lead of Prof. Ing. Adam Herout, Ph.D.

....
Tomáš Milet
August 12, 2020

Acknowledgements

I would like to thank people who helped me create this thesis. Namely: Tomáš Chlubna, Adam Herout, Pavel Zemčík, Jozef Kobrtek, Tomáš Starka, Jan Pečiva, Michal Kula, Michal Tóth, Tomáš Lysek. I would also like to thank Jitka Miletová and Matěj Petružela for their support. Next, I would like to thank 허준, 세종, 예진 and 대장금; 감사합니다. I would also like to remember close relatives who passed away during my studies.

제망매가

삶과 죽음의 길은
여기 있으매 머뭇거리고
나는 간다는 말도
못 다 이르고 어찌 갑니까

어느 가을 이른 바람에
이에 저에 떨어질 잎처럼
한 가지에 나고
가는 곳 모르는구나

아아 그림자에서 만날 나
도 닦아 기다리겠노라

월명사 (762–765)

Contents

Preface

In this short part of the thesis, I will describe some personal things that give context to the rest of the text. This section is informal and readers are welcome to skip it, but I feel the need to write it down. I would like to start with how I worked on this thesis and how other school activities interfered with it.

This thesis was large part of eight years of my life. I worked on it during my Ph.D. studies at Faculty of Information Technology in Brno. I worked on it semi-regularly since the deadline was always someday in the future. I was distracted by way too many other school activities and side projects. These other activities took big chunk of my time which led to a lot of postpones. A lot of ideas presented in this thesis were formed in the first two years of my study, but I was unable to finish the implementation back then. In the rest of the preface I will describe some selected activities that distracted me.

In the first year of my doctoral studies, I was invited to the team reSound with Michal Zachariáš, István Szentandrási and Rudolf Kajan to participate in Imagine Cup 2013 competition. We created an augmented reality game where player's goal was to „control“ music. The game was quite innovative due to the augmented reality nature and we won the national round and went to St. Petersburg. In the international round, we placed ourselves in top 5 teams and were told that we chose the wrong category - game category instead of innovation category. An the end of the event, I had an some ideas of how to accelerate shadow rendering witch led to further research later. This event was one of my first distractions and it predetermined my inability to refuse work on side projects.

I helped friends (Jan Navrátil, Jozef Kobrtek) with their doctoral theses and took a part in other projects and cooperated with different IT companies.

I have work with embedded systems for NXP company. I designed and implemented some vision algorithms for their automobile computer systems. All algorithms were designed for their i.MX8 processors and were GPU accelerated using OpenVX for its build-in Vivante GPU. Other task was to design and implement GPU accelerated inference engine for convolutional neural networks. I used OpenCL and together with Roman Juránek we developed system for training and inference neural nets for their embedded platforms.

In the last one or two years I have cooperated with Smarter Instruments company and have worked with Looking Glass holographic displays. We have been developing our own multi-platform holographic renderer together with Tomáš Chlubna. As the renderer works on Linux, I combined my knowledge from times when I worked for NXP and created holographic renderer for embedded systems, see Figure 1.

I also helped with project for company Honeywell. The goal was to analyze properties of vision based navigation for flying drones in situations where the GPS is not reliable - in cities with high buildings. We simulated the algorithms and drones in virtual environment, see Figure 2.

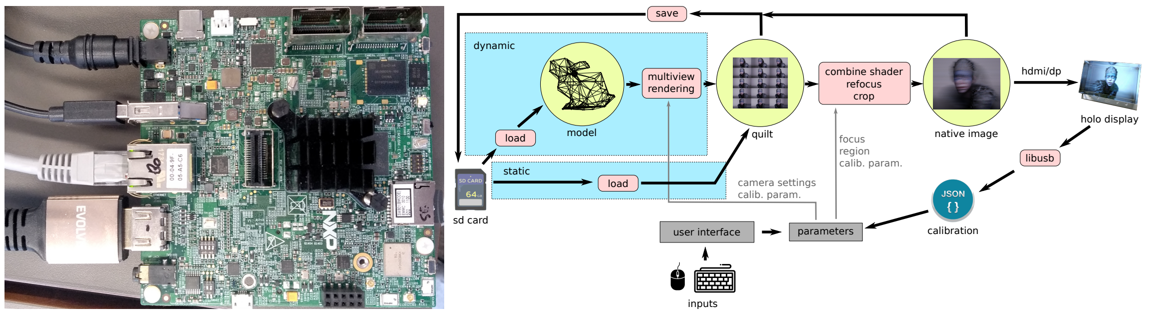


Figure 1: The left image shows NXP development board i.MX 8MQuadLite. This passive cooling device is able to render holographic images for Looking Glass display using our open renderer. The right image shows simplified diagram of holographic renderer.

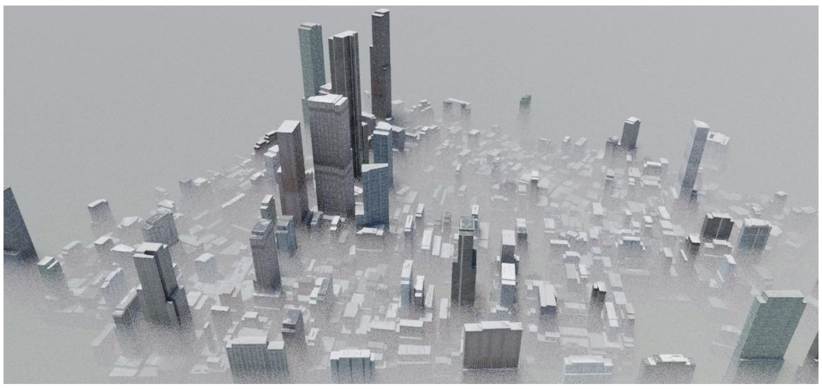


Figure 2: The image shows environment where virtual drones were tested. The environment and drone trajectories were created in Blender.

Maybe the biggest part of my doctoral studies was (and still is) teaching. I love teaching, sharing knowledge and helping students. Nowadays, I teach in six different courses - all related to compute graphics. I also lead (with Tomáš Starka) voluntary seminars for students on different IT topics and arrange so called „rendering readings“. These readings are weekly meetings with others interested in computer graphics where we dissect a scientific paper. I constantly try to innovate lectures, unify and redistribute information between lectures and improve student projects and lab tasks. Every doctoral student has to teach. I usually teach at least 8 times more than common doctoral student. It is not uncommon for me to have more teaching hours than most academics in my faculty (even docents and professors). This determination or even obsession led to one of the most memorable moments in my life. I was chosen by students as the best teacher for bachelor studies in my faculty in 2019.

This is just a small sample of all project I have participated in. All of side projects had one obvious drawbacj. I didn't invest enough time into finishing this thesis. It took much longer that I thought. It could have been easily finished four years earlier.

The development of algorithms, methods and approaches described in this thesis started as early as in 2012. The company Cadwork informatik AG have been co-operating with Faculty of Information Technology in Brno. Tomáš Starka and Jan Pečiva have been developing rendering algorithms for CAD software and one of the rendering tasks was to design real-time shadowing algorithm. There were specific requirements for the shadowing algorithm. The main four requirements were:

- The algorithm had to be real-time or at least interactive.
- It had to handle arbitrary models with complex triangle settings, including non-manifolds, models with holes and edges with any number of connected triangles.
- It had to produce per-sample precise shadows with no refinement over time.
- It had to be multi-platform, running on any graphic processing unit and operating system.

There is a very large body of scientific papers published on topic of shadow algorithms and there is a lot of diverse methods for rendering of several types of shadows with different attributes. From the vast spectrum of methods, the algorithm called Shadow Volumes was chosen as it fulfills all requirements. Other methods for rendering shadows were considered, mainly Shadow Mapping or ray-tracing. Ray-tracing produces per-sample precise shadows and works with arbitrary models, but it is/was slower that other methods. Ray-tracing requires good acceleration structures in order to run at least interactively. With recent introduction of hardware specific units (ray-tracer cores) on NVIDIA Turing graphics cards, designed to handle sub-tasks of ray-tracing, ray-tracing is starting to be viable option.

On the other hand, Shadow mapping is by far one of the fastest methods for rendering shadows. That fact is the reason why it is used in most today's video games. There is a lot of different variants of Shadow Mapping algorithm and the base algorithm can also be used as subpart of other visual algorithms. The main difficulty of Shadow Mapping is it's visual quality. It produces wide variety of visual artifacts and for each of them there exists a lot of scientific papers of how to solve them or at least how to mitigate them.

The selected algorithm - Shadow Volumes can render per-sample precise shadows in real-time. However, some of its faster variants have problems with non manifold models and other complex triangle settings. Furthermore, the algorithm sometimes produces imperfect shadows due to hardware specific reasons. Some of the methods described in this thesis are used by Cadwork informatik AG company in their applications today.

The goal of the thesis is to describe new approaches and additions to the vast topic of rendering of shadows. It summarizes algorithms and methods developed during eight years.

Chapter 1

Introduction

Lighting and shadows are important parts of our everyday life. They are not as obvious as air or water, but without them our lives would be greatly different. They help us navigate in our environment, help us understand the composition of the world and even create feelings. Good lighting can create mood and atmosphere, while shadows can highlight important objects. They are well understood in arts, movies and video games. Without them, our lives would be much blander, see the Figure 1.1. The goal of this text is to present new ideas of how to compute shadows that are precise and robust in every situation and lighting algorithms.

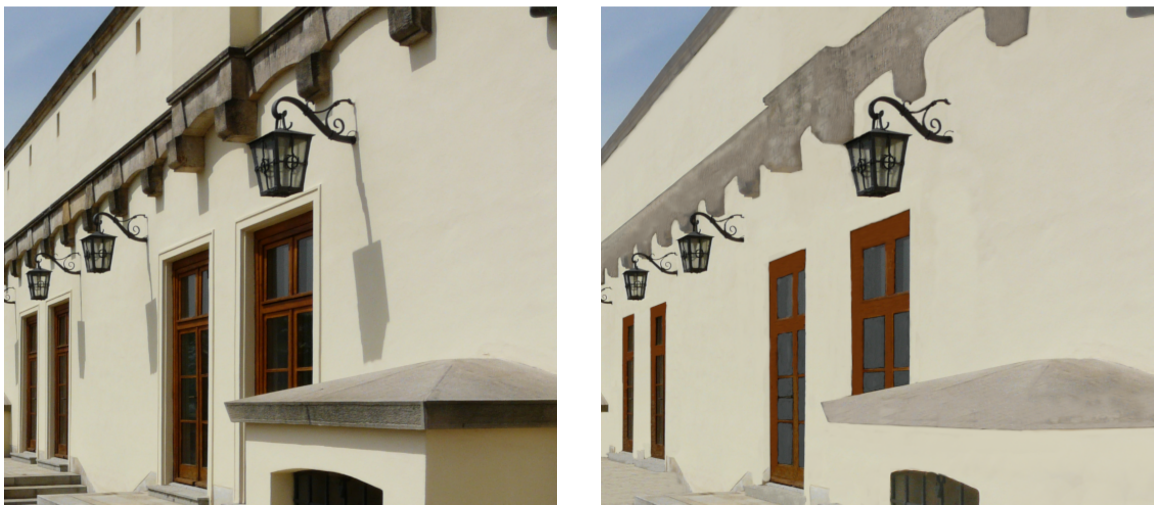


Figure 1.1: The image shows photograph of Špilberk castle. Shadows and lighting are removed on the right side leaving only textures and colors.

Chapter 2

Basics of Lights and Shadows

The goal of this thesis is to present methods for acceleration of shadow computation and lighting on different hardware platforms. This chapter describes overall properties of lights and shadows and their importance in rendering of scenes.

2.1 Light Sources

Lights and light sources illuminate 3D scenes and create shadows. Shadows can only arise in scenes with a light source. There are many types of light sources. Algorithms for shadow computation are usually specialized for different kinds of lights. This short section describes the most common light sources in virtual environments.

A light source can be small or large and can create soft or hard shadows, see Figure 2.1.

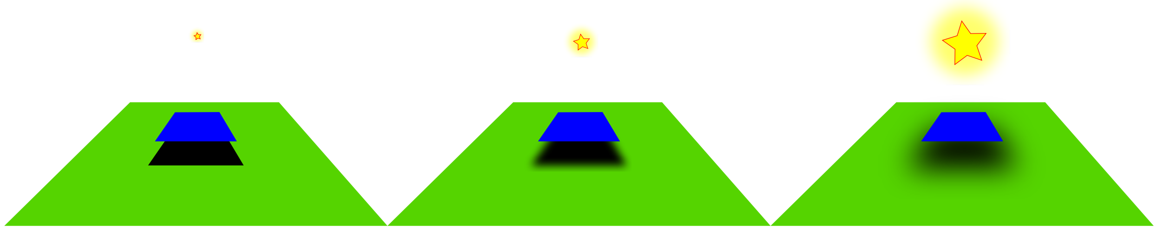


Figure 2.1: Point light sources create hard shadows, the left side. Large light sources (or area lights) create soft shadows, the right side. The shadow border smoothness depends on the light source size and the distance of the light and the shadow caster to the shadow receiver. Soft shadows are usually more computationally expensive.

Formally, the light source can be described as a set $\mathfrak{L}$ of points in 3D space that emit light energy. If the set $\mathfrak{L}$ contains infinite number of points, it is usually described as area or volume light source. If the set contains only one point, it describes point light source. If the point emits the light only to certain directions, the light is called spot light. If the point is located in infinity, the light is called directional light. Types of point light sources are visualized in Figure 2.2.

Light sources can be primary or secondary, creating direct or indirect illumination, see Figure 2.3.

The most of this work is focused on omnidirectional point light sources.

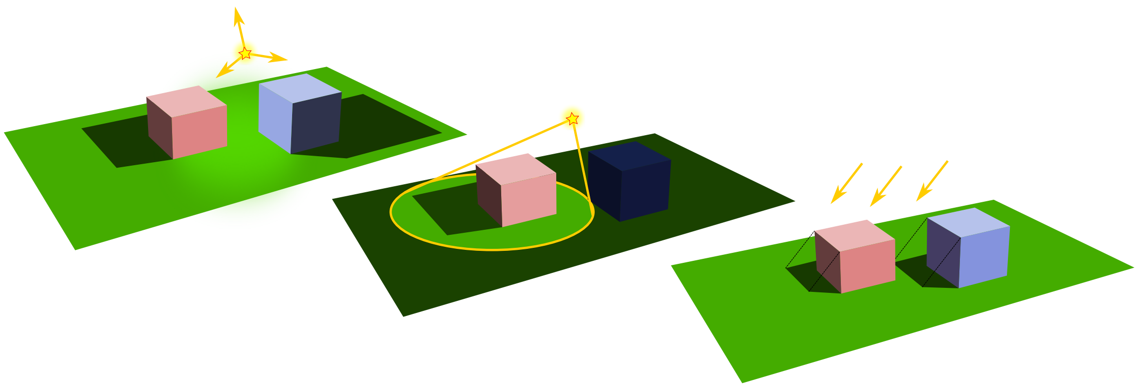


Figure 2.2: The left side of the image shows omnidirectional point light. The middle part shows spot light source. The right side shows directional light source. All light types are able to create hard shadows. Omnidirectional light sources are usually the most computationally expensive point light sources.

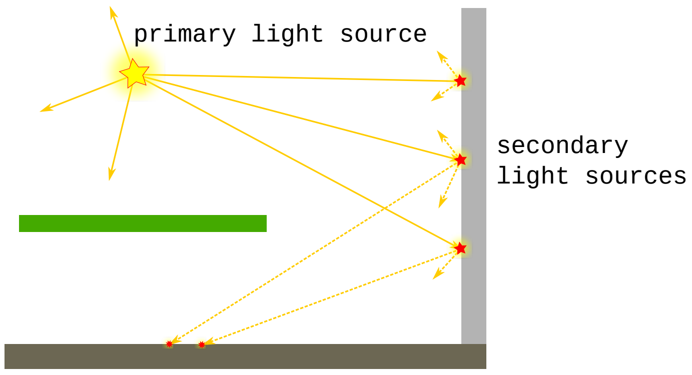


Figure 2.3: The image shows primary light source and secondary light sources. Primary light sources emit light by themselves, secondary light sources are created when a light is reflected off a surface. Primary light sources create direct illumination while secondary light sources create indirect illumination. Secondary point light sources are sometimes called virtual point light sources (VPL).

2.2 Shadows

Shadows are an important part of 3D rendering. They improve depth perception and visual quality, see Figure 2.4 and Figure 2.5.

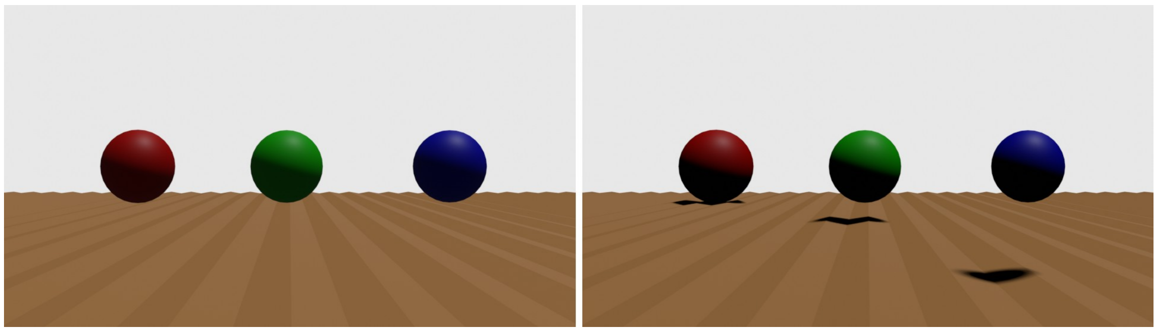


Figure 2.4: The left side of the figure shows a scene with three spheres without shadows. All spheres seem to be located at the same height above the ground. Also, spheres appear to have the same radius. The right side of the figure shows the same scene but with shadows. The shadows improved the depth perception and the ability of the viewer to recognize the relation between objects in the scene. The red sphere is the largest, while the blue sphere is the smallest. The red sphere is located much closer to the ground. The shading alone cannot help the viewer as the scene is illuminated with directional light.

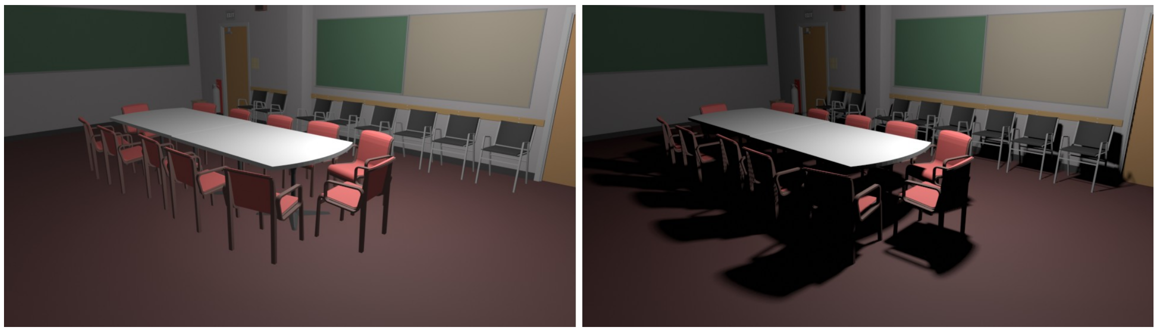


Figure 2.5: The left side of the figure shows the Conference room without shadows. The right side of the figure shows the same scene with shadows. The visualization quality is improved.

There are many ways of defining shadows. A mathematical way to define a shadow (as presented in the Siggraph 2013 course Efficient Real-time Shadows by Eisemann [26]) can be seen in Figure 2.6.

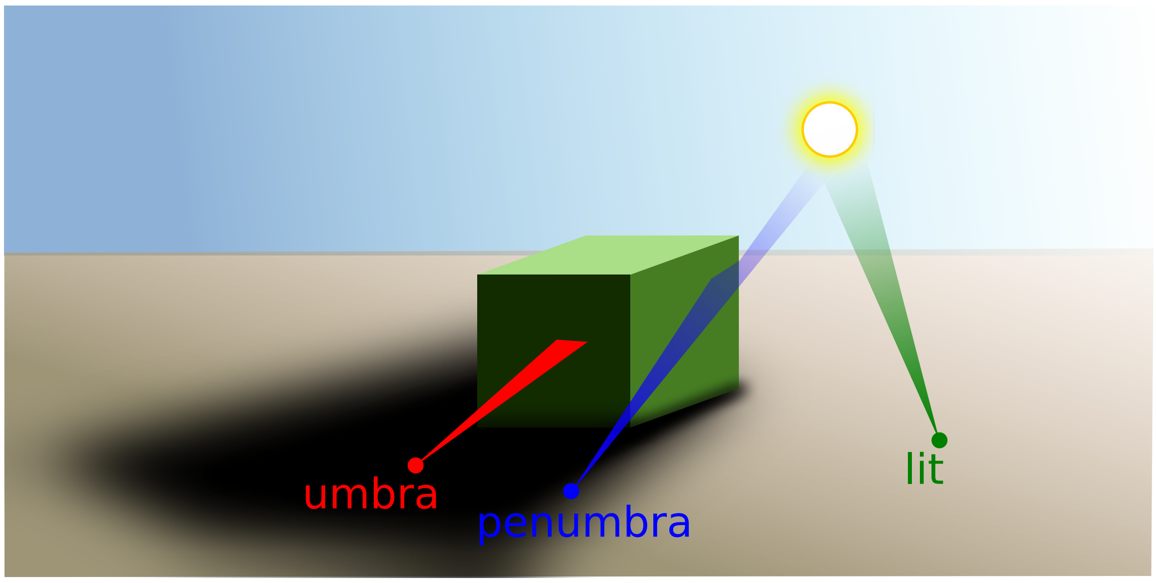


Figure 2.6: This Figure is inspired by Figure 1.4 from the book Real-Time Shadows by Eisemann [25]. The omnidirectional area light source $\mathfrak{L}$ is a set of points that emits a light. The surface point P is considered shadowed if there is an occluding geometry between P and $\mathfrak{L}$ and not all rays from $\mathfrak{L}$ reach the point P. If none of rays from $\mathfrak{L}$ reaches the point P, the region is called umbra. If some rays from light $\mathfrak{L}$ reaches the point P, the region is called penumbra. Penumbra regions or soft shadows can only arise in scenes lit by area light sources or in scenes with transparent objects. Hard shadows arise in scenes lit by point light sources.

Following algorithms require the definition of shadow caster and receiver, see Figure 2.7.

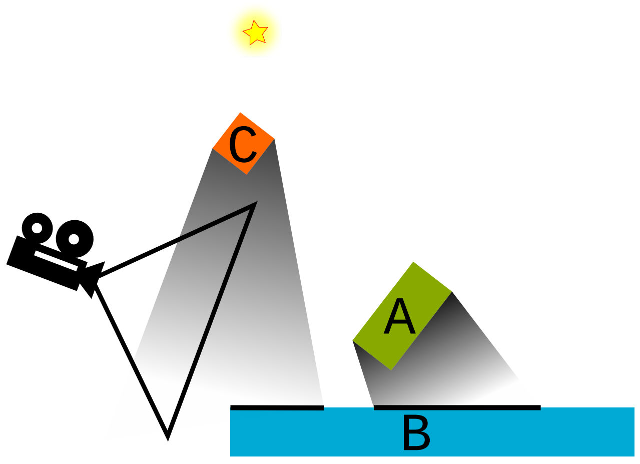


Figure 2.7: Objects A and C are shadow casters. Object B is shadow receiver. In reality, all object are shadow casters and receivers. However, It is useful to distinguish between objects that can receive shadows (shadows can be seen by camera on their surfaces) and shadow casters that influence observed shadows. Shadow casters are sometimes called blockers or occluders.

There are different kind of shadows. Hard shadows and soft shadows are just two categories. There are also indirect shadows, see Figure 2.8.

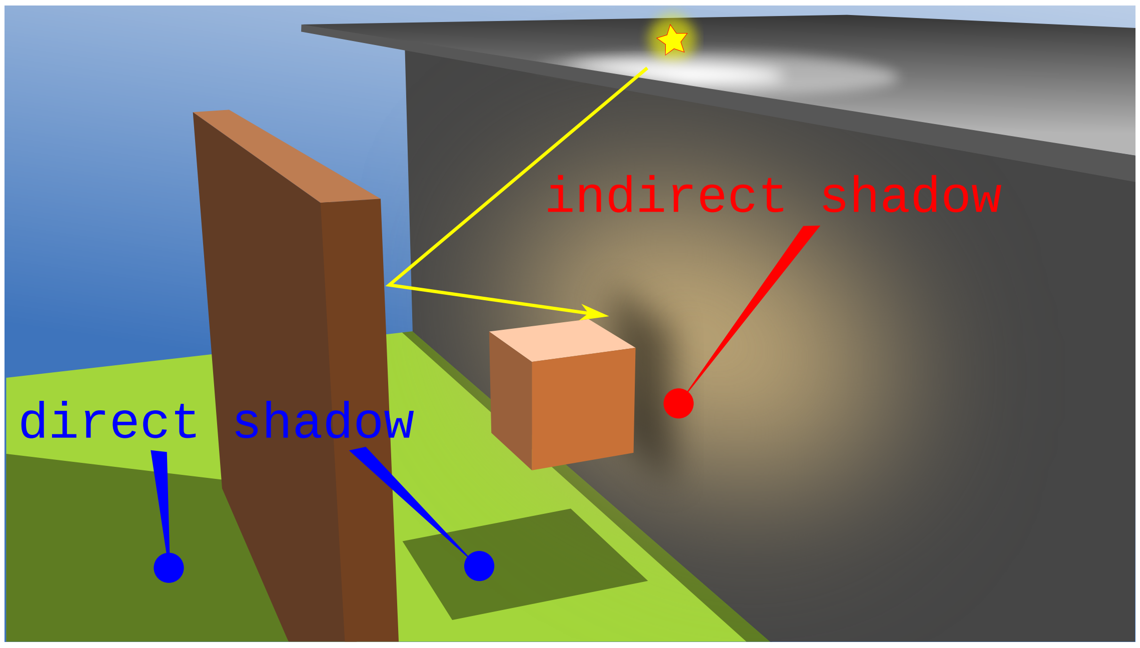


Figure 2.8: The image shows direct and indirect shadows. The indirect shadows arise as the light from light source bounces off the big orange block and illuminates gray wall. The indirect shadows are soft even if the light is point source. Indirect shadows are caused by indirect lighting.

Other than that, ambient occlusion can be considered as a form of shadow, see Figure 2.9.

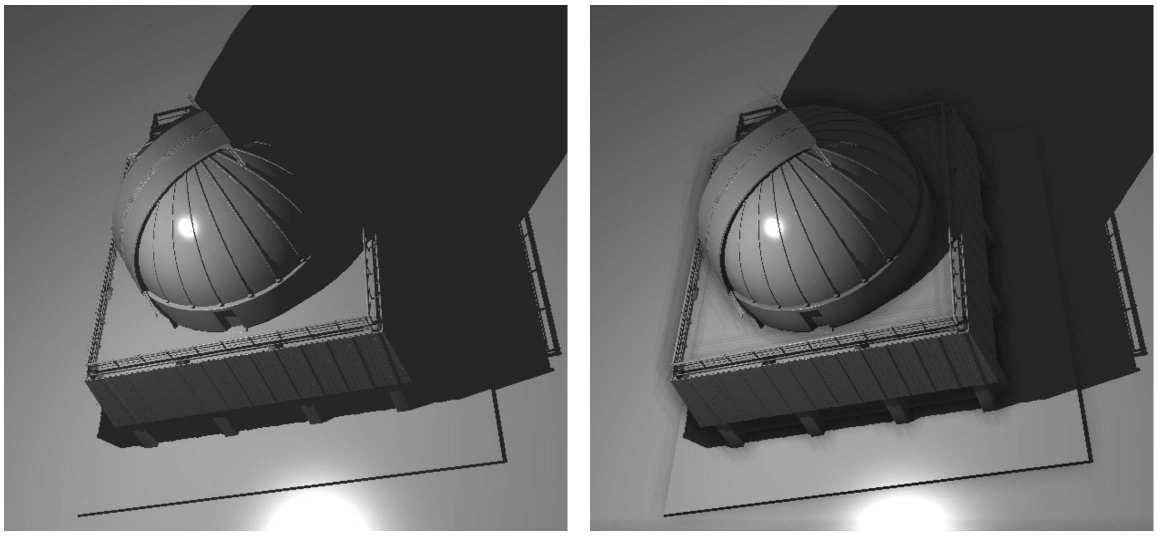


Figure 2.9: The left side of the image shows Observatory scene illuminated with point light source. All surfaces are gray and geometric details are lost in shadow regions. The right side of the image shows effects of ambient occlusion. Corners and slits are much darker preserving details. Ambient occlusion is a cheap method for computing indirect shadows and other effects of indirect illumination.

Generally, concept of shadows is redundant as it can be defined in other way. Outgoing surface radiance can fully describe light and dark regions as is defined in rendering equation, Equation 2.1. The rendering equation was introduced by Kajiya [50] and Immel et al. [44] and can describe illumination in a scene:

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + L_r(\mathbf{p}, \omega_o) \tag{2.1}$$

The terms in Equation 2.1 are:

- $\mathbf{p}$ is a surface point.
- ω_o is a direction from the surface point $\mathbf{p}$.
- L_o is an outgoing surface radiance from the point $\mathbf{p}$ in the direction ω_o.
- L_e is an emitted radiance from the point $\mathbf{p}$ in the direction ω_o. This term is zero for surface points that are not light sources.
- L_r is a reflected radiance from the point $\mathbf{p}$ in the direction ω_o. This term is summation of all incoming radiance to the point $\mathbf{p}$, see Equation 2.2.

$$L_r(\mathbf{p}, \omega_o) = \int_\Omega f_r(\mathbf{p}, \omega_o, \omega_i) L_i(\mathbf{p}, \omega_i) \cos(\mathbf{n}_\mathbf{p}, \omega_i) d\omega_i \tag{2.2}$$

The equation Equation 2.2 describes reflected radiance. The reflected radiance is summation of all incoming radiance over the hemisphere with a center in the surface point $\mathbf{p}$.

- L_r is the reflected radiance from the point $\mathbf{p}$ in the direction ω_o.

- f_r describes surface properties and it is called bidirectional reflectance distribution function BRDF. This term modulates how much of the incoming light energy from direction ω_i to direction ω_o for point p is reflected.
- L_i is an incoming radiance at the point $\mathbf{p}$ from another surface point in the direction ω_i.
- $\mathbf{n_p}$ is the surface normal of the point $\mathbf{p}$.

The rendering equation is fundamental basis in computer graphics. The equation describes equilibrium of light energy in a scene. Every surface point emits or reflects some amount of light. Shadows or dark regions of a scene produce less light while lit and light regions produce more light. With this concept, shadows are the difference in observed outgoing radiance.

The rendering equation is recursive and hard to compute. A lot of research is aimed at solving it effectively and precisely. Partial approximations and specializations are useful, because they allow the computation in real-time. The concept of shadows is one of these specializations.

2.3 Basic Shadow Algorithms

There is large spectrum of real-time shadow algorithms. A lot of algorithms is based on core ideas of shadow mapping or shadow volumes. This section describes basis of shadow mapping and shadow volumes and defines common terms.

Shadow algorithms require definitions of a scene, a view-sample, a shadow-sample and a light source, see Figure 2.10.

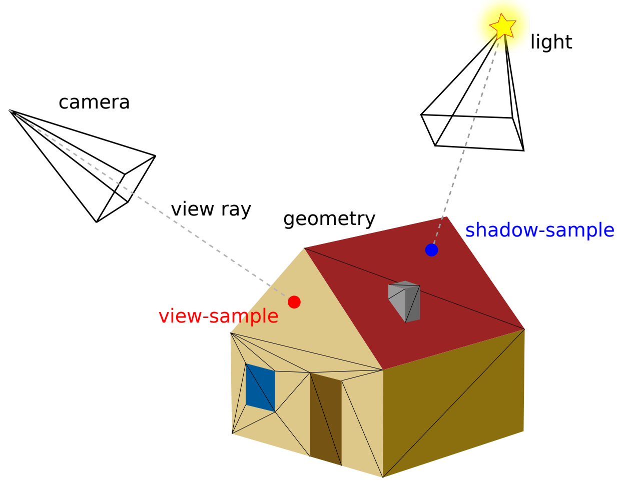


Figure 2.10: A scene geometry is composed of triangles. A view-sample (red point) is the closest intersection point of a view ray from camera with the scene geometry. A view-sample is associated with a screen space pixel. There is usually one view-sample per pixel. If multi-sampling is enabled, there can be multiple view-samples per pixel. Point light source illuminates the scene. Some algorithms draw the scene from light point of view and create shadow-samples (blue point).

2.3.1 Shadow Mapping

Shadow mapping is probably the most widespread shadow algorithm today. There are many reasons. It is very fast algorithm with support in hardware. The performance is quite stable. It is relatively easy to implement. It supports an arbitrary soup of triangles. It is commonly used as a part of other algorithms. Shadow mapping and its variants are used by wast majority of computer games today.

The main idea behind shadow mapping is to render a scene two times, see Figure 2.11.

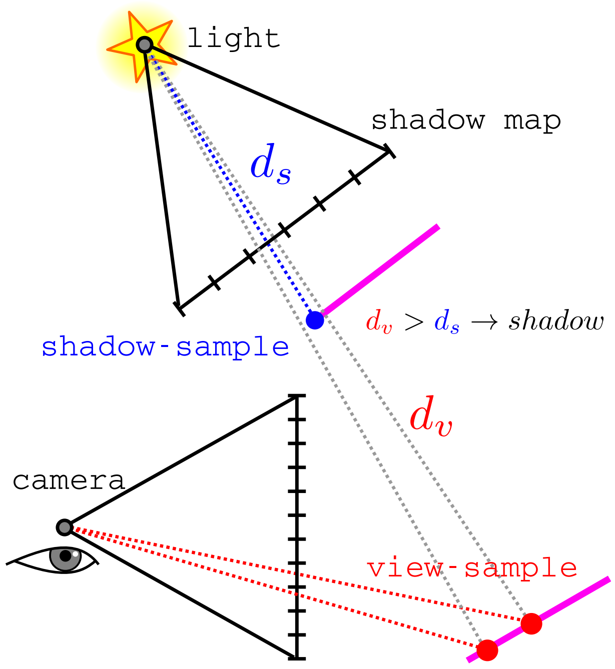


Figure 2.11: The image shows shadow mapping algorithm. The first render pass rasterizes a scene from light point of view. The rasterization creates shadow-samples that are stored in a 2D texture called shadow map. Shadow map stores depth values for each shadow-sample d_s. A shadow sample could be viewed as a small square facing light source, which casts shadow into the scene. The second render pass rasterizes a scene from camera point of view. Each view-sample is projected into the shadow map and the algorithm finds the closest shadow-sample. If the shadow-sample depth d_s is less than the view–sample depth d_v, the view–sample is shadowed.

Shadow mapping in its bare form suffers from wide variety of visual artifacts. A lot of research is aimed at quality improvements, performance improvements and extensions such as transparency, soft shadows and omnidirectional light sources. Shadow mapping is described with more details in later sections.

2.3.2 Shadow Volumes

Next algorithm is Shadow volumes. The main idea is shown in Figure 2.12. The algorithm produces per sample precise shadows. Shadow volumes are supported by hardware with stencil operations. Generally, shadow volumes are slower than shadow mapping. While quality improvements is the main research topic for shadow mapping, most of the research on shadow volumes is focused on performance. The algorithm is geometry based which can cause some problems with complex geometry. Shadow volumes were widely used in gaming industry but today are mostly replaced by shadow mapping due to its speed. Detailed description of shadow volume algorithms is presented in following sections.

There are other shadow techniques like projected shadows and ray-tracing, but they are out of scope of this text.

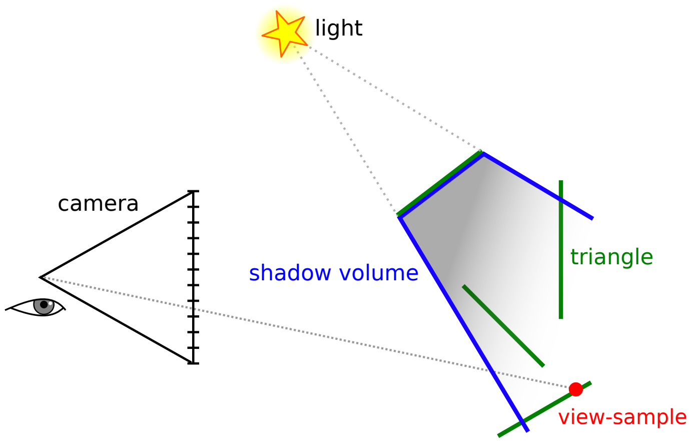


Figure 2.12: The image shows shadow volume algorithm. The algorithm creates shadow volume geometry that encloses shadows. Shadow volumes are created using a scene geometry. If a view-sample lies within any shadow volume, it is shadowed.

Chapter 3

Precise Hard Shadows

This section describes algorithms and methods for per sample precise hard shadows rendering, see Figure 3.1. There is a lot of different shadowing methods. Some methods are focused on speed, some are useful for soft shadows, some for transparent or volumetric shadows. Even if the category of methods is limited to just precise hard shadows, it contains tens of scientific papers and additional topics. The goal of this section is to describe most of the real-time per sample precise methods for rendering hard shadows. The first part of the section describes floating-point arithmetic, geometry and topology. These general topics are important for precise algorithms. Next part describes shadow volume algorithms and their variants. The last section describes various sample precise shadow methods.

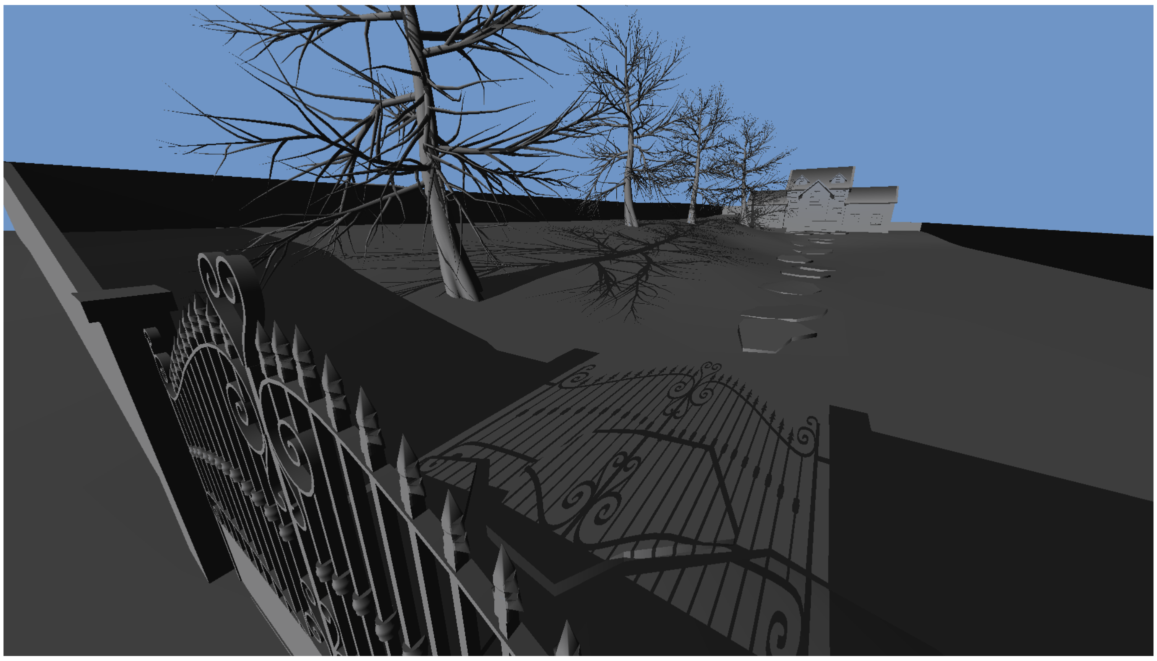


Figure 3.1: The figure shows Villa scene illuminated with point light source. Sample precise shadows are cast from each part of the geometry: thin tree branches and iron gate bars. The image was rendered using silhouette shadow volume algorithm.

3.1 Robustness and Floating-point Arithmetic

This section describes common problems that arise when dealing with floating-point numbers and their significance for rendering algorithms. In-depth discussion about floating-point numbers can be found in „What Every Computer Scientist Should Know about Floating-Point Arithmetic“ by Goldberg [34] which describes most of the issues. The motivation for this section can be seen in Figure 3.2.

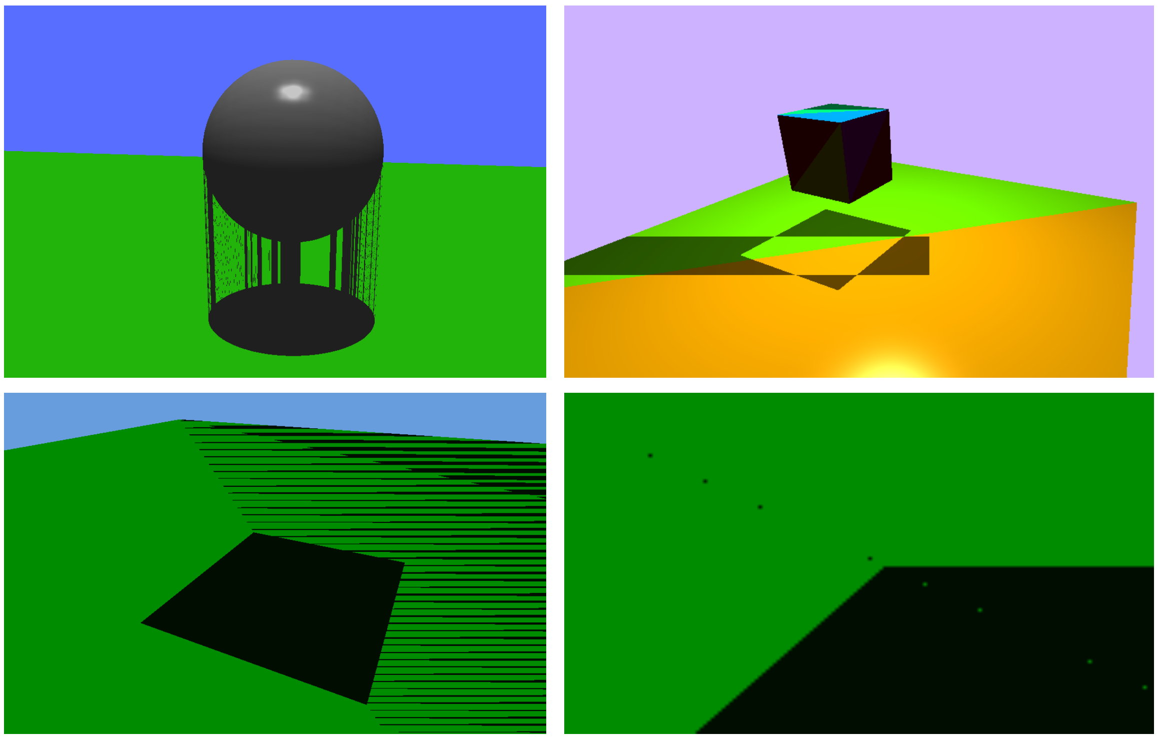


Figure 3.2: The image shows different visual artifact that can affect precise shadow rendering. The top left image shows incorrect silhouette computation. The bottom left image shows z-fight caused by different vertex winding orders. The top right shows artifacts caused by incorrect plane computation for silhouette algorithm with hierarchical rasterization. The bottom right shows incorrectly lit or shadowed pixels. All of these issues can be traced to floating-point arithmetic.

3.1.1 Floats

The floating-point standard IEEE 754 [95] specifies the definition of floating-point values, restrictions and operations on them.

The binary format of float is usually divided into three parts: sign, exponent and significand, see Figure 3.3.

In mathematics, addition and multiplication of three real values $a, b, c \in \mathbb{R}$ is commutative and associative. Resulting value does not depend on the order of variables: $a + b + c = a + c + b = c + b + a = \cdots$ and $a \cdot b \cdot c = a \cdot c \cdot b = \cdots$. This is true for some floating-point values, see Table 3.1, but it is not generally true, see Table 3.2.

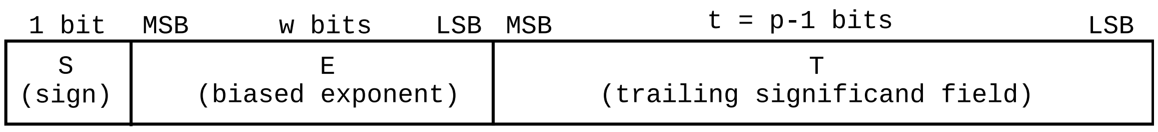


Figure 3.3: The figure shows binary representation of floating-point value as specified in [95]. A value is encoded in k bits, where S is 1-bit sign, biased exponent is encoded in w-bits and trailing significand is encoded in $t = p - 1$ bits. For 32 bit floating-point value: $k = 32$, $p = 24$, $s = 1$ and $w = 8$ according to Table 3.5 in [94]. The significand is encoded in 23 bits because the leading bit is implicitly encoded in the biased exponent. The Figure is modified Figure 3.1 from [94].

	a	b	c	(a+b)+c	a+(b+c)
float	2	3	5	10	10
hex	0x40000000	0x40400000	0x40a00000	0x41200000	0x41200000

Table 3.1: This table shows three floating-point values a, b, c stored in IEEE 754. The third row shows hexadecimal representation. The last two columns show result of addition in different order of variables. The computation of these expressions gives the same result.

	a	b	c	(a+b)+c	a+(b+c)
float	1e+08	100	100	1e+08	1e+08
hex	0x4cbebc20	0x42c80000	0x42c80000	0x4cbebc38	0x4cbebc39

Table 3.2: This table shows three floating-point values a, b, c stored in IEEE 754. The third row shows hexadecimal representation. The last two columns show result of addition in different order of variables. As can be seen, the addition of three floating-point values is not associative. The problem arises (but is not limited) when adding extreme values (either small or large).

This simple fact introduces issues for certain precise algorithms as well as for precise shadow computation. Problems with floating-point values affect geometry computing as is described in Section 3.1.2.

3.1.2 Geometry and Plane Construction

The construction of geometric primitives, such as lines and planes, is affected by floating-point arithmetic. The ordering of variables determines the resulting values, discussed in Section 3.1.1. This section describes manifestation of floating-point issues in geometry construction, see Figure 3.4.

As can be seen in Figure 3.4 and Figure 3.5, even the most simple task of computing a plane from three points in 3D space can introduce problems. One of the solutions to plane construction is points ordering. The ordering ensures the deterministic computation. Two 3D points can be ordered using less operator described in Algorithm 1. However, the ordering can introduce additional computational cost.

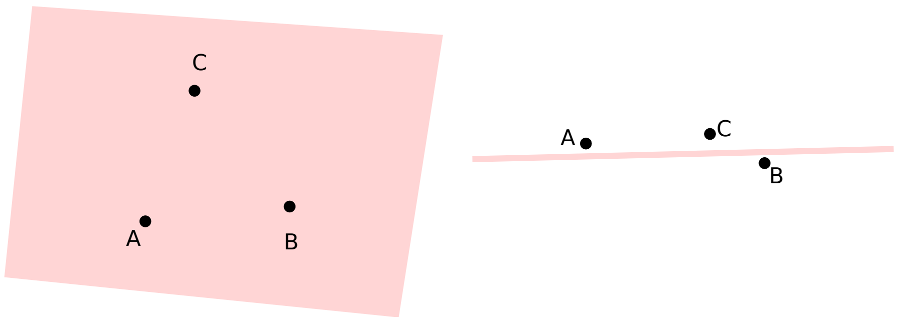


Figure 3.4: The figure shows three points A, B and C, each represented using 3 floating-point values. A plane (4 floating-point values) is constructed using these three points. Due to floating-point precision, points A, B, C may not lie on the constructed plane. Next issue (ordering) is described in Figure 3.5.

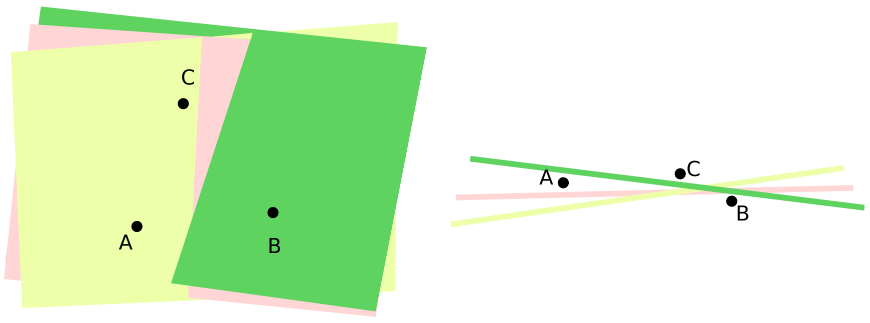
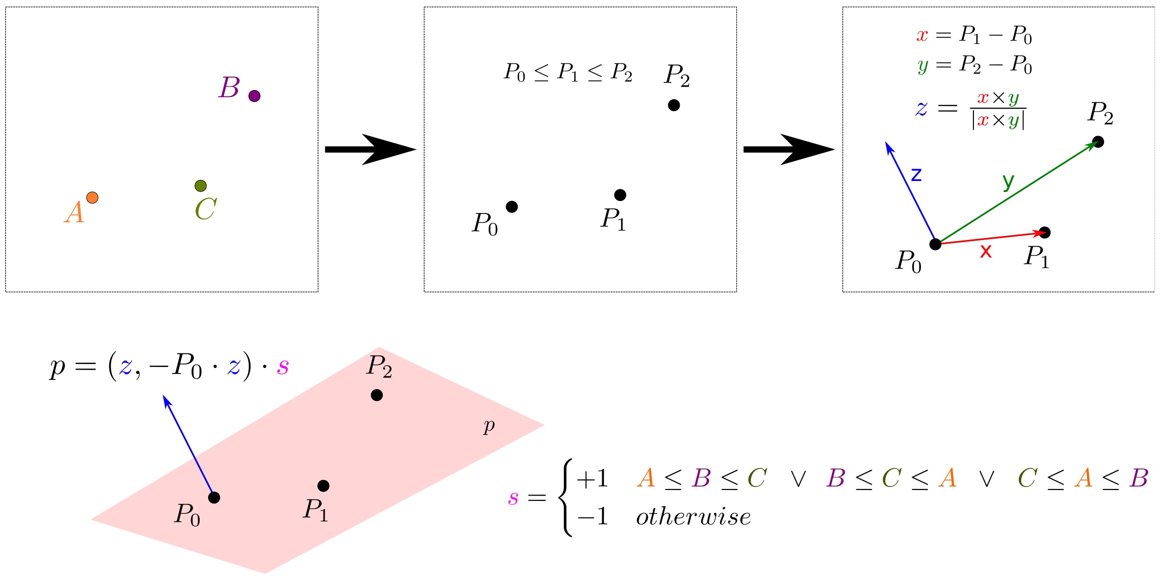


Figure 3.5: The figure shows three points A, B and C. Plane normal is constructed using function: $\mathrm{nor}(A, B, C) = (B - A) \times (C - A)$. Plane is constructed using function $\mathrm{plane}(A, B, C) = (\mathrm{nor}(A, B, C), -\mathrm{nor}(A, B, C) \cdot A)$. Different order of points during construction does not result in the same plane, $\mathrm{plane}(A, B, C) \neq \mathrm{plane}(B, A, C)$. One of the solution to this problem can be seen in Figure 3.6.

Algorithm 1: This algorithm describes ordering operator (relation <) for two 3D points.

```
   Data: 3D points $\mathbf{A}, \mathbf{B}$
   Result: true if $\mathbf{A} < \mathbf{B}$, false otherwise
1  i = 0;
2  while $i < 3$ do
3      if $\mathbf{A}_i < \mathbf{B}_i$ then
4          return true
5      if $\mathbf{A}_i > \mathbf{B}_i$ then
6          return false
7      i = i + 1;
8  return false
```



B
A
C
$P_0 \leq P_1 \leq P_2$
P_2
P_0
P_1
$x = P_1 - P_0$
$y = P_2 - P_0$
$z = \frac{x \times y}{|x \times y|}$
z
y
x
P_0
P_1
P_2
$p = (z, -P_0 \cdot z) \cdot s$
P_2
p
P_0
P_1
$s = \begin{cases} +1 & A \leq B \leq C \;\vee\; B \leq C \leq A \;\vee\; C \leq A \leq B \\ -1 & otherwise \end{cases}$

Figure 3.6: This figure shows algorithm for deterministic construction of a plane in 3D. The input of the algorithm is in form of three points in 3D space (top left). First step of the algorithm is to sort input points (top middle). Then it constructs normal vector from sorted points (top right). In the final step (bottom), the plane is constructed using normal vector and the first sorted point. The orientation of the normal vector (and the plane) is flipped if the ordering step swapped exactly two points. All six orderings can be seen in Figure 3.7. Ordering operator can be seen in Algorithm 1. Similar algorithm for plane construction was used in implementation of [93].

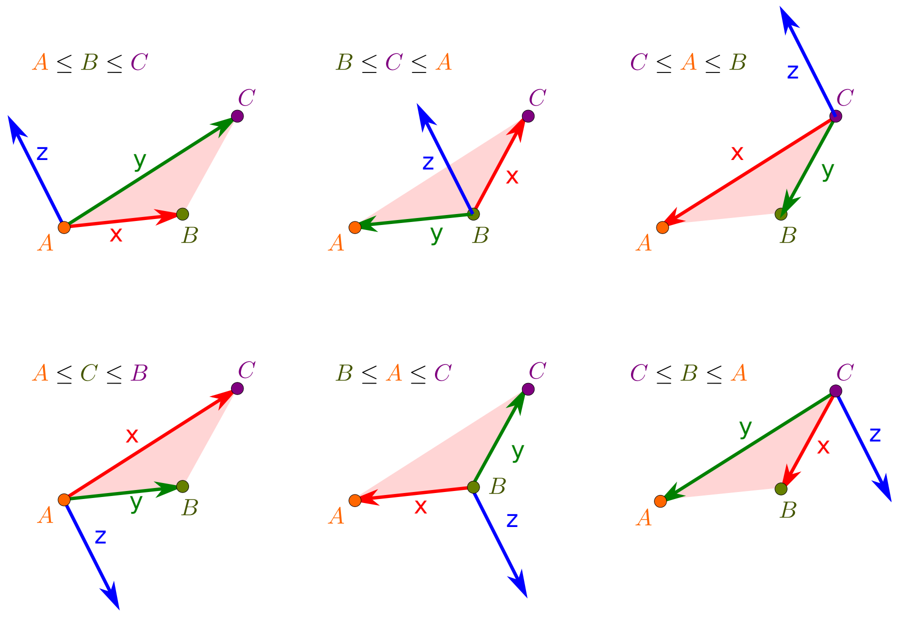


Figure 3.7: This figure shows six possible cases of plane construction from three points, see Figure 3.6. Three points A, B and C are sorted. The top row shows three cases, where none or all points were swapped during sorting. The bottom row shows remaining three cases, where exactly two points were swapped during sorting. The normal vector and plane orientation have to be flipped for bottom cases.

3.2 Geometry and Topology

Precise shadow algorithms require detailed geometry description. Algorithms like shadow volumes are sensitive to geometrical properties of models. Some simple algorithms are able to compute shadows only from watertight models and fail when working with more general geometry. This section describes geometry and topology of models that are suitable for algorithms described in following sections.

First, algorithms usually require triangulated model. Polygons have to contain exactly three points, see the Figure 3.8.

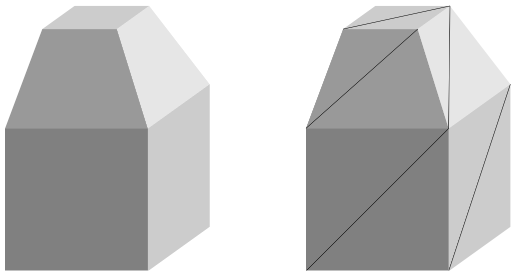


Figure 3.8: Most modeling software support polygons with arbitrary number of vertices. These polygons can be triangulated without loss of generality. The left side shows a model as usually displayed in modeling tool and the right side shows preprocessed, triangulated model.

Edges can be shared among multiple triangles. Intersection edges are not considered as a part of a model. Triangles are oriented using vertex window order, see the Figure 3.9.

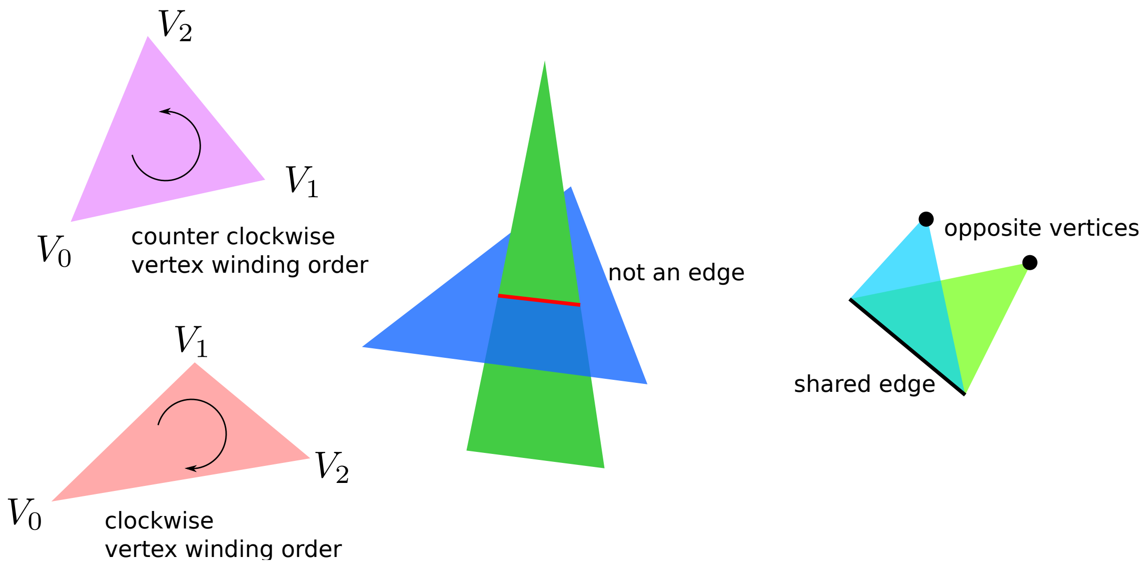


Figure 3.9: Triangles are composed of three 3D vertices described by three floating-point numbers and three edges. Vertices are ordered in clockwise or counter clockwise manner. The ordering is called vertex winding order. The triangle winding order determines its orientation. If two triangles intersect, the intersection edge is not considered as a part of the geometry. Two or more triangles can share an edge; the third vertex of each adjacent triangle is called opposite vertex.

Simplest models are usually watertight. More general models can contain holes or slits, see Figure 3.10.

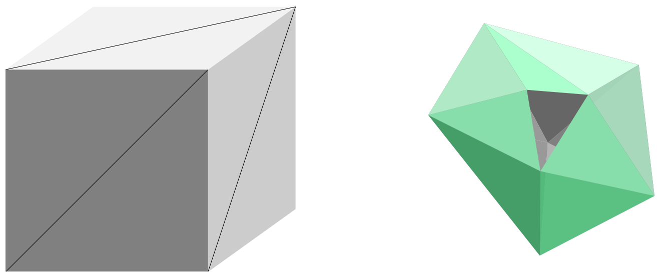


Figure 3.10: The left side shows an example of watertight model, the right side shows a model with a hole.

Good models are usually oriented. However, it is not always the case and some faces can have flipped normal vectors, see Figure 3.11.

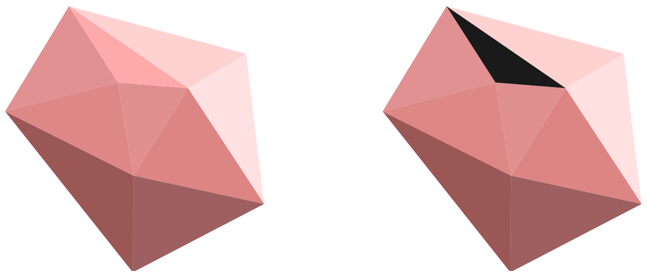


Figure 3.11: The left side shows properly oriented model. Every face has the same vertex winding order. All normals point outside. The right side shows non-oriented model. The black face has opposite vertex winding order and its normal vector points inside.

A model can be 2-manifold or contain edges with arbitrary number of adjacent triangles, see Figure 3.12. Non-manifold models are usually harder to process.

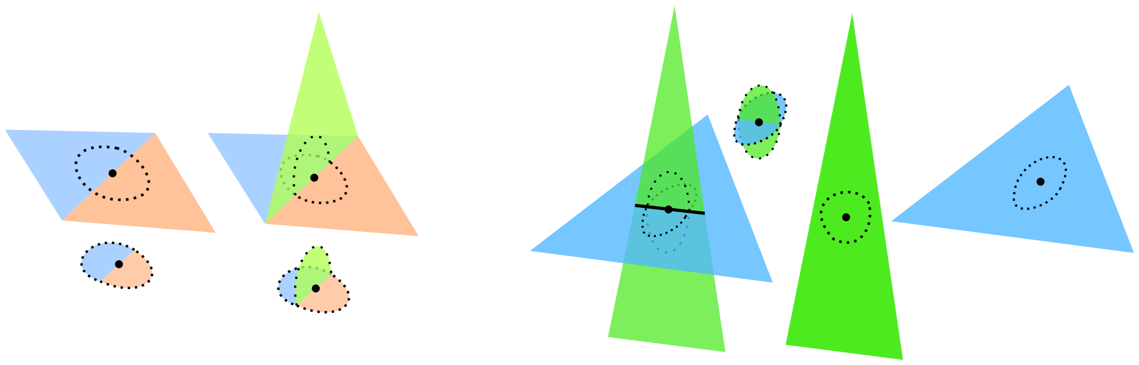


Figure 3.12: The pair on the left shows 2-manifold and non-manifold object. A neighborhood around any point has to be homeomophic to 2D euclidean space. The neighborhood around a point on a shared edge does not resemble 2D space for non-manifold. The right side shows two triangles intersecting each other. This can still be considered as 2-manifold for following algorithms. Two triangles represent separate surfaces.

A model can contain degenerated or near degenerated triangles, see the Figure 3.13.

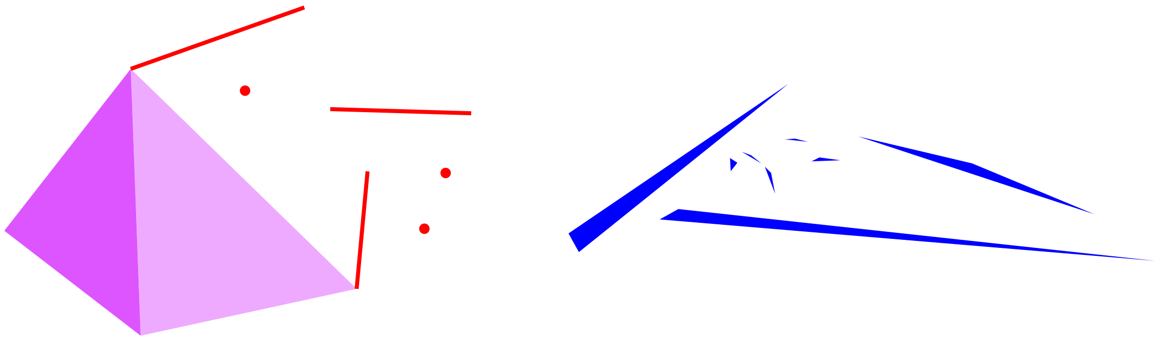


Figure 3.13: If a triangle contains two or three vertices that are located at the same position, the triangle is degenerated. Triangles degenerate into lines or points. Small triangles or triangles with very small interior angles are near degenerated. Such triangles can cause several problems due to floating-point arithmetic.

Finally, following shadow algorithms consider the most general models that can contain arbitrary geometry described in this section, see the Figure 3.14.

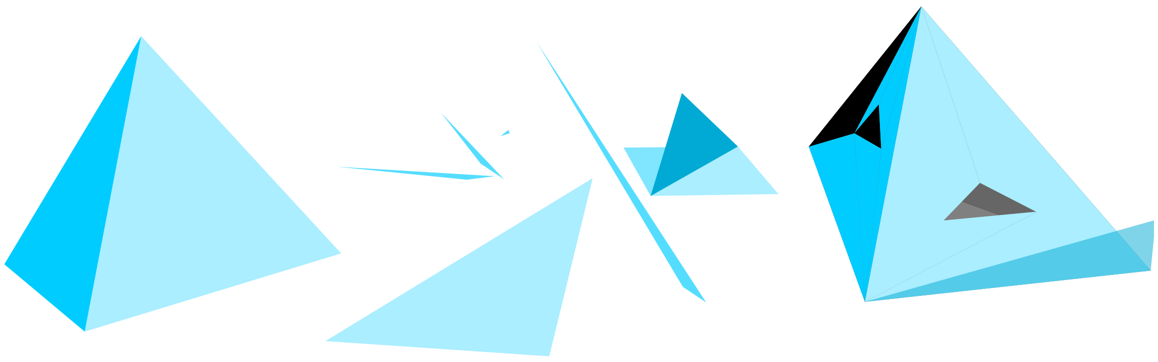


Figure 3.14: The image shows the most general model. A model does not have to be watertight, oriented nor 2-manifold. It can also contain degenerated or near degenerated triangles. Robust algorithm have to cope with every possible geometric configuration.

3.3 Shadow Volumes

This section describes shadow volume algorithms for computing precise hard shadows. Additional information about shadow volumes can be found in the sections 2.3 Shadow Volumes of the book Real-Time Shadows [25].

The concept of shadow volumes (or shadow polygons) was first introduced in 1977 by Franklin Crow [17]. A shadow volume is a volume in the space that contains a shadow. It is usually defined as a set of polygons enclosing shadowed space. If a point in space (view-sample) lies within any shadow volume, it is shadowed. There are several ways of shadow volume construction. There are also several ways of using shadow volumes for shadow rendering.

3.3.1 Shadow Volume Construction

The most simple way of shadow volume construction involves homogeneous coordinates as shown in Figure 3.15.

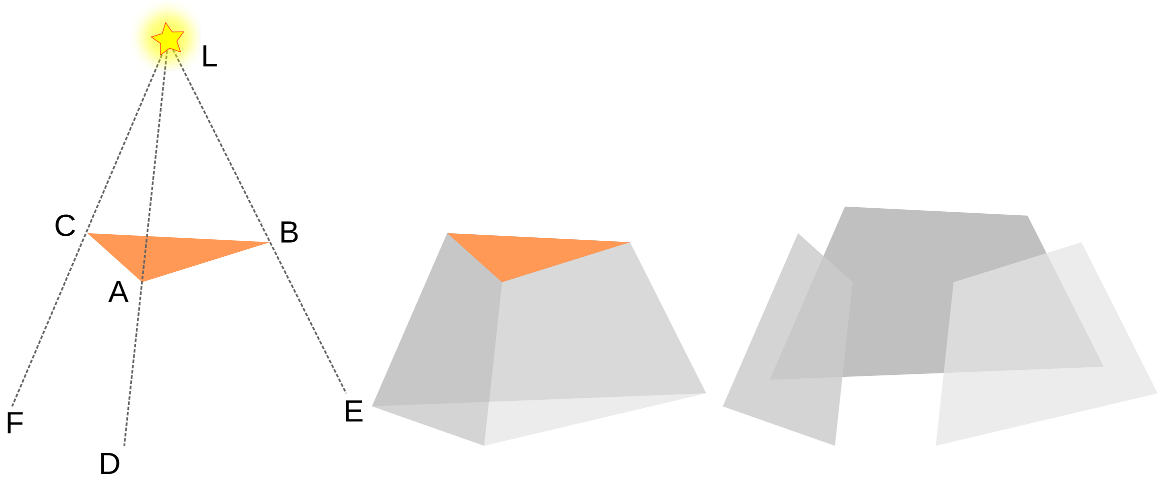


Figure 3.15: The left side of the image shows a triangle ABC and point light source L. The middle image shows the shadow volume. The right side of the image shows exploded shadow volume. Points D, E and F lie in infinity and are constructed using the light source and triangle points. Let $P = (p_x, p_y, p_z, 1)$ be a vertex of the triangle and $L = (l_x, l_y, l_z, 1)$ be the light position in homogeneous coordinates. The corresponding point P' in infinity is defined as $P' = P - L = (p_x - l_x, p_y - l_y, p_z - l_z, 0)$. This construction is applicable only for omnidirectional point light sources. If the light is directional, the construction has to be modified, see Figure 3.16.

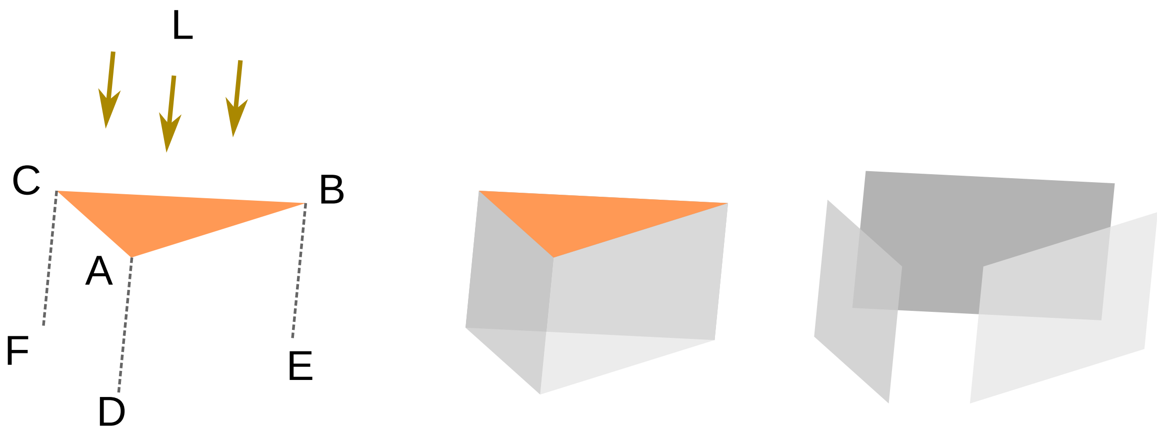


Figure 3.16: The left side of the image shows a triangle ABC illuminated using directional light source L. Let $P = (p_x, p_y, p_z, 1)$ be a vertex of the triangle and $L = (l_x, l_y, l_z, 0)$ the light position in homogeneous coordinates. The light from light sources illuminates the scene in direction $-L$. The corresponding point P' in infinity does not depend on P and can be constructed using formula $P' = -L$. The point P' coordinates are therefore $P' = (-l_x, -l_y, -l_z, 0)$. The construction of shadow volume for directional light sources and omnidirectional light sources (Figure 3.15) can be unified. Without loss of generality, let $L = (l_x, l_y, l_z, l_w)$ be a light source. If $l_w = 1$ then the light source is point light source. If $l_w = 0$ then the light source is directional light source. The corresponding vertex P' in infinity is constructed using $P' = (p_{xyz}l_w - l_{xyz}, 0) = (p_x l_w - l_x, p_y l_w - ly, l_z l_w - l_z, 0)$. The orientation of shadow volume polygons is important, otherwise shadows will not be correctly computed, see Figure 3.17.

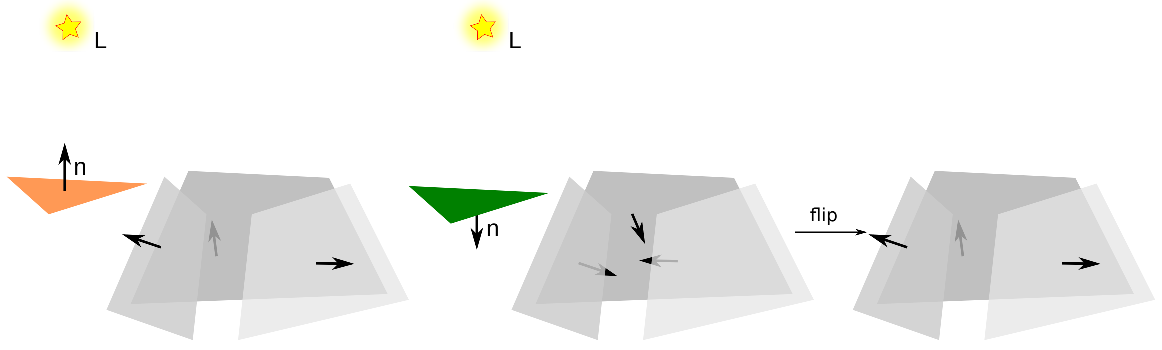


Figure 3.17: The figure shows two triangles, an orange and a green and their corresponding shadow volume polygons. The normal of orange triangle is facing the light source L. The normal of green triangle is back-facing the light source. All shadow volume polygons have to be oriented in the same way with normals pointing either inside or outside. The green triangle is back-facing the light source, therefore the orientation of its polygons is incorrect. Shadow volumes of back-facing triangles need to be flipped. When a shadow volume is constructed for every triangle in the scene, the shadow volume algorithm can compute the shadow, see Figure 3.19.

3.3.2 Naive Algorithm

One of the first implementations of shadow volumes in hardware was presented by Fuchs et al. [32] and can be seen in Algorithm 2 and Figure 3.18.

Algorithm 2: This algorithm describes modified implementation of shadow volumes using Pixel-Plane machines by Fuchs et al. [32]. The description of the algorithm can be found in section Naive Shadow-Volume Algorithm [25]. The main idea is to rasterize a full screen quad for each triangle and test underlying view-samples against shadow volume. The algorithm was implemented using Pixel-Plane machine that allowed fast evaluation.

```
   Data: scene, light
   Result: shadowMask
1  clear(shadowMask);
2  viewSamples := rasterize(scene);
3  foreach triangle in scene do
4      shadowVolume := computeShadowVolume(triangle, light);
5      sampleCoords := rasterizeFullScreenQuad();
6      foreach coord in sampleCoords do
7          viewSample := viewSamples[coord];
8          if viewSample is inside shadowVolume then
9              shadowMask[coord] = 1;
10 return shadowMask
```

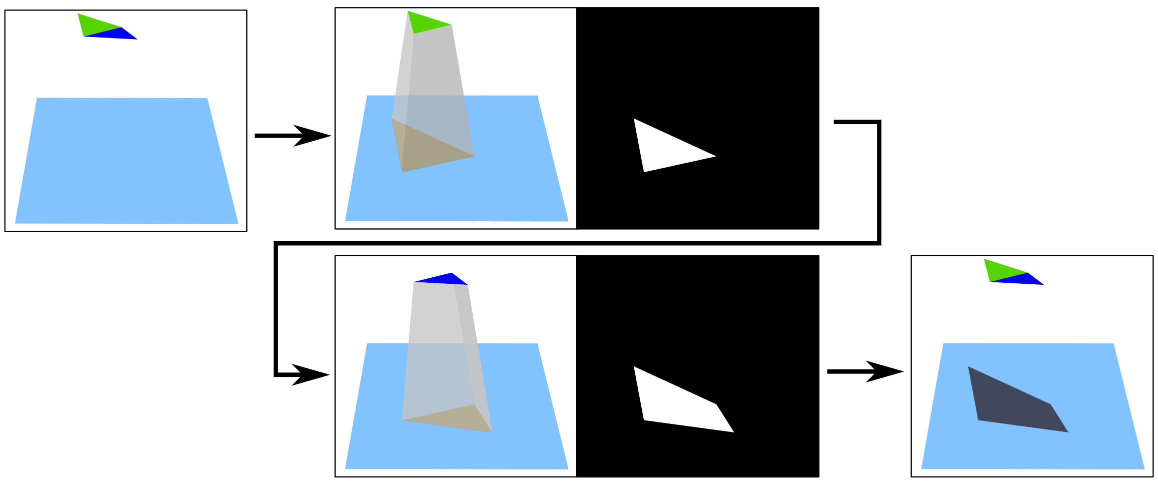


Figure 3.18: This image shows hardware implementation of shadow volumes using Pixel-Planes. The left part shows the scene without shadows from camera point of view. The middle part sequentially processes each triangle in the scene and creates shadow mask. The right side shows the final image.

3.3.3 Z-pass Stencil Algorithm

Z-pass algorithm was implemented using stencil buffer in 1991 by Tim Heidmann [41]. The basic idea behind stencil algorithm is to count how many times does a ray from camera intersect front-facing and back-facing shadow volume geometry. The counting is performed during rasterization of shadow volume geometry into stencil buffer. If fragment *passes* the depth test, it increases or decreases stencil value depending on the fragment orientation. The stencil buffer is a memory on GPU that can store integer values (usually 8-bit values) for each screen-space sampling location. The stencil buffer and specialized hardware allows atomic counting of rasterized fragments. The algorithm is visualized in Figure 3.19

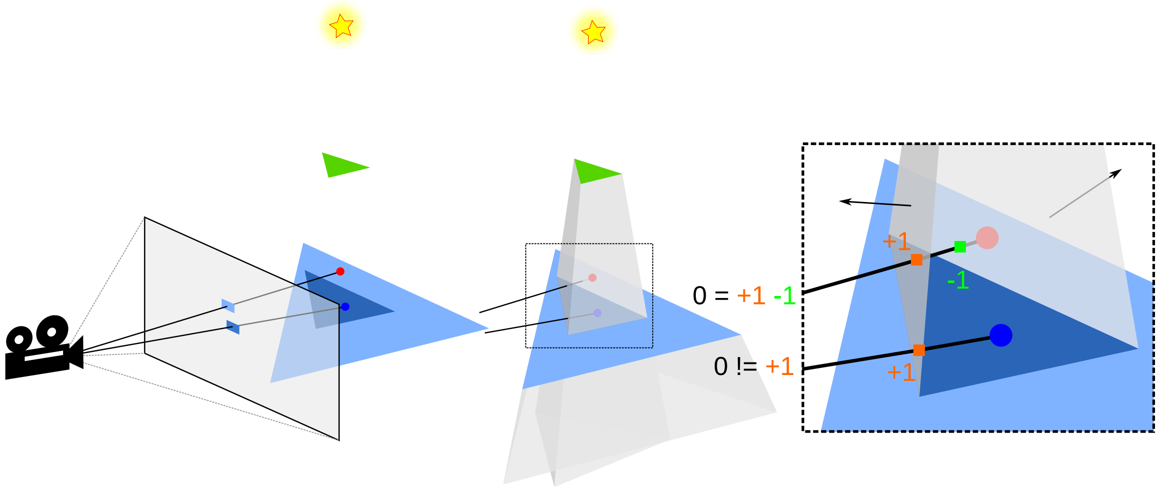


Figure 3.19: This figure visualizes the Z-pass shadow volume algorithm. The left part of the image shows a scene with two triangles - blue and green. The scene is illuminated by point light source. The camera views the scene and renders two view-samples - red and blue points. The blue view-sample is shadowed. The middle part of the image shows shadow volumes of both triangles. The right part of the image shows the shadow volume algorithm. The ray from camera to the blue view-sample intersects one front-facing shadow volume polygon. The stencil value of the blue sample is +1. The ray from camera to the red view-sample intersects one front-facing and one back-facing shadow volume polygon. The stencil value of the red sample is 0. Each time a ray from camera intersects front-facing shadow volume polygon, the stencil value is incremented and each time a ray from camera intersects back-facing shadow volume polygon, the stencil value is decremented. This is the reason why the orientation of shadow volume polygon is important, as stated in Figure 3.17. If the stencil value is not equal to 0, then the view-sample is shadowed. Z-pass algorithm is not without a problem, see Figure 3.20.

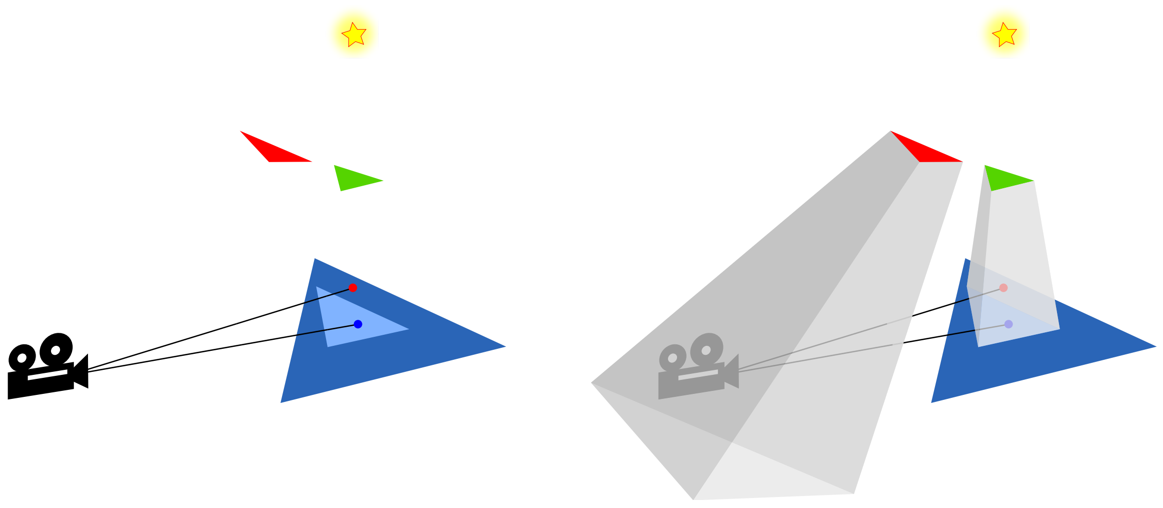


Figure 3.20: This figure shows one of the problems of Z-pass algorithm. The left part of the image shows the same scene, as in Figure 3.19, but with one additional red triangle. The blue triangle is shadowed by green triangle. Shadows are inverted, parts of the scene that should be lit are shadowed and parts of the scene that should be shadowed are lit. The right part of the image shows shadow volumes of red and green triangle. The Z-pass algorithm, in the Figure 3.19, counts wrong stencil values for red and blue samples. The counting is biased by shadow volume from blue triangle that decreases stencil value by 1. One of the solutions to this issue is to count the number of times the camera is inside a shadow volume and modify stencil values accordingly. This might work, but the problem is more complicated, see Figure 3.21.

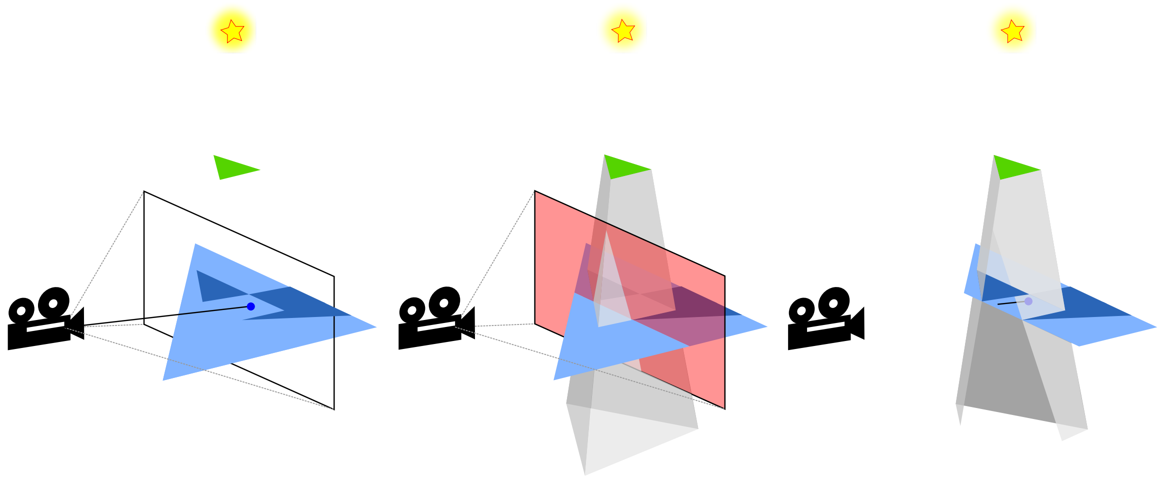


Figure 3.21: This figure shows full impact of problem with Z-pass algorithm. The left part of the image shows the same scene, as in Figure 3.20, but the camera is much closer to triangles. The shadow is not trivially inverted as can be seen in Figure 3.20. In the middle part of the image, the near plane of the camera intersects the scene and shadow volume geometry. The right part of the image shows clipped triangles and shadow volume geometry. A view-ray does not originate in the center of projection but it originates on the near plane. The blue view-sample is lit, because the ray does not intersect any non-clipped shadow volume polygon.

3.3.4 ZP+ Algorithm

ZP+ algorithm aims to solve Z-pass algorithm problems (see Figure 3.21). The algorithm was developed by Hornus et al. [43]. The main idea behind this approach is to project geometry onto the camera near plane from light position, see Figure 3.22.

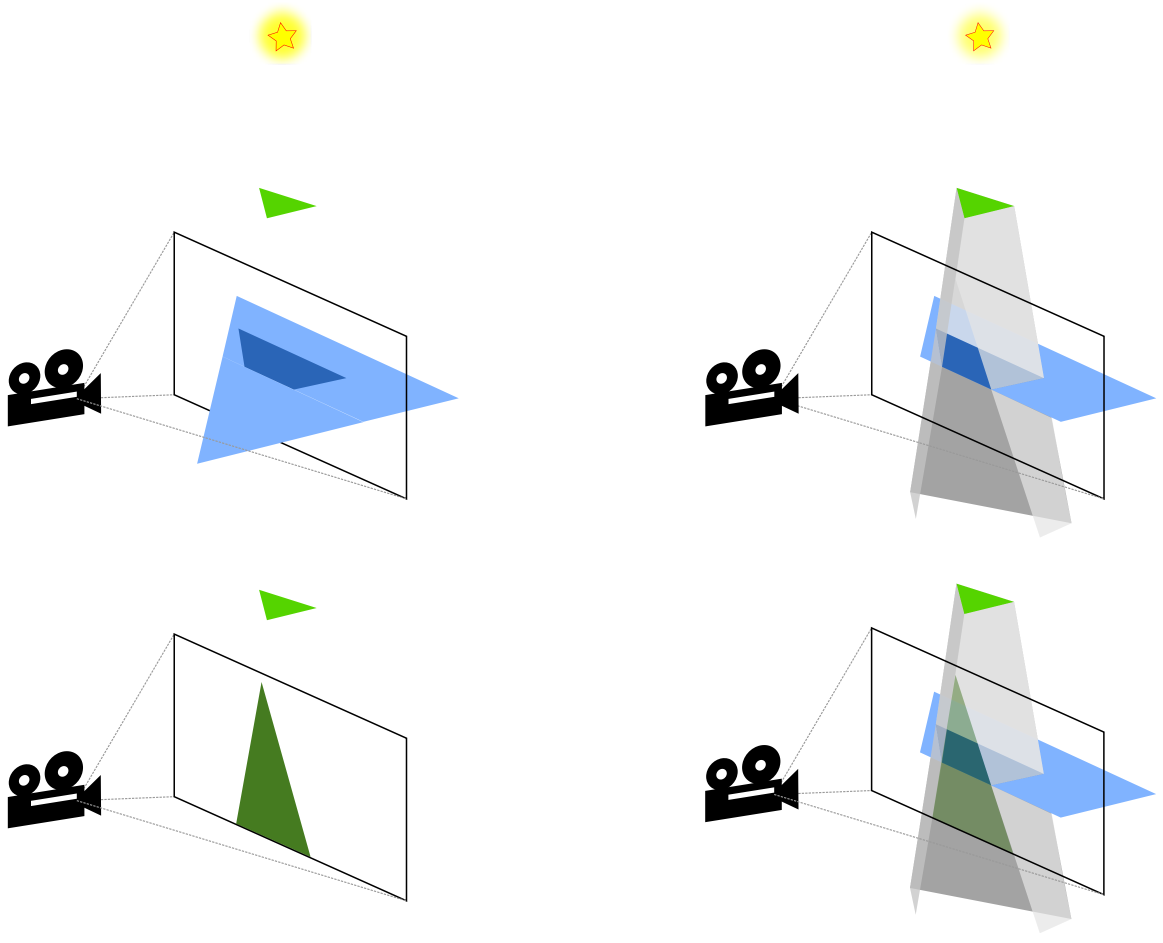


Figure 3.22: The image shows ZP+ algorithm that solves robustness of Z-pass algorithm. The top left part of the image shows the same scene as in the Figure 3.21. The scene and the shadow volume is clipped by the camera's near plane, top right part of the image. In the bottom left part, the green triangle is projected onto the near plane using light position. The bottom right part of the image shows fixed shadow volume geometry.

The ZP+ algorithm computes correct shadows under certain conditions. The main problem of ZP+ algorithm is that it does not work if the light source is close to the near plane. If the light source is close to the near plane, the projection of the scene geometry onto the plane is heavily affected by numerical precision.

3.3.5 Z-pass with Near-plane Clamping

Another fix to basic Z-pass algorithm is to use ++ZP algorithm. The algorithm was introduced in the book Real-Time Shadows [25]. The near-plane clamping of shadow volumes can be seen in the Figure 3.23.

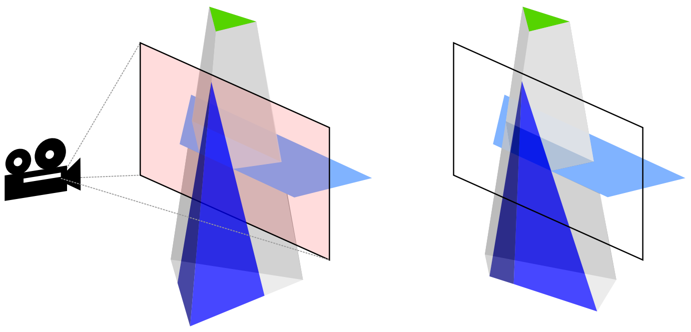


Figure 3.23: This image shows Z-pass algorithm with near-plane clamping supported by GPU hardware. The left part of the image shows the shadow volume intersecting the near-plane. Blue parts of the shadow volume should be clipped. If the near-plane clamping is active, blue parts of the shadow volume are projected onto the near-plane - the right side of the image. This mitigates one of the problems of the Z-pass algorithm, see Figure 3.21. However, triangles that do not intersect near-plane can still bias stencil value, see Figure 3.20. The solution for that can be seen in Figure 3.24.

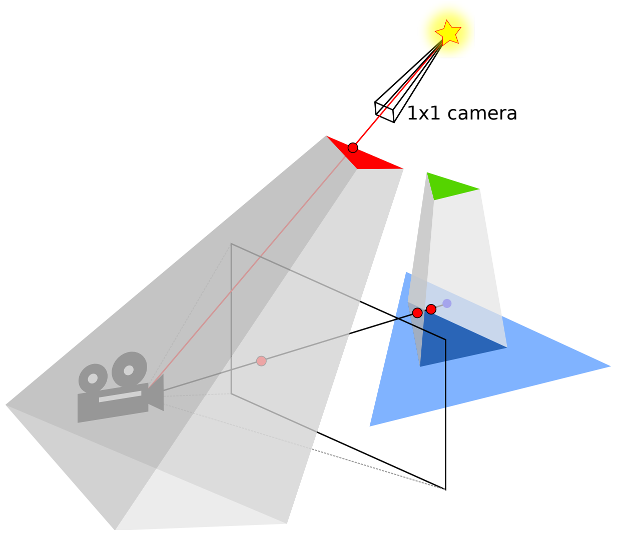


Figure 3.24: This figure shows ++ZP algorithm. In addition to near-plane clamping (Figure 3.23), a small 1x1 pixel size camera is created at the light position. The small light camera points to the original camera. This small light camera rasterizes scene polygons and counts the number of times the original camera is inside a shadow volume. This algorithm is still not completely robust, because it requires two different projections and rasterizations. Due to floating-point issues, the counting might be inconsistent.

3.3.6 Z-fail Algorithm

Another algorithm for computing shadows using shadow volumes is Z-fail algorithm. The Z-fail algorithm was first introduced by Bilodeau [9] and was also independently used in Doom 3 by Carmack [14]. It was also mentioned in GDC by Dietrich [21].

Z-fail algorithm solves issues with the basic Z-pass algorithm. Z-pass algorithm counts ray intersections with shadow volumes in front of the nearest surface to the camera. Z-fail algorithm counts ray intersections behind the nearest surface, see Figure 3.25.

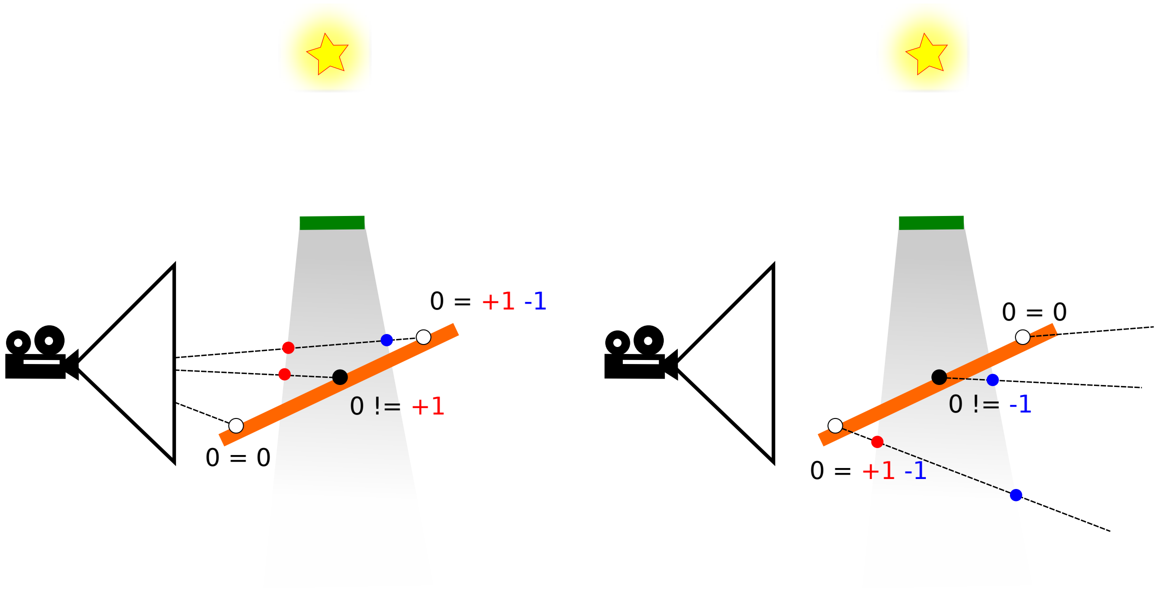


Figure 3.25: This image shows the difference between Z-pass and Z-fail algorithm. Z-pass algorithm on the left counts ray intersections with shadow volume that are closer to the camera than the nearest surface. Z-fail algorithm on the right counts ray intersections that are farther than the nearest surface. As can be seen, the different way of intersection counting is interchangeable. The Z-fail algorithm still needs modification in order to work properly, see Figure 3.26.

Z-fail algorithm requires rasterization of triangles that are located in infinity. In order to avoid far plane clipping, the camera far plane has to be moved to infinity. Perspective matrices allow this modification [37] and Cass Everitt [28] showed such matrix. The matrix is constructed from standard projection matrix by limiting the far plane distance. The new matrix does not significantly affect depth precision [100], [55]. Today's hardware supports triangle clamping to the far plane so such modifications are not necessary.

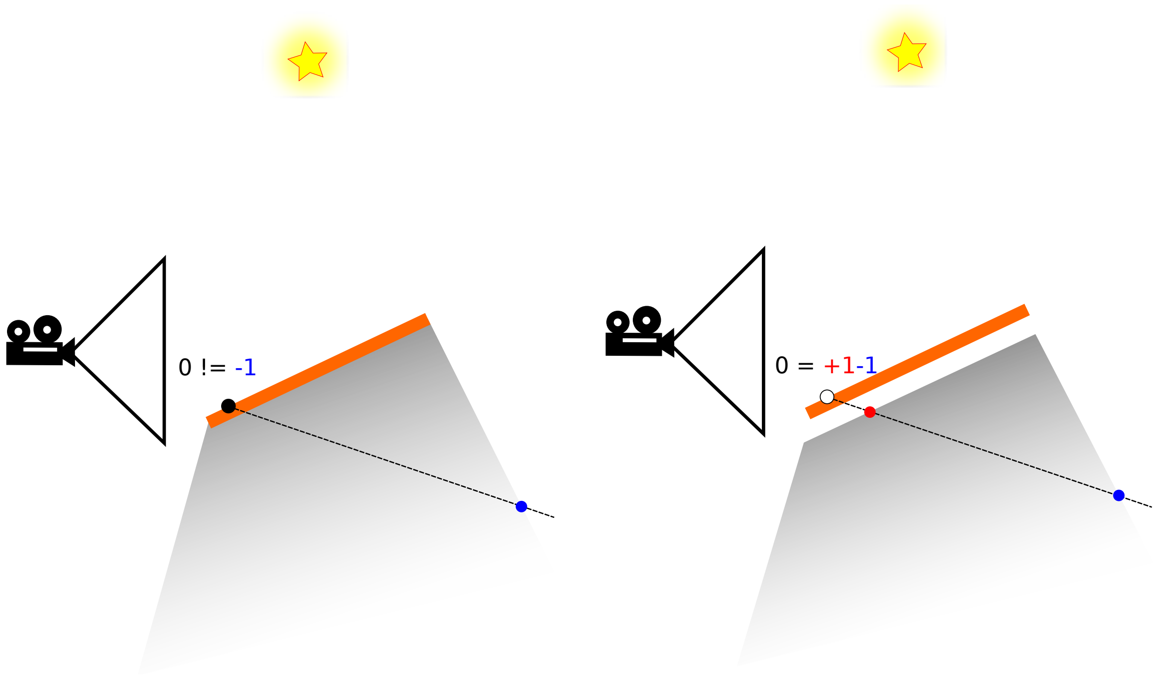


Figure 3.26: This image shows incorrect computation of shadows for Z-fail algorithm. On the left, the black sample is considered shadowed even if it is not inside any shadow volume. The stencil value is decremented once as the ray intersects back-facing shadow volume (blue point). The way to solve this issue is to modify definition of shadow volume geometry and introduce near caps. Near caps extend shadow volume polygons by shadow caster triangles. The right side of the image shows the shadow volume that is slightly shifted away from the triangle due to visualization purposes. The ray, in addition, intersects the front facing near cap which fixes stencil value. Near caps has to be located exactly at the same position as the shadow casting geometry. The introduction of near caps does not solve all issues, see Figure 3.27.

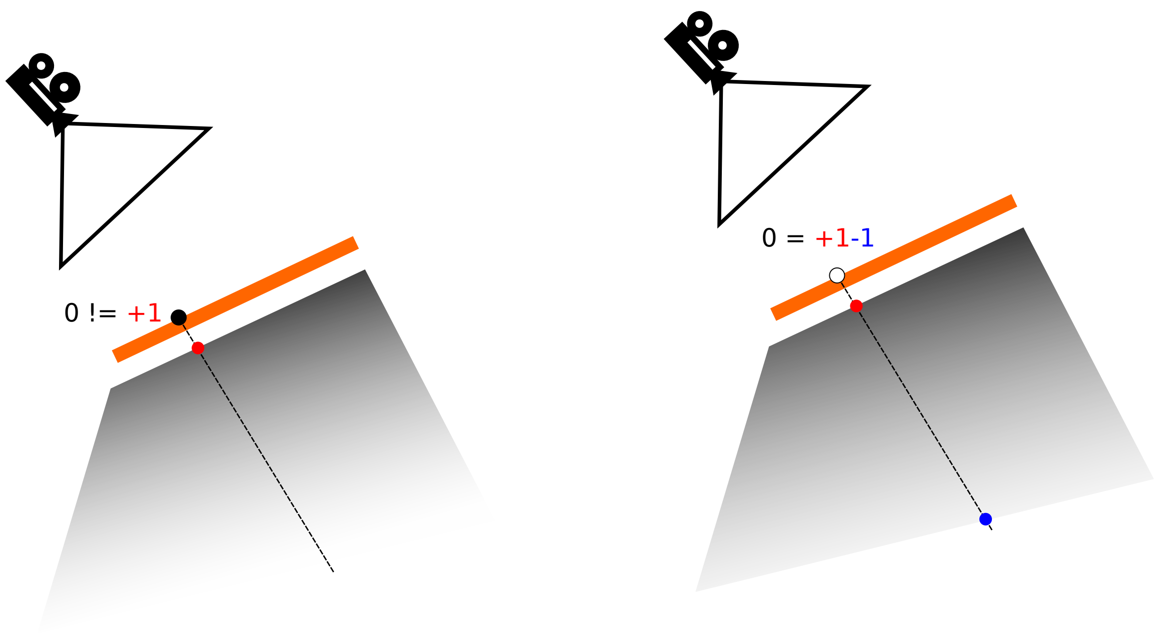


Figure 3.27: This image shows side effects of near caps for Z-fail algorithm. If the camera views the geometry from roughly the same position as the light source, the stencil value will never be decremented by back-facing shadow volume geometry. The way to solve this is to introduce far caps. Far caps, together with near caps close shadow volume geometry and create watertight volumes. While near caps have to lie at the same position as the shadow casting geometry, far caps have to be projected into infinity. The right side of the image shows Z-fail algorithm with introduction of both types of caps. The near cap is shifted away from the triangle and the far cap is not drawn in infinity for visualization purposes. The modified shadow volume definition can be seen in Figure 3.28.

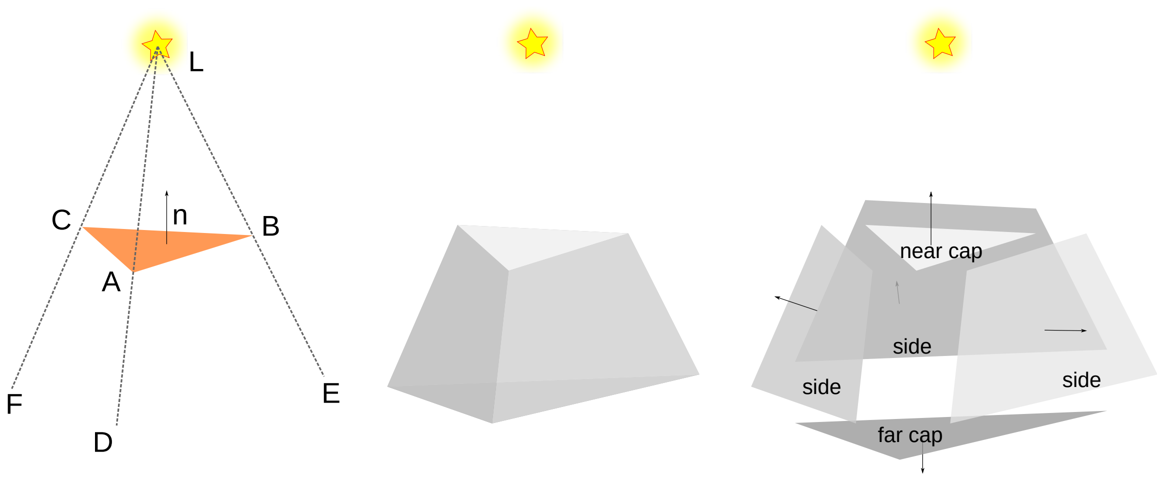


Figure 3.28: The figure shows extended definition of shadow volume geometry for Z-fail algorithm, see figure Figure 3.15. Shadow volume geometry is extended by near and far cap, (sometimes called front and back or light and dark caps). Near cap is identical to shadow casting triangle. Far cap is shadow casting triangle projected to infinity from light source.

3.3.7 Acceleration

Shadow volume algorithms require rasterization of big amount of fragments. They are view dependent, fill rate intensive algorithms. The research is aimed not only at robustness but also at acceleration. There are methods for reduction of geometry complexity as well as rasterization cost.

Subsamping

The easiest method for acceleration of shadow volume algorithms is to reduce resolution of shadows. This can be done by reduction of the stencil buffer resolution and upsampling of the resulting shadows. By sacrificing the quality, the resolution reduction provides roughly linear speedup as was shown by Röttget et al. [89].

Light Range and Attenuation

Another simple acceleration method is to compute shadows only in parts of a scene affected by a light source, see Figure 3.29.

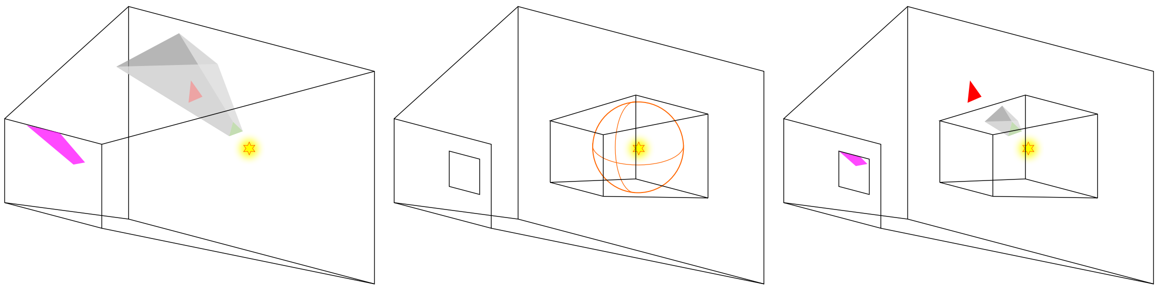


Figure 3.29: Light intensity decreases with distance. There are many ways how to attenuate a light, but the most common is quadratic falloff. Point light sources can be enclosed within a bounding sphere with finite radius. If a view-sample lies outside of the bounding sphere, the light contribution is smaller than the smallest displayable intensity difference. Therefore, any geometry outside the bounding sphere is in shadow. On the left side of the image, the camera frustum views the scene with two triangles. The shadow volume of green triangle is clipped by frustum planes and covers large portion of screen space (purple color). In the middle, the light is enclosed in the bounding sphere and in the bounding sub-frustum. The light contribution outside of the bounding volume is negligible. On the right side, the shadow volume is clipped by light's sub-frustum reducing screen space footprint. The two dimensional bounding quad using GPU was proposed by [60] and [70]. Later, it was extended to 3D bounding sub-frustum by [28] and [69].

Z-pass or Z-fail

Z-pass based algorithms are usually faster than Z-fail based algorithms. Z-fail algorithm requires raterization of capping geometry which increases the fill rate. However, Z-fail algorithm can be faster, if the number of changes to the stencil buffer is smaller. This can occur if most of shadow volumes are closer to the camera than the view-samples. Z-fail can be faster for certain subset of view-samples, see Figure 3.30, so the choice of algorithm can be made per view-sample.

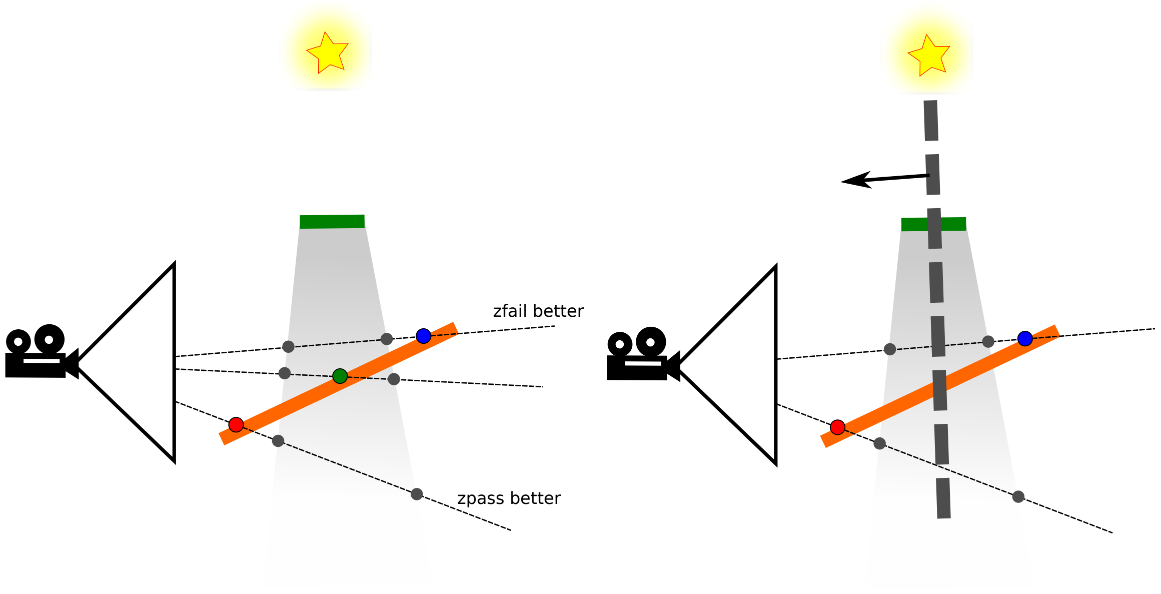


Figure 3.30: The image shows three view-samples, red, green and blue. The number of stencil buffer changes is smaller for red view-sample, if the selected algorithm is Z-pass. The number of stencil buffer changes is smaller for blue view-sample, if the selected algorithm is Z-fail. The right side shows possible position of split plane. If a view-sample is in front of split plane the algorithm chooses Z-pass. If a view-sample is behind, it chooses Z-fail.

Laine [56] proposed solution which choose the algorithm according to split plane depth and suggested possible split plane constructions. The algorithms requires unavailable hardware extensions in order to be viable and could be implemented with use of hierarchical depth buffer [38].

Level of Detail

The shadow volume rasterization is usually the slowest part, but sometimes, with complex geometry, it might by useful to accelerate geometry computation as well. Level of detail algorithms reduce geometric complexity of shadow volumes, but usually don't significantly affect the number of rasterized shadow volume samples. The topic of level of details and shadows can be traced to Clark [16], who proposed shadow quality adaptation. The quality adaptation is based on screen space shadow size. Level of detail is usually applied to shadow casters. However, simplified shadow caster geometry can create rendering artifacts, imprecise shadow borders or self-shadowing, see the Figure 3.31.

For shadow volumes, Zimoa [111] proposed mitigation which modifies shadow volume extrusion. Govindaraju et al. [36] also proposed level of details for shadows. Mattausch et al. proposed method that exploits hardware tessellation for smooth shadow borders [67].

Some today's video games use level of details for shadows. For example, The Witcher 3: Wild Hunt uses separate geometry for rendering view-samples and for creating shadows. The game uses shadow map based algorithms which renders simplified world geometry. This accelerates shadow rendering, but can also cause unwanted artifacts. In one instance, the geometry simplification process removed roofs of some houses leaving their interiors always lit by the sun.

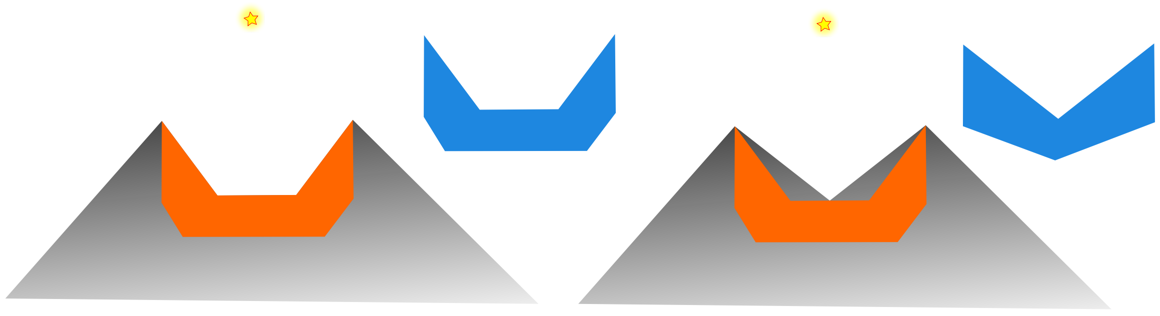


Figure 3.31: The image shows one of problems with level of details applied to shadow volumes. The left part shows orange object that casts shadows from shadow caster that is not simplified (blue). The right part shows the same object but the shadow volume is cast from simplified object (blue). The inner parts of orange objects are self shadowed by simplified shadow caster.

Culling

Another option of shadow acceleration is to cull shadow casters whose shadows do not influence final image. Casters that lie completely inside shadow volumes of other casters do not need to be processed. Likewise, casters that only influence invisible parts of a scene can be culled as well, see Figure 3.32. Some of the ideas were presented by Clark [16].

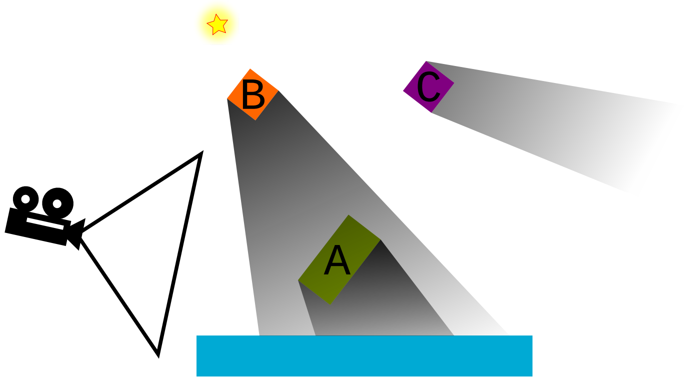


Figure 3.32: Shadow volume of the object A can be culled, because it is completely enclosed in shadow volume of the object B. Shadow volume of the object C can be culled, because it does not affect any visible shadow receivers.

Shadow volumes can be clipped by each other [6], [4]. Clipped shadow volumes contain less geometry which leads to performance improvements, see Figure 3.33. However, the clipping is costly operation making the approach unusable.

Approach by McCool [68] tries to reconstruct shadow volumes using shadow map. The approach produces many small shadow volume quads and also produces artifacts on shadow borders. Lloyd et al. [64] proposed casters culling that do not influence shadows. The method is based on depth and stencil buffer from a light point of view. Other method by Stich [97] uses culling based on hierarchical structures and visibility of bounding boxes' shadow volumes.

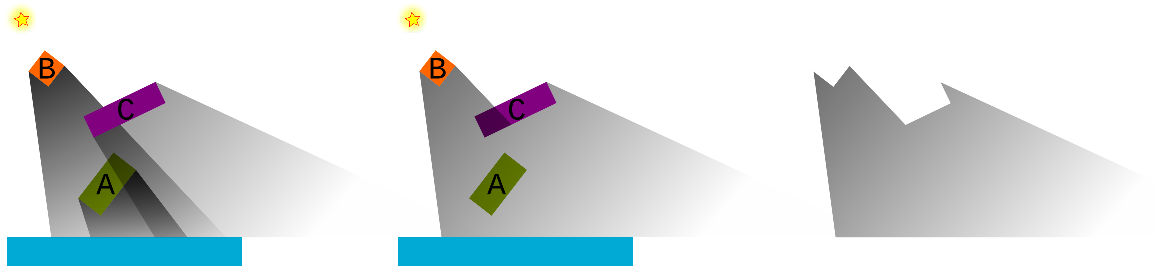


Figure 3.33: The left side shows 3 shadow volumes. Some shadow volume polygons are inside other shadow volumes. The middle and the right side shows clipped and merged shadow volumes.

Clamping and Clipping

Another viable way to to accelerate shadow volume rendering was presented by Lloyd [64]. The idea is to clip parts of shadow volumes that do not contribute to shadows, see the Figure 3.34

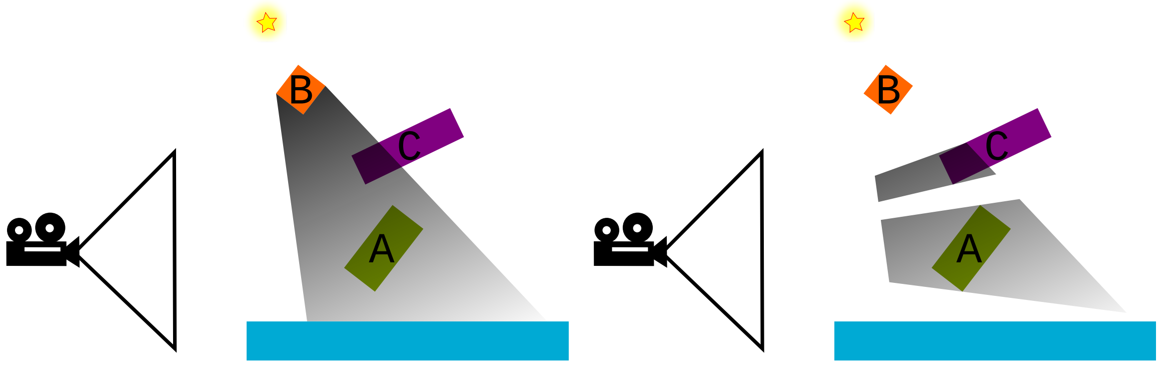


Figure 3.34: The left side shows full shadow volume of the object B. Some parts of the shadow volume cannot contribute to the shadow because they don't contain any shadow receiver geometry. The right side shows clipped and clamped shadow volume. The process creates more shadow volume polygons, but the fill rate is reduced creating balance between geometric and rasterization load.

Hierarchical Approaches

Shadow volume algorithms are fill rate intensive. The performance is lower for larger screen resolution. Some researchers focused on hierarchical rasterization approaches. Aila [1] proposed solution that tests 8x8 pixel tiles with shadow volumes. If a tile does not contain shadow border, a single pixel can be used for shadow computation. This can dramatically reduce the number of rasterized samples. Only boundary tiles require full rasterization, see Figure 3.35. However, the solution requires specialized hardware modifications in order to be viable.

The idea was further developed by Sintorn et al. [93]. Instead of using 2 level hierarchy with 8x8 pixel tiles, they use full hierarchical depth and stencil buffer, see Figure 3.36. They use tiles with 8x4 or 4x8 subtiles, this ensured good mapping to GPU warp size. While the

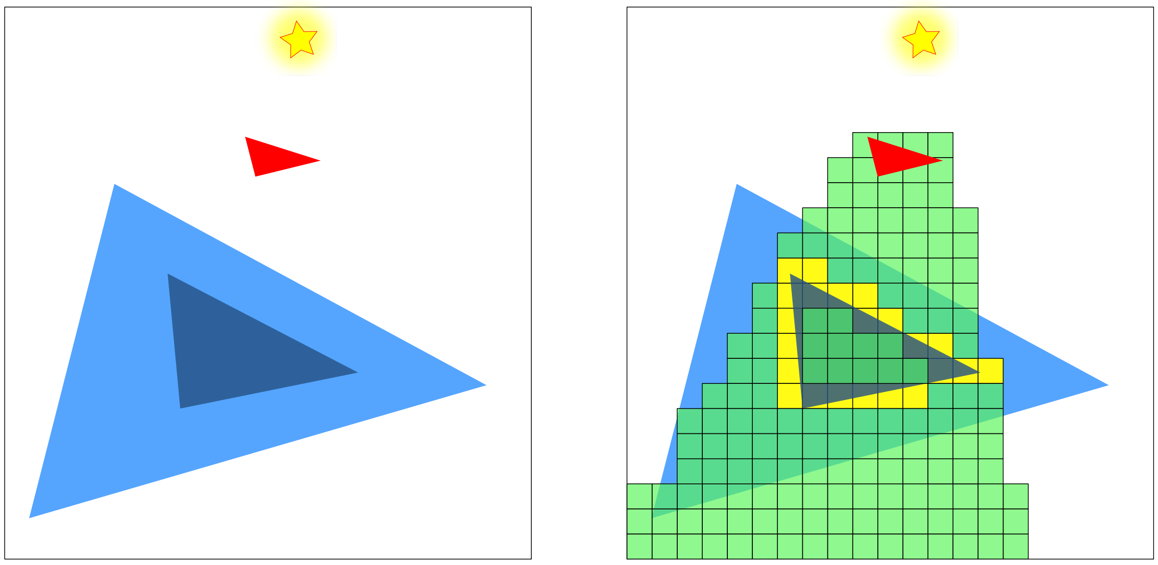


Figure 3.35: The image shows Aila's algorithm, where only boundary tiles require full rasterization. The left part shows a simple scene with two triangles. The right part shows shadow volume rasterization. Green tiles require single pixel to correctly compute shadow. Yellow tiles are rasterized in full resolution.

solution reduces number of changes to shadow mask, it is only starting to be viable for large resolutions (4k x 4k).

Sintorn et al. [92] also proposed an extension. The extension builds view-sample cluster hierarchy and use full 3D test for cluster elimination. Cluster hierarchy is built using morton code, see Figure 3.37.

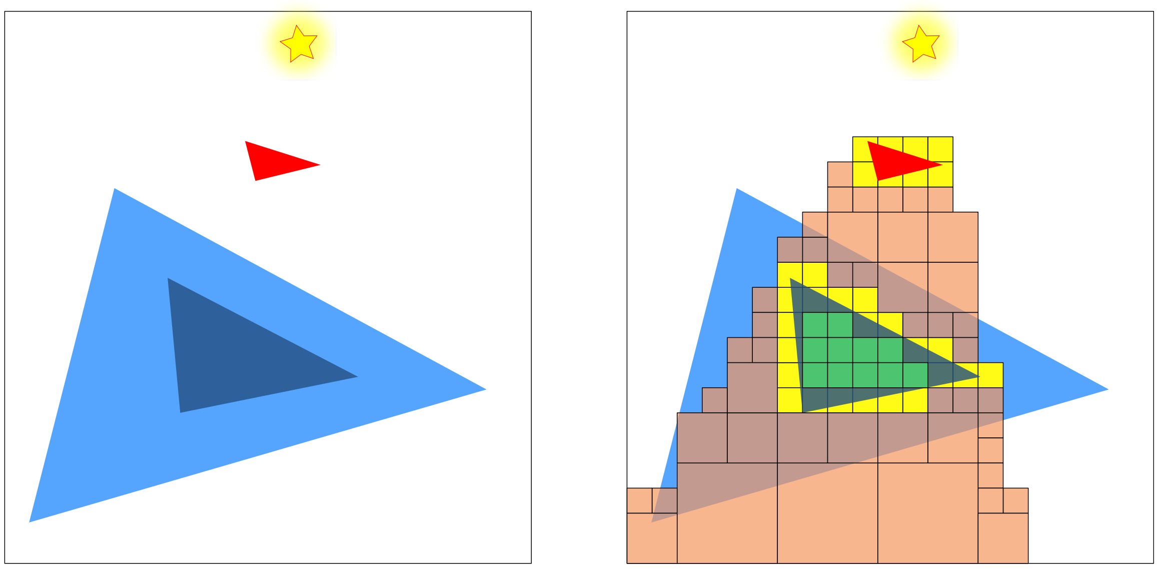


Figure 3.36: The image shows Sintorn's hierarchical CUDA based shadow volume rasterization. The idea is based on testing depth buffer tiles with shadow volumes. If a tile lies outside of a shadow volume, it does not need to be traversed deeper. If a tile lies inside a shadow volume, corresponding bit is set in hierarchical stencil shadow mask. Only if a tile intersects a shadow volume, it needs to be traversed deeper. Green tiles are trivially accepted as they lie inside the shadow volume. Orange tiles are trivially rejected. Yellow tiles intersect shadow volume and need to be traversed deeper.

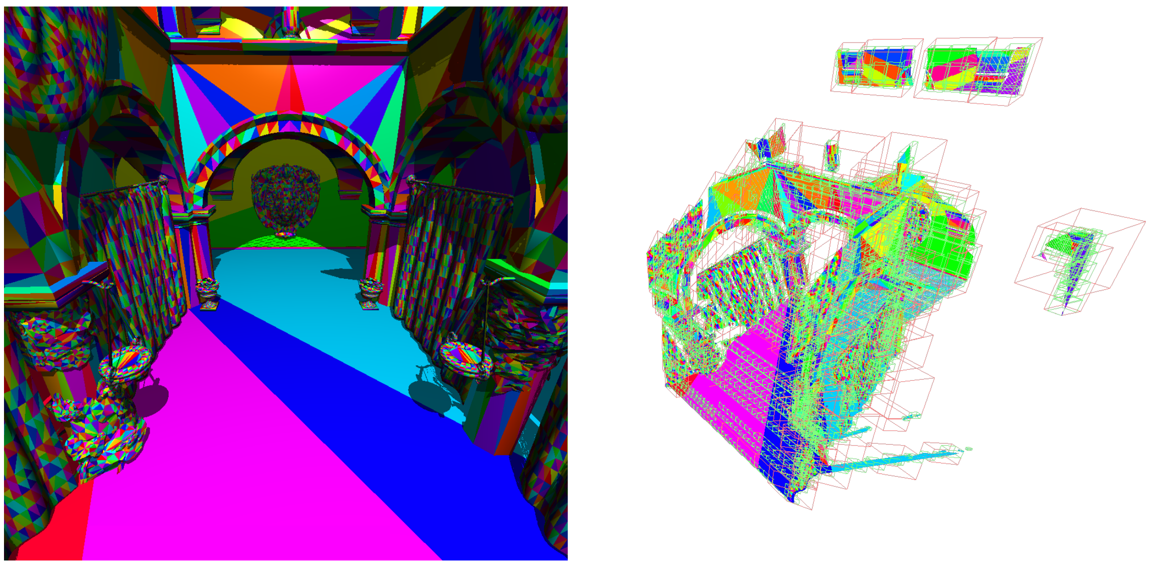


Figure 3.37: The left part shows Sponza scene from camera point of view. The right part shows view-samples in 3D space and two selected levels for view-sample hierarchy. Sintorn's algorithm for every triangle shadow volume traverses the hierarchy and compute its contribution to parts of the shadow.

3.3.8 Silhouette Algorithms

So far, all described shadow volume algorithms use per-triangle shadow volume geometry. This works well, but the performance of shadow volume algorithms is reduced, see Figure 3.38.

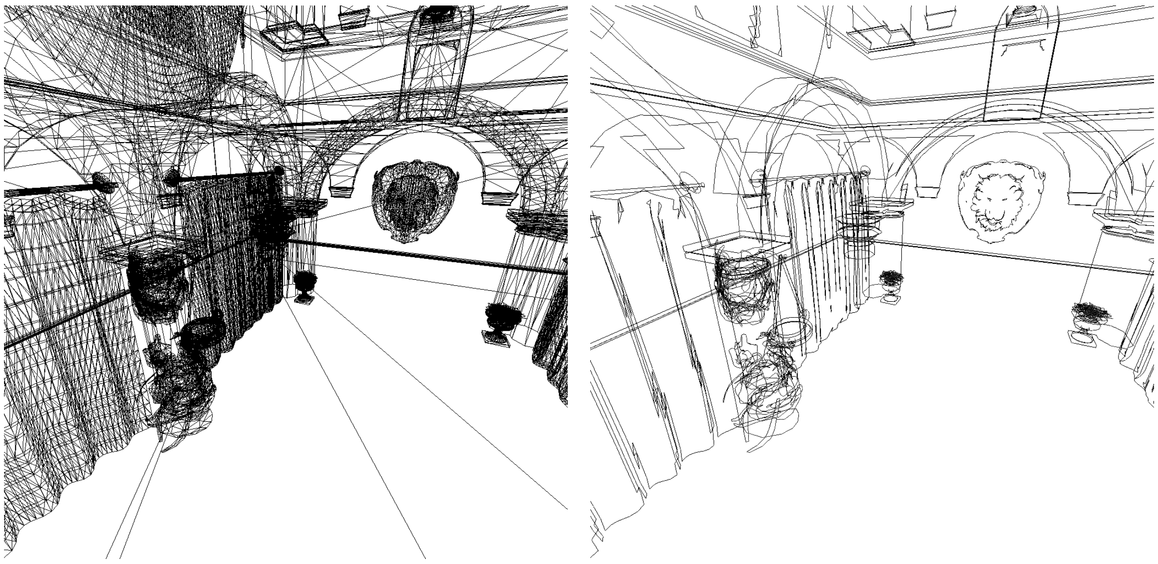


Figure 3.38: The left side shows edges in Sponza scene and the right side shows silhouette edges. The number of silhouette edges is much smaller which improves shadow volume performance.

Shadow volume rasterization is commonly the slowest part. Per-triangle shadow volumes contain unnecessary polygon which have to be rasterized, see Figure 3.39.

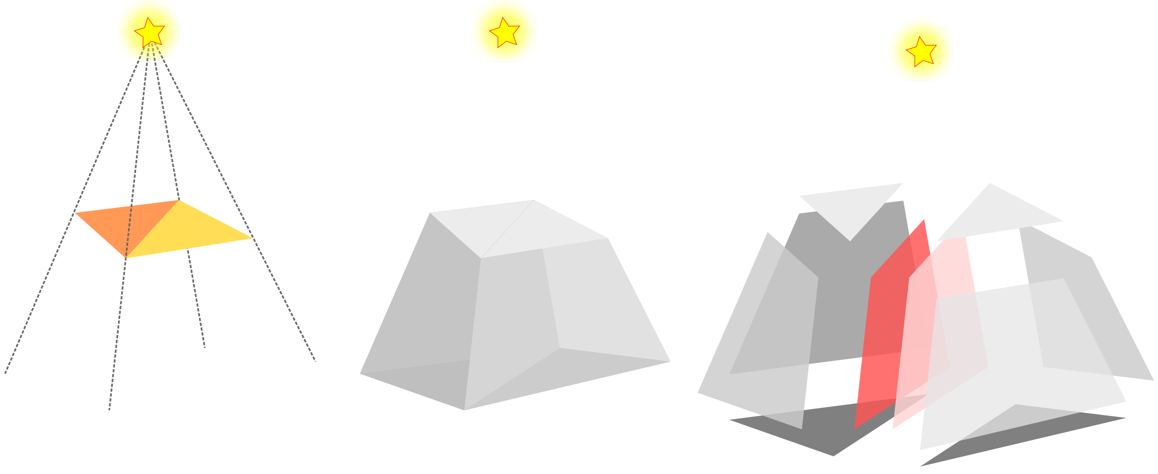


Figure 3.39: The figure shows two triangles and their corresponding shadow volumes. Red polygons are identical, but they have different orientation. A ray from camera has to intersect both or none. Therefore, the stencil value is decremented and incremented at the same time leading to no change. Silhouette based algorithms remove such polygons and reduce rasterization cost, see Figure 3.40.

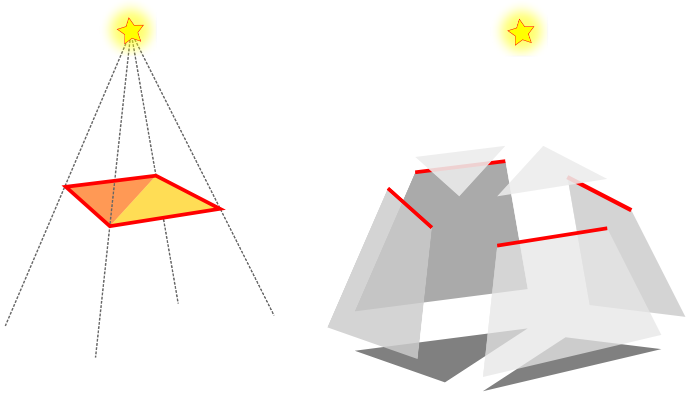


Figure 3.40: The image shows two triangles and the silhouette shadow volume. The red edges are silhouettes and only these silhouette edges are used during sides construction. By removing unnecessary edges, the shadow volume becomes simpler and the rasterization cost is reduced.

The Shadow Volume algorithm, in its basic form, constructs shadow volume from every triangle in a scene. Therefore, almost all of shadow volume sides are rasterized twice. Shared edges produce same shadow volume sides multiple times. A shared edge among two triangles produces two identical sides with same or opposite orientation, see Figure 3.41.

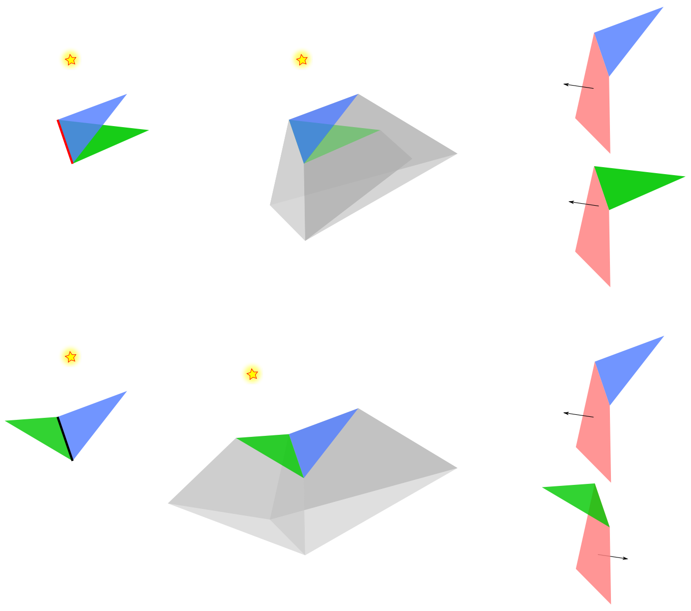
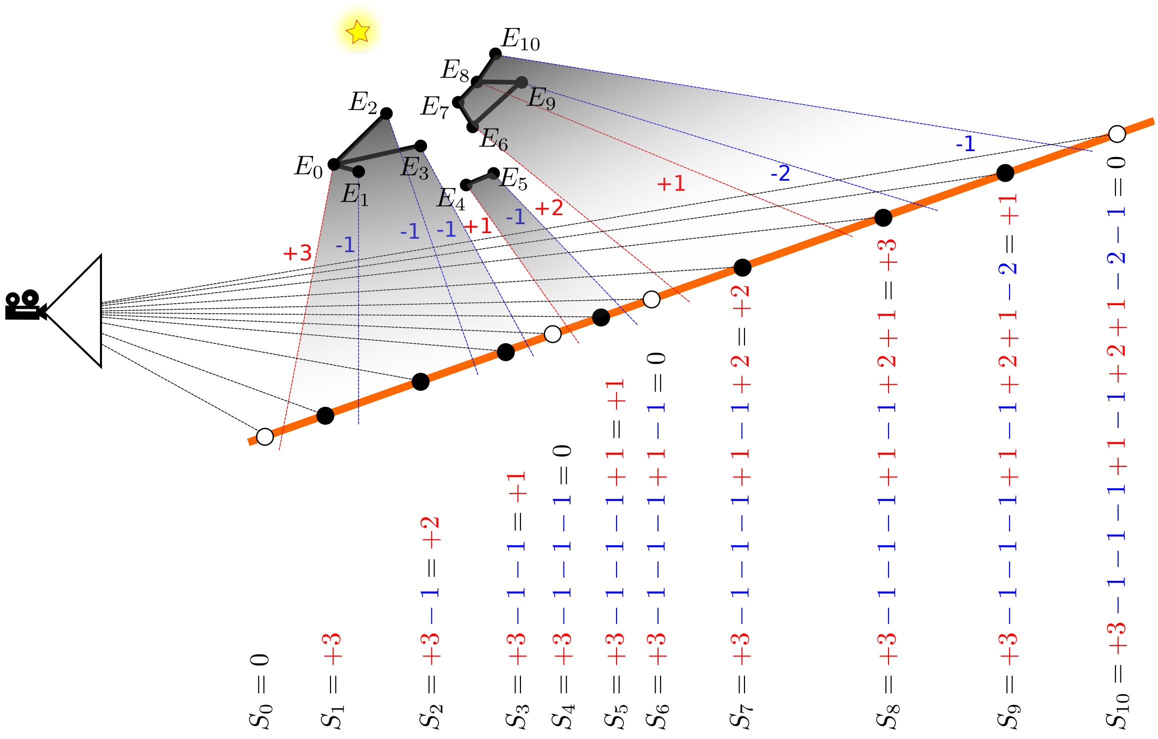


Figure 3.41: The top row shows two triangles that share an edge. The edge is a silhouette edge, because shadow volume sides have the same orientation. The bottom row shows a non-silhouette edge.

Non-silhouette edges can be skipped during rasterization. A silhouette edge should be shared by two triangles, one front facing a light and the other back facing the light. If a model is 2-manifold, watertight and oriented, this assumption is correct. The assumption breaks if one of the conditions is not true. The research over the years incrementally extended the silhouette extraction algorithms to more general cases. One of the first implementations in hardware was proposed by Brennan [13] and Brabec [12]. McGuire [70] implemented the whole algorithm in vertex shader. Van Waveren in 2005 [105] extracted silhouettes using SSE instructions on CPU for DOOM 3. Geometry shader based solution was proposed by [97]. Many of the solutions requires 2-manifold models. Bergeron [8] focused on manifold objects with boundary edge cases. Aldridge [3] further removed constraints and proposed algorithm for non-manifold oriented meshes. Finally, Kim [52] presented algorithm for any non-manifold object. Overview of silhouette algorithms is provided by Kolivand [54].

The most general algorithm by Kim Byungmoon in 2008 [52] is able to extract silhouettes from arbitrary triangle models. The principle of the algorithm is visualized in Figure 3.42 and in Figure 3.44.



E_{10}
E_8
E_9
E_7
E_2
E_6
E_0
E_3
E_1
E_4
E_5
-1
-2
+1
+2
+1
-1
-1
-1
+3
-1
$S_0 = 0$
$S_1 = +3$
$S_2 = +3 - 1 = +2$
$S_3 = +3 - 1 - 1 = +1$
$S_4 = +3 - 1 - 1 - 1 = 0$
$S_5 = +3 - 1 - 1 - 1 + 1 = +1$
$S_6 = +3 - 1 - 1 - 1 + 1 - 1 = 0$
$S_7 = +3 - 1 - 1 - 1 + 1 - 1 + 2 = +2$
$S_8 = +3 - 1 - 1 - 1 + 1 - 1 + 2 + 1 = +3$
$S_9 = +3 - 1 - 1 - 1 + 1 - 1 + 2 + 1 - 2 = +1$
$S_{10} = +3 - 1 - 1 - 1 + 1 - 1 + 2 + 1 - 2 - 1 = 0$

Figure 3.42: The image shows principle of Kim's algorithm [52]. Points E_0 - E_{10} represent edges and black line segments represent triangles. The only non-silhouette edge is edge E_7. Red and blue dashed lines represent shadow volume sides. During rasterization, shadow volume sides do not simply increment or decrement stencil value, but they add a value called multiplicity m. The algorithm computes multiplicity value for every edge using light-plane, see the Figure 3.43. Multiplicity value is in range $[-n, n]$, where n is the number of triangles connected to an edge. Multiplicity value is computed by examining every triangle connected to an edge. If a triangle lies in front of edge's light plane, multiplicity is incremented. Multiplicity is decrement for triangles that are behind the light plane. Light plane is constructed using edge vertices and light position. Multiplicity values of E_0, E_7, E_8 and E_9 are $+3$, 0, $+1$ and -2. If $m \neq 0$, the edge is silhouette. The algorithm requires rasterization of shadow volume sides multiple times in order to modify stencil value by m. Some hardware support incrementation and decrementation of stencil value by arbitrary value.

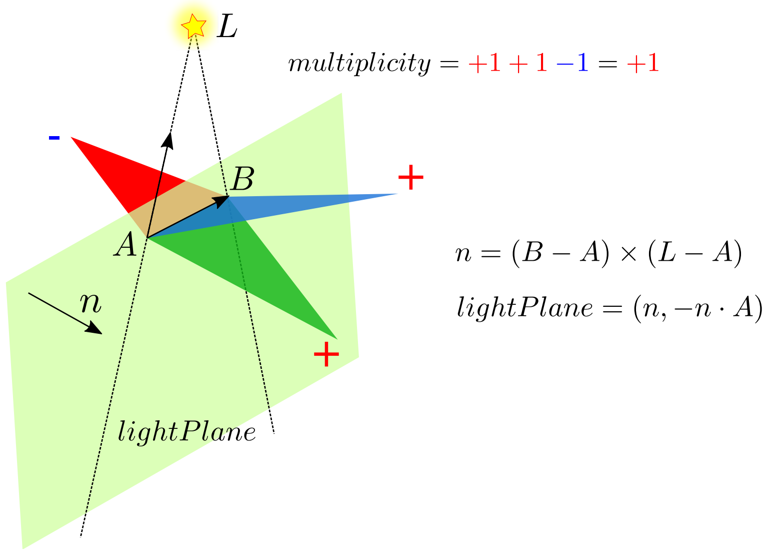

L
$multiplicity = +1 + 1 - 1 = +1$
-
B
+
A
$n = (B - A) \times (L - A)$
n
$lightPlane = (n, -n \cdot A)$
+
lightPlane

Figure 3.43: The image shows light-plane and multiplicity computation. The edge is shared by 3 triangles, two are in front of the light-plane and one is behind. The multiplicity value is +1.

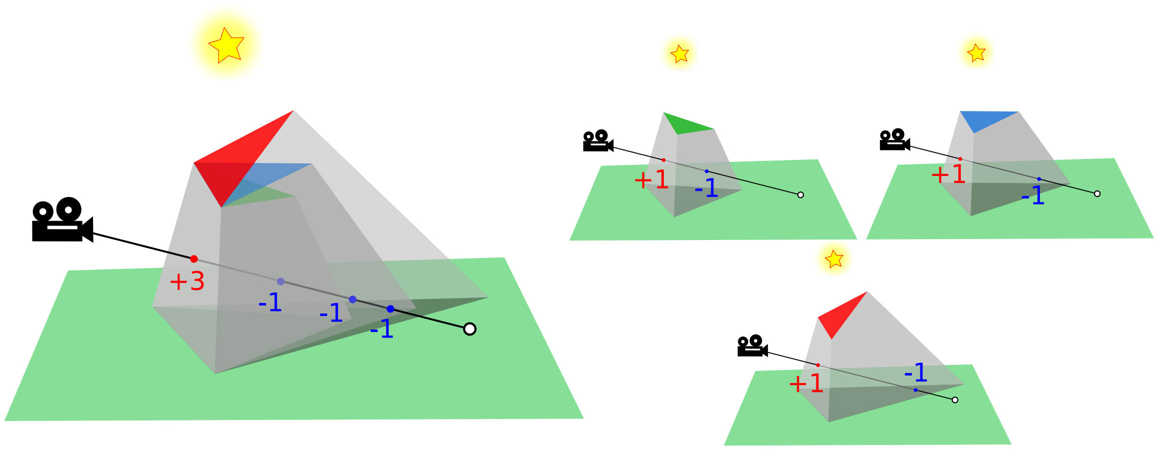


Figure 3.44: The image shows the idea of Kim's algorithm. Three triangles share an edge with multiplicity +3. A ray from camera colides with silhouette shadow volume and computes stencil value, left side. This can be imagined as three separate triangles with per triangle shadow volumes, right side.

While Kim's algorithm supports arbitrary geometry, it still suffers from visual artifacts if special care is not taken during GPU implementation. Visual artifacts arise when the algorithm computes multiplicity. The multiplicity value has to be consistent among all three triangle edges. The consistency ensures that shadow volume sides have proper orientation, see Figure 3.45.

Eisemann et al. [25] suggested an algorithm which precomputes triangle planes and creates edge list. The edge list contains all edges with adjacent precomputed triangle planes. Precomputed planes ensured correct multiplicity computation. Pečiva et al. [85] proposed similar edge list structure. However, the solution does not require precomputed triangle

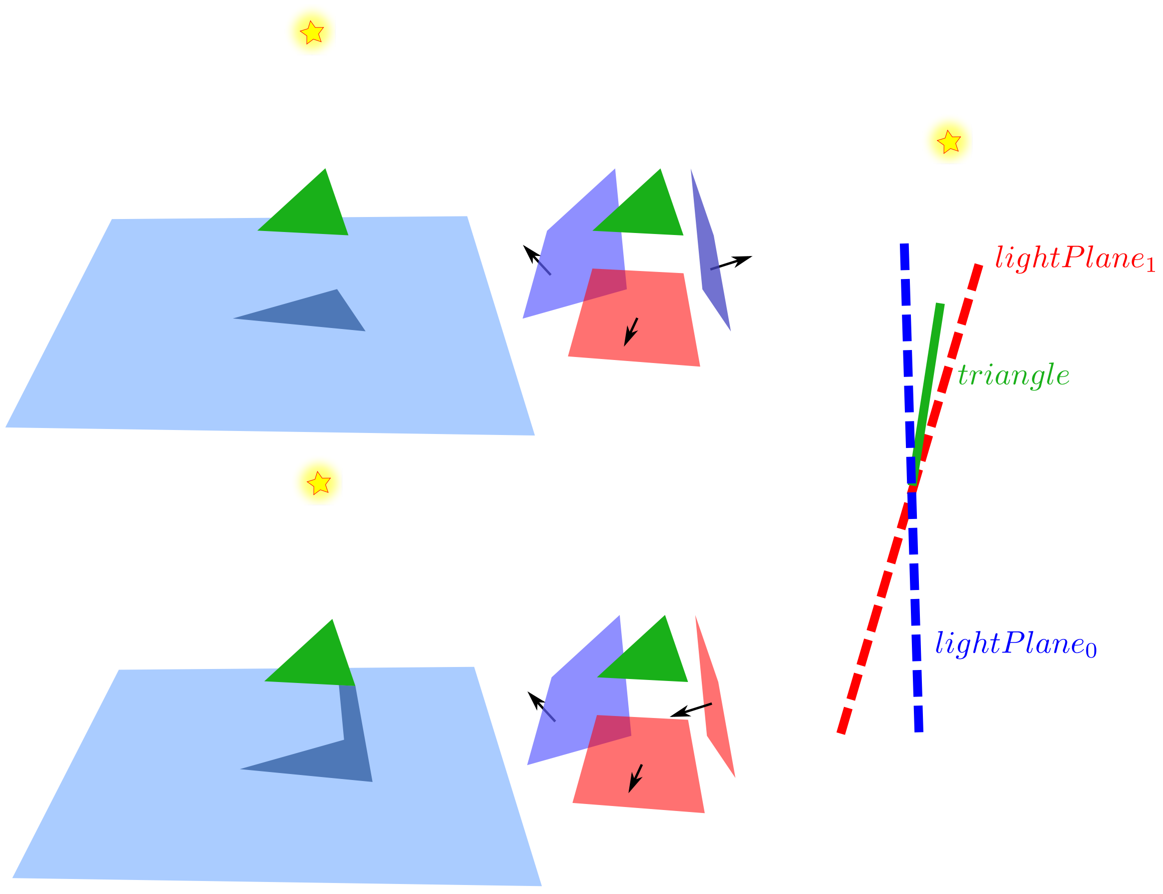


Figure 3.45: The left column shows a scene with a single triangle shadow caster. The middle column shows corresponding shadow volumes. Red polygons face the viewer while blue polygons back face the viewer. The bottom part shows visual artifacts caused by inconsistent multiplicity computation. The inconsistency is caused by different light plane computation and floating-point arithmetic. The issue arises when the light source is close to triangle plane. The right side shows side view of two light planes. Both planes are computed using the same edge and light source, but due to floating-point precision and order of operands, they are different.

planes. The edge list contains opposite vertices for each adjacent triangle. When the algorithm detects inconsistency (by computing all possible cases), it discards the problematic triangle. Milet et al. [72] presented algorithm that robustly computes multiplicity using reference edge. The idea is to always compute multiplicity using the same reference edge. Then the algorithm transforms the reference multiplicity to edge multiplicity. For Z-fail algorithm, the multiplicity has to be computed even for near and far caps.

Robust Silhouette Computation

The robust silhouette computation is essential for correct silhouette shadow volume rendering. The algorithm proposed by Milet et al. [72] computes silhouettes in parallel using Kim's general method with concept of reference edge. The algorithm requires special data structure for each model's edge, see Figure 3.46.

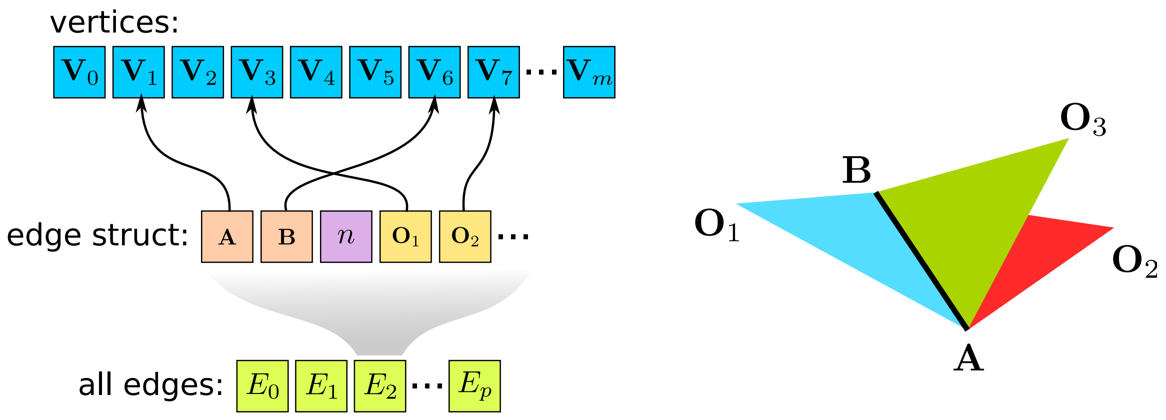

vertices:
$\mathbf{V}_0$ $\mathbf{V}_1$ $\mathbf{V}_2$ $\mathbf{V}_3$ $\mathbf{V}_4$ $\mathbf{V}_5$ $\mathbf{V}_6$ $\mathbf{V}_7$ ⋯ $\mathbf{V}_m$
edge struct:
A B n $\mathbf{O}_1$ $\mathbf{O}_2$ ⋯
all edges:
E_0 E_1 E_2 ⋯ E_p
$\mathbf{O}_3$
$\mathbf{B}$
$\mathbf{O}_1$
$\mathbf{O}_2$
$\mathbf{A}$

Figure 3.46: The right side shows a shared edge A, B with three adjacent triangles. The edge with opposite vertices is stored in edge structure. The left side shows model vertices V_i and the edge structure. The edge structure contains number of opposite vertices, indices to edge vertices and opposite vertices. The edge structure can contain vertices directly, if the indirection is not required. If the implementation requires fixed-sized edge structure, edges with many adjacent triangles can be split into multiple instances.

The algorithm extracts silhouette edges in parallel. Each thread computes multiplicity value using one edge structure, see Algorithm 3.

Algorithm 3: This algorithm draws silhouette shadow volume sides. The algorithm processes every edge, computes multiplicities and draws correctly oriented sides multiplicity times. The function computeMultiplicity robustly computes multiplicity value using reference edge, see Algorithm 4. The function drawSide is visualized in Figure 3.47.

```
Data: edge structures - $edges$, $light$
foreach $edge$ in $edges$ do // in parallel
    $multiplicity = 0$;
    foreach $O_i$ in $edge.oppositeVertices$ do
        $multiplicity$ += computeMultiplicity($edge.A, edge.B, O_i, light$);
    $i = 0$;
    while $i \leq |multiplicity|$ do // draw side multiplicity times
        if $multiplicity > 0$ then
            drawSide($edge.A, edge.B, light$);
        else // flip winding order
            drawSide($edge.B, edge.A, light$);
        $i = i + 1$;
```

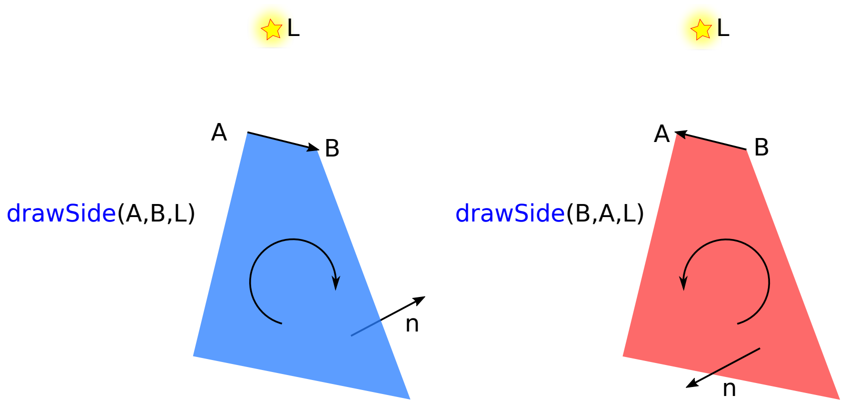


Figure 3.47: The function drawSide draws one side with correct orientation given by the order of input variables. The vertex winding order matches the order of edge vertices.

The algorithm computes multiplicity of reference edge (the largest oriented edge of adjacent triangle) and transforms it to correct edge multiplicity, see Algorithm 4.

Algorithm 4: The computeMultiplicity function computes multiplicity of one adjacent triangle. First, it finds the smallest and the largest vertex and creates reference edge. Then it computes reference multiplicity and transforms it to corresponding edge multiplicity.

```
  Data: edge vertices $A$,$B$, opposite vertex $O$ and $light$
  Result: $multiplicity$
1 $[R_{min}, F, R_{max}] = $ sort$([A, B, O])$;        // $R_{min} \to R_{max}$ is the reference edge
2 $n = (R_{max} - R_{min}) \times (light - R_{min})$;        // light plane normal
3 $referenceMultiplicity = $ sign$(n \cdot (F - R_{min}))$;        // -1 0 +1
4 if hasReferenceEdgeOppositeOrientation$(A, B, O)$ then
5     return -$referenceMultiplicity$
6 else
7     return $referenceMultiplicity$
```

The orientation has to be computed consistently. Edge orientation in edge list can be arbitrary and the algorithm has to cope with it, see Figure 3.48.

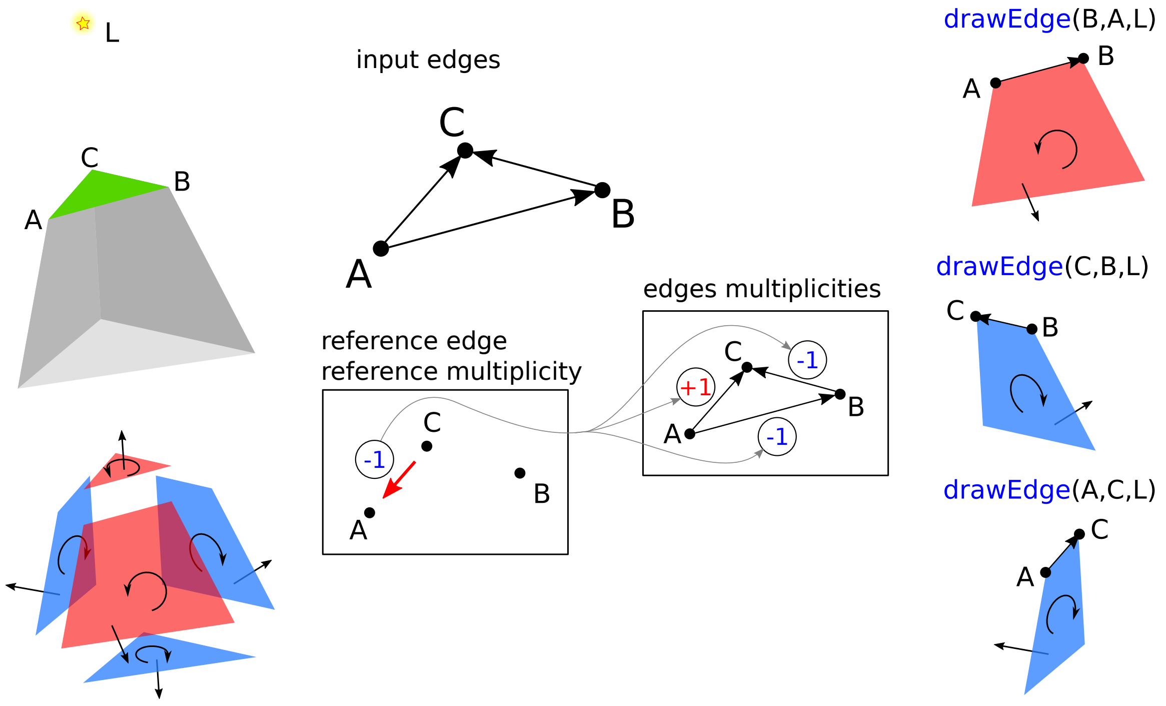


Figure 3.48: The image shows an example of computation of reference edge algorithm. The left side shows one triangle with shadow volume. The exploded shadow volume shows vertex winding orders and orientations. All normals point outside. Counter clockwise sides front face the viewer. The middle part shows the principle. Input edges are arbitrary oriented. First, the algorithm finds reference edge and computes reference multiplicity. A reference edge starts in the smallest and ends in the largest vertex of a triangle. Then, the algorithm transforms the reference multiplicity to edge multiplicity. If an edge has opposite orientation than the reference edge, the transformation negates reference multiplicity. The right side shows rendering. If an edge multiplicity is negative, the algorithm flips vertex winding order.

3.3.9 Rasterization Issues

Methods for computing shadows using shadow volumes require counting of ray intersections with shadow volumes. The counting is usually performed by rasterization of shadow volumes into stencil buffer. Although some of the robustness issues with shadow volumes were addressed by one or the other method, there are still other problems.

Rasterization on the GPU by itself can produce inconsistent results, see Figure 3.49.

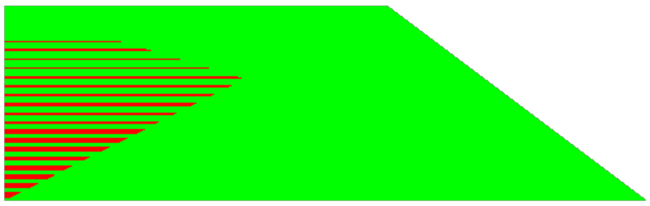


Figure 3.49: The image shows two clipped triangles - red and green. Both triangles are located at the same depth relative to the camera. Vertices of both triangles are identical, but the image still suffers from depth fight artefact. The reason is different vertex winding orders. The red triangle is drawn with counter clockwise winding order while the green with clockwise winding order. Depth values are not same. This issue affects some consumer hardware (AMD and Intel GPUs) and can create inconsistent shadows, see Figure 3.50.

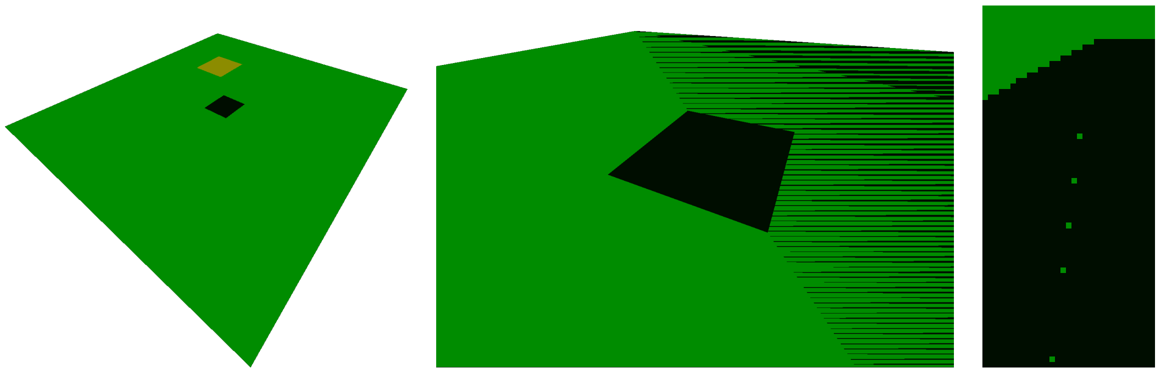


Figure 3.50: This image shows shadow volume issues caused by rasterization. The left side shows a simple scene with 4 triangles illuminated by a point light source. Yellow triangles cast shadows using Z-fail algorithm. The Z-fail algorithm requires rasterization of near-caps. Near-caps are located at the same position as shadow casters. Orientation of some of near-caps has to be flipped in order to keep consistent shadow volume orientation, see Figure 3.17. The flipping changes triangle winding order. Some hardware produces inconsistent depth values for different vertex winding orders, see Figure 3.49. The right green triangle has clockwise vertex winding order and has to be flipped which leads to z-fight artefact. One of the solution to this problem was proposed by Milet at el. [75], where near-caps are projected to infinity. Another issue arises with per-triangle shadow volumes algorithms. They require the same shadow volume sides for shared triangle edges, see Figure 3.39. Usually, these identical sides have different vertex winding orders. This creates empty space in between them, right side of the image. This can be imagined as light leaking through slit between two connected triangles. The solution for this issue is to store such sides in separate buffer and rasterize them later with correct vertex winding order and flipped stencil operation. The issue is not limited to per-triangle shadow volumes and can occur with silhouette based shadow volumes as well. If an edge is shared among many triangles, it can be split into multiple identical edges with fewer number of connected triangles.

3.4 Other Precise Shadow Methods

There are other real-time methods for precise shadow rendering. Some methods are based on shadow volumes, other on shadow mapping. This section briefly describes these methods.

Sample Based Visibility

Sintorn et al. developed precise method based on shadow mapping [91]. The idea extends the concept of shadow map, see Figure 3.51.

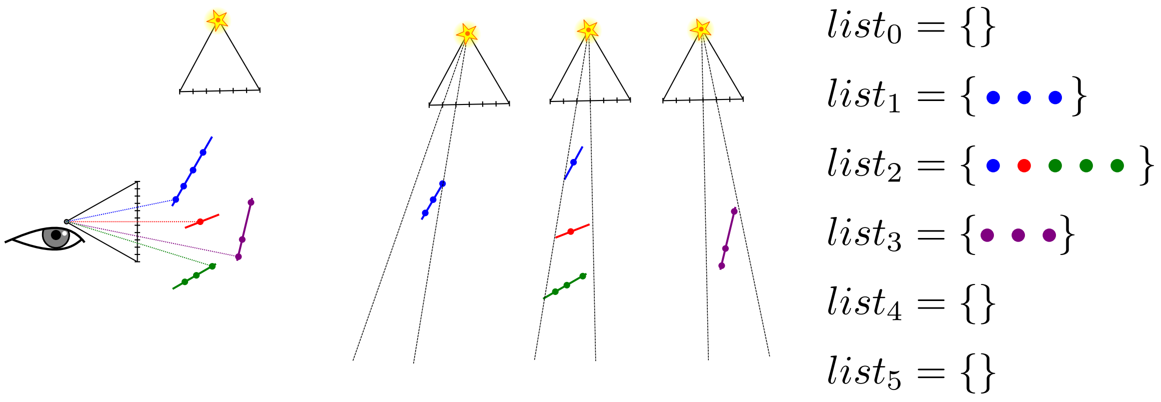


Figure 3.51: The image shows Sintorn's method. The shadow map does not store depths of shadow-samples, but it stores all projected view-samples. The algorithm first renders a scene from camera point of view. Then it projects all view-samples into shadow map space. Each shadow texel contains a list of view-samples.

When the extended shadow map construction is finished, the algorithm renders every triangle from light point of view. During triangle rasterization, each fragment looks into the shadow map and tests underlying view-samples against triangle's shadow volume, see the Figure 3.52.

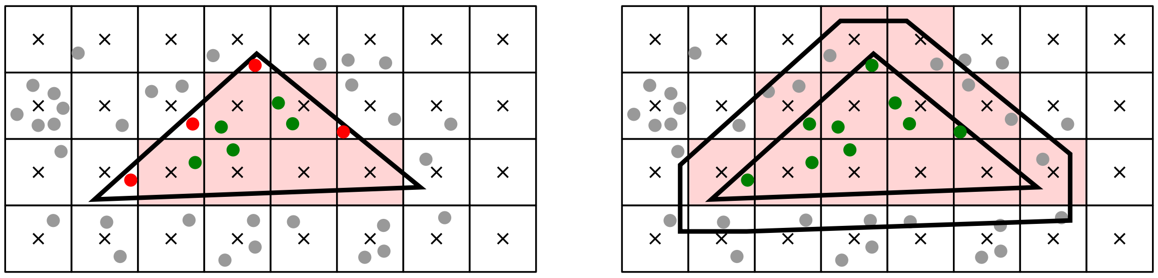


Figure 3.52: The image shows shadowing step of Sintorn's algorithm. Triangles are rasterized from light point of view. Each rasterized fragment invokes routine which tests all underlying view-samples against triangle's shadow volume planes. The left side shows standard rasterization. Red view-samples are shadowed by the triangle, but their fragment is not rasterized. The conservative raterization extends borders of the triangle by half of a pixel. The right sides shows the same process but with conservative rasterization. No view-samples are missed.

Triangles have to be conservatively rasterized in order to avoid artifacts on triangle's edges. Conservative rasterization is available in some consumer GPUs, but it can also be implemented manually. Conservative rasterization also suffers from floating-point problems for near degenerated triangles. The method is 3 times slower than regular shadow mapping with $8k^2$ resolution.

Hybrid Ray-traced Shadows

The algorithm was developed by Story [98] and is based on shadow mapping. The main idea is to store triangle indices into deep primitive map rather than to store depth value. The algorithm is similar to Sintorn's method. While Sintorn's method stores view-samples into shadow map texel list, this method stores triangles, see Figure 3.53. The method conservatively rasterizes scene triangles from light point of view and stores triangles IDs into every covered fragment list. Then it rasterizes the scene from camera point of view and tests view-samples against shadow volumes of triangles stored in texels lists. The size of triangle lists is scene and light dependent and has to be chosen manually.

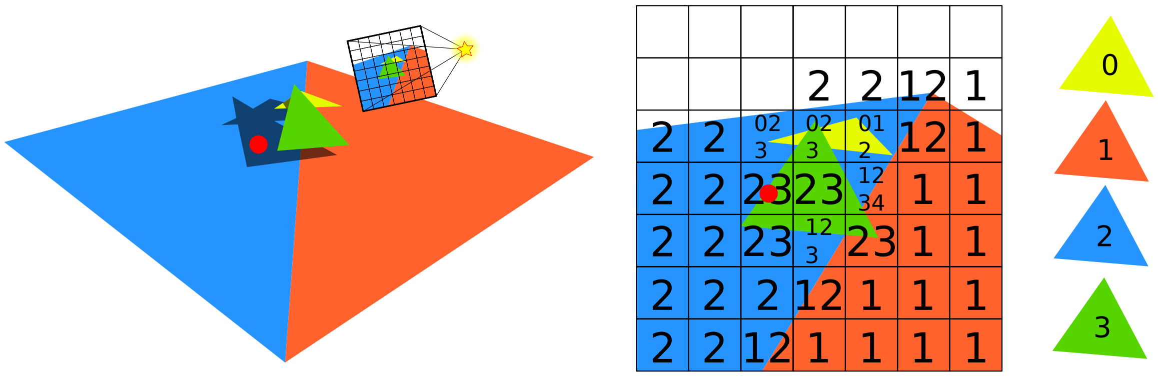


Figure 3.53: The image shows the principle of Story's algorithm. The left side shows a scene rendered from camera point of view. The right side shows deep primitive map. Each texel contains a list of triangle IDs. View-samples are projected into the map and tested against lists. The red view-sample is inside shadow volume of green triangle.

Partitioned Shadow Volumes

Gerhards at al. developed method that partitions scene space according to shadow volume planes [33]. The main idea is to quickly build acceleration structure from per-triangle shadow volumes. The structure uses ternary object partitioning tree, see Figure 3.54.

The algorithm incrementally builds TOP tree with randomized triangle order. An example of TOP tree can be seen in the Figure 3.55.

The algorithm was further developed and extended by Mora [76] and Deves [20].

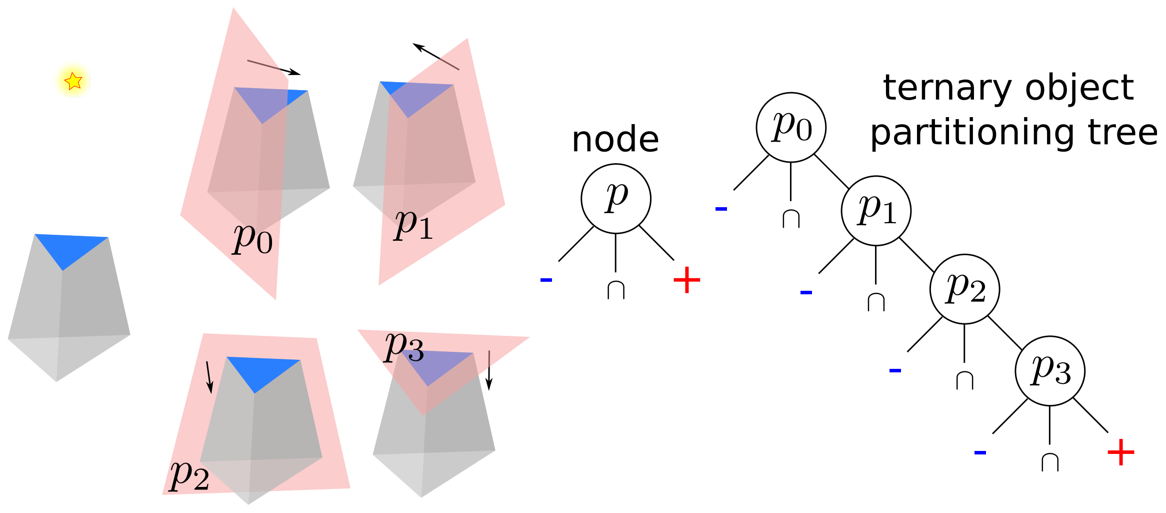


Figure 3.54: The image shows ternary object partitioning tree (TOP). The left side shows a triangle with a shadow volume. The middle part shows four shadow volume planes. Tree nodes represent one plane. Each node has three children. The right child contains objects strictly in front of the node's plane, the left child contains object strictly behind and the middle child contains intersecting objects. The right side of the image shows TOP tree for single shadow volume.

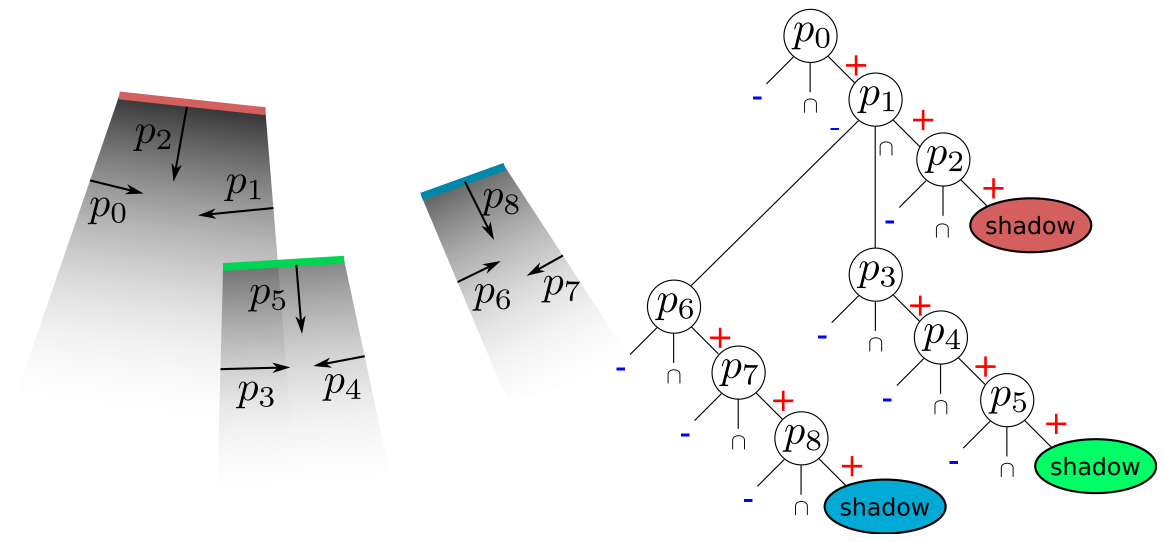


Figure 3.55: The image shows simple TOP tree structure in 2D. The left side shows three triangles with shadow volume planes. The tree is built with random triangle order. When the tree is built, it can be used for querying shadow information of view-samples.

Irregular Z-buffer Methods

The concept of irregular z-buffer was proposed by Johnson [48]. The idea is to enable irregular sampling of the scene. Common sampling stores samples in regular grid. Johnson presented irregular shadow mapping method. The method was further developed later by Wyman [109]. Z-only prepass creates samples that are inserted into linked lists of light-space texels, see Figure 3.56. A linked list contains only view-sample pointers and is constructed using atomic comp-swap operation. View-samples are tested by conservatively rasterized triangles. The process is similar to Sintorn's algorithm.

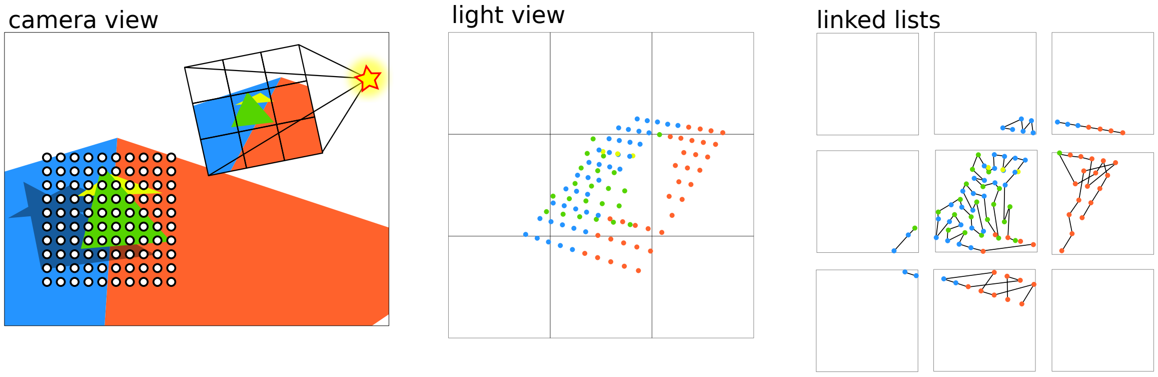


Figure 3.56: The image shows Irregular z-buffer method for shadow rendering. The left side shows a scene render from camera point of view. View-samples form regular grid. The middle part shows view-samples projected to light space forming irregular pattern. The right side shows linked lists of shadow map texels.

Chapter 4

Robust Silhouette Shadow Volumes on Contemporary Hardware

The section describes an algorithm, which produces shadow volumes for an arbitrary triangle model without visual artifacts. The algorithm has been implemented, optimized, and evaluated for a number of contemporary hardware platforms. The main contribution is removal of visual artifacts caused by limited precision of floating-point arithmetics (see Figure 4.1), overview of the implementation and result of the optimizations on individual platforms. The full content of this section can be found in paper [85].

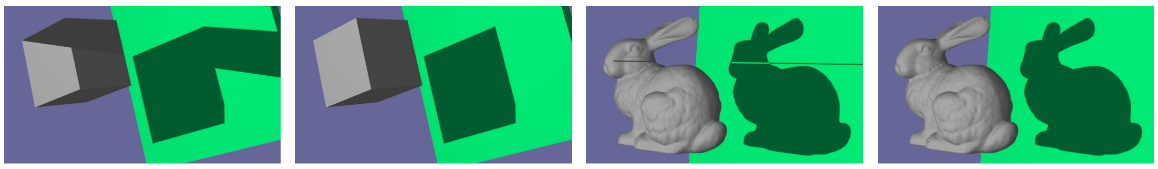


Figure 4.1: The Figure shows the difference between the original algorithm and the new robust algorithm. The right image of each couple shows result of new algorithm. The first couple of images shows very simple model, where artefacts are most visible. The second couple shows artefacts on more complex model, which could appear in real applications.

4.1 Introduction

Shadow volumes (SV) method is a traditional and popular method for shadow casting in computer graphics. The research on shadow volumes is summed up in the previous sections.

An alternative for rendering shadows is shadow mapping [106]. It is very frequently used as it generally offers high performance; however, the shadow maps approach suffers from visual imperfections caused by the limited shadow map resolution. The shadow map approach is massively used in game industry where high performance is critical and scenes can be adjusted so that visual artefacts are not too visible or do not occur. On the other hand, these features limit the applicability of the approach in e.g. CAD systems, where the scene is given by the model being constructed and may be incompatible with the requirements of the shadow maps approach to work well.

While the SV approaches produces per-pixel correct results, they are affected by some performance issues. In its simple form, the applications were using a SV generated for each triangle of the object geometry, which could be very rasterization demanding. This fact lead to development of more sophisticated methods, which construct a SV only from the possible silhouette edges of the occluder geometry, that has positive impact on fill-rate. Many of the of silhouette algorithms expect 2-manifold, oriented or watertight objects as their input. Kim in 2008 [52] presented the algorithm that works with any mesh objects.

Unfortunately, during the implementation of the algorithm presented in [52] one can find out that the algorithm is not completely robust and often produces results with visual artefacts. This is caused by limited precision of floating-point arithmetics. Thus, this section describes robust algorithm, based on Kim's algorithm, that is free from visual artifacts. Additionally, it summarizes optimized implementation for a number of contemporary hardware platforms (CPUs and GPUs).

4.2 Algorithm Description and Robustness improvement

The algorithm, described in [52], generates the output shadow silhouette based on the triangular mesh representing the model and the position of light source.

4.2.1 Description of the Algorithm

The input of the algorithm is a triangular mesh and light source position and the output is a silhouette represented by a set of edges selected from the input model. The algorithm can be briefly described as follows:

1. The triangular model is converted into an edge representation. Every edge occurs only once even though it is shared among more triangles. Each edge in the new representation is described by its vertices and list of all opposite vertices (OVs). The OVs are vertices of the triangles sharing the edge that do not belong to the edge. See Figure 4.2.

2. For each edge from the edge set, an oriented light plane (LP) is evaluated from edge vertices and the position of the light source.

3. For each OV belonging to the edge, multiplicity is calculated as +1 or -1 depending on which side of LP the OV lies. If the OV lies exactly on the LP, its multiplicity is 0. The final multiplicity of an edge is given by the sum over the multiplicities of every OV. See Figure 4.2.

4. Finally, the set of edges forming the silhouette is a subset of all the edges such that their multiplicity is not 0.

4.2.2 Implementation and Problems

The above mentioned algorithm processes the model with the "by edge" approach. The multiplicity could also be calculated with the "by triangle" approach (with identical results). In parallel implementation, the "by edge" approach, is better than "by triangle" implementation, although the latter may seem more natural. The main reason is that the edges are independent to each other, so this avoids concurrent memory writes. While the

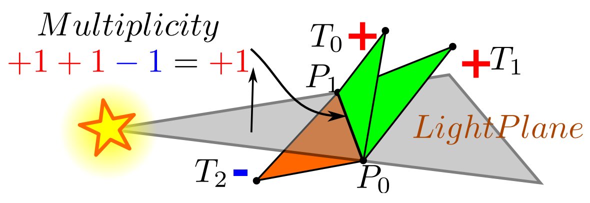


Figure 4.2: Multiplicity of Edge P_0-P_1 for the Opposite Vertices (OVs) T_0-T_2.

"by triangle" approach would lead to usage of atomic operations. Therefore, the "by edge" implementation is exploited.

The algorithm assumes that the evaluation of multiplicity is consistent within each triangle. Unfortunately, this is not the case for floating-point arithmetics used in HW. In the example shown in Figure 4.3, the "by edge" approach could evaluate multiplicities inconsistently for the triangle which is (almost) parallel to the LP. While the error was demonstrated for a single triangle model, such error can occur in a more complex model for individual triangles and ruin the whole silhouette leading into visible artefacts in shadows (see the Figure 4.1).

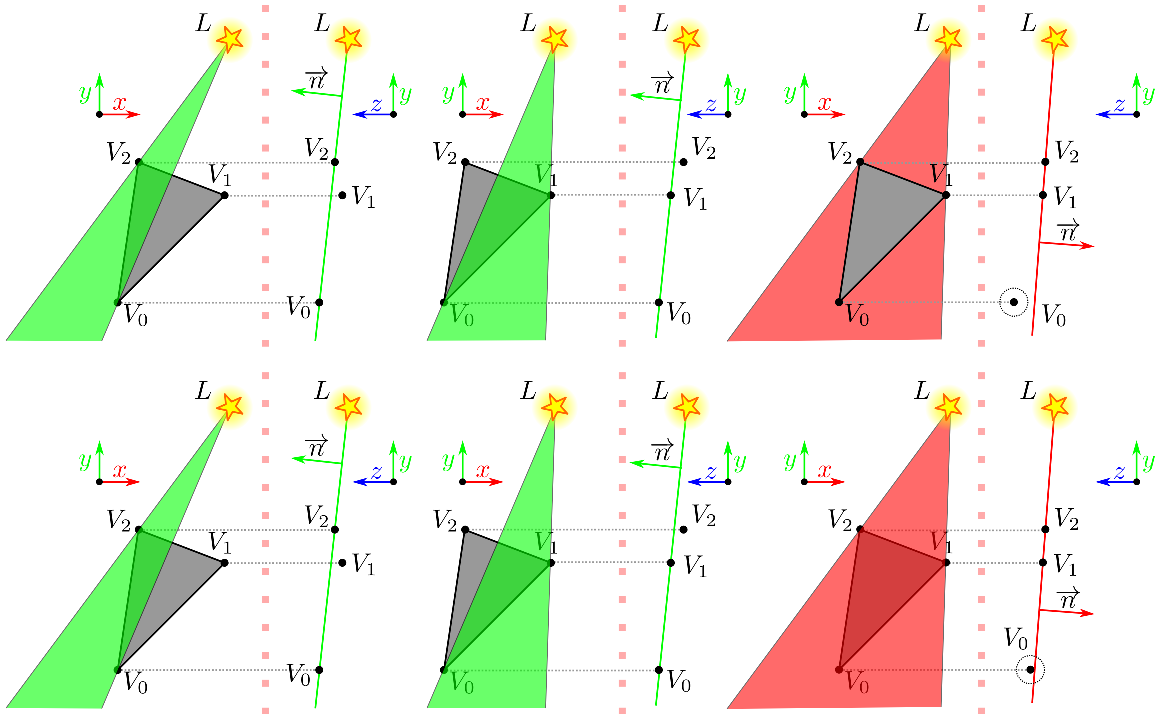


Figure 4.3: The grey triangle generates a SV. The green and red triangles represent the two orientations of the light plane (LP). The lower part shows erroneous calculation. In the upper right part, triangle is in front of LP with respect to the viewer. In the lower right, this is incorrectly evaluated, therefore the vertex V_0 is assumed to be behind.

4.2.3 The Proposed Robust Algorithm

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.

The question is “What is the most efficient way to remove such triangles?”. Note that while the “by triangle” approach permits to simple discard the computed triangle, the “by edge” approach does not. One possible solution would be to evaluate “how close to parallel” the triangle is to the LP but in this case, the evaluation would have to be consistent for each triangle edge leading more or less to the same problem. Therefore, a solution was taken to “simulate” evaluation of the “other two edges” of the triangle formed by the edge and each OV. The modification to the original algorithm changes the step 3:

3. For each OV belonging to the edge a triangle is formed from OV and the edge. The multiplicity is evaluated for **every side** of this triangle and its remaining vertex as +1 or -1 depending on which side of LP the vertex lies. If the vertex lies exactly on the LP, its multiplicity is 0. If the evaluation of the multiplicity is inconsistent, the triangle is disregarded (the OV multiplicity is set to 0). Note also, that the same order of vertices and edges in triangles must be preserved for each edge evaluation. The final multiplicity of an edge is given by the sum over the multiplicities of every OV. See Figure 4.2.

The actual multiplicity evaluation for the edge AB for the light source position **L** and set $\mathfrak{O}$ of all OV, each in homogeneous coordinates, is as follows:

The LP itself is defined as:

$$\begin{aligned} \mathbf{V} &= (\mathbf{L}_x - \mathbf{A}_x\mathbf{L}_w, \mathbf{L}_y - \mathbf{A}_y\mathbf{L}_w, \mathbf{L}_z - \mathbf{A}_z\mathbf{L}_w) \\ \mathbf{N} &= \text{normalize}((\mathbf{A} - \mathbf{B}) \times \mathbf{V}) \\ \mathbf{LP} &= (\mathbf{N}_x, \mathbf{N}_y, \mathbf{N}_z, -\mathbf{N} \cdot \mathbf{A}) \end{aligned} \tag{4.1}$$

The multiplicity of the edge is:

$$m = \sum_{\mathbf{o} \in \mathfrak{O}} \text{sign}(\mathbf{LP} \cdot \mathbf{o}) \tag{4.2}$$

Where $|m|$ denotes the number of times the side of SV, extruded from this particular edge, is actually drawn/rendered.

Of course, the evaluation of the above set of expressions, for each edge of the triangle (instead of just once for each triangle), introduces a computational overhead. While some subexpression results can be reused, a significant overhead remains. However, it turns out that the cost of additional arithmetics, especially in case of exploitation of powerful computational platform, is less costly than increased memory traffic or synchronization operations needed in alternative approaches.

4.3 Hardware Platforms

The hardware platforms selected for implementation of the presented algorithms are selected in order to represent the wide range of contemporary hardware being used in the segment of “personal computers”. Measurements were carried out on Windows 7 x64 SP1. As for the accelerated graphics, samples of the most frequently used “mainstream” boards and not necessarily the highest performance ones were selected:

4.3.1 Intel Platforms

Intel HD 3000 - driver version: 9.17.10.3062 This is a representative of the frequently occurring boards in low cost devices, such as notebooks and netbooks. Processor: i7-2600K, RAM: 16GB

Intel HD 4000 - driver version: 9.18.10.3071, release date: 2013-03-19 This is a representative of the frequently occurring boards in low cost devices, such as notebooks and netbooks. Processor: i5-3570K, RAM: 8GB

4.3.2 GPU Platforms

GeForce 650 Ti, 2GB GDDR5, 128bit This board was selected as one of the mid-range graphics boards by NVidia, being a mainstream in the graphics boards. GeForce 650 Ti provides the Nvidia Keppler architecture while it is a cost effective solution, providing 768 computing cores compared to 1536 of hi-end GeForce 680 and 192 of low-end GeForce GT 630. Driver version: 314.07 Processor: i7-2600K, RAM: 16GB

Radeon HD 7950, 3GB GDDR5, 384bit This is a representative of the high performance graphics board. Driver version: 13.1 Processor: i7-2600K, RAM: 16GB

4.3.3 Platforms Summary

The boards were plugged into a personal computer. To enable reasonable comparison of performance, all the graphics accelerators were used in the same PC with exception of the HD4000 graphics, where it was impossible for technical reasons (HD3000 and HD4000 graphics cannot be generally included in one computer).

4.4 Experiments and Results

The experiments were conducted in order to evaluate the achieved results, to assess feasibility of exploitation of the presented approach at the above mentioned platforms and in different applications, and to verify that the approach works well.

The performance of the algorithm was tested under these conditions: the robust version compared to the original version with no robustness, objects of simple shape compared to more complex shapes with similar number of triangles, varying levels of geometric complexity of the same object and scenes containing several separated objects.

4.4.1 Robust Versus Traditional Implementation

The purpose of this test was to evaluate whether at all and how much the implementation of the robustness of the algorithm adversely affects performance. The conditions for the test were made similar to the real applications conditions in terms of the size of the scene ($\sim 10^5$-10^6 triangles). The test was performed on a scene that did not exhibit significant occurrence of visual artefacts caused by the traditional implementation.

The results of experiments are shown in the Table 4.1. Note that performance of the robust algorithm was slightly worse than in the traditional implementation (the adverse effect of additional calculations was less than 10% at all platforms), but this is true only in case of scenes exhibiting no or little visual artefacts. In scenes with larger amount of artefacts, performance of the robust implementation was mostly better and the larger the amount of artefacts, the worse the traditional implementation performed also from the

	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	30.0	40.0	92.0	98.0	n/a	n/a	4.4	8.7
OpenCL	42.0	41.0	83.0	80.0	n/a	n/a	7.6	7.7

Table 4.1: Results of experiment (the first value shows FPS of the robust implementation, the second value is the original implementation)

	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	**10.0**	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	30.0	30.0	**92.0**	95.0	n/a	n/a	4.4	n/a
	34.0	34.0	**156.0**	124.0	n/a	n/a	6.0	n/a
OpenCL	**42.0**	**52.0**	83.0	**96.0**	n/a	n/a	**8.1**	**8.2**
	54.0	**67.0**	142.0	**147.0**	n/a	n/a	11.3	6.0

Table 4.2: Results for one particular GPU and one platform consisting of 4 tests on different scenes: 10 individual bunnies (top left), 10 baked bunnies (top right), 10 individual spheres (bottom left), and 10 baked spheres (bottom right). Bold values highlight the fastest implementation for each scene for selected GPU.

computational time point of view probably due to the fact that the artefacts caused increase in the fill rate. Overall, quite surprisingly, the decrease of performance in the robust method is not a problem on any platform.

4.4.2 Simple Versus Complex Shapes

The purpose of the next test was to evaluate how the shape of the objects influences the rendering times. Therefore, scene consisting of simple shapes, spheres, and scene consisting of complex shapes, bunnies, similarly large in terms of triangles, were compared. The size of the scene in this case was about the usual application size ($\sim 6.5 \cdot 10^5$ triangles). One measurement was performed for the scene consisting of more (10) objects, one for the scene consisting of a single object but with the same complexity as in the previous case; this was done in order to check whether the number of objects affects the calculation speed.

4.4.3 Number of Triangles in the Scene

The consequent test was focused on the behavior of the algorithm in rendering of the scene depending on the number of triangles in the scene. The goal was to learn how the algorithm performs when the number of triangles changes from relatively low ($\sim 10^5$ triangles) to relatively high ($\sim 10^6$ triangles).

It can be seen that, as expected, performance of the algorithm decreases with the increased size of the object in terms of triangles. However, what was not as expectable is

	GF650Ti		AMD7950		HD3000		HD4000	
CPU	3.9	35.0	5.1	81.0	**5.3**	**34.0**	**37.0**	20.0
	3.4	27.0	8.4	61.0	6.5	32.0	4.4	18.5
AVX+OMP	3.9	35.0	5.7	81.0	4.9	32.0	4.2	**23.0**
	3.4	28.0	9.3	63.0	**6.7**	**34.0**	**5.0**	**24.0**
GS	23.0	165.0	85.0	**650.0**	n/a	n/a	3.7	15.0
	18.7	**100.0**	**84.0**	**482.0**	n/a	n/a	3.1	19.3
OpenCL	**43.0**	**192.0**	**91.0**	430.0	n/a	n/a	**6.0**	19.3
	13.4	26.0	22.0	54.0	n/a	n/a	4.4	9.1

Table 4.3: Computational performance based on the number of triangles in the scene. Results for one particular GPU and one platform consisting of 4 tests on different scenes: one sphere with 10^6 (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 fact that on different platforms, performance does not decrease uniformly and also that performance is not affected uniformly in different implementations on the same platforms. Additionally, in cases where the number of triangles is large, performance is also adversely affected by subdivision of the scene into separate objects as described below.

4.4.4 Number of Isolated Objects in the Scene

The goal of this final test was to demonstrate how the algorithm performs in dynamic scenes.The test was performed along with the above mentioned testing of performance with changing number of triangles in the scene. It shows that the number of isolated objects does have impact on results, especially in case of some platforms, probably due to synchronization and data transfer issues.

4.4.5 Hardware Platforms

The implementation was tested on following platforms:

- AMD Radeon HD 7950 (driver version: 13.1)
- GeForce 650 Ti (driver version: 314.07)
- HD3000 integrated Intel GPU (driver version: 9.17.10.3062)
- HD4000 integrated Intel GPU (driver version: 9.18.10.3071)

First three GPUs were tested with Intel i7-2600K with 16GB RAM. The HD4000 was tested with Intel i5-3570K with 8GB RAM. All the measurements were carried out on Windows 7 x64 SP1.

4.4.6 Interpretation of the Results

Some of the results observed in the above tests are especially worth mentioning and they are listed below:

- Performance of “OpenCL” implementation is very good on all the platforms for which OpenCL is available. The current solution heavily suffers from the synchronization between OpenCL and OpenGL contexts. This mainly occurs in scenes containing many separate objects because synchronization must be performed for each object. Unfortunately, OpenCL solution is not supported by the HD3000.
- The “Geometry Shader” implementation performs well in scenes with many separate objects. The performance declines with increasing complexity of models more than in OpenCL implementation.
- CPU+AVX+OMP performance is sometimes surprisingly less than the standard CPU implementation. On the other hand, measurements carried out on i5-3570K (Ivy Bridge architecture) showed 4-30% performance increase (16% in average), compared to standard one. Despite the fact that the CPU is capable of processing 8(4) threads concurrently, maximum performance increase is only 30% in its peak. The reason is that not all parts of the algorithm can be parallelized or rewritten using AVX intrinsics.

It is most suitable to use OpenCL implementation (where available) for scenes which do not contain too many (less than 100) objects. Geometry shader solution can be used in situations where the scene contains larger number of separated objects. CPU+AVX+OMP implementation should be used on modern CPUs in situations where the above solutions are not available. Standard CPU implementation should be used otherwise.

4.5 Conclusions

This section presented a novel approach to Shadow Silhouette Shadow Volumes that leads into a more robust implementation, which has been tested and evaluated on a number of different hardware platforms.

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. The algorithm 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 most efficient achieved implementation was in OpenCL for a scene containing 10^6 triangles, followed by the Geometry Shader implementation that is usable also with Intel HD 4000 platform. However, the OpenCL implementation suffers from synchronization slowdowns in case the scene is divided into more independent objects. As for the CPU implementations, while they are generally slower than the GPU ones, the SIMD instructions (AVX) and parallelism boosts performance on the latest CPU architectures.

Overall, the proposed approach performs very well and at the same time it is robust and precise in terms of “per pixel” precision of the shadows. Therefore, it represents a very good alternative to shadow methods.

Future work includes improvements of the OpenCL implementation in terms of synchronization in scenes containing more objects and general improvements of the triangle tests. The future work also includes more thorough evaluation on more platforms and more scenes.

Chapter 5

Fast and Robust Tessellation-Based Silhouette Shadows

This section presents a new, simple, fast and robust approach in computation of per-sample precise shadows. The method uses tessellation shaders for computation of silhouettes on arbitrary triangle soup. The robustness is reached by novel silhouette computation based on reference edge. New method was compared with other methods and evaluated on multiple hardware platforms and different scenes (Figure 5.1), providing better performance than current state-of-the art algorithms. The full content of this section can be found in paper [72].

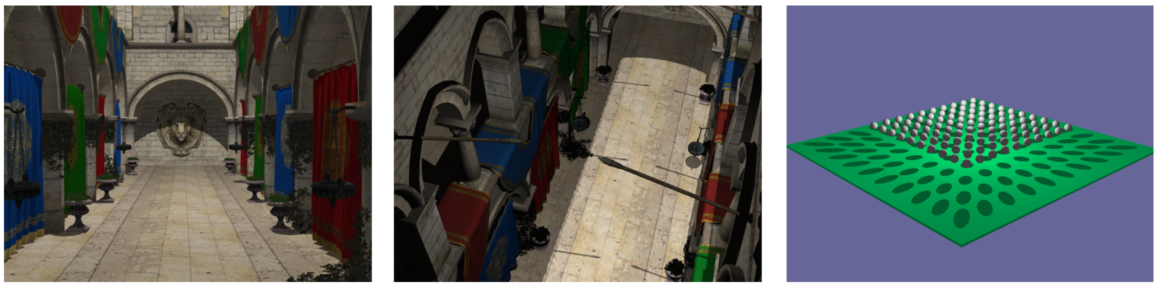


Figure 5.1: Test scenes - Crytek Sponza and Spheres

5.1 Introduction

Novel tessellation methods are based on silhouette shadow volumes. Shadow volumes are closely described in previous sections.

Shadow Mapping (SM) algorithm, proposed by [106], is an alternative approach to shadow volumes. It uses depth information from light source stored in a texture. Shadow mapping is nowadays massively used in games thanks to its performance, but suffers from spatial and often temporal aliasing problems and produces imperfect shadows because of limited shadow map resolution [23]. Low resolution is not an issue for games, because scenes can be adjusted so that visual artifacts are suppressed or a filtering method is applied, but applications for visualization in architecture or industrial design require pixel-correct shadows for object visualization. Per-sample precise alias-free shadow maps (AFSM) algorithm was proposed by [91]. The method stores multiple samples into a list for each shadow map pixel and conservatively rasterizes triangles into shadow map using CUDA. Individual samples stored in the lists are then tested against shadow volume of the triangle. As authors stated, their per-sample precise method is three times slower than standard shadow mapping with resolution of 8096 by 8096 texels.

While producing per-sample correct shadows, SV are affected by performance issues. In its naive form, when a volume is generated for every triangle in the scene, resulting performance is very low due to rasterization of a large amount of triangles. More efficient way is to construct shadow volumes only from silhouette edges of the occluding geometry, which has positive impact on fill-rate.

Several silhouette-based methods were published since then, utilising novel hardware features to speed up silhouette calculation.

Most of the silhouette methods are not completely robust and also cannot handle non 2-manifold casters. [52] proposed an algorithm for any complex triangle caster, but unfortunately it is not robust. Kim's algorithm was improved in [85] using deterministic multiplicity calculation. The novel method further extends and simplifies Peciva's method.

[93] also proposed a shadowing technique based on CUDA software rasterization of per-triangle shadow frusta. This technique uses a small bias when testing sample depth against a triangle plane to avoid self-shadowing. This bias, however, may cause a shadowed fragment to be lit in the final result, moreover, it is also scene-dependent.

5.2 Method Description

This section presents three tessellation based methods - two per-triangle approaches and one robust silhouette method.

The silhouette method is based on the work of [52]. The algorithm calculates so-called multiplicity of an edge. Light plane from light source through the edge is computed and all opposing vertices are tested, if they are in front or behind the plane. According to the test, multiplicity is incremented or decremented. Absolute value of multiplicity is the number of times an infinite quad needs to be drawn from this edge.

5.2.1 Per-Triangle Methods

These methods require no pre-processing and work with arbitrary triangle soup. In the first variant, input tessellation patch has 3 points, which are identical to triangle points. Tessellation factors are 3 (inner) and 1 (outer, for all sides), equal spacing and reversed triangle winding. The resulting patch can be seen in Figure 5.2 b.

The algorithm constructs a simple volume in evaluation shader, see the Algorithm 5. Front cap needs to be rendered in second pass in order to close the volume.

The second per triangle method is a single-pass version for z-fail and tessellates quads instead of triangles. An input tessellation patch contains three points of a triangle. The quad is tessellated using outer factors (1, 5, 1, 5), inner (5,1) and fractional odd spacing, resulting in a shape seen in Figure 5.3 b.

Evaluation shader then twists the shape in order to create a volume, note Figure 5.3.

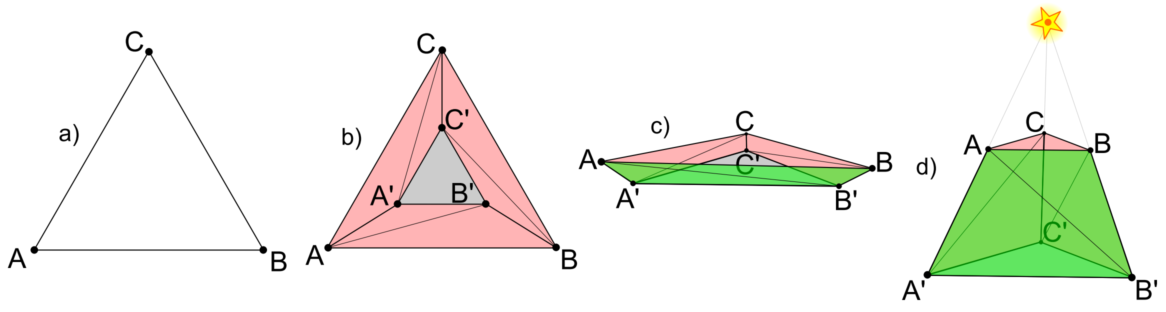


Figure 5.2: The image shows the process of first (two pass) per triangle tessellation method. Initial triangle in *a*) is tessellated using outer factors (1, 1, 1) and inner (3) *b*). Tessellation process creates points A', B', C'. The patch is then deformed *c*) and new tessellated points are pushed to infinity to form a volume with back cap *d*).

Algorithm 5: Evaluation shader in two-pass per-triangle method

```
Data: triangle points $\mathbf{P}[3]$, light position $\mathbf{L}$, tessellation coordinates $\mathbf{T} = (x, y, z)$,
      $x, y, z \in \langle 0, 1 \rangle$
Result: world-space coordinates $X$
1 $c = x \cdot y \cdot z$;            // outer points $A, B, C$ have only one non zero coordinate
2 if $c == 0$ then
3     $\mathbf{X} = \mathbf{P}[0] \cdot x + \mathbf{P}[1] \cdot y + \mathbf{P}[2] \cdot z$;      // points $A, B, C$
4     $\mathbf{X}_w = 1$;
5 else
6     $i$ = getIndexOfLargestVectorElement($\mathbf{T}$);
7     $\mathbf{X} = l_w \cdot \mathbf{P}[i] - \mathbf{L}$;      // points $A', B', C'$
8     $\mathbf{X}_w = 0$;
```

5.2.2 Silhouette Method

The method finds silhouette edges by looping over every edge in the model. Each edge is processed in parallel in Tessellation Control Shader where multiplicity is computed. An input patch can be seen in the Figure 5.4.

A model's vertex buffer has to be extended by E_n vertices, which is the number of edges in the model. Element buffer can be used for reduction of memory requirements, see the Figure 5.5.

Kim's algorithm [52], as in its core proposal, might have problems if multiplicities are not calculated in a deterministic way. In the older approach by Peciva et al. [85], it was fixed by calculating multiplicity per triangle. If the 3 results throughout all 3 edges is not consistent, it discards the triangle from further processing, because it means that the triangle is almost parallel to the light and does not cast a shadow. This novel algorithm further improved the approach - multiplicity is computed only once for each opposite vertex using *reference edge*.

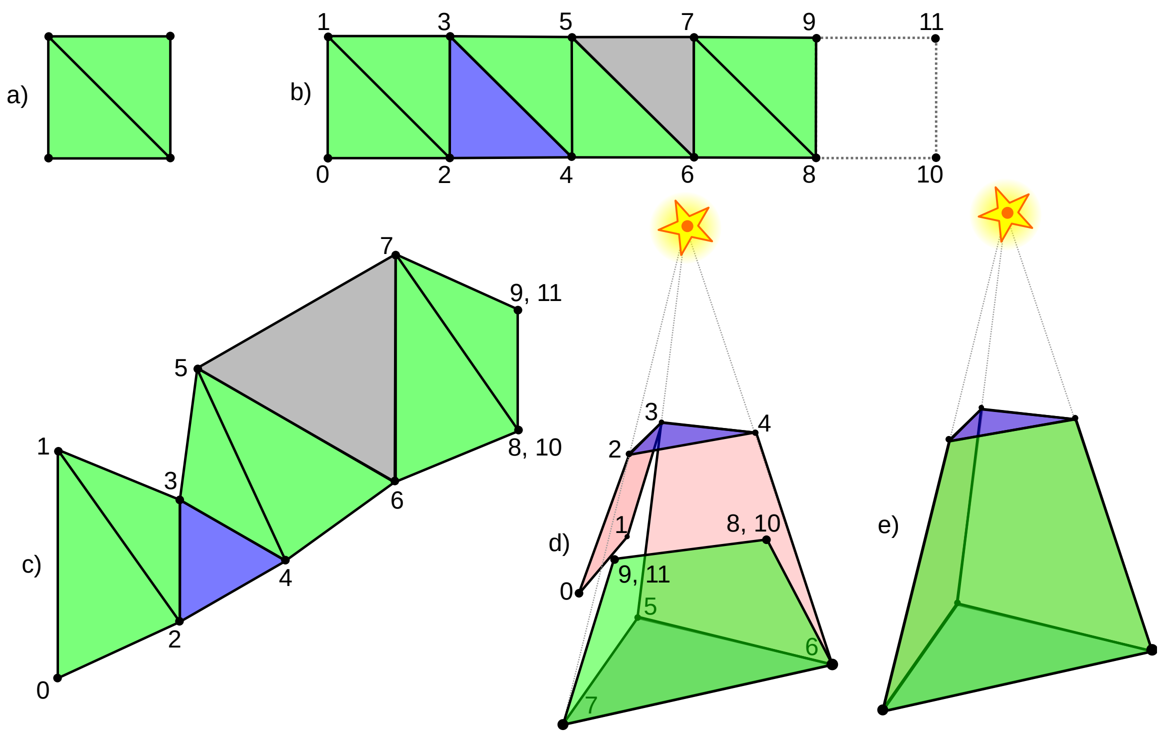


Figure 5.3: Single-pass per triangle method, a full shadow volume is created in a single pass. The method tessellates quads *a*). The tessellation factors are (1, 5, 1, 5),(5, 1) *b*). Unfortunately, tessellation hardware creates one additional quad (point 10 and 11) at the end of the strip. Points 10 and 11 are merged with 8, 9. Light cap is visualized as blue, dark cap grey *c*). The algorithm then joins points 0-7, 1-5, 2-9, 4-8 and pushed points 5, 6, 7 to infinity *d*) to make the volume *e*).

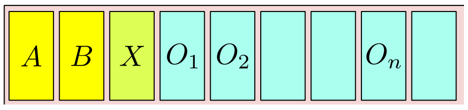


Figure 5.4: The image shows an input patch for tessellation control shader. It is composed of two vertices that describe an edge, one integer that contains number of opposite vertices and n opposite vertices. Because the patch size must be constant, some positions might not be used.

A choice of reference edge has to be the same for all occurrences of a triangle. This can be achieved for example by introducing vertex ordering - Equation 5.1 and Algorithm 6.

$$\begin{aligned}
\mathbf{A} < \mathbf{B} &\Leftrightarrow \text{greater}(\mathbf{A}, \mathbf{B}) < 0 \\
\mathbf{A} = \mathbf{B} &\Leftrightarrow \text{greater}(\mathbf{A}, \mathbf{B}) = 0 \\
\mathbf{A} > \mathbf{B} &\Leftrightarrow \text{greater}(\mathbf{A}, \mathbf{B}) > 0 \qquad (5.1)
\end{aligned}$$

In order to guarantee consistency, reference edge of a triangle in the algorithm is constructed using smallest and larges vertex of a triangle, as in Algorithm 6. More options for such method are available, but evaluation per each triangle occurrence must be consistent in order to get correct results.

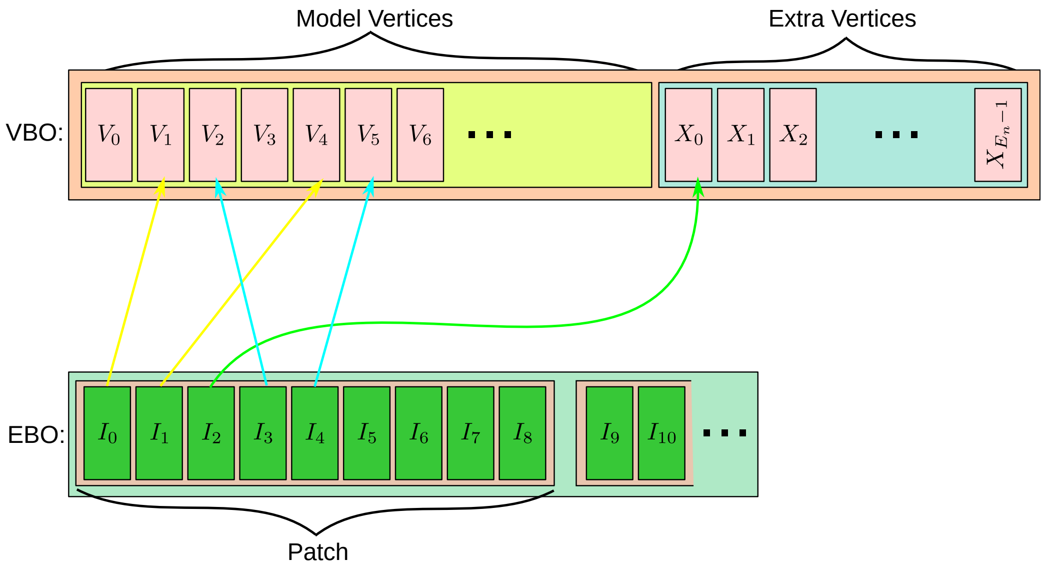


Figure 5.5: Combining per-edge patch data using Element Buffer Object. Vertex Buffer Object needs to be extended by E_n (number of edges) vertices, where X_i is number of adjacent triangles to edge E_i. Because input vertices are 4-component vectors and X_i is scalar value, only a single value per vector is used.

Algorithm 6: Function greater($\mathbf{A}, \mathbf{B}$) used for vertex ordering.

```
Data: Vertices $\mathbf{A}, \mathbf{B}$
Result: Result $r$ of comparison
1 $\mathbf{S} = \text{sign}(\mathbf{A} - \mathbf{B})$;
2 $\mathbf{K} = (4, 2, 1)$;
3 $r = \mathbf{S} \cdot \mathbf{K}$
```

To simulate behaviour of Kim's algorithm (edge casts a quad as many times as it has multiplicity), the algorithm tessellates a quad from an edge using inner tessellation levels $(Multiplicity \cdot 2 - 1, 1)$ and then it bends the tessellated quad in evaluation shader in a way to create m overlapping quads, as seen in the Figure 5.6, which demonstrates edge A-B having multiplicity of 3. The procedure of multiplicity calculation is described in the Algorithm 7 and in the Algorithm 8.

After tessellation, the algorithm transforms tessellation coordinates into vertex position of the shadow volume quad in the evaluation shader. The implementation is described in the Algorithm 9 and in the Equation 5.2.

$$\begin{aligned}
\mathbf{A} &= (a_x, a_y, a_z, 1)^T \\
\mathbf{B} &= (b_x, b_y, b_z, 1)^T \\
\mathbf{C} &= (a_x - l_x, a_y - l_y, a_z - l_z, 0)^T \\
\mathbf{D} &= (b_x - l_x, b_y - l_y, b_z - l_z, 0)^T
\end{aligned} \tag{5.2}$$

Because caps are not generated, this method can also be used with simpler z-pass algorithm.

Algorithm 7: Modified algorithm for computation of final multiplicity of an edge $\mathbf{A}, \mathbf{B}$ using reference edge concept.

```
Data: Edge $\mathbf{A}, \mathbf{B}$, $\mathbf{A} < \mathbf{B}$, set $\mathfrak{O}$ of opposite vertices $\mathbf{O}_i \in \mathfrak{O}$, light position $\mathbf{L}$ in
      homogeneous coordinates
Result: Multiplicity $m$
1 $m = 0$;
2 for $\mathbf{O}_i \in \mathfrak{O}$ do
3     if $\mathbf{A} > \mathbf{O}_i$ then
4         $m = m +$ compMultiplicity$(\mathbf{O}_i, \mathbf{A}, \mathbf{B}, \mathbf{L})$;
5     else
6         if $\mathbf{B} > \mathbf{O}_i$ then
7             $m = m -$ compMultiplicity$(\mathbf{A}, \mathbf{O}_i, \mathbf{B}, \mathbf{L})$;
8         else
9             $m = m +$ compMultiplicity$(\mathbf{A}, \mathbf{B}, \mathbf{O}_i, \mathbf{L})$;
```

Algorithm 8: compMultiplicity$(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{L})$ function used in Algorithm 7

```
Data: Vertices $\mathbf{A}, \mathbf{B}, \mathbf{C}$; $\mathbf{A} < \mathbf{B} < \mathbf{C}$; light position $\mathbf{L}$ in homogeneous coordinates
Result: Multiplicity $m$ for one opposite vertex
1 $\mathbf{X} = C - A$;
2 $\mathbf{Y} = (l_x - a_x l_w, l_y - a_y l_w, l_z - a_z l_w)$;
3 $\mathbf{N} = \mathbf{X} \times \mathbf{Y}$;
4 $m =$ sign$(\mathbf{N} \cdot (\mathbf{B} - \mathbf{A}))$;
```

Algorithm 9: This algorithm transforms tessellation coordinates into the vertex of shadow volume side. Vertices $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ are computed using Equation 5.2.

```
Data: Vertices $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$, tessellation coordinates $x, y \in \langle 0, 1 \rangle$ and multiplicity $m$
Result: Vertex $\mathbf{V}$ in world-space
1 $\mathbf{P}_0 = \mathbf{A}$;
2 $\mathbf{P}_1 = \mathbf{B}$;
3 $\mathbf{P}_2 = \mathbf{C}$;
4 $\mathbf{P}_3 = \mathbf{D}$;
5 $a =$ round$(x \cdot m)$;
6 $b =$ round$(y)$;
7 $id = a \cdot 2 + b$;
8 $t = (id \mod 2)$ ^ $(\lfloor id/4 \rfloor \mod 2)$;
9 $l = \lfloor (id + 2)/4 \rfloor \mod 2$;
10 $n = t + l \cdot 2$;
11 $\mathbf{V} = \mathbf{P}_n$;
```

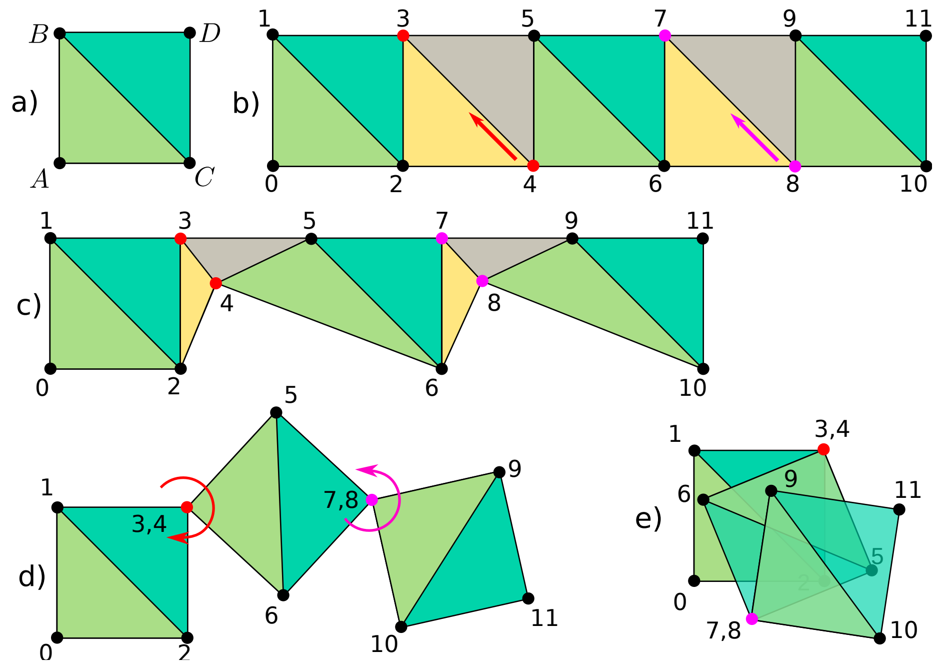


Figure 5.6: The image shows the transformation of a quad into three overlapping shadow volume sides. The transition from part a) to part b) is tessellation of quad with Multiplicity = 3. Only green and blue triangles will be drawn. Yellow and gray triangles will be degenerated. The transition from part b) over part c) to part d) shows degeneration process. Red and purple vertices 3, 4 and 7, 8 from part a) form only one vertex in part d). The transition from part d) to part e) shows rotation around red and purple vertices. This transformation creates three overlapping sides of shadow volume. Positions of vertices A, B, C, D that form initial quad, can be computed according to Equation 5.2.

5.2.3 Implementation

All methods are implemented in Lexolights, an open-source multi-platform program based on OpenSceneGraph and Delta3D, using OpenGL.

Single-pass per-triangle method suffers from inconsistent rasterization of two identical triangles at the same depth but with different winding - depth of fragments from both triangles differs, which resulted in z-fighting artifacts. The algorithm has to push the front cap's fragments into depth of 1.0f, so they would fail the depth test, otherwise self-shadowing artifacts can be observed. Bypassing early depth test in rasterization due to assigning depth values in fragment shader causes significant performance loss over two-pass method. This method served as a basis for silhouette-based approach.

For caps generation in silhouette-based method, the algorithm uses gemetry shader and multiplicity calculation. It calculates triangle's orientation towards light source via reference edge. It was also necessary for keeping calculations consistent throughout the rendering process of shadow volumes.

Because tessellation factors are limited to 64, there is also a limit of maximum multiplicity per edge that this algorithm is able to process. According to equation to calculate tessellation factor $Multiplicity \cdot 2 - 1$, maximum multiplicity of an egde is 32, which should be more than enough for majority of models. But, for example, well-known Power Plant

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 5.1: Performance of two deterministic methods measured in FPS on Sponza scene at different resolutions. First column for each method denotes original algorithm (3 edges) and second value denotes our new method using a single reference edge.

model (12M triangles) has some edges, which have multiplicity of 128. In that case, they would have to be split into more edges.

5.3 Experiments

Methods are compared against already available shadow volumes implementations on modern hardware - robust geometry shader implementation and standard shadow mapping. They are also evaluated against Sintorn's AFSM [91]. Two-pass per-triangle method is also tested against similar geometry shader implementation. For shadow volumes approaches, z-fail was used; shadow map resolution was 8k x 8k texels.

Testing platform had following configuration: Intel Xeon E3-1230V3, 3.3 GHz; 16GiB DDR3; GPUs: AMD Radeon R9 280X 3 GiB GDDR5, nVidia GeForce GTX 680 2 GiB GDDR5; Windows 7 x64; driver version: 13.12 (AMD), 334.89 (nVidia).

5.3.1 Testing Scenes

camera flythroughs were created in two testing scenes, each having one point light source.

- Sphere scene: synthetic scene containing adjustable number of spheres (typically 100) with configurable amount of details. Flythrough has 16 seconds.
- Crytek Sponza: popular model used to evaluate computer graphics algorithms. 262 267 triangles, 40 seconds.

5.3.2 Results

Crytek Sponza scene was used in the first set of tests. Tests evaluated the method in different resolutions and compared it to the geometry shader implementation. Also, both implementations were compared to older variant of deterministic multiplicity calculation as seen in the Table 5.1 and in the Figure 5.7. Geometry shader implementation provides more stable frames per second (FPS) with change of resolution, but fails to outperform the new tessellation algorithm on both graphic cards. Moreover, the new determinism method is faster in majority of cases, with some exceptions in geometry shader implementation, where it is on-par with older type of determinism, using all 3 edges.

Majority of tests were performed on a sphere scene with adjustable amount of geometry. First, a simple flythrough in a scene containing 100 spheres was used with different amount of triangles per test, the results can be seen in the Table 5.2 and graph in the Figure 5.8.

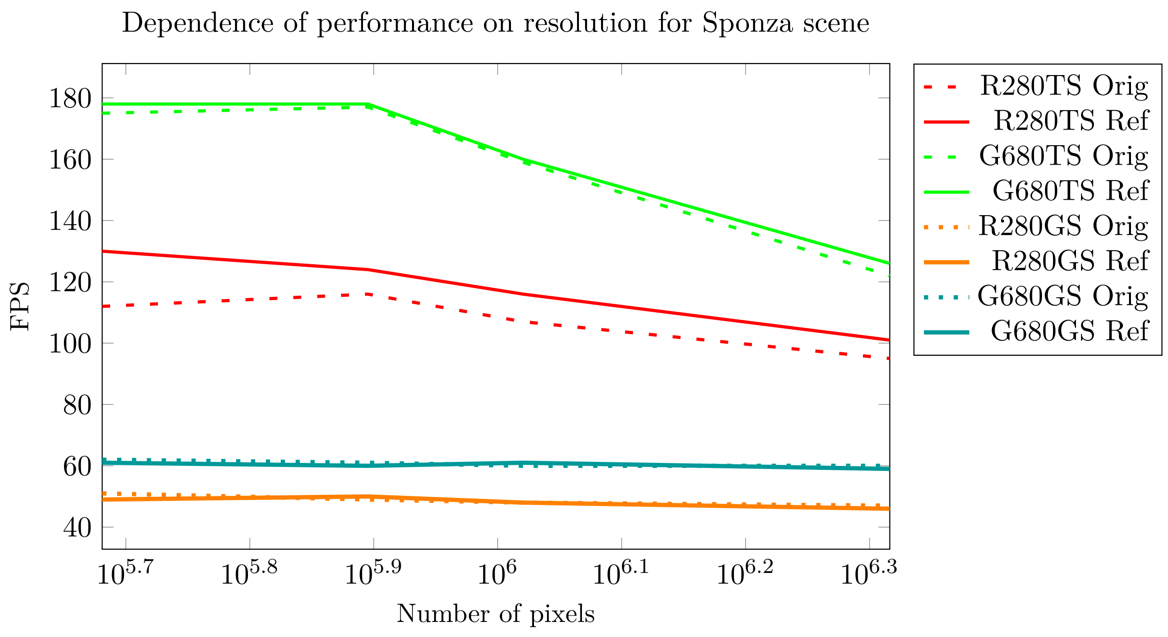


Figure 5.7: Dependence of performance (FPS) on resolution of original and new method, measured on Sponza scene.

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 5.2: Performance of two determinism methods measured in FPS on a scene with 10x10 spheres at different triangle count.

On GTX680, tessellation using reference edge is the fastest, no matter the number of triangles, although the performance gap gets smaller with increasing number of triangles in a scene. R9 280X showed different results, tessellation was more than 2x faster when the scene contained only 32K triangles but at approximately 300K, geometry shader method took a lead.

Another testing scenario contained a scene with only 1 box (12 triangles) and a sphere with different amount of Table 5.3, results can be seen in the Table 5.3 and in the Figure 5.9. The results have the same characteristic as those from a scene having 100 spheres - tessellation is dominant on GTX680, whereas on Radeon R9 280X it is geometry shader. In previous measurement, geometry shader became dominant at about 3500 triangles per sphere, which is also observable in this measurement.

This test was further extended to performance dependency on number of objects in a scene while maintaining constant amount of geometry. This measurement was carried out

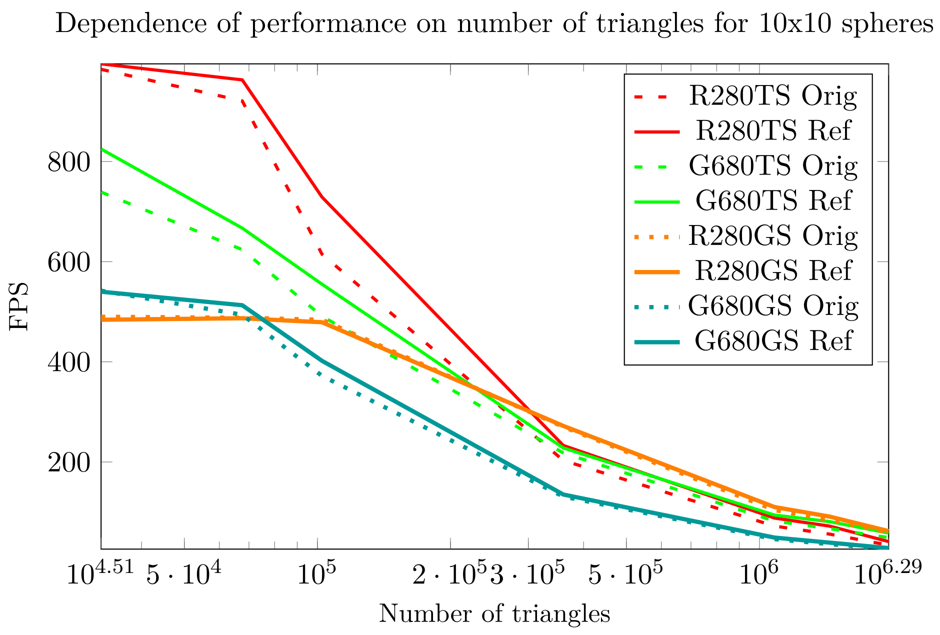


Figure 5.8: Dependence of performance (FPS) on number of triangles on a scene with 10x10 spheres using original and new deterministic method.

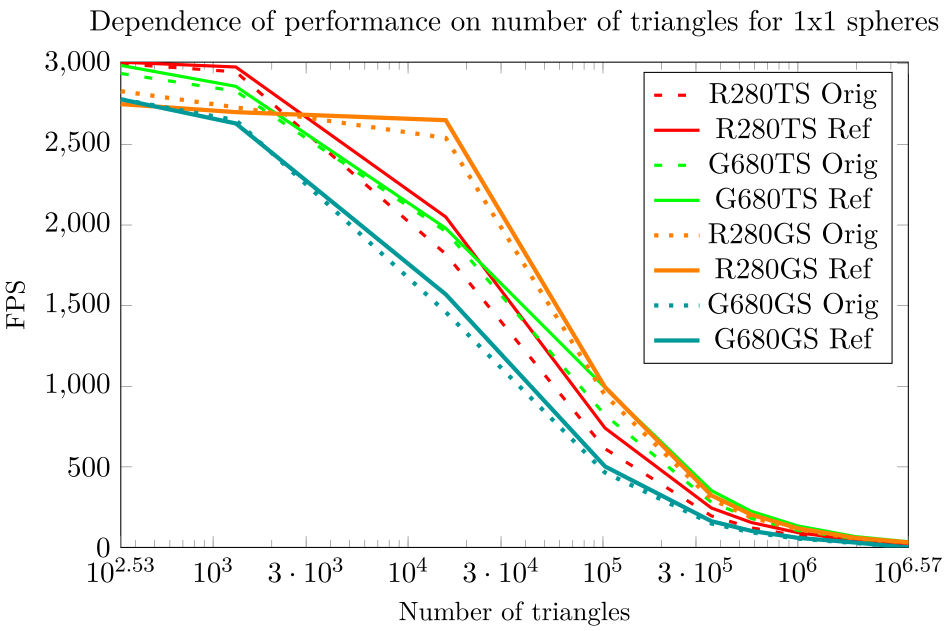


Figure 5.9: Dependence of performance on number of triangles for one tessellated sphere for original and new method.

on Sphere scene, having 1 million triangles (with max 2% deviation) in every case. No hardware instancing was used, every object was drawn via separate draw call. Results can be seen in the Table 5.4 and in the Figure 5.10.

Contrary to previous measurements, tessellation was faster on R9 280X, starting from 10^3 objects, although reference edge was faster only in 40% cases. Moreover, as can be

Spheres1x1	R280				G680			
Triangles	TS		GS		TS		GS	
336	3000	**3010**	2830	2750	2940	**2990**	2780	2780
1308	2950	**2980**	2730	2700	2830	**2860**	2650	2630
15612	1820	2050	2540	**2650**	1960	**1980**	1460	1570
102412	613	741	945	**995**	822	**992**	465	503
360000	196	245	299	**323**	281	**353**	149	163
577600	122	155	186	**203**	180	**224**	94	103
1000000	73	92	111	**119**	106	**134**	55	60
1960000	37	46	56	**60**	54	**68**	28	31
3686400	19	25	30	**32**	29	**36**	0.5	0.5

Table 5.3: Performance of two deterministic methods measured in FPS on a scene with 1 sphere having different amount of triangles. First value denotes original method and second value denotes reference edge based method. On GTX 680, GS and 3.6M triangles, the GPU ran out of its memory and performance dropped to 0.5 FPS.

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 5.4: Dependence on number of objects for spheres scene with 1M triangles. Bold values represent the fastest algorithm/implementation per GPU.

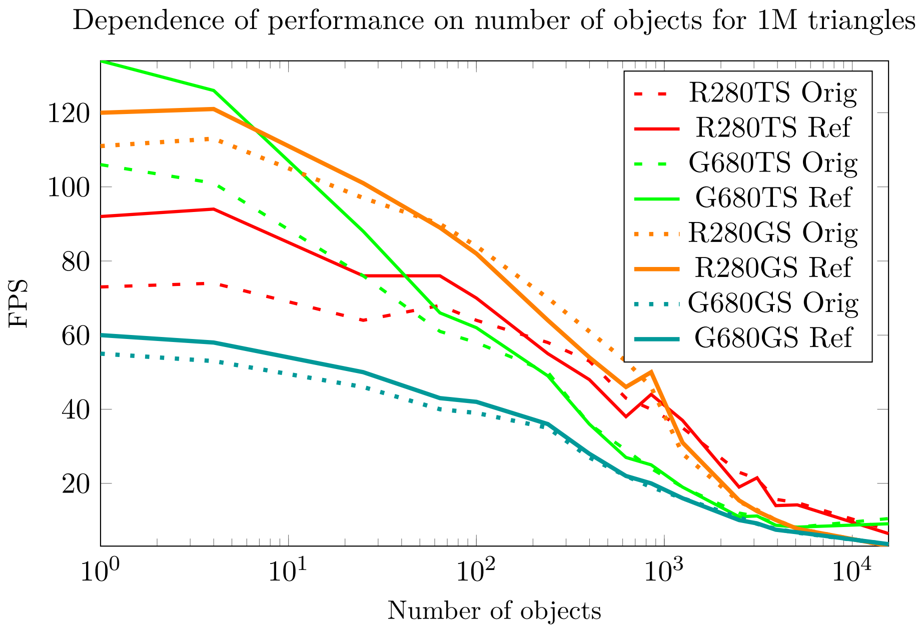


Figure 5.10: Dependence on number of objects for spheres scene with 1M triangles.

seen in the Figure 5.10, there is a slight increase in FPS in both geometry shader and tessellation implementations at about 1000 objects on Radeon. On GTX680, tessellation method was faster in every case; reference edge provided increased performance only in a half of measurements, but in all other cases the difference was only 1-3 FPS.

Sintorn in his AFSM paper [91] stated that his per-pixel precise shadow maps are 3-times slower than standard 8Kx8K shadow mapping. In order to evaluate the algorithm against AFSM, a shadow mapping test was conducted with resolution metioned above. Results can be seen in the Table 5.5 and Figure 5.11.

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 5.5: Shadow Mapping vs Tessellation Silhouettes, 10x10 sphere scene, FPS

Not only the novel algorithm managed to outperform shadow mapping with triangle count up to ~400K triangles, but at almost 2M triangles the method was on par or faster than AFSM - R9 280X dropped to 34% of SM performance whereas GTX680 was only 44% slower than 8K shadow mapping.

Following tests compared silhouette methods with two-pass per-triangle tessellation implementation and 8K shadow mapping, results can be seen in the Table 5.6 and in the Figure 5.12.

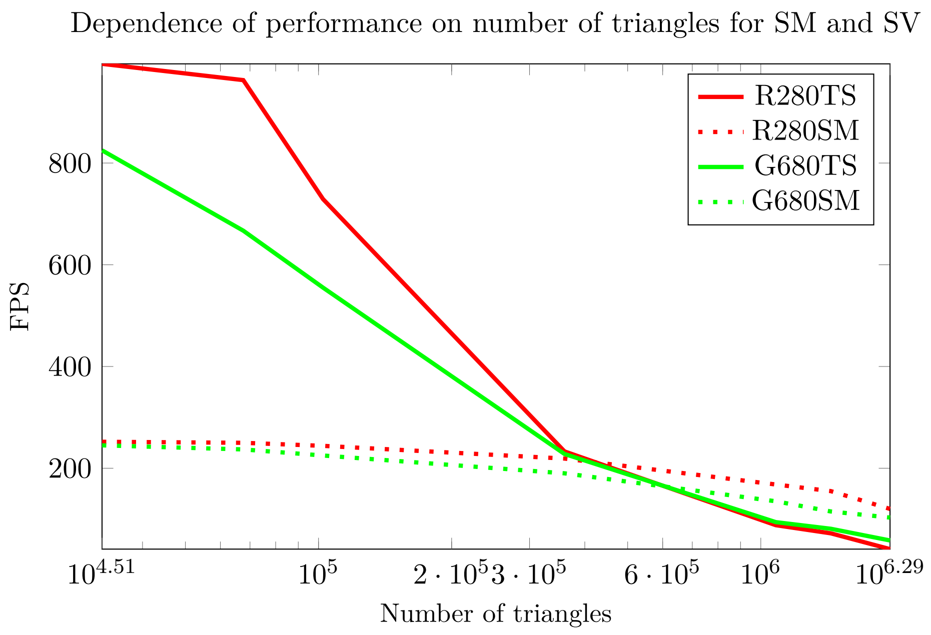


Figure 5.11: Shadow Mapping vs Tessellation Silhouettes on a scene with 10x10 spheres, measured in frames per second.

	R280		G680	
Method	Spheres	Sponza	Spheres	Sponza
TS Triangle	5.8	102	7.9	83
TS Silhouette	23.7	130	32	185
GS Triangle	3.1	51	4.9	73
GS Silhouette	34	49	14.8	62
SM	93	0	74	0

Table 5.6: Overall comparison of GS, TS methods and classic 8K shadow mapping on testing scenes - Sponza, and Spheres with 4M triangles. Shadow mapping was not evaluated on Sponza scene (zeros).

One can observe that per triangle tessellation method is even faster than both geometry shader methods running on Sponza scene. It is also worth noting that per-triangle geometry shader based method provides more performance on this scene than silhouette-based approach. On GTX680, the difference between silhouette and per-triangle tessellation method is 122%, whereas on R9 280X card it is faster only by 27%.

With increased amount of geometry in synthetic test scene, the situation turns around in favor to silhouette methods. Also the performance difference between shadow mapping and tessellation on Radeon drops under 1/3 ratio, but GeForce is still able to maintain 43% of SM performance.

5.4 Conclusions

Three new shadow volume methods were developed using tessellation shaders.

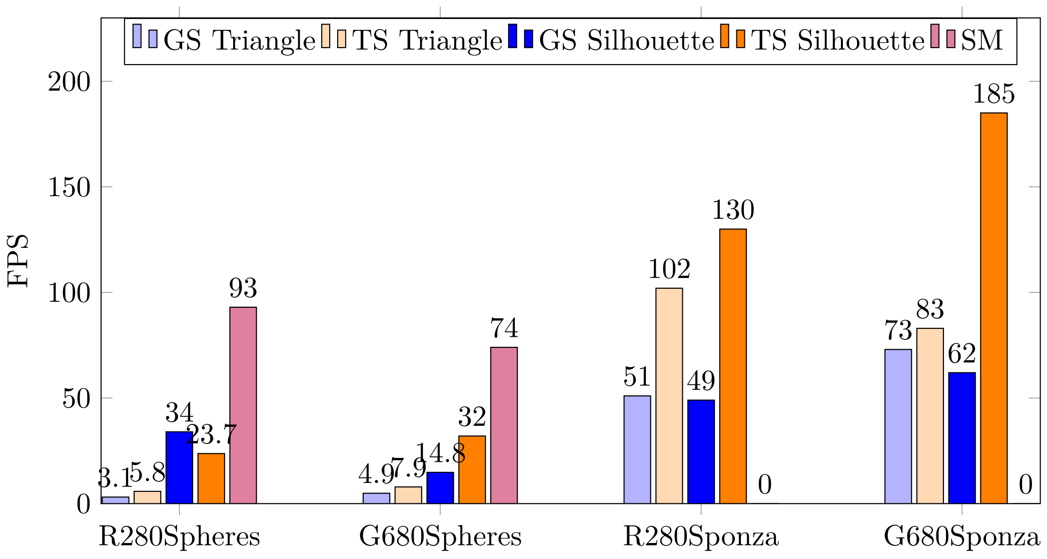


Figure 5.12: Overall comparison of methods on testing scenes - Sponza and Spheres with 4M triangles.

Spheres 100k	R280				G680			
Objects	TS		GS		TS		GS	
1	630	765	962	**1025**	825	**1003**	470	510
4	561	657	881	**920**	660	**742**	410	441
25	534	583	630	**645**	445	**453**	340	352
64	443	448	**485**	475	**297**	289	273	273
100	485	**531**	403	408	234	**237**	211	212
225	**307**	258	199	203	**147**	145	124	125
900	99	**103**	48	49	**61**	61	45.5	45.7
2500	34	**36**	16	17	28.5	**28.6**	17	16.8
3600	24	**25**	11	12	**20.3**	20	11.8	11.7

Table 5.7: Dependence on number of objects for spheres scene with 100k triangles.

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 grows. Compared to geometry shader per-triangle implementation, it was faster in every measurement.

Tests and measurements show that the silhouette method is more efficient, mostly in scenes with higher amount of geometry. GeForce GTX680 benefited mostly from this algorithm, being faster than geometry shader silhouette method. As for Radeon R9 280X, geometry shader method is more suitable. Tessellation method on Radeon proved to be faster in Sponza scene, but synthetic tests on sphere scene showed that its 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. In less detailed scenes it was able to outperform nVidia-based card, but only up to aforementioned 300K triangles.

The robust algorithm was sped up by using a novel method of multiplicity computation based on reference edge, which was able to provide up to 31% performance gain in tessel-

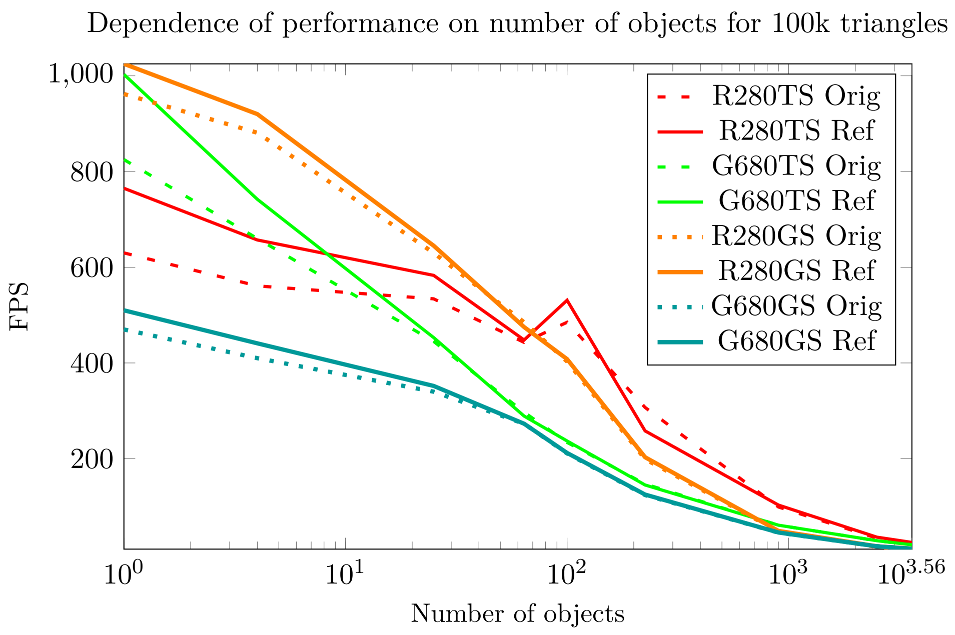


Figure 5.13: Dependence on number of objects for spheres scene with 100k triangles.

lation 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), the new 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 (it's 3-times slower than 8K SM) and is also simpler to implement.

Chapter 6

Fast Robust and Precise Shadow Algorithm for WebGL 1.0 Platform

This section presents fast and robust per-pixel correct shadow algorithm for WebGL platform. The algorithm is based on silhouette shadow volumes and it rivals the standard shadow mapping in terms of performance. The performance is usually superior when compared with high resolution shadow maps. Moreover, it does not suffer from a number of artifacts of shadow mapping and always provides per-pixel correct results.

WebGL 1.0 provides just vertex and fragment shaders. Thus, the algorithms evaluate silhouette edges in vertex shaders. Specially precomputed data are fed to the vertex shaders that extrude shadow volume sides just for silhouette edges. Some optimizations are deployed for performance and data size reasons that are important especially on low performance configurations, such as cost-effective tablets and mobile phones. This section also evaluates the solution on number of models. The solution performs on par with high resolution omnidirectional shadow mapping. The full content of this section can be found in paper [75].

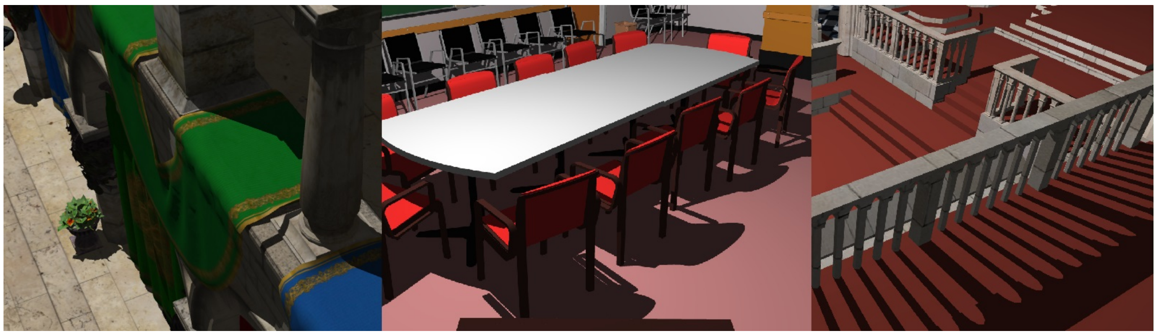


Figure 6.1: Three testing scenes - Sponza, Conference Room and Sibenik rendered using WebGL on NVIDIA Shield.

6.1 Introduction

Shadows are an important visual cue for human perception of 3D space, providing human brain with additional information about the structure of the scene that is usually visualized as a 2D image on the computer screen.

Two popular methods for shadow visualizations are used nowadays - shadow volumes [17] and shadow mapping [106]. Shadow mapping got its hardware acceleration in 2001 with GeForce 3 and became popular shadow method in computer games and other areas. However, shadow mapping is prone to visual artifacts [65]. Tremendous amount of research was done to lessen these artifacts to some extent, for instance [96][107][110][88]. Another approach is to adjust the scene design in a way that artifacts will not become visible. Such approach is probably largely used in computer game industry but it is usually not acceptable in CAD systems, where precise visualization of the model is required.

Shadow volumes got their hardware support in 1991 when hardware stencil buffer was introduced [41]. Unfortunately, Heidmann's algorithm that became known as z-pass was not robust when the viewer himself was in shadow. Thus, [14] and [9] proposed z-fail algorithm. Finally, [28] presented robust z-fail algorithm that should provide correct visual results for any arbitrary scene.

As shadow volumes work partially in image space, they are capable to deliver per-pixel precise shadows. However, rasterizing large shadow volume extents made them fill-rate limited in most cases. The extrusion of shadow volumes only from silhouette edges, instead of every edge, became the main performance optimization. The algorithm for finding silhouette edges on manifold meshes was described by [8]. [12] was the first to show the silhouette algorithm implemented entirely in graphics hardware. [69] described silhouette algorithm implemented in vertex shader only using specially precomputed mesh. [105] developed optimized silhouette construction using SSE2 instructions for computer game Doom 3. [97] used then-new geometry shader for silhouette algorithm.

Many silhouette algorithms require 2-manifold triangle meshes to work correctly. More general algorithms were proposed to alleviate this restriction. [3] requires the mesh to be orientable only. [52] showed an algorithm that works on arbitrary meshes, however it exhibits visual artifacts from time to time due to limited numerical precision of both GPU and CPU computing units. [85] made the algorithm robust and verified its artifact-free property on a number of computing platforms. [72] showed the further performance gains when implemented using tessellation shaders.

6.2 Algorithm

The algorithm is based on the robust silhouette algorithm developed by [85] and enhanced algorithm by [72]. However, they present CPU, multi-core CPU, geometry shader, OpenCL and tessellation shader implementations in their papers, while the only GPU computing capabilities available in WebGL 1.0 are vertex and fragment shaders. This algorithm uses vertex shader and it is based on the idea of [69] where vertex shader is fed by a specially constructed mesh data. These mesh data processed by vertex shaders, extrudes shadow volume only on silhouette edges. [69] designed solution just for 2-manifold meshes so this new method merges McGuire's approach with the algorithm presented in [85]. Furthermore, following sections present a number of data-related optimizations for memory footprint reduction.

6.2.1 Overview

The core idea of the algorithm is to find the edges that form the outline of the possible silhouette on the model for the given light position. Then, the edges of the silhouette are extruded to infinity and closed by caps at the model and at the infinity as required by z-fail algorithm, forming the shadow volume of the model. The silhouette is subset of all edges of a model that satisfies the following condition: edge is considered as silhouette edge when number of light-facing and light-back-facing triangles adjacent to this edge is not equal.

The computed shadow volume is used for shadow visualization using traditional stencil z-fail approach that is described in detail in [24]. Two-sided stenciling optimization is used as it is supported by WebGL 1.0. Briefly, the algorithm works in three steps:

1. render the regular scene, producing z-values to the z-buffer and producing scene with ambient light to the color buffer
2. render the stencil mask using the shadow volume geometry and set stencil function to act whenever z-test fails; front faces are set to increment and back faces to decrement stencil value on z-test fail.
3. render the regular scene with the light switched on while setting stencil test in such a way that the color buffer is updated only in places with zero stencil value, e.g. update only lit regions.

The core of the algorithm lies within the step 2. The algorithm uses two shader programs - a program for sides-data and a program for caps-data. These programs compute Shadow Volume (SV) for particular scene transformation and light position.

6.2.2 Basic Input Data

First step of the algorithm is the construction of shadow-geometry. The shadow-geometry is composed of two parts: data for SV sides (sides-data) and data for SV caps (caps-data) (see Figure 6.5).

Data for sides are composed of all edges of input scene. Every edge is described with its vertices and a set of opposite vertices $\mathfrak{O} = \{\mathbf{O}_0, \mathbf{O}_1, \ldots, \mathbf{O}_{n-1}\}$. An edge and its opposite vertices form triangles that are attached to that edge. There are also additional data for every edge (see Section 6.2.4 and Section 6.2.5).

Caps-data are composed of triangles of input scene with additional information.

These two parts are preprocessed and stored in separate files along the scene files. Though that its not necessary, It is trade of between the on load construction and download time. Neither one of the two approaches influences the measurements of performance.

6.2.3 Multiplicity computation

The multiplicity is computed as follows: The algorithm constructs a plane using the light position and edge's two vertices. The plane is called a lightplane. The lightplane divides space into two subspaces. Then, the algorithm iterates through the opposite vertices of the edge and counts their number in both light plane subspaces. Difference between these two numbers is the edge multiplicity, see Figure 6.2. If it is zero, the edge is not silhouette edge and no edge extrusion happens. If it is non-zero, then absolute value of multiplicity gives the number of how many times the algorithm needs to extrude the side. The number of extrusion translates directly to the number of stencil buffer modification. The sign of the edge multiplicity gives the extruded quad winding order, which will either increment or decrement stencil buffer value.

6.2.4 Vertex Shader for Sides

In order to compute multiplicity, VS has to receive an edge, a set of opposite vertices and a light source. A light source position can be specified using uniform variable. The algorithm uses vertex attributes for edges and sets of opposite vertices (see Figure 6.3).

This section closely describes shader for sides and format of sides-data. Every edge of the scene can extrude n sides of SV (see [52]), where n is the number of attached triangles to this edge, see Figure 6.2. Let n be a maximal multiplicity - $maxMult$. Let $curMult$ be a currently computed multiplicity. It has the following property: $-maxMult \leq curMult \leq maxMult$. A computed multiplicity determines the number of sides extruded from the edge. The algorithm uses the algorithm for computation of multiplicity proposed in [72].

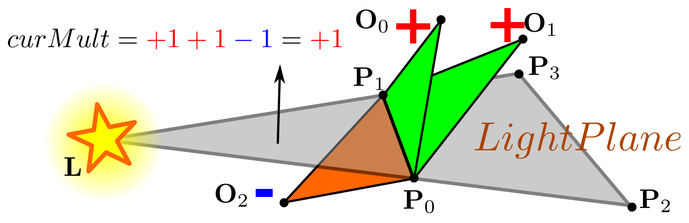


Figure 6.2: Edge $\mathbf{P}_0 \rightarrow \mathbf{P}_1$ has $n = 3$ opposite vertices: $\mathfrak{O} = \{\mathbf{O}_0, \mathbf{O}_1, \mathbf{O}_2\}$. Its maximum multiplicity is $n = 3$. Current multiplicity of this edge, given by light source and transformation, is $+1$. One quad $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ is extruded using edge and light source. This quad has the same orientation as light plane (black arrow).

The computation of multiplicity is accelerated using parallelization by vertex shaders (VS). In order to prevent writes and reads from memory the algorithm computes and draws shadow-data in one step. To make this streaming concept possible, VS has to receive all the data for all potential SV sides. Let s be an index of a potential side.

One side of the volume is composed of 2 triangles and 6 vertices. Let v be an index of vertex. The vertex shader has to receive $maxMult \cdot 6$ vertices per every edge. For example, 18 vertices for $maxMult = 3$. For every vertex, VS computes current multiplicity - $curMult$. $curMult$ determines whether a vertex is useful or not.

Useless vertex lies on a shadow volume side for which the condition $s > |curMult|$ is satisfied. All useless vertices are transformed to $(0, 0, 0)$. This transformation ensures that useless sides of SV will be degenerated and will not be rasterized.

Useful vertices have to be moved to one of the four possible positions: $\mathbf{P}_{id}$ (see Figure 6.2):

$$toInf(\mathbf{P}, \mathbf{L}) = (p_x l_w - l_x, p_y l_w - l_y, p_z l_w - l_z, 0) \tag{6.1}$$

$$\begin{aligned} \mathbf{P}_0 &= (a_x, a_y, a_z, 1) \\ \mathbf{P}_1 &= (b_x, b_y, b_z, 1) \\ \mathbf{P}_2 &= toInf(\mathbf{P}_0, \mathbf{L}) \\ \mathbf{P}_3 &= toInf(\mathbf{P}_1, \mathbf{L}) \end{aligned} \tag{6.2}$$

In Equation 6.2 $\mathbf{P}_0 \rightarrow \mathbf{P}_1$ symbolizes an edge and $\mathbf{L} = (l_x, l_y, l_z, l_w)$ symbolizes light position. id depends on index of vertex v and computed multiplicity $curMult$ according to Equation 6.3, Equation 6.4 and Equation 6.5:

$$id = \begin{cases} idCCW & \text{if } curMult > 0 \\ idCW & \text{otherwise} \end{cases} \tag{6.3}$$

$$idCW = \begin{cases} 6 - v & \text{if } v > 2 \\ v & \text{otherwise} \end{cases} \tag{6.4}$$

$$idCCW = \begin{cases} v - 2 & \text{if } v > 2 \\ 2 - v & \text{otherwise} \end{cases} \tag{6.5}$$

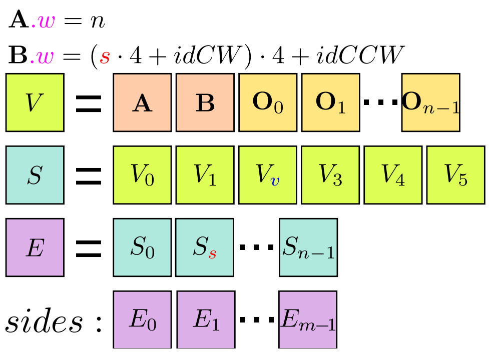

$\mathbf{A}.w = n$
$\mathbf{B}.w = (s \cdot 4 + idCW) \cdot 4 + idCCW$
V = $\mathbf{A}$ $\mathbf{B}$ $\mathbf{O}_0$ $\mathbf{O}_1$ ⋯ $\mathbf{O}_{n-1}$
S = V_0 V_1 V_v V_3 V_4 V_5
E = S_0 S_s ⋯ S_{n-1}
$sides$: E_0 E_1 ⋯ E_{m-1}

Figure 6.3: The image shows data structure for sides of SV. Every vertex (V) contains $n+2$ 4D vectors. **A** and **B** are vertices of an edge and $\mathbf{O}_j$ are opposite vertices. Forth components of **A** and **B** are used for special purposes - storing number of opposite vertices, index of side and indices $idCW$, $idCCW$. Side (S) contains 6 vertices (two triangles). If $maxMult$ is n, there has to be n sides for every edge (E). Sides data are composed of m edges of input model.

Pseudo code implemented in VS for sides-data (Figure 6.3) is in Algorithm 10.

Algorithm 10: Pseudo code for one vertex of a volume side extrusion in Vertex Shader.

```
Data: Edge vertices $\mathbf{A}, \mathbf{B}$, set $\mathfrak{O}$, light position $\mathbf{L}$, side id $s$, indices $idCCW$,
      $idCW$, model-view projection matrix $\mathbf{M}$
Result: $gl\_Position$
1 compute vertices $\mathbf{P}_i$ according to Equation 6.2;
2 $curMult$ = computeMultiplicity$(\mathbf{A}, \mathbf{B}, \mathfrak{O}, \mathbf{L})$;
3 if $s < |curMult|$ then
4     compute $id$ according to Equation 6.3;
5     $gl\_Position = \mathbf{M} \cdot \mathbf{P}_{id}$;
6 else
7     $gl\_Position = (0, 0, 0, 0)$;
```

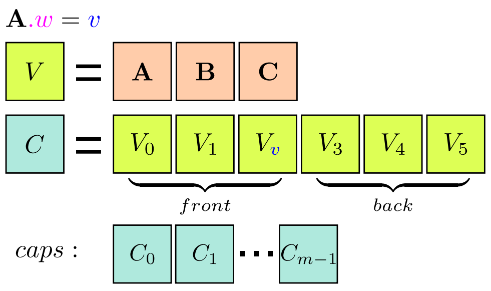
6.2.5 Vertex Shader for Caps

In order to compute multiplicity, VS has to receive all vertices of triangles and light source position. Similarly to sides, all vertices of a triangle are sent to VS using vertex attributes (see Figure 6.4).

A shader for SV caps works with scene triangles. Six vertices are needed in order to create a couple of caps (front and back cap). The vertex shader is therefore executed $6 \cdot m$ times, where m is the number of scene triangles.

In order to prevent errors, both caps have to be properly oriented. The orientation has to be the same as orientation of sides and has to be set deterministically. The algorithm uses deterministic computation of multiplicity destribed in [72]. Compared to sides, there is only one opposite vertex.

First, VS finds reference edge using method described in [72]. Then it computes multiplicity using third vertex and reference edge. The multiplicity determines orientation of particular triangle.



$\mathbf{A}.w = v$
V = A B C
C = V_0 V_1 V_v V_3 V_4 V_5
$front$
$back$
$caps:$ C_0 C_1 ... C_{m-1}

Figure 6.4: The image shows data structure for caps of SV. Every vertex (V) contains 3 4D vectors. $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are vertices of scene triangles. Forth component of $\mathbf{A}$ is used for special purposes - storing index of vertex v. Six vertices form couple of front and back cap. Caps data are composed of m couples for m triangles of input model.

Furthermore, in order to avoid z-fight, the method shifts front cap to infinity. After transforming the vertex into the clip space, it simply sets its z component to its w. The shifting can be seen on Figure 6.5. This ensures that the front cap always fails the depth test. The far plane clipping is avoided by using the modified projection matrix, which effectively sets the far plane into infinity [70]. Let $\mathbf{A} = (a_x, a_y, a_z, a_w)$ be vertex of front cap in clip space. The shifted vertex $\mathbf{B}$ toward infinity from camera can be computed according to Equation 6.6:

$$\mathbf{B} = (b_x, b_y, b_z, b_w) = (a_x, a_y, a_{\mathbf{w}}, a_{\mathbf{w}}) \tag{6.6}$$

Pseudo code implemented in VS for caps-data (Figure 6.4) can be seen in Algorithm 11.

6.2.6 Data Size Optimizations

The size of shadow-geometry can be large in comparison to size of the input scene. For example, an input scene having $e = 1000$ edges and $maxMult = 3$, the size of sides-data can be evaluated using Equation 6.7.

$$e \cdot maxMult \cdot 6 \cdot (2 + maxMult) \cdot 4 \cdot 4 = 1440000[bytes] \tag{6.7}$$

Every edge can produces up to 3 sides. Every side contains 6 vertices. Every vertex contains 5 vertex attributes with size of 4 floats.

Size of sides-data can be reduced $maxMult \cdot 6$ times. In the example shown in Equation 6.7, the reduced size would be $80000[bytes]$. Reduction can be achieved by using combination of instancing and drawing a shadow-geometry using multiple draw calls. The instancing is present in every current mainstream browser as an extension.

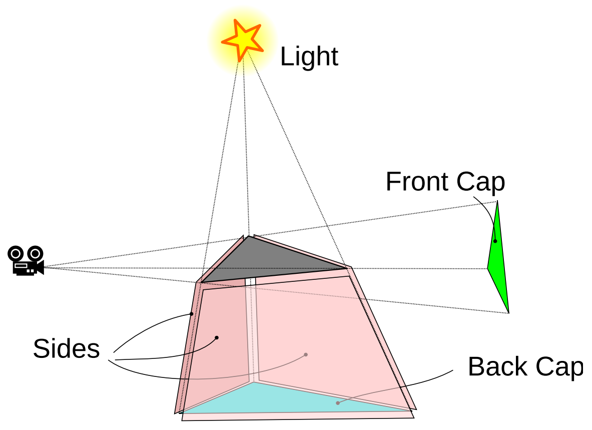


Figure 6.5: The image shows shadow volume that is constructed from a triangle. The front cap is shifted to infinity using homogeneous coordinates in order to prevent self-shadowing artifacts.

Algorithm 11: Pseudo code for caps in Vertex Shader.

```
Data: Triangle vertices $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$, light position $\mathbf{L}$, vertex id $v$, model-view
      projection matrix $\mathbf{M}$
Result: $gl\_Position$
1  $curMult = $ computeMultiplicity$(\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{L})$;
2  if $curMult = 0$ then
3      $gl\_Position = (0, 0, 0, 0)$;
4      return;
5  if $curMult < 0$ then
6      swap$(\mathbf{P}_0, \mathbf{P}_1)$;
7  if $v < 3$ then
8      $\mathbf{V} = \mathbf{M} \cdot \mathbf{P}_v$;
9      $gl\_Position = (V_x, V_y, V_w, V_w)$;
10 else
11     $gl\_Position = \mathbf{M} \cdot toInf(\mathbf{P}_{5-v}, \mathbf{L})$;
```

The modification is simple. First step is to remove additional data $idCW$, $idCCW$ and s from vertex attribute $B.w$, see Figure 6.3. After this step, every vertex V in edge E in the Figure 6.3 has the same data. Therefore, it is necessary to compute $idCW$ and $idCCW$ inside VS using $gl\_VertexID$ and Equation 6.3, Equation 6.4 and Equation 6.5. Unfortunately, WebGL 1.0 does not support $gl\_VertexID$, so the algorithm has to simulate its behaviour using additional vertex attribute. Also, the variable s has to be sent to VS using another vertex attribute. The variable s can be also computed using $gl\_InstanceID$, but it is not present in WebGL 1.0. Another option is to use multiple drawcalls and uniform variables. The final equation for computation of size of sides-data is as follows:

$$16 \cdot (2 + maxMult) \cdot e[bytes] \tag{6.8}$$

e is number of edges in input scene.

Size of caps-data can be also reduced using similar approach. The final equation for computation of size of caps-data is as follows:

$$48 \cdot t[bytes] \tag{6.9}$$

where t is number of triangles in input scene.

6.3 Measurements and Evaluation

The evaluation is focused on three aspects:

- performance comparison with shadow mapping methods
- performance on various devices using popular scenes
- synthetic tests to provide deeper insights

Shadow mapping cannot be directly compared to shadow volumes as shadow volumes provide omnidirectional shadows. Therefore, cube shadow mapping is selected as competing algorithm. Optimizations might be developed to use only cube faces that contribute to the final image. This often lessens the performance requirements to maintain only some faces of the cube shadow map, but indoor scenes, e.g. room with the light in the middle, often requires all the cube shadow map faces, depending on the camera view configuration. The algorithm is compared to full cube shadow mapping.

The comparison uses standard shadow mapping instead of one of its optimized versions, such as LiSPSM [107] or CSM [22]. These optimized versions focus on shadow visual quality which makes the algorithm more complex. This additional complexity does not generally improve the performance but has slightly opposite effect.

Regarding the second point, the algorithm is evaluated on several test scenes and hardware configurations, including handheld devices. Desktop measurements were carried out on high-end graphics hardware. Handheld platform was represented by NVIDIA SHIELD tablet with Tegra K1 SOC running Android 5.0.1, desktop platform by Radeon R9 290X and GeForce GTX 980 running in system with Intel Core i7 4790, 16GB of memory and Windows 7. The web browser was Firefox version 36.0 on both - desktop and NVIDIA SHIELD tablet. Firefox browser was chosen as it provided the best means for measurements. WebGL 1.0 lacks support for OpenGL queries, so the timing had to be manually measured, which was not possible in some browsers like Google Chrome and Opera because glFinish is implemented asynchronously in these browsers. The algorithm was measured using ANGLE[1] and with disabled ANGLE. The measurements were carried out on popular well-known scenes: Crytek Sponza, Conference Room and Sibenik Cathedral using scenes flythrough.

Regarding the third point, specially designed synthetic scenes were used to provide an insight into the algorithm performance in relation to the scene complexity and scene properties, in particular number of edges, max multiplicity, and screen resolution. Following graphs and tables illustrate the algorithm behavior for combination of these three. The main idea behind these metrics is that SV are thought to be mainly fillrate bound, thus change in resolution should change the performance. The cube shadow mapping was tested with

[1] *Almost Native Graphics Layer Engine*, graphics abstraction layer translating WebGL and OpenGL ES 2.0 into DirectX calls, used f.e. in Chrome or Firefox

	Sponza		**Conf. room**		**Sibenik**	
method	R290	G980	R290	G980	R290	G980
no shadow	3.3	3.9	1.0	0.9	0.7	1.1
SM1	13.5	13.8	4.0	3.5	1.7	1.6
SM2	13.4	14.1	5.2	3.8	3.0	2.4
SM4	16.1	15.3	9.8	6.4	8.5	5.7
SM8	37.0	21.8	28.2	15.5	28.9	18.7
SV	14.5	19.2	7.4	11.0	7.1	10.1

Table 6.1: Average frame time (ms) for flythroughs using ANGLE.

	Sponza			**Conf. room**			**Sibenik**		
method	R290	G980	Shield	R290	G980	Shield	R290	G980	Shield
no shadow	2.5	2.5	31.8	1.0	1.3	21.5	0.8	1.1	12.9
SM1	6.8	5.6	77.8	3.8	2.9	43.9	1.9	1.8	37.1
SM2	8.0	5.6	75.9	5.3	3.5	57.5	3.6	2.8	55.0
SM4	15.0	8.4	0.0	11.2	6.0	0.0	10.3	6.0	131.2
SM8	41.7	22.9	0.0	34.2	16.0	0.0	36.0	18.8	0.0
SV	14.7	22.2	207.5	8.0	15.3	165.4	7.6	13.1	73.0

Table 6.2: Average frame time (ms) for flythroughs without using ANGLE.

resolution of 2k, 4k and 8k per side. On NVIDIA SHIELD Tablet, only 2k by 2k was tested due to hardware limitations. Every test scene contains single point light source. The canvas resolution for flythroughs was 1920x1080, unless specified otherwise.

6.3.1 Popular Scenes

The tests were carried out on following well-known models: Crytek Sponza (262k triangles), Conference Room (331k triangles) and Sibenik Cathedral (75k triangles). Performance on these scenes has been measured by camera flythrough. Averaged frame rendering time for each scene is shown in Table 6.1, Table 6.2.

As can be seen in the Table 6.1, the new method is faster than 8k omnidirectional shadow mapping (e.g. 8192x8192x6) on both - high-end NVIDIA and AMD cards. Even on shadow mapping optimized pipeline, the new method is slightly faster. This is no longer true for 4k and 2k shadow mapping, however 2k and 4k shadow mapping does not provide sufficient shadow resolution, especially on modern high-resolution 4k displays, resulting in visual artefacts, see Figure 6.6. On the other hand, the proposed algorithm always provides per-pixel correct solution or per-fragment, if antialiasing is used, because shadow volumes perform its computation in screen space.

The frame times are shown in Figure 6.7, Figure 6.8, Figure 6.9, Figure 6.12, Figure 6.13, Figure 6.10, Figure 6.14, Figure 6.15, Figure 6.11.

Figure 6.7, Figure 6.12, Figure 6.14 show the results for GeForce GTX 980. As can be seen, shadow volume performance is view dependent. Anyway, the performance is higher than 8k shadow mapping for about two thirds of flythrough in Crytek Sponza scene.

On Radeon R9 290X, the performance of shadow volumes is more stable as shown in Figure 6.8, Figure 6.13 and Figure 6.15. Shadow volumes performance roughly equals to the performance of 4k shadow mapping, sometimes even rivaling 2k shadow maps.

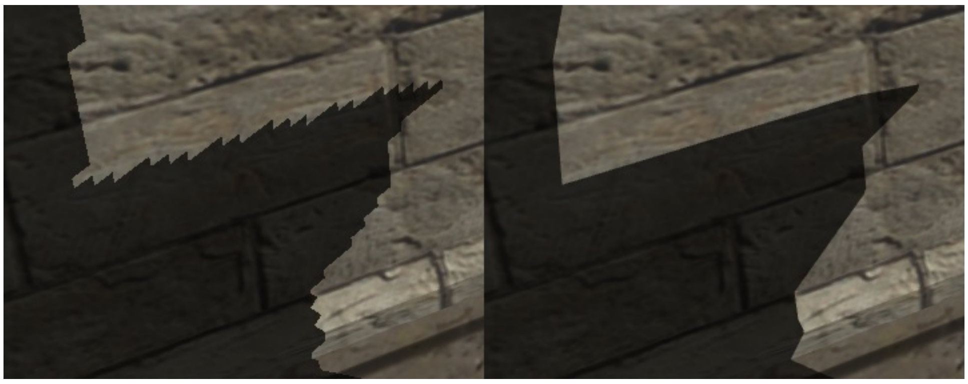


Figure 6.6: The image shows diference in quality between shadow mapping and shadow volumes.

Finally, the Figure 6.9 shows the performance on NVIDIA SHIELD tablet. The performance is not on interactive level, but we consider 4 FPS in FullHD on tablet for such large scene as acceptable. The performance is more stable than on GeForce GTX 980 (only about 20% fluctuation compared to 40%) and roughly only ten times slower than high-end NVIDIA GPU. This seems to go in hand with number of shader processors available on the devices, e.g. 192 vs. 2048. Unfortunately, using larger cube shadow maps was not possible on NVIDIA SHIELD device.

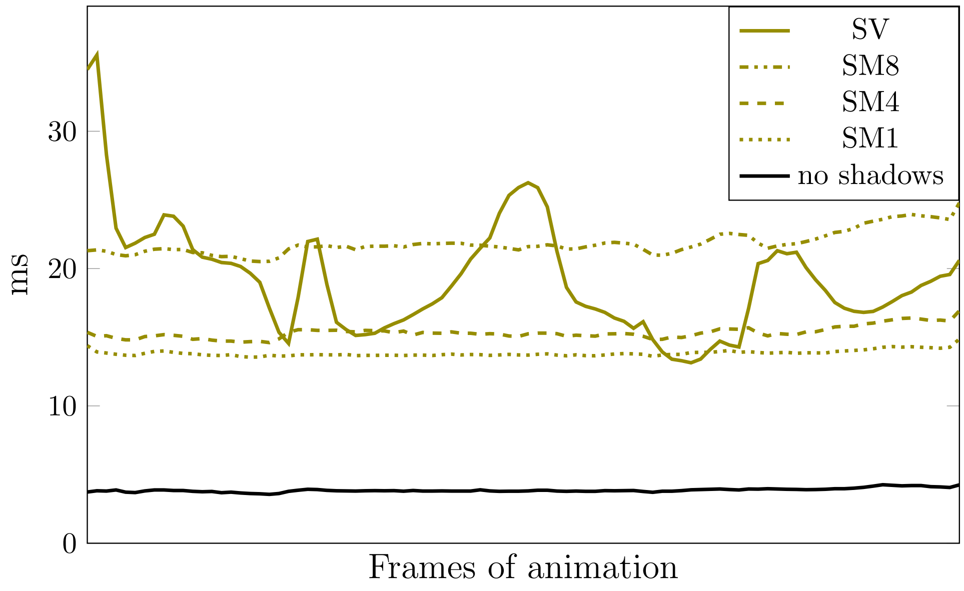


Figure 6.7: The figure shows flythrough Crytek Sponza on GeForce GTX 980. SM is very well optimized, even for large shadow map resolutions, but SV also manage to be real-time for most view points, although not as stable.

6.3.2 In-depth Performance Analysis

Next measurements were focused on rendering stages. The stages are:

1. rendering of ambient scene

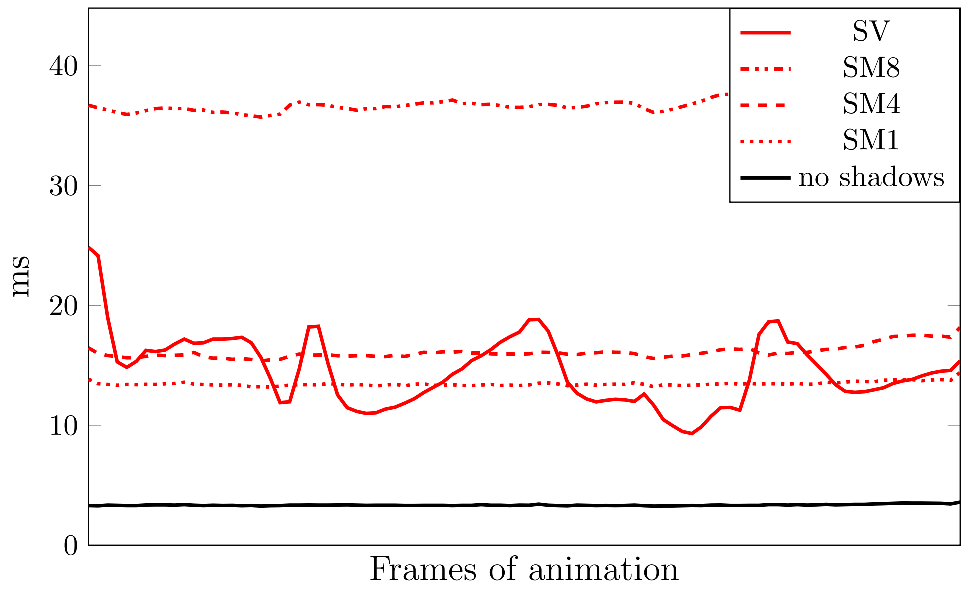


Figure 6.8: The figure shows flythrough Crytek Sponza on AMD platform. The SV seem to provide an interesting alternative to SM. While it behaves more stable than on NV platform, it also manages to be faster than 4k SM in average and completely outperforming 8K SM.

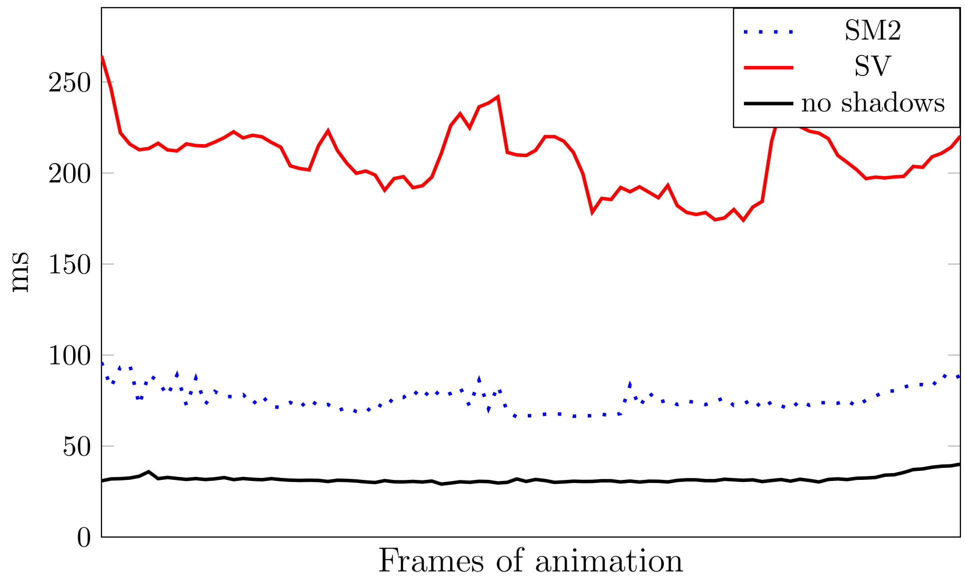


Figure 6.9: The figure shows flythrough Crytek Sponza on NVIDIA SHIELD. Both methods perform are slow and only interactive. Although SM performs better here, it must be taken into account that it is only 2k cubemap, compared to previous results where SV are on-par with 4K SM and difference between 2K SM and SV were much bigger. Thus quality-wise, shadow volumes are better alternative to SM on this platform as well.

2. silhouette edge set evaluation in vertex shaders

3. rasterization of shadow volumes

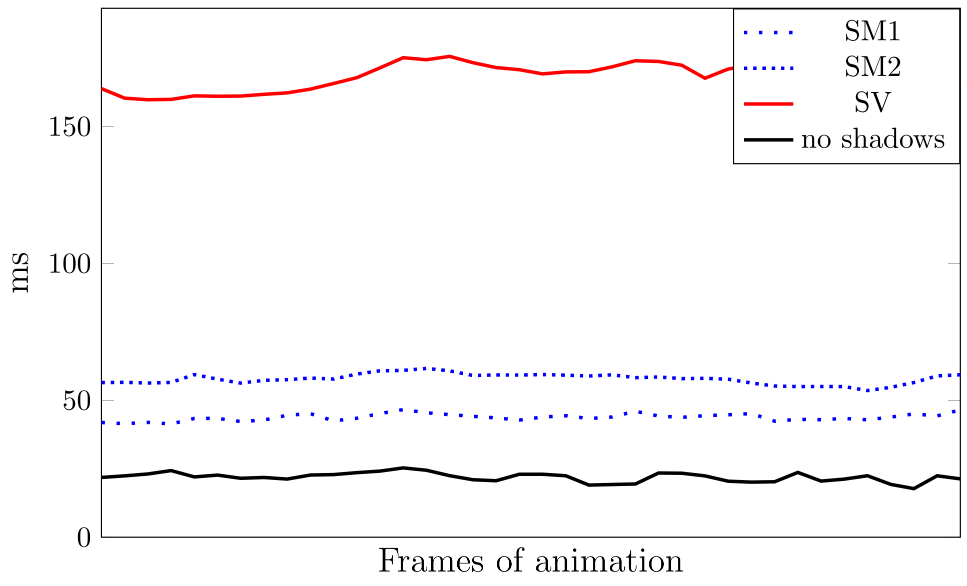


Figure 6.10: Flythrough for Conference room on NVIDIA SHIELD tablet.

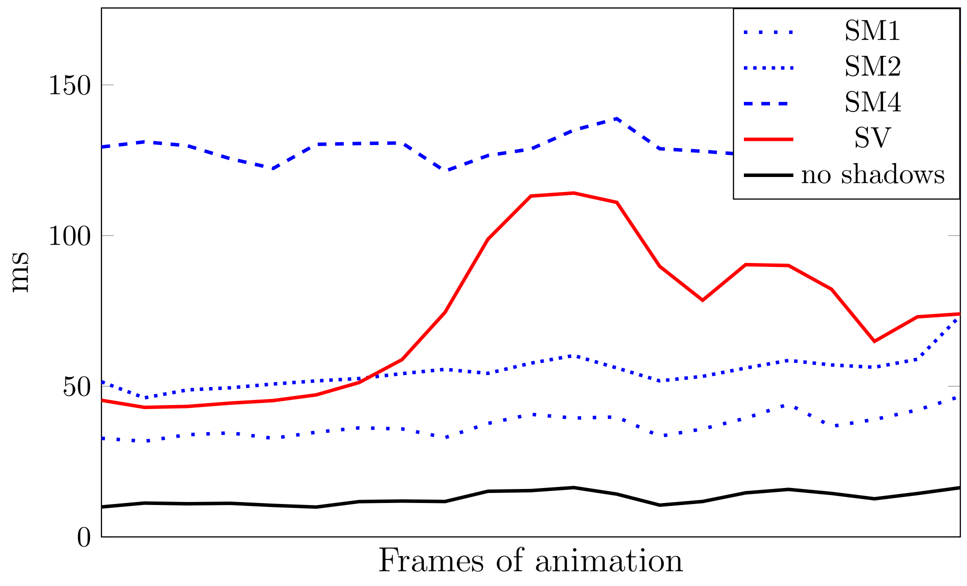


Figure 6.11: Flythrough for Sibenik scene on NVIDIA SHIELD tablet.

4. blending of lit parts of the scene

The results depend on screen resolution. Test were done with resolutions: 800x600, 1920x1080, 2560x1600 and 3840x2160, that corresponds roughly to 0.5, 2, 4 and 8 megapixels video output. The test was carried out on Sponza flythrough.

The results of flythroughs are shown in graphs for 1920x1080 and 800x600 resolution. The average values of flythroughts are presented in tables.

As can be seen in the Figure 6.16, the shadow volume rasterization is the most varying part. As the scene view changes during the animation, the shadow volumes changes occupied space on the screen, sometimes producing smaller, sometimes higher rasterization work.

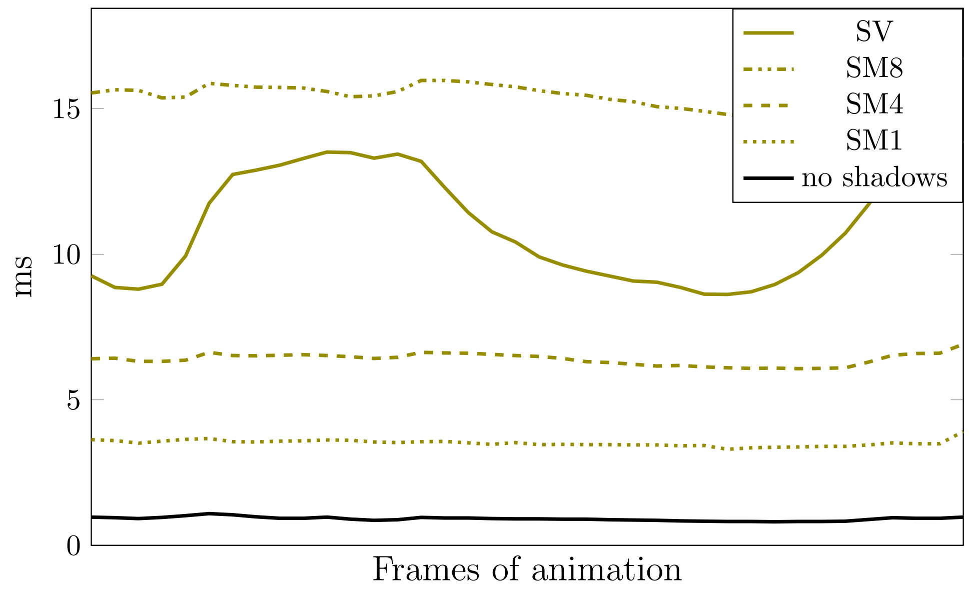


Figure 6.12: Flythrough for Conference Room scene on GeForce GTX 980.

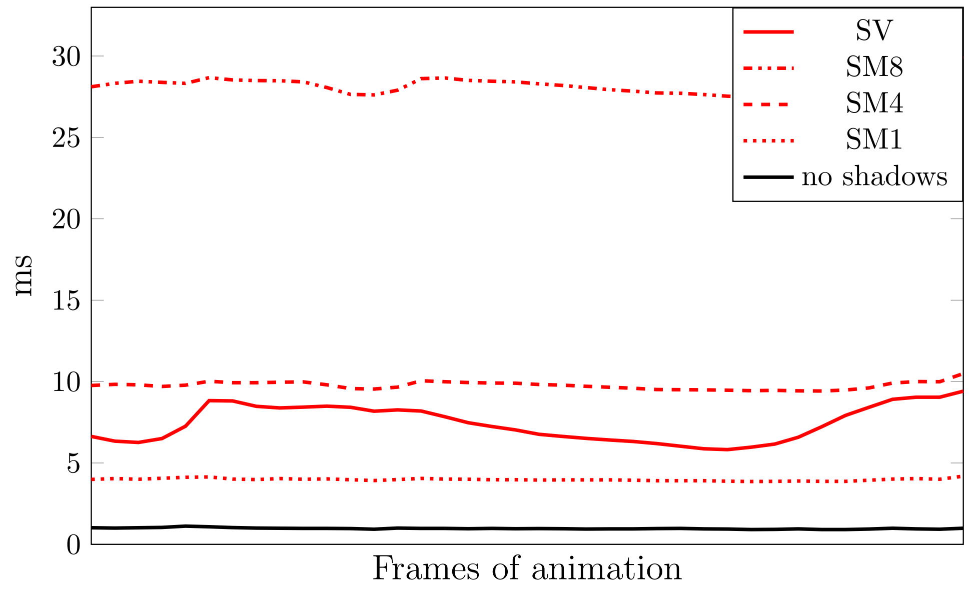


Figure 6.13: Flythrough for Conference Room scene on Radeon R9 290X.

Table 6.3 and Table 6.4 show average times of all four rendering stages on Sponza flythrough for four output resolutions. As can be seen from the results, the first and last stage (ambient scene rendering and blending of lit scene) grows rather slowly with increasing output resolution. For resolution increase from 0.5 to 8 megapixels, e.g. 16 times, ambient scene rendering time increases only about 50% on GeForce GTX 980 and by 25% on Radeon R9 290X. For blending stage, GeForce GTX 980 time increases only about 8% and Radeon R9 290X time increases by 25%. The second stage (silhouette edge set evaluation) takes constant time. Shadow volume rasterization grows rapidly. About 36 times on GeForce GTX 980 and about 16 times on Radeon R9 290X.

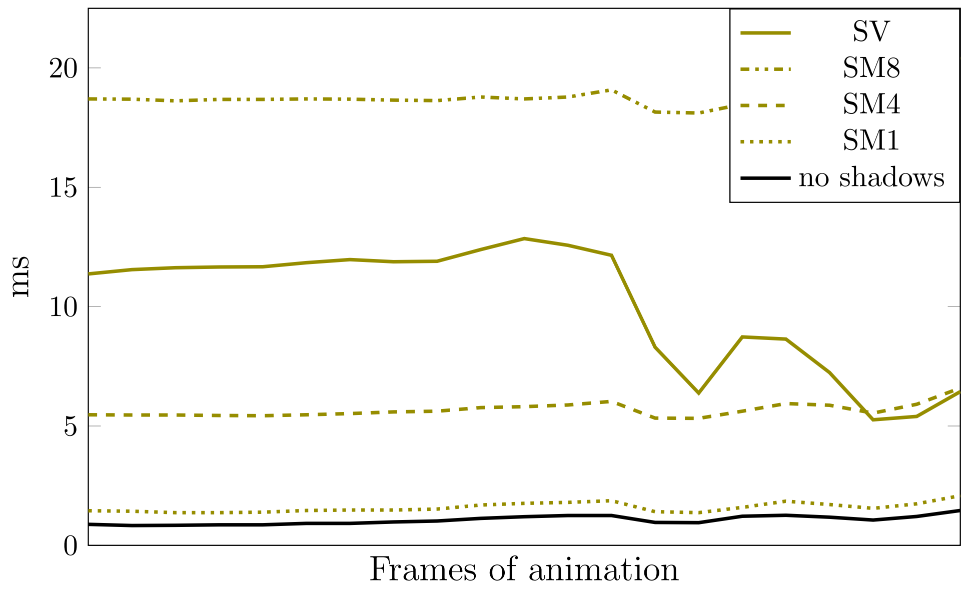


Figure 6.14: Flythrough for Sibenik scene on GeForce GTX 980.

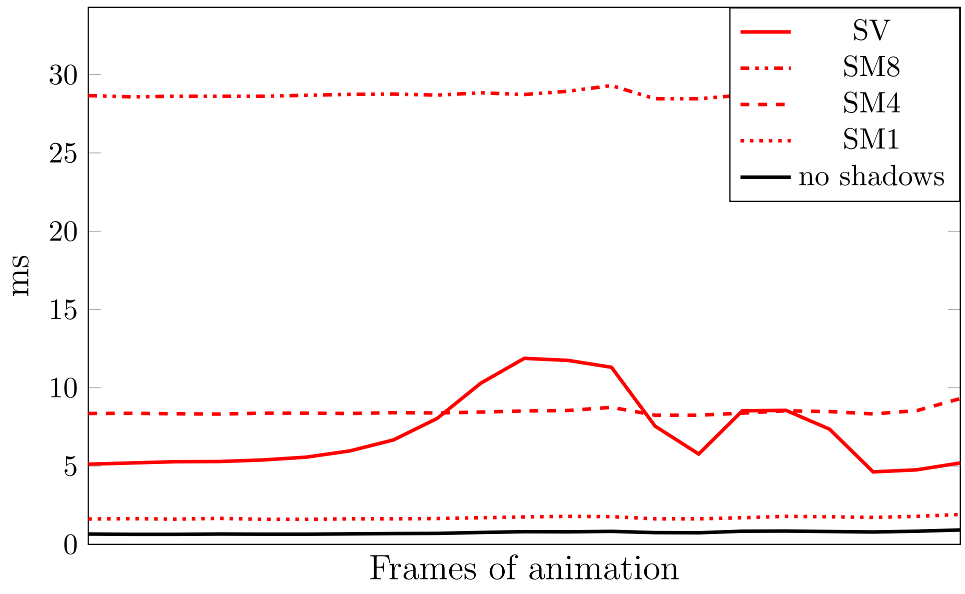


Figure 6.15: Flythrough for Sibenik scene on Radeon R9 290X.

6.3.3 Synthetic tests

Synthetic tests were designed to get more detailed insight into the algorithm performance characteristics. The core of the algorithm is the evaluation of edges whether they are silhouette edges or not. The task is performed by vertex shaders. The synthetic tests were focused on execution time of vertex shader for various numbers of edges and various edge multiplicities. To avoid side effects of shadow volume side rasterization, output triangles were degenerated at the end of vertex shader. Furthermore, all other rendering passes (ambient pass and final blending pass, numbered 1 and 3 in the Section 6.2.1) were removed. Thus, the measured frame time should be equal to the silhouette edge set computation plus

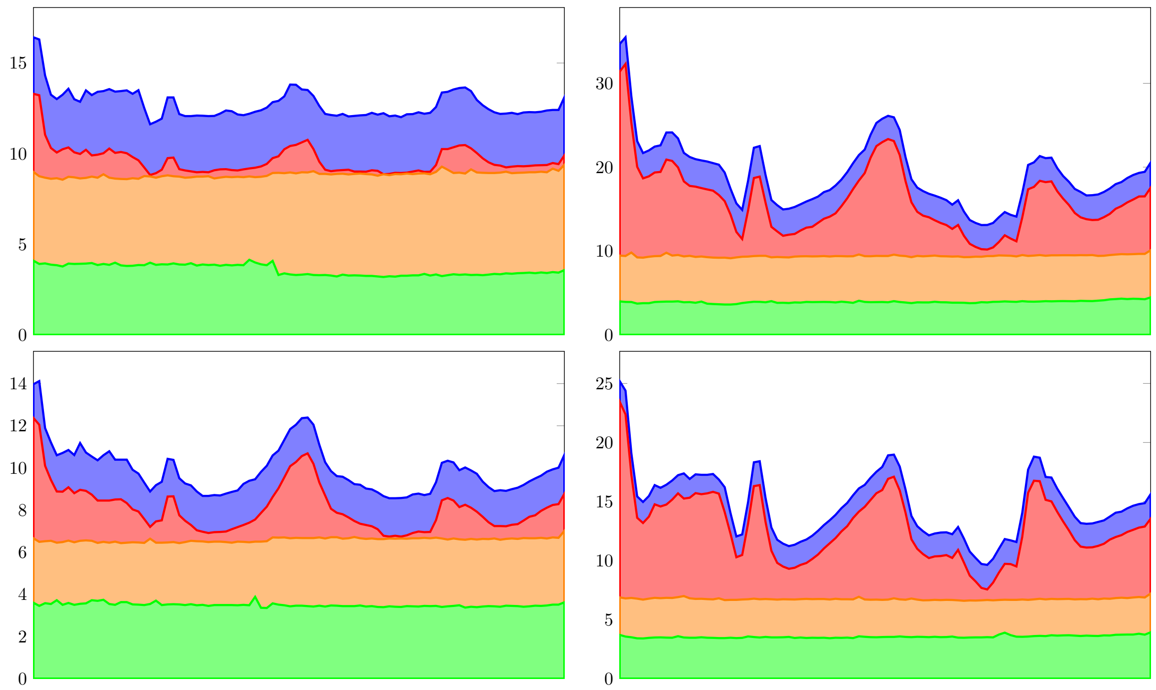


Figure 6.16: The figure shows timings of algorithm stages. Data lines in each sub graph represents stages; from bottom to top: ambient pass, computation of SV, rasterization of SV and blend pass. It was measured using flythrough in Sponza scene. Y axis is time in miliseconds and X axis represents animation frames. The top row shows measurements for GeForce GTX 980 and bottom row for Radeon R9 290X. The left column is for 800x600 resolution while the right column is for 1920x1080 resolution.

some frame overhead. The results for multiplicity 1,2 and 9 are shown in Table 6.5. All the results are shown on Figure 6.17, Figure 6.18 and Figure 6.19 for GeForce GTX 980, Radeon R9 290X and NVIDIA SHIELD tablet respectively.

For the edge multiplicity 1 and number of edges in model increasing from 10'000 to 90'000, the frame time grows linearly from 0.1ms to 0.6ms on GeForce GTX 980, from 0.07 to 0.35 on Radeon R9 290X and from 1.49 to 6.88 on NVIDIA SHIELD tablet. Using the approximation $c_0 e + c_1 = t$, the time of single edge computation (c_0) is 6.13ns and frame overhead (c_1) is 22.9us. For Radeon R9 290X, the edge evaluation time is 4.02ns and frame overhead is 33.0us.

Measured rendering times can be approximated using the Equation 6.10:

stage	800x600	1920x1080	2560x1600	3840x2160
4	3.1	3.1	3.4	3.8
3	0.8	6.7	14.3	28.8
2	5.3	5.5	5.2	5.8
1	3.6	3.9	4.6	5.3

Table 6.3: GeForce GTX 980 average times in milliseconds for Sponza flythroughs for all four stages and resolutions.

stage	800x600	1920x1080	2560x1600	3840x2160
4	1.8	1.9	2.0	2.3
3	1.5	6.1	12.2	24.0
2	3.1	3.2	3.1	3.2
1	3.5	3.5	3.9	4.4

Table 6.4: Radeon R9 290X average times in milliseconds for Sponza flythroughs for all four stages and resolutions.

edges	G980			R290			Shield		
9999	0.10	0.15	1.14	0.07	0.11	0.58	1.49	2.64	82.26
19998	0.15	0.28	2.23	0.10	0.18	1.11	2.70	4.77	155.46
30000	0.21	0.41	3.32	0.14	0.25	1.64	2.96	6.71	226.99
39999	0.28	0.54	4.39	0.17	0.32	2.16	3.45	9.02	302.42
49998	0.34	0.67	5.48	0.21	0.39	2.69	4.05	11.01	393.20
60000	0.40	0.80	6.56	0.24	0.46	3.22	5.02	13.32	445.64
69999	0.47	0.93	7.64	0.28	0.53	3.75	5.47	15.31	519.09
79998	0.54	1.05	8.73	0.31	0.60	4.27	6.14	17.28	592.94
90000	0.60	1.18	9.81	0.35	0.67	4.80	6.88	19.09	664.40

Table 6.5: Silhouette edge computation time for different edge counts and multiplicities. Three columns for each device represent 1,2 and 9 multiplicities.

$$f(e,m) = (k_0m^2 + k_2m + k_4)e + (k_1m^2 + k_3m + k_5) = t \tag{6.10}$$

Where m is edge multiplicity, e is number of edges in thousands and t is measured rendering time in milliseconds. Using weighted least square approximation, the following coefficients that characterizes each of our tested graphics devices were estimated:

The approximation by the Equation 6.10 allows easy computation of edge evaluation time for various multiplicities.

For the constant number of edges and increasing maximal multiplicity, the execution time grows quadratically as can be seen on Figure 6.17, Figure 6.18 and Figure 6.19. Radeon R9 290X seems to be faster about two times in edge evaluation time.

All the tests above were carried out on silhouette edges. For the purpose of verification, the special test composed of all non-silhouette edges was designed and tt verified that the

koef	G980	R290	Shield
k0	0.0008437815	0.0004229571	0.0940159968
k1	0.0017681964	0.0001344101	0.309259332
k2	0.0044036999	0.0017166787	-0.1703702257
k3	-0.0119359553	0.0017862163	-1.3526614536
k4	0.000884234	0.0014015652	0.1437560834
k5	0.0430166459	0.0333773652	2.022957563
MWSE [%]	0.9806859032	2.8303320077	9.4299797482

Table 6.6: Coefficients for the polynomial approximation on different platforms. The MSWE means mean weighted least square error.

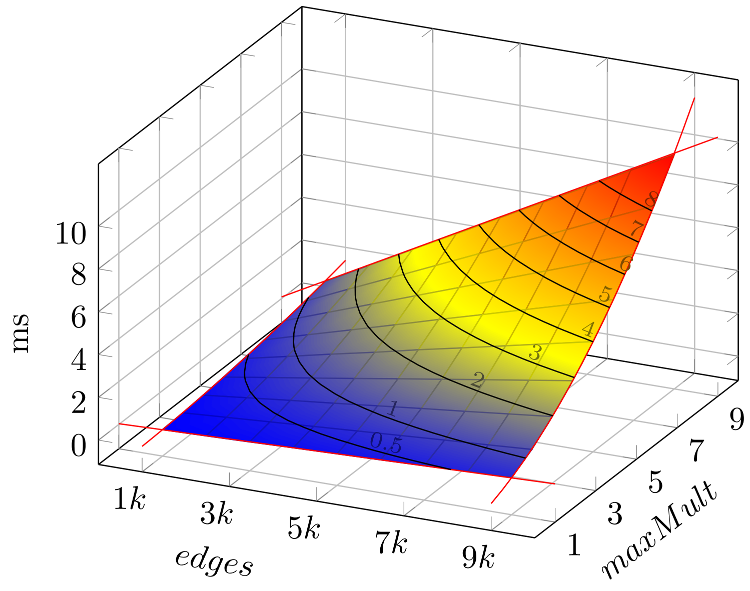


Figure 6.17: Behavior of the algorithm for sides implemented using VS on GeForce GTX 980. Red lines are approximated using polynomial in Equation 6.10.

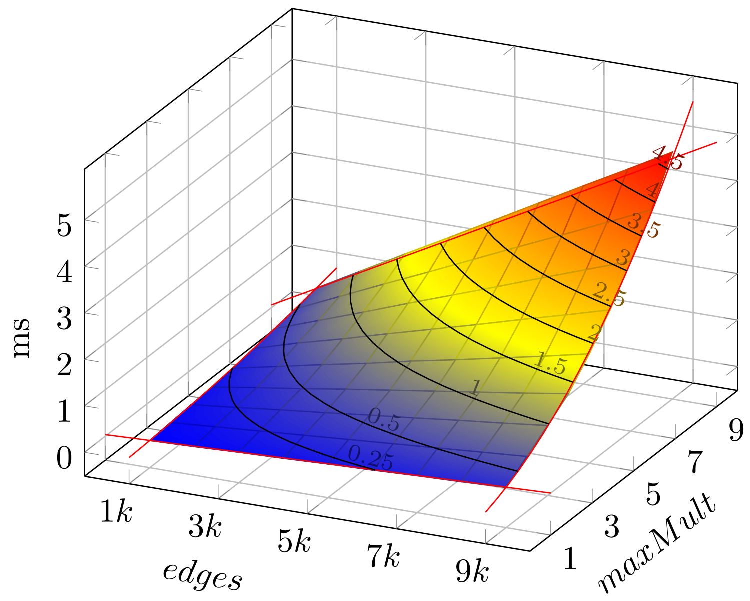


Figure 6.18: Behavior of the algorithm for sides implemented using VS on Radeon R9 290X. Red lines are approximation using polynomial in Equation 6.10.

execution time was the same. The test scene was a sphere with light source inside. In such scene, there are no silhouette edges.

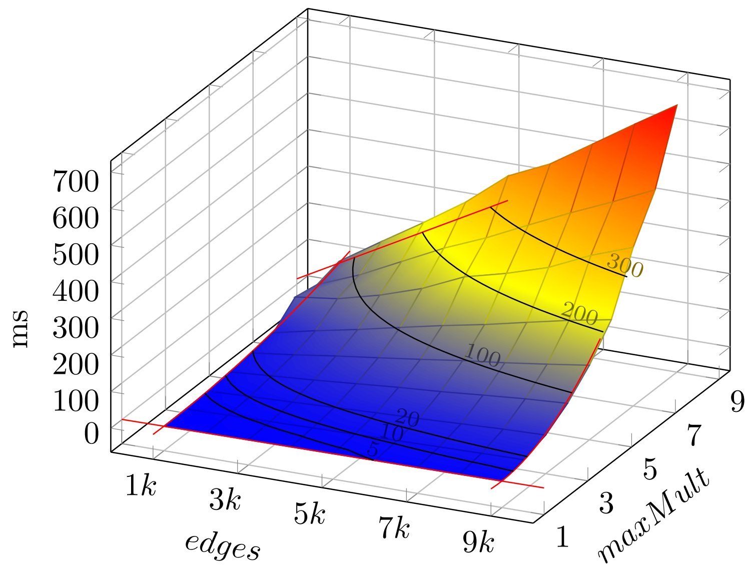


Figure 6.19: Behavior of the algorithm for sides implemented using VS on NVIDIA SHIELD. Red lines are approximation using polynomial in Equation 6.10.

6.4 Conclusion and Future Work

This section presented fast and precise method for shadow rendering in web applications using WebGL API. Despite limitations of WebGL 1.0 (only vertex and fragment shader, no queries, etc.), the algorithm is able to compute silhouette shadow volumes on GPU. The algorithm is robust and works with any arbitrary triangle meshes, producing precise per-pixel (or per-sample) correct shadows.

The algorithm was tested on number of well-known scenes and it proved to be faster than omnidirectional shadow mapping with resolution 8k on GeForce GTX 980 and Radeon R9 290X while not suffering from any artifacts seen on shadow mapping. On Radeon R9 290X, it performs even better and outperforms 4k omnidirectional shadow mapping. Among the algorithms for precise shadows, this algorithm is one of the fastest for WebGL.

Contribution of this algorithm is the silhouette edge evaluation for arbitrary model implemented in vertex shader, shadow data space optimizations to limit large amounts of shadow-data transmitted over network and robust rendering of shadow volume caps by moving them to infinity.

The future work might reduce the bandwidth requirements by not transmitting the topology data for shadow rendering but generating them on the client side from the scene data. The Javascript performance might limit its usefulness. A tempting idea is to use compute shaders for the algorithm once they are available as WebGL extension or included in future WebGL releases.

Chapter 7

Silhouette Extraction for Shadow Volumes Using Potentially Visible Sets

The full content of this section can be found in paper [53]. This section presents a novel approach for accelerated silhouette computation based on potentially visible sets stored in the octree acceleration structure. The scene space, where the light source can appear, is subdivided into voxels. The octree voxels contain two precomputed sets of edges that potentially or always belong to the silhouette, see the Figure 7.1. The novel method of octree compression for reduction of the memory footprint of the resulting acceleration structure is also presented. Using the novel technique, the algorithm is able to considerably decrease the computational complexity of finding the silhouette. The performance is also less sensitive to the number of edges.

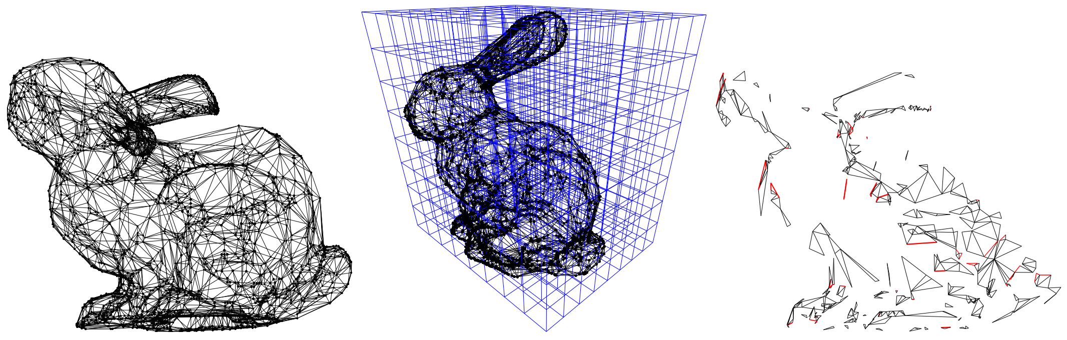


Figure 7.1: **left:** Wireframe representation of a given model. **middle:** Voxelized space around the model. One voxel on the lowest octree level is selected, based on the light position, and all potentially-silhouette (need to be tested) and silhouette edges (guaranteed to be silhouette) can be collected by ascending the octree hierarchy. **right:** Red coloured edges are those that are a part of the silhouette after testing the set of potentially silhouette edges (all red and black ones). Only a small subset of model edges needs to be tested, which considerably reduces the computational complexity.

7.1 Introduction

Solving surface visibility from a light source (or another point in space in general) is a very fundamental problem of computer graphics. Determining, whether a point on a surface is lit from a point light source has been subject of research for decades, as documented by Woo and Poulin [108]. Over the course of history, two major techniques were developed to address this problem in the field of rasterization – Shadow Maps and Shadow Volumes. Although the majority of the derived methods of the above mentioned techniques are based on Shadow Maps, Shadow Volumes still provide an important option for scenarios requiring sample-precise shadows, which can be problematic when Shadow Maps are involved due to their discrete nature and limited resolution.

Crucial part of the algorithm of Shadow Volumes is silhouette extraction, i.e. finding the subset of edges that have both visible and non visible triangles connected to them from the light's perspective. Such edges are subsequently extruded as the shadow volume side and rendered into the stencil buffer on the GPU. Usually, all the edges are tested during the rendering of a single frame to determine whether they get extruded or not.

The new method is focused on improving the silhouette extraction performance of the Shadow Volumes by reducing the number of edges that need to be tested during the Shadow Volumes rendering of an arbitrary triangle soup. Edge indices are stored in an octree-like structure created by voxelizing selected scene space. This octree is later traversed by the GPU to acquire two sets of edges – one set that requires further testing (potentially silhouette edges, **PE**), the second set is known to be silhouette (silhouette edges, **SE**) and edges inside the set are extruded immediately.

The remainder of the chapter is organized as follows. Section 7.2 outlines the previous work, focusing on Shadow Volumes and silhouette extraction. The following Section 7.3 introduces the reader to the details of novel algorithm, being divided into octree construction, compression and traversal. Section 7.4 discusses implementation details and problems. Performance and practical analysis as well as limitations are described in Section 7.5. Finally, conclusions are drawn in Section 7.6.

7.2 Related Work

Shadow volumes and silhouettes are described in previous sections.

Silhouette extraction methods can be divided into 3 categories – image space, object space, and hybrid (computing in object space, displaying in image space), as categorized by Isenberg et al. [45]. The majority of these methods were used to provide object contours for non-photorealistic rendering, but some of the object-based methods are interesting from the perspective of Shadow Volumes. Johnson and Cohen [47] use a hierarchy of normal cones to determine edge visibility. Olson and Zhang [79] propose octree as an acceleration structure to store Hough transform of a 3D mesh. Gooch [35] and Benichou and Elber [7] designed a preprocessing method based on projecting face normals onto a Gaussian sphere. However, all of these methods are limited to 2-manifold objects.

With the introduction of programmable graphics pipeline, research started to focus more on the ad-hoc algorithms, mostly due to the fact that the new pipeline provided mechanisms to determine the silhouette during the rendering process with zero or minimal preprocessing. Silhouettes can be computed in almost any programmable shader stage, specifically vertex [13] [72], geometry [97], [85], requiring practically no preprocessing, tessellation [75] or compute (OpenCL) [85] shaders.

Gerhards et al. [33] use BSP trees constructed from per-triangle frusta. Fragments are then tested against this structure, whether they are lit or shadowed. This method, however, needs to rebuild the data structure each time the light source or geometry is changed by even a rigid transformation.

The proposed method does not require rebuilding (unless the light source moves outside the targeted space) and it is also invariant to affine transformation – the correct voxel in the octree is selected by applying an inverse transformation of the object to the light source and then traversing from the corresponding voxel.

7.3 Algorithm

The algorithm is based on the concept of the *potentially visible set* (PVS) introduced by Airey et al. [2]. It precomputes the results of brute force silhouette extraction for a discrete set of world-space voxels. The brute force extraction process therefore does not need to be executed on all scene edges but only on a small subset that cannot be precomputed, see Figure 7.1. This section describes the construction of a compression structure for storing the PVS in an effective manner. It also describes the modified extraction process.

The algorithm can be broken down into two major stages: **construction** and **traversal**.

7.3.1 Silhouette Extraction

The brute force silhouette extraction process is described in previous sections and it is based on Kim's algorithm [52] with concept of reference edge [72].

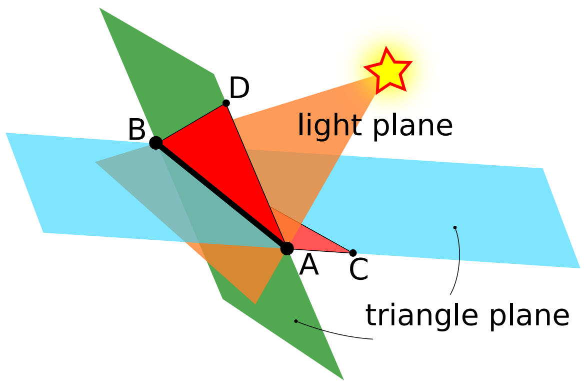


Figure 7.2: Edge AB is not a silhouette edge because triangles ABC and ABD do not lie on the same side of the light plane. Two triangles partition the world space into four subspaces.

A model is composed of vertices that are connected by edges/triangles. In general, from 1 to N triangles can be connected to a single edge. Kim et al. [52] proposed a technique that computes the difference in the number of triangles on the left and the right side of the light plane (Figure 7.2) called edge multiplicity $m \in [-N, N]$. Without loss of generality, edges with more then 2 connected triangles can be transformed into several simpler edges by splitting and duplicating. If an edge is connected to only one triangle, it is considered a silhouette edge in every case. The new method works with edges having maximum of 2 adjacent triangles connected to them.

7.3.2 Octree Construction

The voxelization step voxelizes space around a model. The size of the space is given by scaled bounding box (AABB) with user-specified scaling factor, Figure 7.3. The scaling factor depends on the user's needs and on the type of the scene (closed-space scenes will do even with factor 1, open scenes or simple models require larger factors, around 5–10).

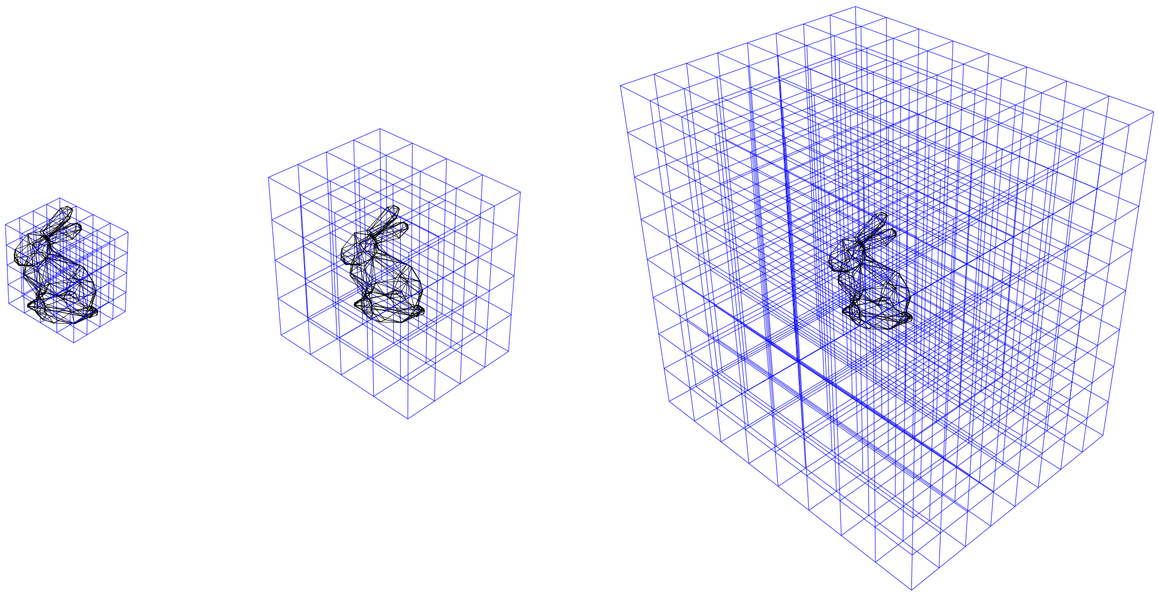


Figure 7.3: The algorithm supports custom scales of the scene bounding box. If a larger scale is selected, the light can be moved farther from the model. The image shows three different scales with the same voxelization level (in this case 2 levels of octree, $4 \times 4 \times 4$ voxels).

This scaled bounding volume circumscribes all the possible light positions. The user can then choose the maximal level of the octree hierarchy, see Figure 7.4. AABB scaling and maximum octree depth define the octree granularity and voxel size on the deepest level.

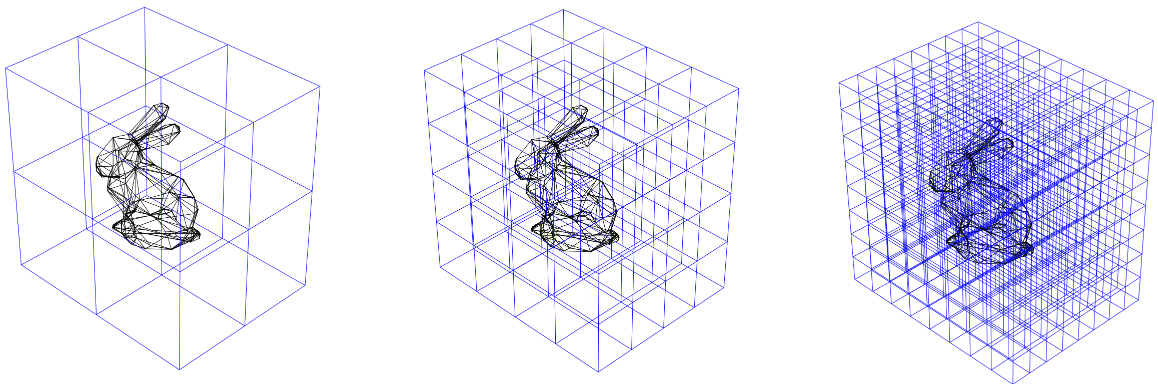


Figure 7.4: The algorithm supports a custom level of voxelization. The image shows three levels (1,2,3) of depth of the octree for the same scale of the scene bounding box.

The depth level of 3–5 is suitable in most scenarios. Larger scales tend to consume too much memory (as described in Section 7.5.1, each octree level increases the amount of memory by a factor of 4). The next step is to find two sets (**SE** and **PE**) for every voxel in the lowest level of the octree. The algorithm tests each edge against all voxels on the lowest level of the octree, as seen in Figure 7.5. If any plane constructed from the triangles adjacent to edge E intersects the voxel, E is considered a **PE**. If none of the triangle planes intersects the voxel and multiplicity of E is non-zero, it is stored among **SE** (set of silhouette edges). The multiplicity can be computed against any point inside the voxel because the whole voxel lies within one of the four subspaces, as demonstrated in Figure 7.2.

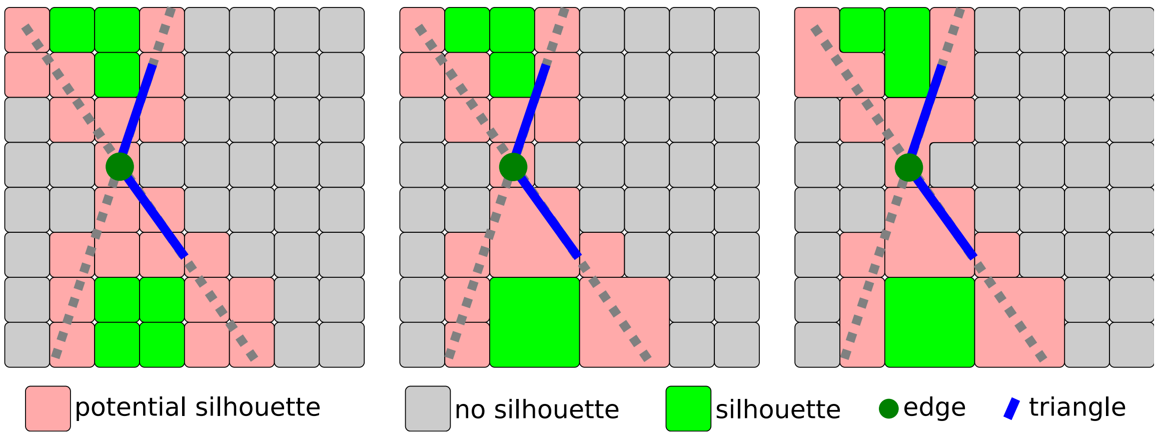


Figure 7.5: Overview of the proposed approach in 2D. The image shows voxels for one edge. The left side of the image shows the first step of the voxel building algorithm. Voxels are classified into 3 categories – no silhouette, silhouette and potentially silhouette. The next step is to propagate this 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.

The next step is to propagate **PE** and **SE** into higher levels of the octree. An edge can be propagated to the parent node if it is contained in all of its children. Both types of edges are propagated. The propagation process already significantly reduces the memory footprint. This propagation scheme is referred as “basic compression”.

The last optional step in octree construction is advanced compression. It extends the propagation step by allowing edges to be moved to their parent node even if not all of them are contained in all of its children. These sets of edges are marked with bitmasks corresponding to voxel shapes, see Figure 7.6. Every subvoxel in these voxel shapes contains the same set of edges, see Figure 7.7. This extended propagation is referred as “8-bit compression” as the bit mask size is 8-bit integer. Edges can also be propagated into grandparents (from 64 sibling voxels) which can further improve the compression ratio. That compression scheme is referred as “64-bit compression”. Octree node data are shown in Figure 7.6.

7.3.3 Traversal

The traversal part of the algorithm has to copy **SE** and **PE** subsets from the octree into two continuous buffers. The light position determines which subsets of edges have to be copied to the linear buffers, see Figure 7.8.

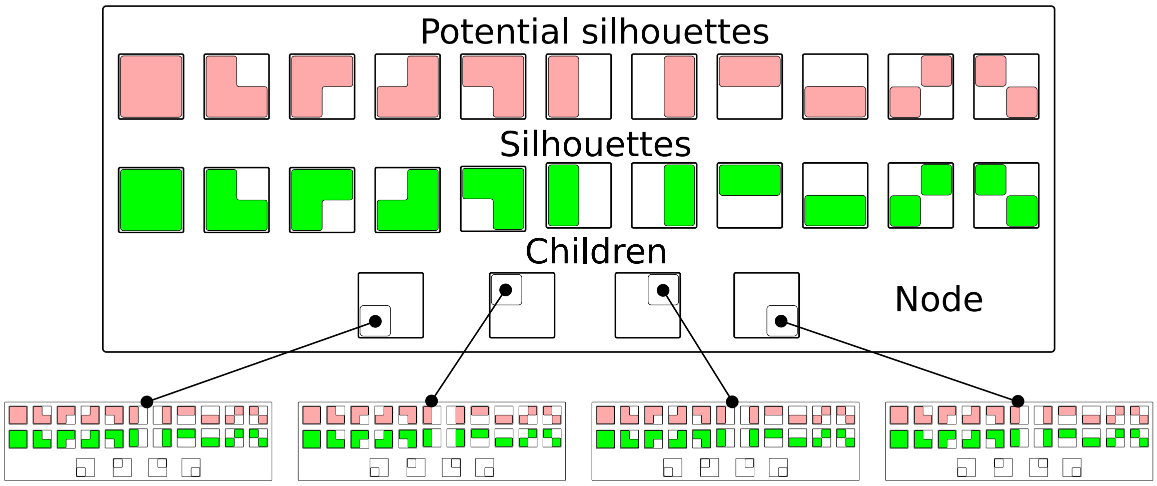


Figure 7.6: Node data in 2D space for the 8-bit compression. One node contains sets of silhouette and potentially silhouette edges, each addressed by its bitmask value. If a set shape does not intersect the triangle planes of an edge, the edge is stored into the set. The largest set shape is chosen if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

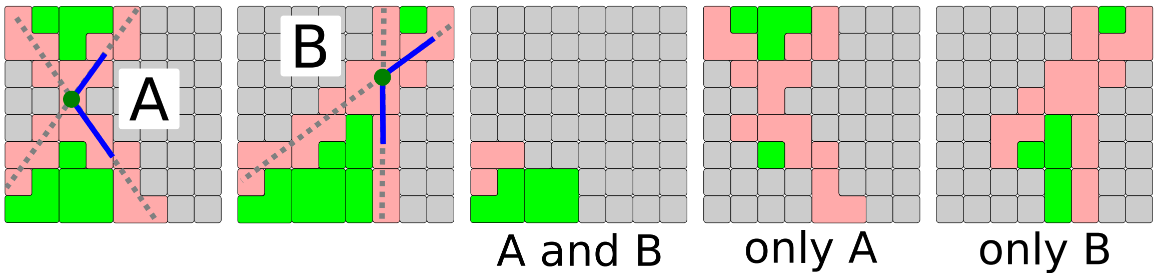


Figure 7.7: The first two images show two edges – A and B. Each edge partitions all voxels into voxel shapes for silhouette case and of potential silhouette case. If some voxel shapes are the same for both edges, the edge subsets of those voxel shapes contain both edges (middle image). Otherwise, voxel shapes contain only one edge.

The **PE** linear buffer is in the final part of the brute force silhouette extraction process. However, the **PE** set is very small compared to the set of all edges which leads to performance improvement.

7.4 Implementation

The whole process was implemented both on CPU and GPU (OpenGL). The construction process relies on two compute shaders: the first one is used to load the data to the octree, the second one for propagation to upper levels.

For larger models, adding edges to the lowest level of the octree is running in batches. The batch size is limited by the GPU memory size.

Usually, not all edges get stored neither in **PE** nor in **SE** buffer of a voxel, thus their size can be limited to a percentage of total edge count (the factor of 0.8 works for most

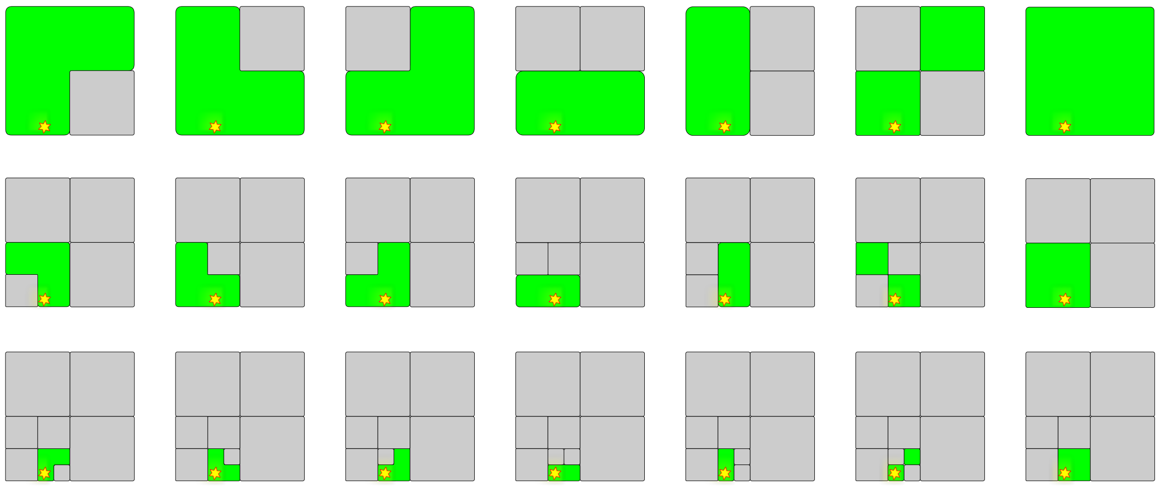


Figure 7.8: 2D illustration of all edge subsets that contain silhouette edges for a given light position. The hierarchy level is 3. The union of all subsets forms the set of all precomputed silhouette edges. Similar subsets are selected for all potentially silhouette edges. Note that some subsets could be empty. A single edge is contained only in one of the subsets (the largest possible).

cases and can be seen for example in Table 7.1). This increases the batch size and speeds up the building process.

Data are then copied back to the system memory. Before the edge propagation, which is also implemented in a compute shader, the algorithm needs to sort the edges, which is carried out on the CPU in parallel.

At first, the compression was implemented as a multi-core post-processing of the octree, but such implementation, although parallelized, was $20 - 170\times$ slower than 8-bit compression, based on particular scene. For the 8-bit scenario, the advanced compression was moved to the compute shader which tests the edges against octree voxels, because it is in this very step that the bitmask is already known. However, porting the 64-bit compression scheme to GPU seemed problematic as the potential number of sub-voxels is 2^{64}, which would lead to excessive memory footprint, thus it was performed as CPU post-processing. Due to implementation reasons, compressed nodes using bitmasks other than all-bits-set are located only in `max_depth-1` for 8-bit compression or `max_depth-2` for 64-bit.

During traversal, the algorithm first determines the (X, Y, Z) voxel coordinates within the octree from the light position (flooring the floating point coordinates), which are then converted to linear voxel index. If a light source lies on the boundary between two or more voxels, only one voxel is selected according to Equation 7.1 where l is the light position and A and B are minimal and maximal corners of the voxel:

$$l_x \in [A_x, B_x) \land l_y \in [A_y, B_y) \land l_z \in [A_z, B_z). \tag{7.1}$$

The traversal process depends on the compression level. For non-compressed and 8-bit compression scenarios, the algorithm traverses the octree and computes an exclusive scan of sizes of all necessary sub-buffers, which serves as the input to the second stage that performs the actual data copy from selected subsets to two linear buffers, one for **PE**, the other for **SE**. Traversal for 64-bit compression splits the pre-processing and prefix scan into

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

Table 7.1: Build test of Šibenik 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 build with basic compression scheme. Columns tagged "c8" and "c64" show build times and sizes when using 8-bit or 64-bit advanced compression schemes. 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 next to last column 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.

two steps as the amount of sub-voxels increases to several thousands. Splitting the stage also helps reducing the count of the global memory reads.

7.5 Results

Evaluation took place on the following test setup: Intel Core i5 6500, 16GB of DDR4, nVidia GeForce RTX 2080Ti (11GB of GDDR6, driver version 419.17), Windows 10 Pro x64. The test application was built using Visual Studio 2015 x64.

7.5.1 Build and Compression Tests

Comprehensive tests were conducted on the Šibenik, Conference, and Sponza scenes. The aim was to evaluate the building time and the size of the octree structure, based on the octree deepest level, size of the voxelization area, and compression type.

Table 7.1, Table 7.2, and Table 7.3 show the performance evaluation of the building process under various octree settings, with respect to the selected light source position inside the scene's bounding box. 3 types of builds are compared: with basic compression only (first two coloured columns), 8-bit GPU compression (c8) and 64-bit CPU compression (c64).

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

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

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

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

One of the first noticeable things is that 8-bit compression on GPU performs actually faster than the non-compressed version of the algorithm. 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 compression is performed as a post processing step and it happens on the CPU, thus being very slow, even though the algorithm was written using OpenMP. The 64-bit compression was also tested 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.

It can be seen that the amount of memory required to store the octree increases with a factor of 4 with each octree level, but also the average amount of extracted SE increases by around 10 % and the average number of PE that needs to be tested is almost halved with increasing octree depth, for each of the tested scenes.

This test, however, shows the biggest weakness of the algorithm. The memory consumption is, for a particular model, strongly dependent on the algorithm's settings. For practical use, the 8-bit compression scheme seems to be the best choice, in terms of both the building speed and the size of the resulting octree structure.

The memory consuption can be approximated using Equation 7.2, 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:

$$S(e, d, c) = e \cdot 8^d \cdot V_d \cdot c \tag{7.2}$$

Based on the results in Table 7.1, Table 7.2, and Table 7.3, the average compression ratios can be estimated to 0.32 for 8-bit compression scheme and 0.11 for 64-bit scheme by dividing the compressed octree size with non-compressed octree. Values V_d define the approximate size of a single voxel per 1 million edges. These values were calculated as $V_d(d, e) = S_m/8^d/e$, 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 7.2 is 6 %.

7.5.2 Silhouette Extraction Tests

The new method (with 8-bit compression) was compared to a brute-force compute shader implementation of silhouette extraction, based on an OpenCL implementation and multiplicity theorem described in [85]. Both methods output edge indices as their result. This test contained 26 models in total, mixing popular models (Sponza, Šibenik, Buddha, Conference, Gallery, Bunny[1]) with two types of synthetic scenes – the first type were scenes consisting of uniform grid of increasing amount of spheres, having 33 750 to 1 574 640 edges. The second type consisted of randomly positioned spheres differing in numbers, having 124 200 to 933 120 edges. The algorithm was evaluated with two levels of octree depth (3 and 5) posing as best and worst case, and scene scales 1, 2, 4, 8, and 16.

In a single test run, the light source moved through the octree volume in a $10 \times 10 \times 10$ grid, both for novel method and the brute-force approach. From each light position, the traversal time was evaluated as average of 5 runs. In total, 75 000 measurements were made in each scene: 50 000 for novel approach and 25 000 for brute force method.

The result can be seen in Figure 7.9. The novel accelerated approach has reduced the sensitivity of the algorithm to the number of edges, compared to the brute force approach.

[1]freely available at https://casual-effects.com/data/

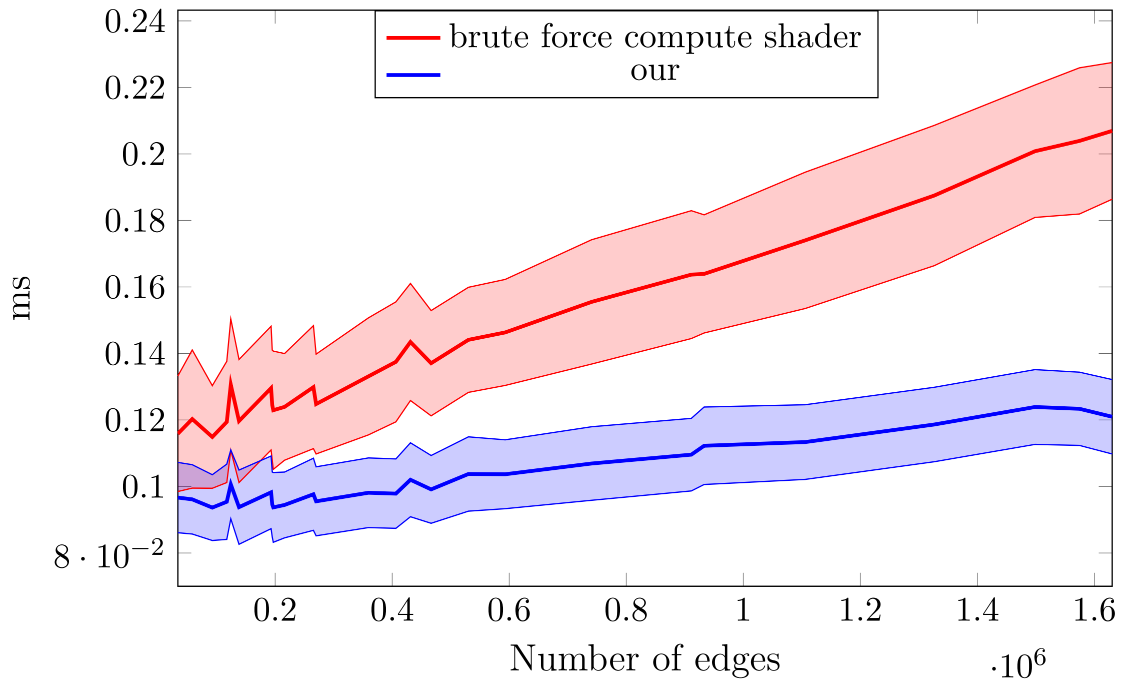


Figure 7.9: Average extraction times for compilation of 26 scenes (sorted by the number of edges). Red line represents bruteforce compute shader method, blue line represents our new proposed method. The area around the lines represents (+-) mean absolute deviation.

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.

The new method performs always better on models having more than 200 000 edges and it is also more stable. Its average absolute variance is almost half, compared to the brute force method.

The performance difference when using different compression ratios was also evaluated. The light source moved around in Sponza scene in the same way as described in the previous test. The results can be seen in Table 7.4. The complexity of the 64-bit compression traversal outweighs its benefits in the form of lower size, thus the 8-bit compression scheme seems to be the best choice.

The extraction time for brute force approach can be estimated as $t_b = E \cdot K$ where E is the number of edges and K is extraction complexity. Novel method yields $t_t = P \cdot E \cdot K + T$ where T is traversal cost and P is the ratio of potential edges, which can be seen in 2nd last column of Table 7.1, Table 7.3 and Table 7.2. Based on the build test results, the P can be estimated to be 0.2, 0.1 and 0.05 for octree levels 3, 4 and 5. According to the measurements, T was 0.075 ms in average and was not dependent on the number of edges.

7.6 Conclusion

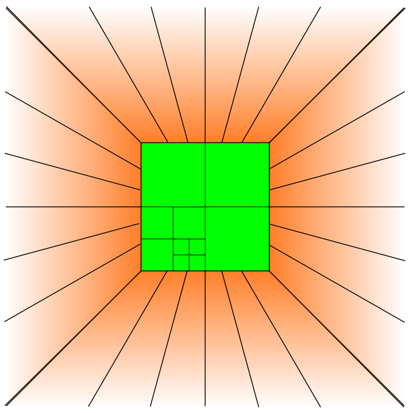


Figure 7.10: One of the possible extensions to the algorithm. Instead of using one hierarchical structure, the algorithm would use two – one for the close vicinity of the model and one for all the other space around. The green part shows the hierarchical structure as presented, the orange parts is the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

This section presents a novel approach to accelerated silhouette extraction by storing pre-computed PVS in an octree, as well as novel octree compression schemes.

The majority of the building process was implemented on GPU using OpenGL and compute shaders. The building process is reasonably fast when using 8-bit compression scheme, as it processes less data then the basic compression scheme. The resulting octree can be stored in a file to avoid repetitive builds in subsequent runs. Experimentally, the octree size can be reduced using 64-bit compression to around 12 % of basic compression scheme, but the compression itself was used as a post-processing step on CPU, thus not performing as fast as the GPU implementation. In terms of performance, 64-bit compression also lagged behind due to having a more complicated traversal. The 8-bit compression scheme provides the best results in terms of octree size and traversal speed.

Compared to the brute-force approach, the new method is less sensitive to the number of edges. It is also more stable, in terms of maximal absolute deviation. The biggest drawback of the method is its memory consumption and also spatial limitation due to the nature of voxelization. The method also would not work well on scenes with dynamic geometry (e.g. morphing).

This method could be further improved by storing triangle indices instead of edges. In theory, it could reduce memory footprint of the method even more. The whole structure does not need to reside on the GPU but can be streamed as needed. Future research could also evaluate usage of homogeneous coordinates, which may create hierarchy with unlimited spatial span, see Figure 7.10.

Chapter 8

Shadow Mapping, Many Lights

Previous sections describe sample precise shadow methods focused main on robustness and performance. This section briefly describes shadow mapping methods with focus on quality improvements and algorithms for scenes with many light sources.

8.1 Shadow Mapping

Basic shadow mapping principle is described in the previous section. Shadow map based methods are fast real-time algorithms affected by variety of visual artifacts. A lot of research is aimed at visual artifacts mitigation. Most of the problems can be traced to difference in samplings from camera and light point of view, Figure 8.1. Jagged edges and self shadowing (Figure 8.2) are one of the most common problems. Self shadowing is usually mitigated using a small bias introduced to depth comparison. This bias, however, can create other problem - missing shadows. Jagged edges are mitigated by large spectrum of methods. A lot of methods try to modify shadow sampling to improve coverage of view-samples. Such methods are based on fitting, partitioning, warping or other techniques. Fitting tries to find minimal required shadow frusta. Partitioning splits space into multiple areas each with separate shadow map. Warping reparameterizes sampling using scene deformation techniques. Most methods can be combined together, some are viable only under certain circumstances.

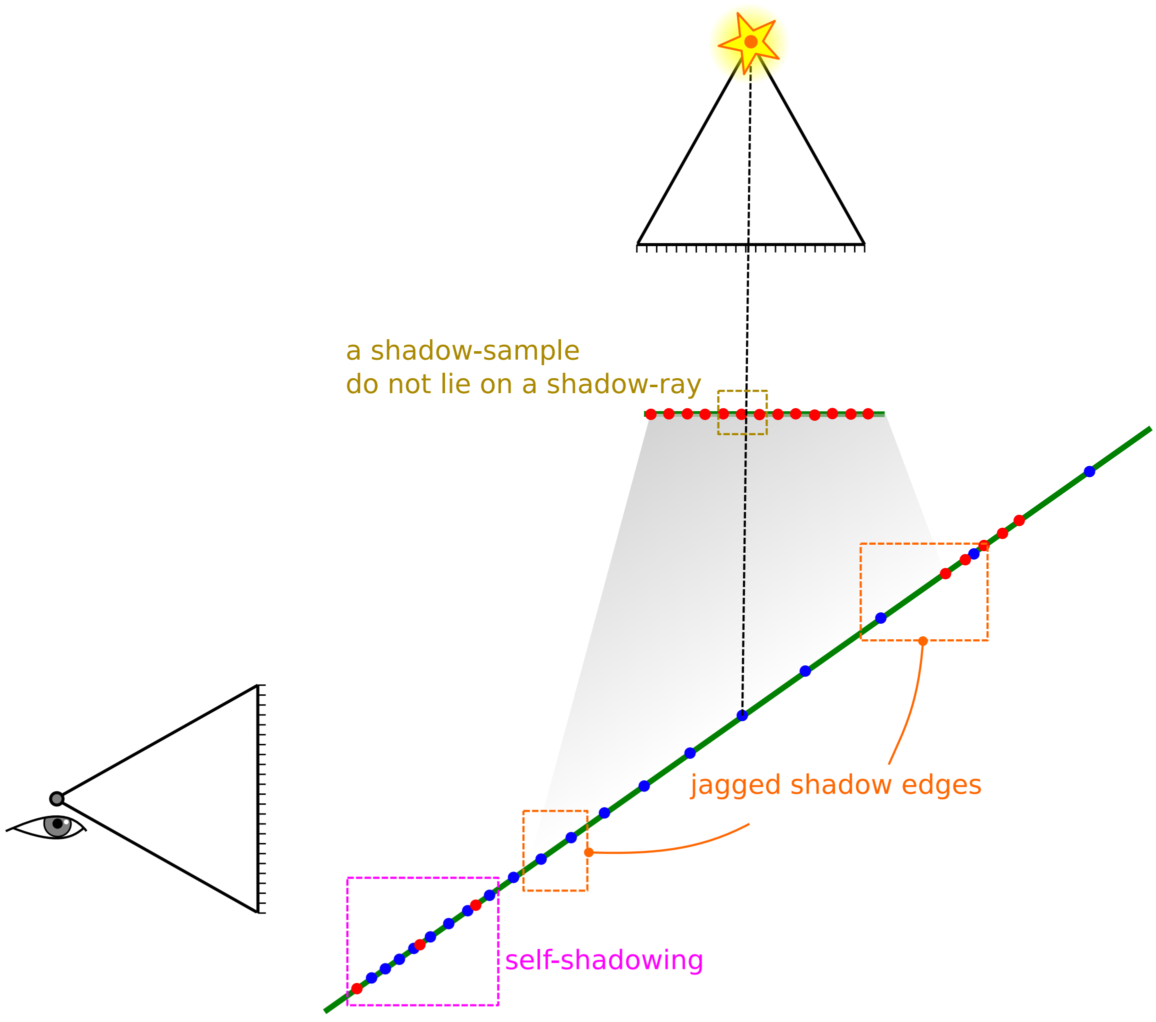


Figure 8.1: The image shows difference in samplings from camera and light point of view. The distribution of view-samples (blue) is different than the distribution of shadow-samples (red). In the ideal situation, every shadow-sample lies on a ray from the light to the view-sample and every view-sample has exactly one corresponding shadow-sample. Some possible visual artifacts can be seen in the Figure 8.2

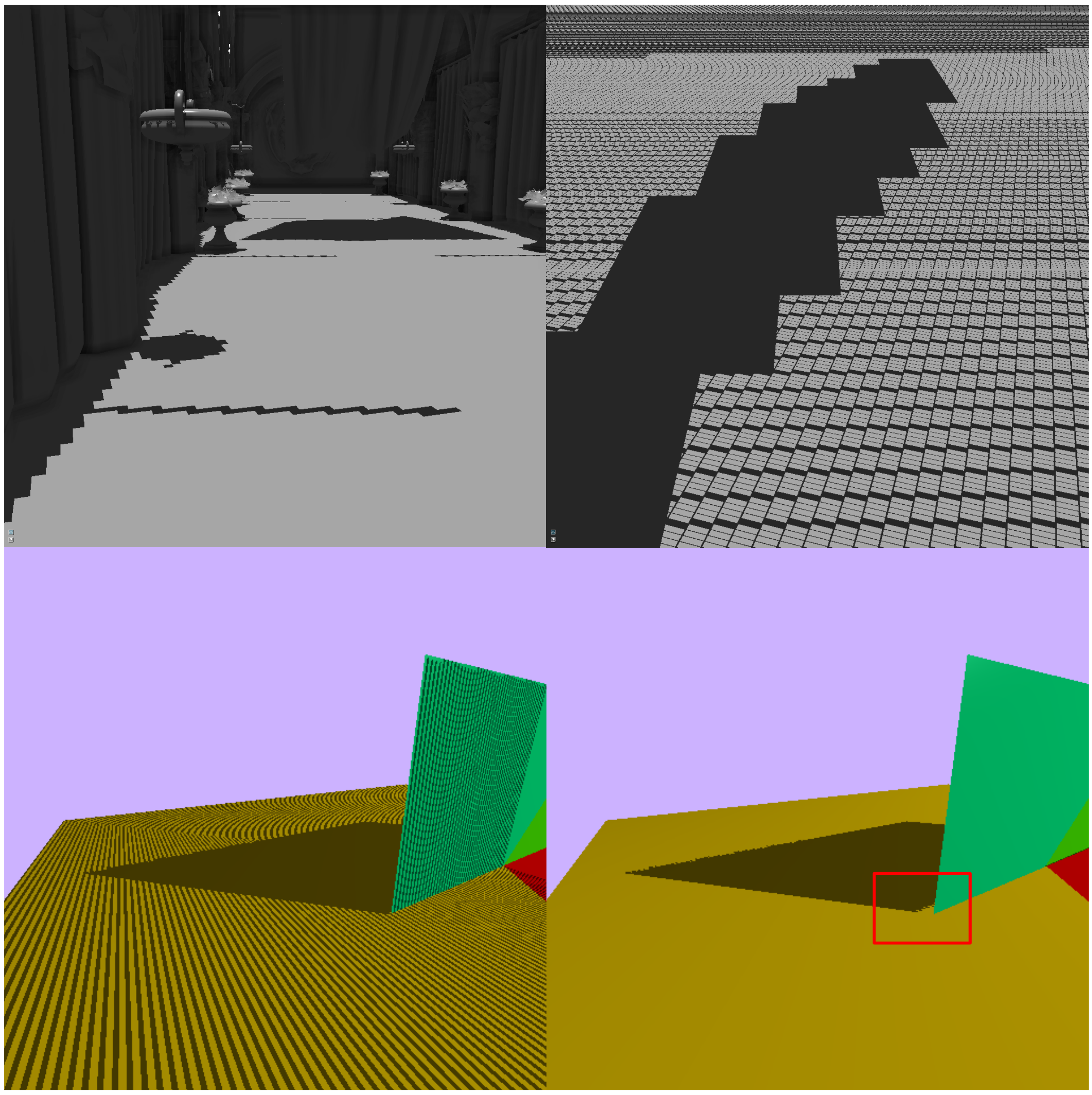


Figure 8.2: The image shows common visual artifacts that affect shadow mapping algorithm. The top left part shows jagged shadow edges caused by a low shadow sampling rate. The top right part shows self shadowing caused by regular sampling and imprecise shadow-sample placement. The bottom left part shows the self shadowing artifact (also known as shadow acne) and the right one shows one possible mitigation. The mitigation biases the depth comparison removing shadow acne, but it also introduces new artifact. The shadow is missing near the shadow caster.

8.1.1 Fitting

The fitting tries to find the smallest shadow frusta. When a scene is rendered from a light point of view, it is desirable to render only the necessary parts of the scene. Brabec [11] introduced fitting and focusing techniques. In order to find minimal shadow frusta, it is necessary to construct convex hull containing useful geometry, Figure 8.3. The process of finding minimal shadow frusta can be improved using visibility information. Bittner et al. [10] proposed method for caster culling. Lauritzen et al. [58] analyzed distribution of samples in order to further reduce shadow frusta size. The fitting mitigates jagged shadow edges but also introduces a new problem - temporal aliasing. When the camera, the light source or objects in the scene move, the fitting produces different shadow frusta creating flickering artifacts. Valient [102] and Zhang et al. [29] tried to address this issue. More information about fitting can be found in the book Real-Time Shadows [25].

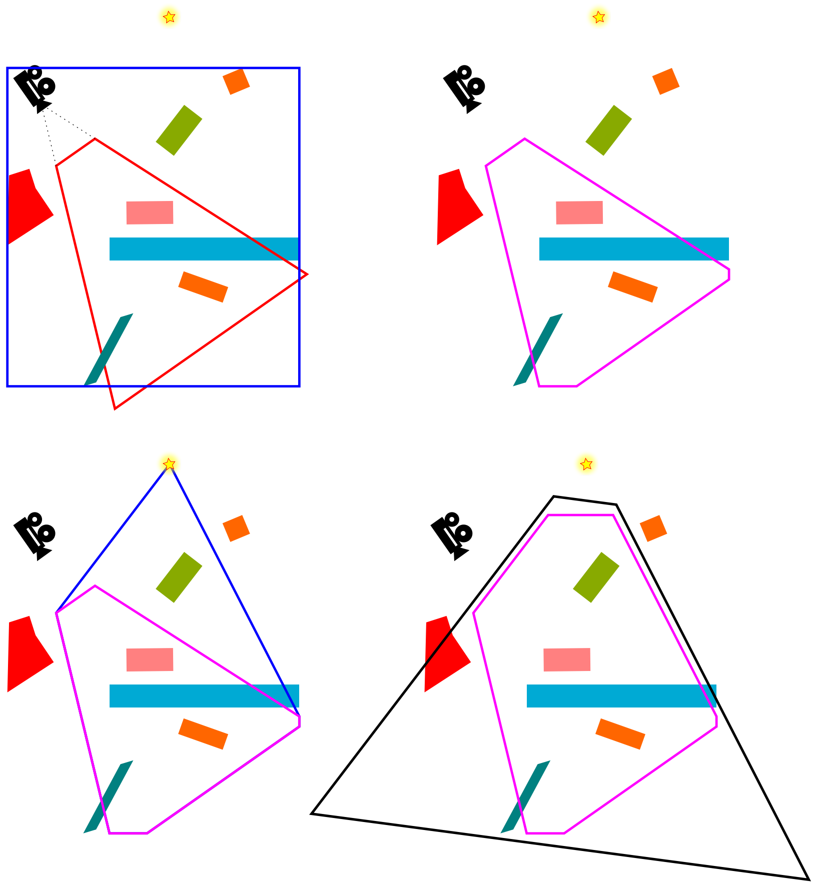


Figure 8.3: The image shows process of finding minimal shadow frusta. The top left image shows a scene enclosed in a bounding box (blue), a light source, and a camera with view frustum (red). The top right image shows intersection of the view frusta and the scene bounding box (magenta). The intersection forms a volume containing all possible shadow receivers. The volume is extended with the light source in the left bottom image, Wimmer 2006 [71]. The extension is then clipped with the scene bounding box creating the convex hull. In the final step, the convex hull is tightly enclosed within shadow frusta. The process can be improved with view-samples and visible surfaces examination, Figure 8.4.

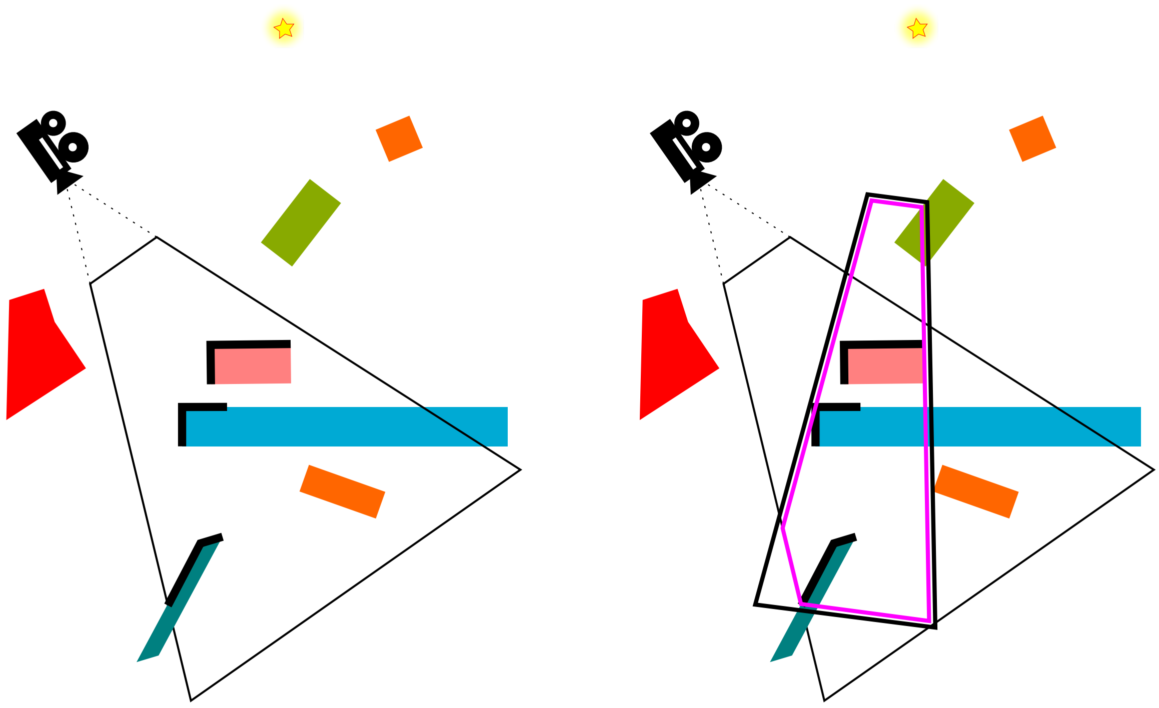


Figure 8.4: The image shows improved process of finding minimal shadow frusta. The process involves examination of view-samples and visible surfaces (black line segments). It is more computationally expensive but provides tighter shadow frusta with better sampling.

8.1.2 Partitioning

While fitting finds more optimal shadow frusta, it cannot solve all the problems. View-samples can easily be distributed in such a way that the minimal shadow frusta size is not reduced at all. Other approaches are trying to solve this issue by space partitioning and multiple shadow maps, see Figure 8.5

Z-Partitioning is one of the most common global partitioning schemes. It is named *cascaded shadow maps* and it was proposed by Engel [27]. It was also proposed by other authors: *z-partitioning* by Lloyd [63] or *parallel split shadow maps* [110]. The positions of splits are important. Some authors suggested logarithmic splitting or weighted average between the logarithmic and linear approaches. Others proposed methods that analyze view-sample distribution.

Researchers over the years introduced many partitioning schemes and in detail description is out of scope of this thesis. More information about partitioning can be found in the book Real-Time Shadows [25] in section Global Partitioning.

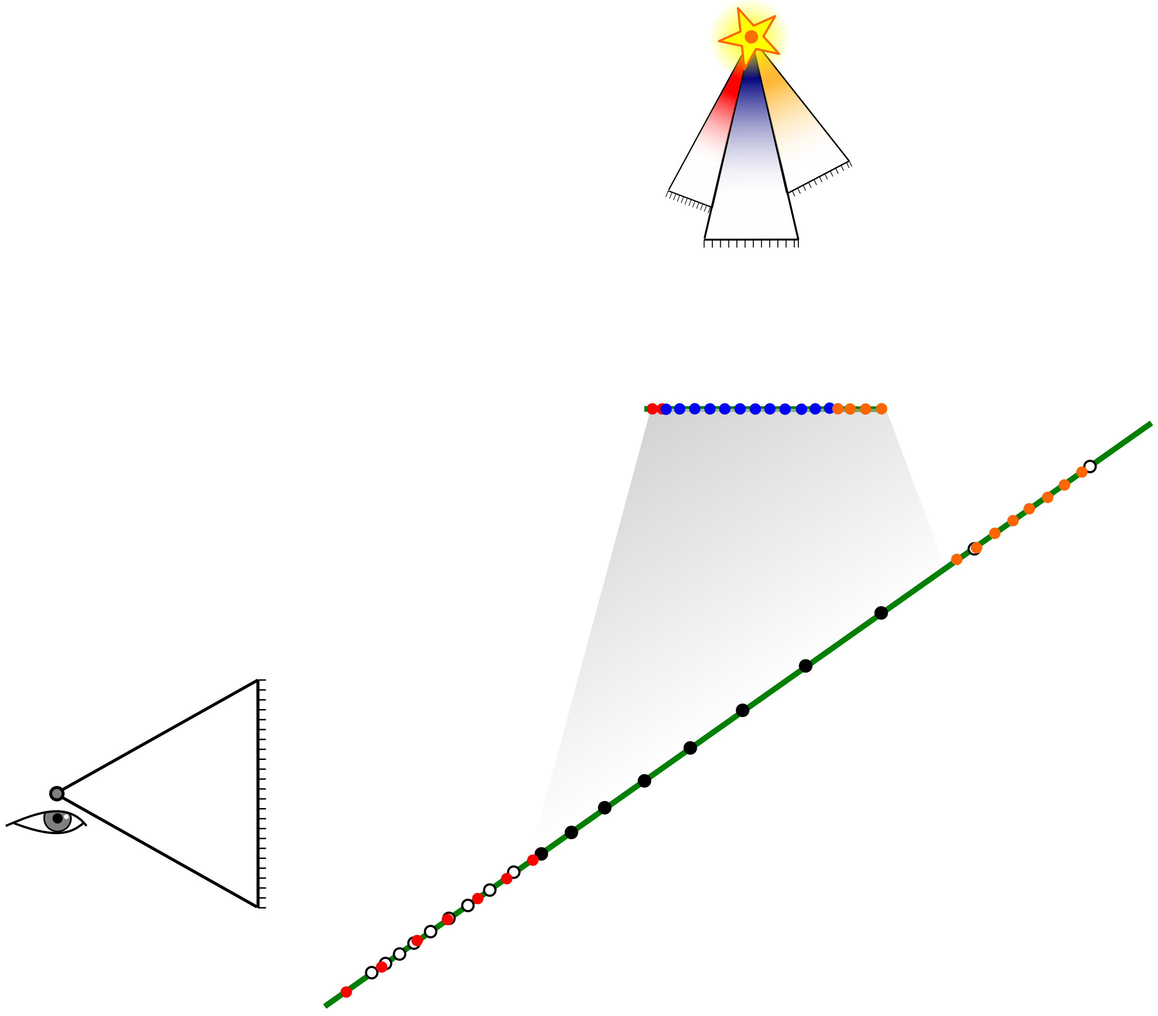


Figure 8.5: The image shows partitioning process for sample distribution improvement. In this example, three shadow maps (red, blue, orange) are used, each focusing on a different part of the scene. The space is partitioned along camera's z-axis.

8.1.3 Warping

While partitioning generally creates multiple shadow maps, warping can be used with a single shadow map. Warping tries to modify the distribution of shadow-samples that are generated in regular grid during shadow map rasterization. Today's hardware does not support irregular rasterization. Warping algorithms warp and deform scene geometry in order to achieve irregular sampling, Figure 8.6.

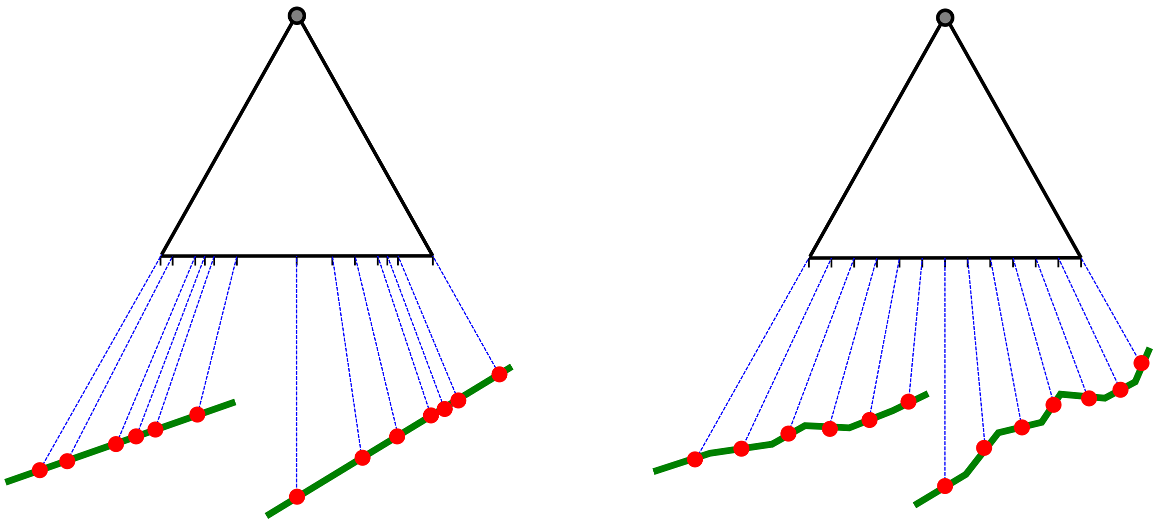


Figure 8.6: The left side shows irregular sampling that is unsupported by hardware rasterization. The right side shows regular grid based sampling. The scene is warped and deformed in such a way that samples are distributed in the desired irregular pattern. If the scene contains large triangles, warping process cannot correctly deform the space. Green line segments are tessellated in order to allow better sample distribution.

There are global warping schemes that apply single warping function to the whole scene. Stamminger [96] introduced concept of perspective shadow maps which uses perspective projection as the warping function. Perspective shadow maps were further extended by other authors. Wimmer proposed light space perspective shadow maps [107]. Martin et al. proposed trapezoidal shadow maps [66]. Lloyd proposed logarithmic perspective shadow maps [62]. Global warping schemes have smaller degree of freedom and do not drastically deform scenes.

Local warping schemes, on the other hand, can significantly deform scenes. The warping process is localized, which means that some parts of a scene are warped more than others. The local warping function adapts to the scene, light, camera and samples properties. Rosen proposed rectilinear texture warping [88]. The method uses two 1D warping functions that are derived using so-called importance map. The importance map highlights parts of the scene that require higher sampling and the warping functions enlarge them. The importance map is constructed using view-samples. Jia proposed distorted shadow mapping [46] which tries to enlarge shadow borders. Milet et al. proposed non-orthogonal texture warping [73] extending Rosen's method. Instead of using two 1D warping functions, they used two sets of 1D warping functions giving the warping process more degree of freedom.

8.1.4 Other Techniques

Different approaches have also been proposed. Some researchers combined shadow mapping and shadow volumes, others used information from previous frames or tried to filter shadow maps.

Temporal Reprojection

Daniel Scherzer proposed shadow maps with temporal reprojection [90]. The algorithm uses information from previous frames stored in history buffer. Each frame is rasterized with subpixel jitter in order to improve sampling overtime. The history buffer is updated every frame. The algorithm can be summed as follows:

1. Evaluate shadows for every view-ample using standard shadow mapping algorithm.
2. Transform history buffer to the current frame.
3. Update the history buffer using the current shadow map.
4. Shadow every view-sample using updated history buffer.

The method uses a term *confidence* which represents the closeness of a view-sample to the closest shadow-sample. In other words, it measures the correctness of shadow map test. If the view-sample is exactly at the same position as shadow-sample, the confidence is one. The history buffer is updated in order to increase *confidence*. The method converges into per sample correct shadows after sixty frames.

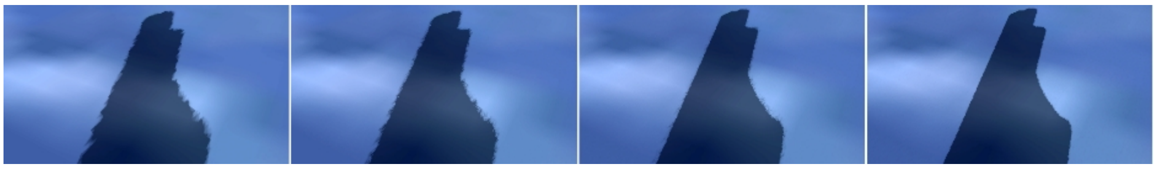


Figure 8.7: The shadow adaptation over time after one, twenty, forty and sixty frames. The image is taken from paper [90].

Hybrid Approaches

Shadow volume algorithms are sample correct shadowing techniques, but they are slower than shadow map based methods. On the other hand, shadow mapping algorithms are not sample correct but they are fast. Hybrid methods try to combine both approaches. Shadow mapping alias artefacts mainly occur on the edges of shadows. The hybrid method described by Eric Chan [15] uses shadow volumes on shadow borders and shadow mapping elsewhere. Shadow volumes algorithm is computed only on a small subset of pixels.

First step of the algorithm creates standard shadow map. Shadow mapping is applied only on view-samples whose shadow-samples are surrounded by eight samples with the same shadow result. This constraint prevents usage of shadow mapping on the shadow edges. View-samples that do not meet this constraint are marked. These marked pixel are shadowed using Shadow volume algorithm. Algorithm suffers from visual artifacts caused by imperfect rasterization, Figure 8.8.

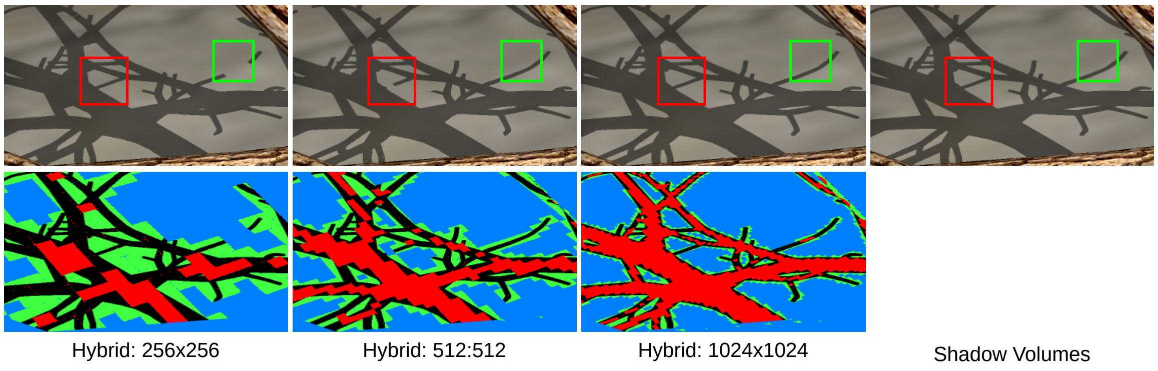


Figure 8.8: The top row shows result of hybrid method. The bottom row shows regions, where the standard shadow mapping is used (visualized in red) and where the shadow volume algorithm is used (visualized in green and black). Visual artifacts are highlighted using red and green squares. Artifacts are caused by imperfect shadow mapping rasterization. If a triangle is too small, it can be missed by rasterization. Also, a small slit between two triangles can create incorrect shadows. The image is taken and modified from Chan's paper [15].

8.2 Many Lights

Recent scenes in compute graphics contain huge amount of light sources. The computation of lighting becomes more and more expensive with the increasing number of light sources. The common lighting model in computer graphics application is Phong lighting model [86]. The final color is computed from three components: ambient, diffuse and specular colors. Computation of these components have to performed on every view-sample. If a scene contains L_n light sources and S_n view-samples, the Phong lighting equation has to be evaluated $L_n \cdot S_n$ times. Some methods are focused on reduction of S_n while others on L_n.

8.2.1 Deferred Shading

Deferred shading [40] reduces the number of view-samples S_n. This method rasterizes a scene into so-called geometric buffer or G-Buffer. The buffer usually contains three 2D arrays storing positions, normals and colors. After the G-Buffer is constructed, the lighting is computed using the stored information. This process ensures that the Phong lighting equation is done only on the closest view-samples. This can also be achieved using forward shading with pre z-pass.

8.2.2 Tiled Shading

Tiled shading was introduced in 2011 by Ola Olsson et al. in paper called Tiled Shading [80]. Their method reduces an average number of light sources per view-sample. The method assumes that if the light source is far away from a view-sample, its contribution will be negligible. The intensity of light decreases with the distance according to the inverse square law [61]. A Light source can be enclosed within a bounding sphere and it can only affect samples which lie inside. A list of light sources can be assembled for each view-sample, which would be inefficient. Instead, the algorithm constructs lists for screen tiles. The algorithm can be described as follows:

1. Screen samples are grouped together forming 2D grid of tiles in screen space.
2. The light list is assembled for each screen tile.
3. View-samples are illuminated using information stored in lists.

A Light source insertion can be seen on Figure 8.9. If a screen-space bounding quad of a light source intersects a bounding quad of a tile, the light is inserted into the tile's list. The intersection test can be extended by usage of bounding boxes. In order to extend bounding quads to bounding boxes, the minimal and maximal depth of the tile has to be found.

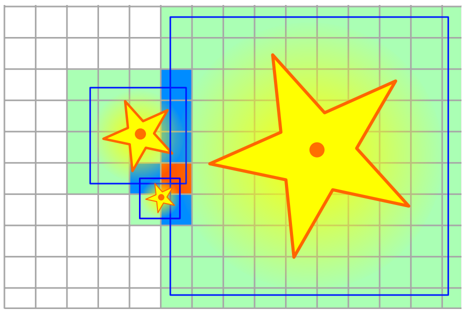


Figure 8.9: The image shows screen divided into 2D grid of tiles. Lights are represented as blue bounding quads, green tiles contain only one light source, blue tiles contain two light sources and orange tile contains three light sources

Tiled shading provides lighting acceleration, but it also increases precision. The light accumulation is done in register using full floating-point precision. It has one drawback. The total illumination of the scene is decreased. This is caused by reduction of radii of influence of light sources. Tiled shading was extended by Harada in paper A 2.5D Culling for Forward+ [39]. The extension provides better, depth-based light culling.

8.2.3 Clustered Shading

Tiled shading clusters screen samples based on their 2D position. Clustered shading described in paper Clustered Deferred and Forward Shading in 2012 by Ola Olsson [81] creates clusters differently.

The algorithm creates clusters containing samples that have similar 3D position. Normals of screen samples can be used for clustering as well. The clustering creates clusters with smaller number of screen samples. A light list of each cluster contains a smaller number of light sources. A light is inserted into light list only if its 3D bounding box intersects cluster and normal cone of the cluster points toward the light source. Clusters can be seen in Figure 8.10. Clustered shading speeds up lighting even more than Tiled Shading; however, the construction of clusters is more computationally demanding.

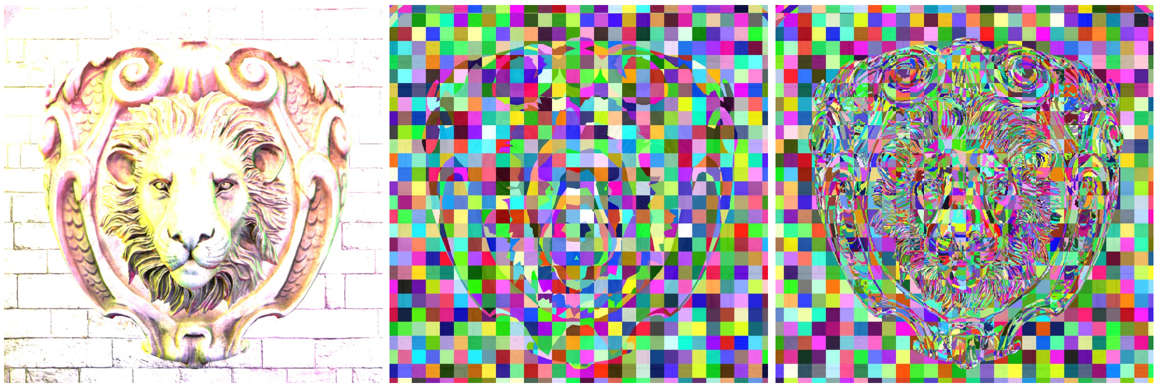


Figure 8.10: The left part of the image shows the original scene. The middle part shows clustering based on 3D positions of screen samples. Flat regions produce clusters that are similar to tiles in Tiled Shading. The right part shows clustering based on 3D positions and normals of screen samples. The image is taken from paper [81].

O'Donnel et al. introduced Tiled Light Trees [84]. The method adapts hierarchical acceleration structure according to light source distribution. Tokuyoshi et al. proposed Stochastic Light Culling for VPLs on GGX Microsurfaces [99]. The approach uses ellipsoid to enclose virtual point light sources. Archer et al. introduced Hybrid Lighting for faster rendering of scenes with many lights [5]. The algorithm improved building of a 3D light grid.

While tiled or clustered shading only computes lighting, Olsson et al. proposed method that also computes shadows using virtual shadow maps [82] and [83].

8.2.4 Light Cuts

Light cuts is an algorithm that clusters light sources. It was introduced in 2005 by Bruce Walter in paper Lightcuts: A Scalable Approach to Illumination [104]. Light sources are grouped together by their influence on a scene. If light sources are close to each other and have similar color and intensity, they are clustered. The clustering is done hierarchically. Hierarchical clusters could greatly reduce computation time of illumination. Screen samples can be illuminated using clusters of lights, in case of no significant visual artefact. Light cuts can be seen in Figure 8.11. The algorithm is scalable and it is commonly used for realistic illumination computation.

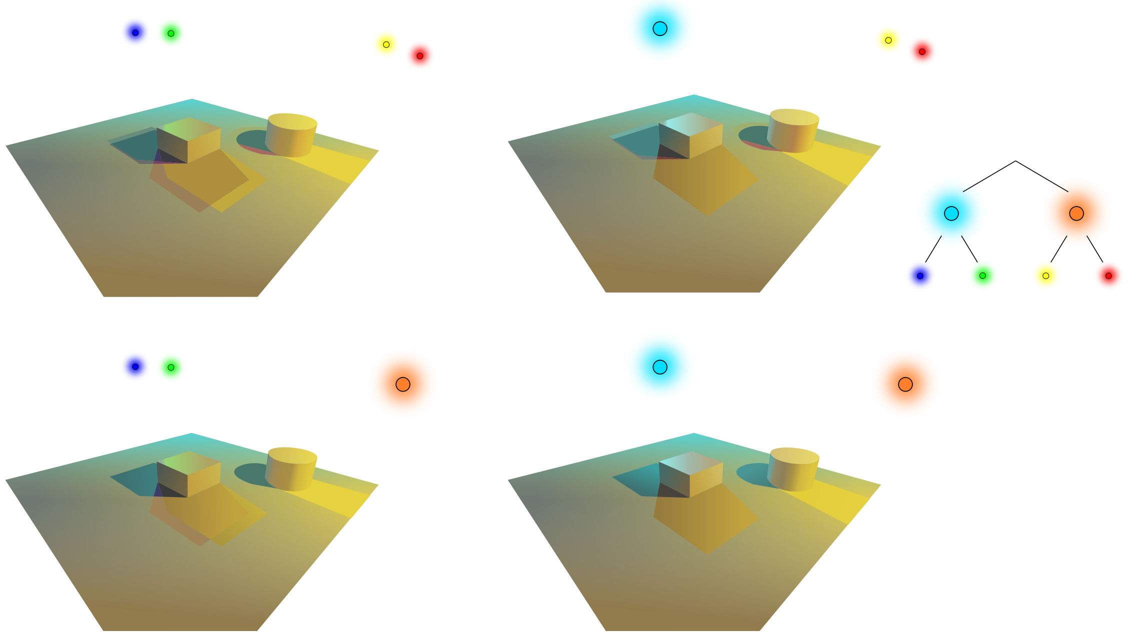


Figure 8.11: The image shows a scene illuminated with four light sources. Four light sources are clustered into a tree (right side). Four separate images show different cuts of the light tree.

Chapter 9

An Improved Non-Orthogonal Texture Warping for Better Shadow Rendering

In interactive applications, shadows are traditionally rendered using the shadow mapping algorithm. The disadvantage of the algorithm is limited resolution of the depth texture which may lead to aliasing artifacts on shadow edges. This section introduces an improved depth texture warping with non-orthogonal grid that can be employed for all kinds of light sources. For instance, already known approaches for reducing aliasing artifacts are widely used in outdoor scenes with directional light sources, but they are not directly applicable for omnidirectional light sources. The new method shows that the improved warping parameterization reduces the aliasing artifacts and it is able to present high quality shadows regardless of a light source or a camera position in the scene, Figure 9.1. The full content of this section can be found in paper [73].

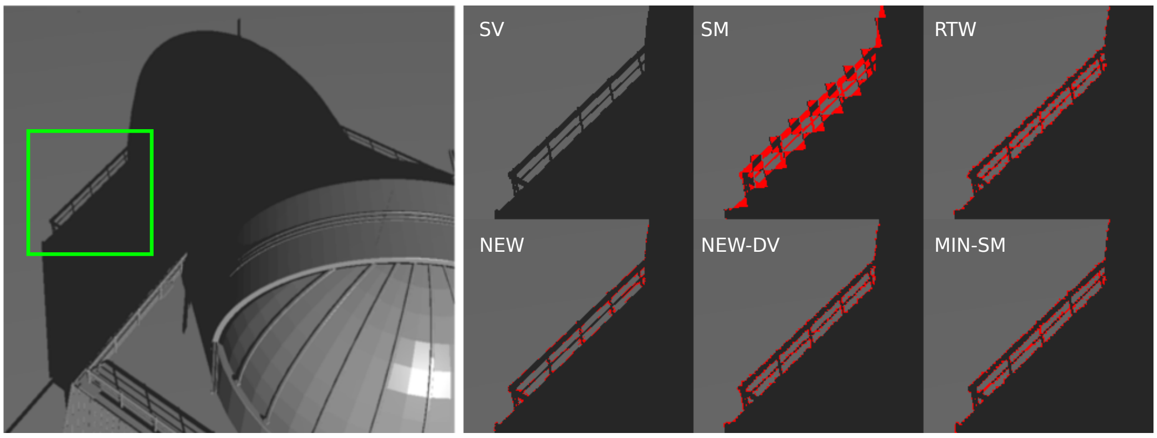


Figure 9.1: The figure shows the difference in quality of shadows cast from Observatory scene using different methods. Red pixels are wrongly evaluated. From left to right: Shadow Volumes (SV), Shadow Mapping (SM), Rectilinear Texture Warping (RTW), the new solution, the new solution using only desired view (DV), SM + minimal shadow frustum.

9.1 Introduction

The quality of computer-generated imagery is still being improved. Nowadays, computers can synthesize images in nearly photorealistic quality and in real-time. One of the most important visual cues, still worth improving, are shadows. The two key algorithms for shadow rendering [106, 17] have been accelerated on GPUs. The shadow volumes approach [17] suffers from the need to render a large amount of data to gather necessary information for rendering shadows. On the other hand, shadow mapping approach [106] is limited by the size of depth texture. In this section, this problem is addressed with the improved depth texture parameterization that makes use of the available resources more efficient.

The resolution of the shadow map determines the number of samples that can be utilized. Some approaches were introduced that improve sampling of the important parts of the scene. These approaches work well for outdoor scenes where the major part of lighting comes from the directional light source [110]. This type of light sources can be processed efficiently with the shadow mapping algorithm. On the other hand, the basic shadow mapping algorithm cannot easily address omnidirectional light sources and it needs additional improvements, e.g. using cube shadow maps, or an alternative parameterization [11, 88, 46]. In this case, the improved sampling is difficult to achieve using approaches mentioned above because the algorithms would need non-trivial modification.

Since the omnidirectional point light sources can cast shadows into all directions, the regions where the need to improve scene sampling exists can be distributed throughout the depth textures. It all depends on the scene complexity and also on the mutual position of the light source and the camera. While outdoor scenes are lit by directional light source, the important parts are located in front of the camera and the importance decreases with a distance from the camera.

Various approaches have been introduced that can parameterize the shadow map and thus improve sampling on selected parts of the scene based on the scene analysis [103, 78, 88, 46]. However, neither of these approaches is fully automatic or robust enough and they work only in few cases when the scene is lit with directional light sources or the light source is outside a camera view frustum.

The new solution computes an improved parameterization based on importance driven depth texture warping. Regions in the depth texture, where the sampling is not optimal, can be identified. These regions can be enlarged in order to get higher sampling rate. The novel method employs modern GPUs in the warping process thus these additional computing steps have no crucial impact on an overall performance.

The new approach introduces an additional step that is performed before the traditional shadow mapping algorithm is applied. In this step, a non-orthogonal warping grid is computed and this grid is used during the shadow rendering step.

Main contributions are:

- Introduction of a novel importance function for determining sampling rate of depth texture. This function extends the set of functions introduced by Rosen et al. [88],
- The non-orthogonal warping grid which leads to better control of importance-based warping without affecting the nearest regions in the texture (in the same row and/or column).

9.2 Previous Work

The shadow mapping algorithm was first published in 1978 [106]. Since then, many approaches addressing its aliasing issues have been published.

Stamminger and Drettakis [96] introduced an idea of creating depth texture after performing of perspective projection. This step emphasizes regions in front of the camera where the aliasing errors are mostly observable. However, results of this approach are dependent on the mutual position of a camera and a light source. In some cases, creating depth texture in post-perspective space may lead to very unpleasant results because of the perspective transformation function. Also, the overall results are influenced by objects outside the camera view frustum because they introduce additional complexity to the computation.

Fernando et al. [30] introduced a hierarchical structure by subdividing shadow map into smaller shadow map pages having different resolutions due to different level of aliasing in different parts of the current camera view frustum. With camera being dynamic, this hierarchical structure needs to be updated per frame and due to limitations of graphics hardware of that time, significant part of the algorithm runs on CPU.

This method was further optimized by Lefohn et al. [59]. Evolution of GPU hardware allowed more of the algorithm to move on the graphic chip itself by programmable vertex and pixel shader pipeline stages.

Parallel-split shadow maps approach was introduced by Zhang et al. [110]. The idea is to split view frustum into multiple parts according to depth, split light frustum into multiple ones and then independent shadow maps are rendered for each layer. Splitting view frustum is based on a practical splitting algorithm which averages logarithmic and uniform splitting scheme. However, this method is targeted and optimized for outdoor scenes and it would need some amount of work to adapt it for indoor scenes and namely omnidirectional point light sources. Also, the approach does not deal with the perspective aliasing error correctly.

Based on Zhang, Lauritzen et al. introduced Sample distribution shadow maps which further improves partitioning, [58]. The camera frustum is partitioned automatically based on receiver sample distribution given by depth buffer, eliminating areas with no shadow samples. This sample distribution is also used to compute tightly-bound light-space partition frusta.

The first method that addressed the problem of important regions distributed in the depth texture was introduced by Rosen [88]. He introduced the rectilinear warping maps that could easily control the sampling in particular parts of the depth texture. This could be controlled by importance function and the approach could be used for point light sources without complex modification. Nevertheless, the rectilinear warping schema is not completely local. Other parts of a scene may receive unneeded resolution. This can lead to reduction in overall quality.

Similar approach was published by Jia et al. [46]. They do not limit the approach to perpendicular splitting planes; therefore, they can control the results more precisely. However, this approach needs multiple render passes of the scene to analyze the scene and then decides the dividing schema. This might introduce certain issues for complex scenes.

Finally, some approaches that were focused only on point light sources have been published recently [103, 78]. They discussed possibilities for improving shadow quality using Dual-Paraboloid shadow maps [11]. But this technique is not sufficient due to its nonlinear transformation during generation of depth textures. This introduces additional limitations regarding model quality and especially size of polygons.

9.2.1 Scene Sampling and Parameterization

The shadow mapping algorithm works with two types of samples. A view-sample is a point on a scene surface that is described by its 3D position (and other properties such as color, normal vector etc.). The view samples are generated by sampling the scene's geometry from a camera point of view. Secondly, shadow-samples are generated by sampling the scene from a light source point of view. In both cases, the sampling is performed using an orthogonal grid with a predefined resolution.

However, multiple view samples can be projected onto one shadow sample and then aliasing can be observed in the final image as jagged edges of the shadows. This is caused by uniform rasterization of a texture produced by a graphics hardware.

Another solution is to parameterize the sampling using a *warping function* $y = f(x)$. The function enlarges important parts of a scene in order to increase shadow sampling rate. This technique increases a probability that shadows for different view samples are resolved by different shadow samples. There are two types of the warping function - *global* and *local*. The global warping function can be defined by a transformation matrix. This warping function mostly depends on a mutual position of a camera, a light source and geometry and ignores properties of view samples [96]. The local warping function is derived from properties of view samples and scene analysis [88, 46].

9.2.2 Rectilinear Texture Warping

The novel algorithm is partially based on Rectilinear Texture Warping (RTW) approach [88]. RTW approach utilizes various properties of view samples, e.g. distance to a camera, normal vector or edge detection. The warping function can be constructed using forward, backward or hybrid analysis.

The first step in the forward analysis is rendering of the scene from a light source point of view. Then, the importance map is computed.One additional rendering step is necessary to compute shadows. In the backward analysis, the G-buffer with the scene's depth and color is rendered from a camera point of view. Then, the importance analysis is performed using samples projected into the light space. The hybrid analysis combines both approaches.

The backward analysis is the fastest method because it requires the scene to be rendered only two times. The first rendering pass is used to create a depth buffer from a camera. The second rendering pass creates a warped shadow map. Its complexity is linear with relation to the number of light sources.

The warping function in RTW is composed of two 1D warping functions that operate on projection plane of a light source (see Figure 9.2). These functions are derived from an importance map. The *importance map* is constructed by projection of view samples onto the projection plane of a light source. Multiple view samples can be projected into one pixel of the importance map. For every pixel, the importance value is computed using the view sample properties. The 1D warping functions are derived separately for column and rows according to a maximal importance value. Since the functions parameterize vertical and horizontal component of the shadow map separately they produce an orthogonal warping grid.

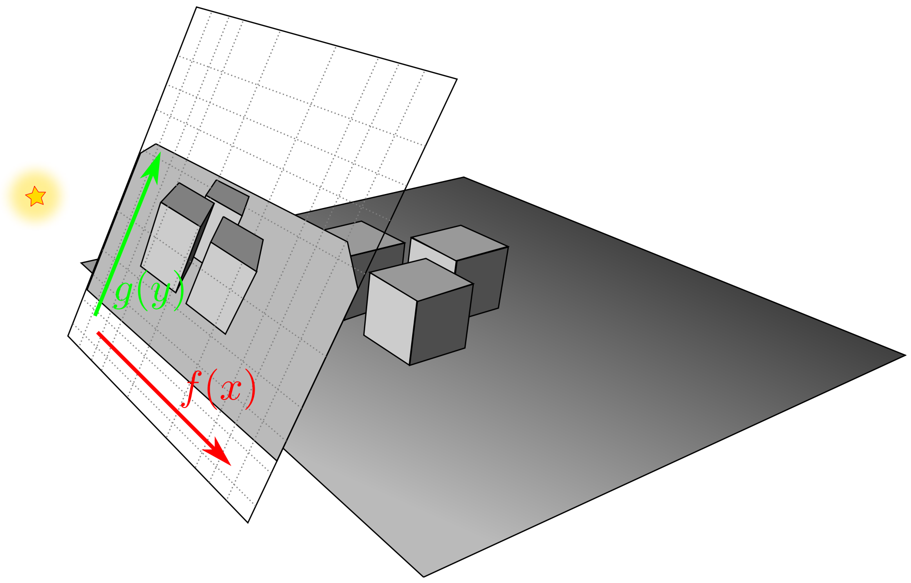


Figure 9.2: Two 1D warping functions in the light space enlarge parts of the scene that are important according to the importance map.

9.3 Shadow Rendering Using Non-Orthogonal Warping Grid

The basic idea of the novel algorithm is to achieve better distribution of view samples in the shadow map. Every shadow sample resolves shadow for all view samples that were projected on it. The ideal situation occurs when one texel from the shadow map samples a surface that is projected onto one pixel in the image space. However, this is hardly achievable in most of the scenes because of the scene complexity, geometry and mutual position of the camera and the light source. Because of this fact, it can be assumed that the best result is observed when the number of view samples for all shadow samples is the same.

In the new approach, the importance map has the same resolution as the shadow map. Every pixel in the importance map stores the number of view samples that are sampled by the given shadow sample. The importance map can be created by projection of view samples into to the light space and incrementing a counter by one. This step can be easily accelerated by contemporary GPUs.

The complete algorithm for shadow computing consists of the following steps:

1. Render a scene from a camera point of view to G-buffer
2. Project every view sample into the importance map
3. Compute prefix-sum for every row in the importance map
4. Construct the set of warping functions for rows according to Equation 9.4. Use the prefix-sum from the Step 3
5. Smoothen the set of warping functions, e.g. using weighted average
6. Project every view sample onto the importance map (and increment by 1) leveraging the set of warping functions created in the previous step
7. Repeat the Steps 2-5 for all columns

8. Create shadow map using both sets of warping functions
9. Evaluate shadows in the scene using G-buffer, the set of warping functions and the warped shadow map

The first step is generation of the G-buffer. Apart from other properties, it contains positions of view samples that are needed for importance estimation.

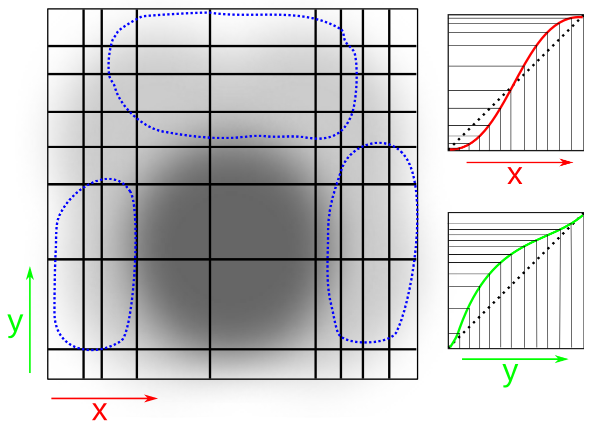


Figure 9.3: Importance map for RTW, Left: Combination of two 1D warping functions, Right: two 1D warping functions. It can be seen that blue parts are oversampled. The larger cells cover more important areas of the shadow map.

The most important are the steps 2-7 where the algorithm constructs the set of 1D warping functions. Warping functions are derived in different manner than Rosen [88]. For every row and every column, the algorithm constructs one 1D warping function separately and thus it does not allocate unneeded resolution in other parts of the shadow map. The degree of freedom for warping functions is increased using this approach and it should not allow the situation illustrated on the Figure 9.3. The steps are described in detail in the following section.

9.3.1 Construction of 1D Warping Functions

For one row of the importance map, let $f(x)$ be a function that returns the number of view samples on a normalized position x and let $g(x)$be its corresponding prefix-sum function:

$$n = f(x) \quad x \in \langle 0, 1 \rangle \tag{9.1}$$

$$s = g(x) = \int_0^x f(x)dx \tag{9.2}$$

For evenly distributed view samples in the row, the ratio of the number of view samples on all positions before x, i.e. $g(x)$, and the total number of view samples $g(1) = N$ is equal to ratio of the position x and the row length:

$$\frac{g(x)}{g(1)} = \frac{x}{1} \tag{9.3}$$

Expression $g(x)/g(1) > x/1$ implies that there are more view samples than the number of samples x and thus the area needs to be enlarged to achieve uniform sampling rate. On the other hand, expression $g(x)/g(1) < x/1$ implies that there are less view samples and the area can be smaller.

The warping function can be derived as an offset $o(x)$ that has to be added to the actual view sample position. The offset function is given by:

$$o(x) = \frac{g(x)}{N} - x \tag{9.4}$$

If a view sample is projected onto a particular row in the shadow map, then a new sample position x' in the row is given by:

$$x' = x + o(x) \tag{9.5}$$

Before warping functions for columns are constructed, the importance map needs to be recomputed. The newly derived set of 1D warping functions for rows is applied to the importance map. After this step, the number of view samples that have to be redistributed in a given column is nearly constant (see Figure 9.4). When 1D warping functions for columns are derived, all view samples are distributed almost uniformly.

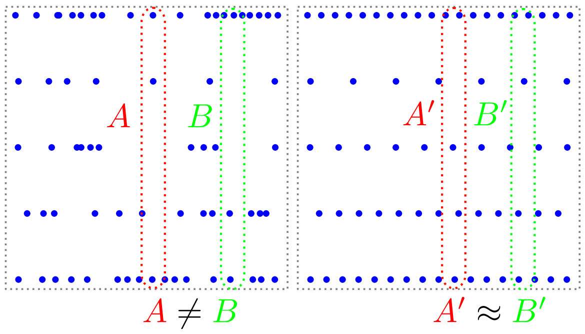


Figure 9.4: Left: Five rows of the importance map. Blue dots indicate view samples. Right: the importance map constructed using the set of row warping functions. Columns in the left do not contain the same number of view samples. Columns in the right contain approximately the same number of view samples.

As it is mentioned in the Section 9.2.2, the RTW algorithm constructs two 1D warping functions - one for rows and one for columns. The novel algorithm improves this approach and constructs a set of warping functions for all rows and all columns. Nevertheless, these functions need to be smoothened in order to prevent them from providing too different offsets. Otherwise, the large polygons that are linearly rasterized would not be processed correctly. The smoothing step is included in the RTW algorithm as well. Rosen performs this step on the warping functions. However in the novel approach, the smoothing is performed among all warping functions. It can be implemented, for instance, as a weighted average of the results based on the number of view samples on a row or a column respectively (see Figure 9.5).

The complete warping function can be expressed as:

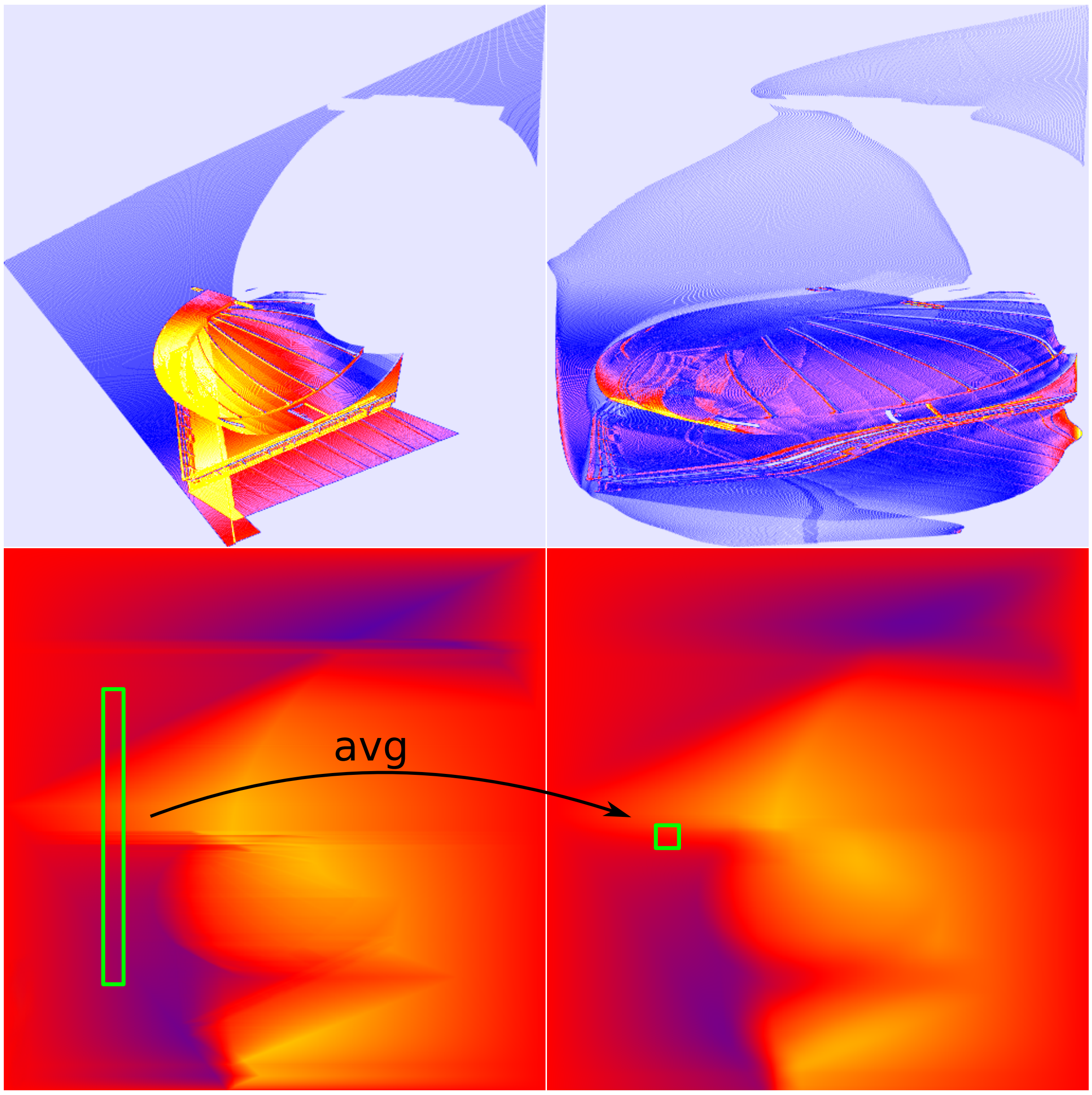


Figure 9.5: Top Left: Importance map. Top Right: The importance map after application of row warping functions - importance map for columns. Bottom left: a set of warping functions for every row of the importance map. Bottom right: warping functions smoothed using an averaging window shown in green. Yellow color in warping functions means positive offset to the right for certain position in the row. Blue color means offset to the left.

$$\begin{aligned} warp(x,y) &= (x + o_x^{(i)}(x), y + o_y^{(j)}(y)) \\ i &= \lfloor y \cdot w \rfloor \\ j &= \lfloor (x + o_x^{(i)}(x)) \cdot w \rfloor \end{aligned} \tag{9.6}$$

where w is the shadow map resolution (number of pixels in a row), $o_x^{(i)}(x)$ is a warping function for i^{th} row, $o_y^{(j)}(y)$ is a warping function for j^{th} column.

When both sets of warping functions are applied, the view samples projected onto the projection plane of a light source are better spread as can be seen on the Figure 9.6.

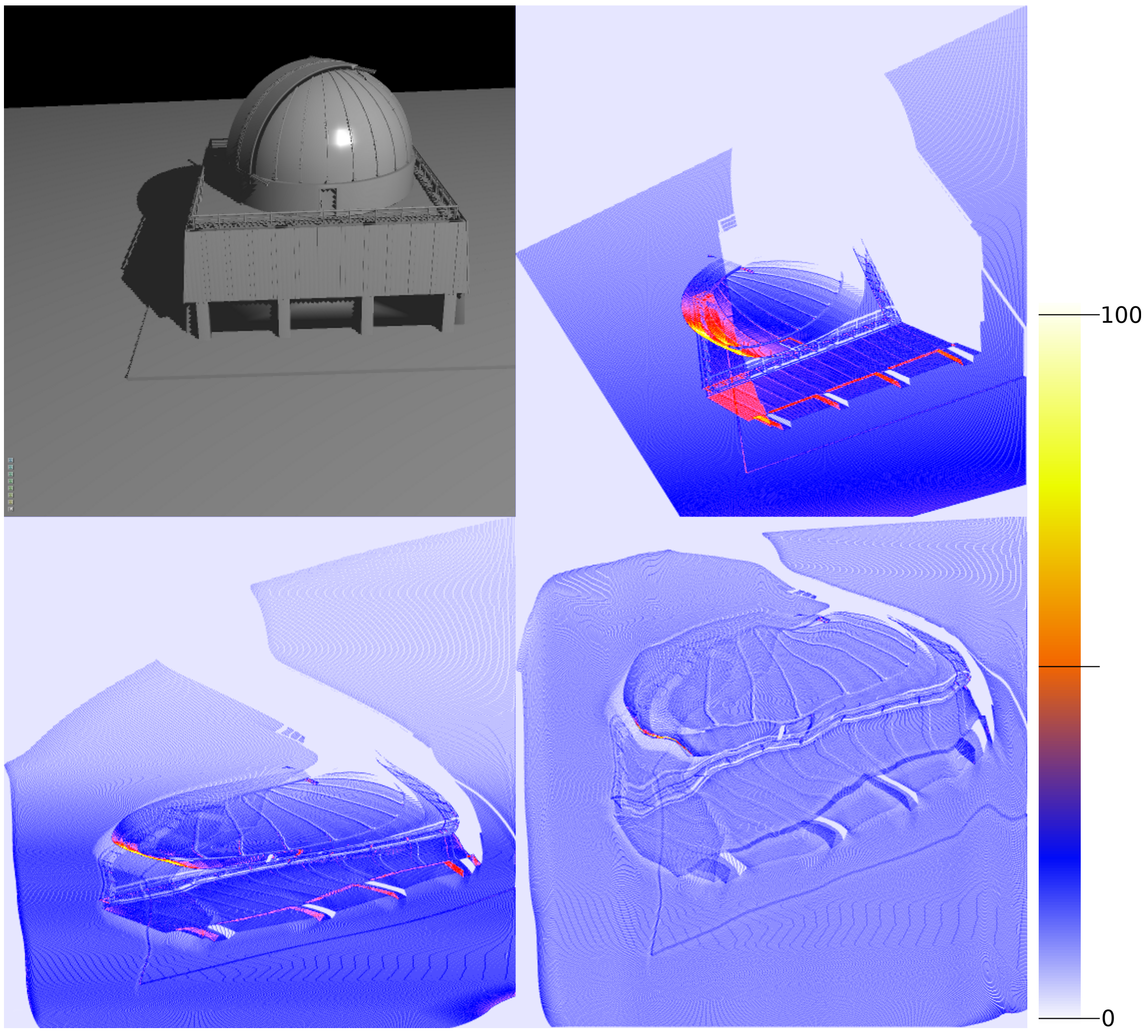


Figure 9.6: Top Left: Scene rendered from a camera point of view. Top Right: the importance map created from view samples. Bottom Left: reprojected view samples using only row warping functions. Bottom right: reprojected view samples using both sets for warping functions. It can be seen that view samples are more spread across the importance map in the final stage. Light parts of second image are pixels with no view samples. These pixels correspond to those shadow map pixels that are unused - they resolve shadowing equation for invisible parts of the scene. In the final image, these light parts almost disappear and the number of projected view-samples for each shadow map texel is reduced.

Once the algorithm constructs both sets of the warping functions, the shadow map can be generated (see Step 8 of the proposed algorithm). A surface point with a world-space coordinate $\mathbf{v} = (v0, v1, v2, 1)$ is projected onto the shadow map in Algorithm 12. Final shadow map can be seen in Figure 9.7.

Algorithm 12: Warping function that can be used in vertex / evaluation shader. Steps 1, 2 project vertex into normalized coordinates of shadow map. Step 3 moves vertex according to warping funcions. Steps 4, 5 project vertex back into shadow map clip space.

```
Input: v - vertex in world space, $M$ - light projection view matrix
Output: p - vertex in the shadow map clip space
1 $\mathbf{a} = M \cdot \mathbf{v}$;
2 $\mathbf{b} = ((a_1, a_2)/a_4 + 1)/2$;
3 $\mathbf{c} = \texttt{warp}(\mathbf{b})$;
4 $\mathbf{d} = (\mathbf{c} \cdot 2 - 1) \cdot a_4$;
5 $\mathbf{p} = (d_1, d_2, a_3, a_4)$;
```

9.3.2 Minimal Shadow Frustum Extension

The algorithm was improved by finding a minimal shadow frustum (MSF) [96] and it was extended using rotating caliper. Using this technique, the algorithm projects only parts of the scene that are visible in the camera view frustum and occluders outside the frustum that cast shadows on objects inside the frustum.

However, since the algorithm is complex, it runs on CPU and thus it may influence rendering speed. Moreover, issues caused by precision of floating point operations have to be considered during implementation.

The goal of this additional improvement is to verify whether the MSF does not provide better results with a less cost.

Rosen et al. presented a desired view (DV) function that works similarly to the MSF. However, they did not clearly show how it influences the overall quality. The novel algorithm supports the DV as well, but it is only used as pre-process step before computing the importance map.

DV simply finds minimum and maximum view samples coordinates in the importance map. In addition, the MSF rotates the bounding box to an optimal position and adjusts near and far planes.

Rosen et al. computes DV in RTW approach from the importance map by finding first/last row and column that contains an importance value greater than zero. In the new algorithm, DV is computed by parallel reduction over the set of view samples projected into the shadow map space. DV does not contribute to warping process, it only focuses the relevant part of shadow map. The DV function can be applied before construction of the warping functions (before the Step 2 of the Algorithm 12):

9.3.3 Implementations Details

The algorithm was implemented in OpenGL 4.4 using compute shaders. The importance map is created using image atomic operation *imageAtomicAdd* delivered with OpenGL.

The solution requires additional memory as compared to the basic shadow mapping algorithm. It uses deferred shading for creating the G-buffer that requires set of 2D textures. For storing the warping functions, it needs two one-channel floating point 2D textures that have the same resolution as the shadow map. Further, the algorithm requires few textures for storing temporary results - the importance map, prefix sum map and storage for not smoothed warping functions. The additional memory requirements are thus dependent on the shadow map resolution. For instance, when the shadow map with resolution $w = 1024$ is used, the algorithm needs to allocate additional 20 MB of the memory.

The memory requirements can be decreased by using e.g. another format of textures. For instance, 16bit textures for the importance map or prefix-sum map. Also, with increasing number of lights, the memory requirements increase only for storing the warping functions: $8w^2$[bytes] for one light source.

9.4 Results and Discussion

The results were measured on a PC running Intel Core i7 4790 with 16GB of memory with a high-end GPU: NVidia GTX 980. Operation system was Linux Ubuntu 14.04.2.

The novel algorithm was compared with the Rectilinear Texture Warping algorithm (RTW) [88], the basic shadow mapping algorithm (SM), accelerated silhouette-based shadow volumes algorithm (SV) [72] and the shadow mapping algorithm extended with the minimal shadow frustum (MIN-SM). The quality and speed of all approaches was measured (see Table 9.1 and Figure 9.1).

Method	time per frame [ms]
SM	1.596
MIN-SM	1.7
SV	8.750
RTW	3.296
New	**4.708**
New-DV	**2.521**

Table 9.1: Performance comparison of implemented methods. Times are in miliseconds. Measured for Observatory scene on 1024×1024 resolution with 512×512 resolution for the shadow map.

Regarding quality comparison, the shadow volumes algorithm was selected as the ground truth. It provides sample-precise shadows and, moreover, it also defines a lower boundary for speed. No solution based on the shadow mapping algorithm can be slower than the silhouette-based shadow volume approach, otherwise the shadow volume is better option.

The RTW algorithm is the most similar approach to the new solution. RTW algorithm was implemented with backward analysis used for creation of the importance map. The timings for the crucial steps of both approaches can be seen in Table 9.2. The implementation uses both the distance to eye importance function and the desired view function in all reference images (Figure 9.1).

Also, comparison with the shadow mapping algorithm extended with the minimal shadow frustum (MSF) shows some interesting results. The main reason for including this method is to find out if the MSF is not sufficient enough for rendering images with a similar quality. Rosen did not describe this extension and did not show any results.

The algorithm was measured on three scenes (see Figure 9.8). Various types of scenes (outdoor as well as indoor) were selected in order to show that the new solution can be adapted to different environment and types of light sources. Times for all scenes are shown in Table 9.3.

Figure 9.8 shows differences from the reference solution (shadow volumes algorithm). Red pixels are incorrectly computed.

As it can be seen in Table 9.3, the novel method is slightly slower than RTW but it produces better visual results (see Figure 9.1 and Figure 9.8). Measurements show that computing MSF is not expensive and it may be suitable in some situations. Also, it did not provide the best quality. Computing desired view (DV) function in the new method is the third fastest method but visual results are worse than using MSF. Results also show that DV function plays a major role in decreasing alias error, but in some situation it is not sufficient.

scene	Conf. room		Sponza		Observatory	
	new	rtw	**new**	rtw	**new**	rtw
desired view	**13.9**	89.1	**15.4**	82.5	**18.4**	86.8
imp. map	**68.4**		**61.0**		**69.0**	
shadow map	**14.4**	7.3	**19.9**	14.1	**8.2**	9.3
final pass	**3.3**	3.5	**3.7**	3.2	**4.4**	3.8

Table 9.2: Overhead of algorithm steps in new and RTW algorithm for different scenes. Values are in percent.

Scene	Conf. room	Sponza	Observatory
triangles	126665	261978	52583
gbuffer	2.16	2.229	1.84
SV	9.64	18.41	14.96
SM	0.21	0.40	0.16
RTW	3.14	3.47	3.02
New	**3.63**	**3.84**	**3.23**

Table 9.3: Performance comparison of implemented methods for different scenes. Times are in miliseconds.

Minimal Shadow Frustum

The experimental results show that performance of DV and MSF depends on current hardware setup. MSF performs better than DV when running on fast CPU and slow GPU.

The impact of both approaches on the quality was compared. The MSF or DV performs better when a small part of a scene is rendered. However, in real world scenes the camera renders a bigger part of a scene and in these cases the warping techniques perform better (see Figure 9.8 and Figure 9.7). The MSF or DV do not generate the view frustum small enough and thus artifacts on shadow edges are more apparent.

9.4.1 Limitations

The novel algorithm as well as the RTW algorithm have to deal with linear rasterization unit. The result of new warping process can be seen in Figure 9.6 (bottom right). The warping functions distort the space. Nowadays, the rasterization pipeline can handle only the triangle vertices. If the warping function changes rapidly between two vertices, some errors can be observed (see Figure 9.8 top, right for missing shadows under curtains). The novel approach uses a few techniques to deal with these errors.

First, The adaptive tessellation provided by OpenGL is utilized. The similar improvement was suggested by Rosen et al. Further, the size of smoothing window when averaging the warping functions is optimized. The wider the window is, the less different are the warping functions. In the extreme case, the new solution converts to the RTW algorithm. Another solution is to use weights during smoothing step. It can influence sizes of offset values.

9.5 Conclusion and Future Work

This chapter presents an extension of the Rectilinear Texture Warping algorithm achieved through the improved non-orthogonal warping grid constructed using the set of 1D warping functions. The novel importance warping function results in less artifacts at the shadow edges.

The improvement has been evaluated on various testing scenes. The proposed method is fast and provides better results than the RTW algorithm. Also, various improvements and extensions that can be used together with the solution are discussed.

Standard methods for alias reduction globally change sampling rate using partitioning of a scene where directional light sources are commonly used. The novel method changes sampling rate locally and thus it can be used with other kinds of light sources using DPSM or cube maps.

The future work includes adaptation the algorithm for other visual effects, e.g. mirrors, refraction etc.

The novel algorithm is applicable to cube shadow maps and DPSM, see the Figure 9.9 and the Figure 9.10. Interesting observation is that the warping process works better on dual paraboloid shadow maps than on cube shadow maps, Figure 9.11. The warping on DPSM has more freedom to redistribute view-samples in the importance map.

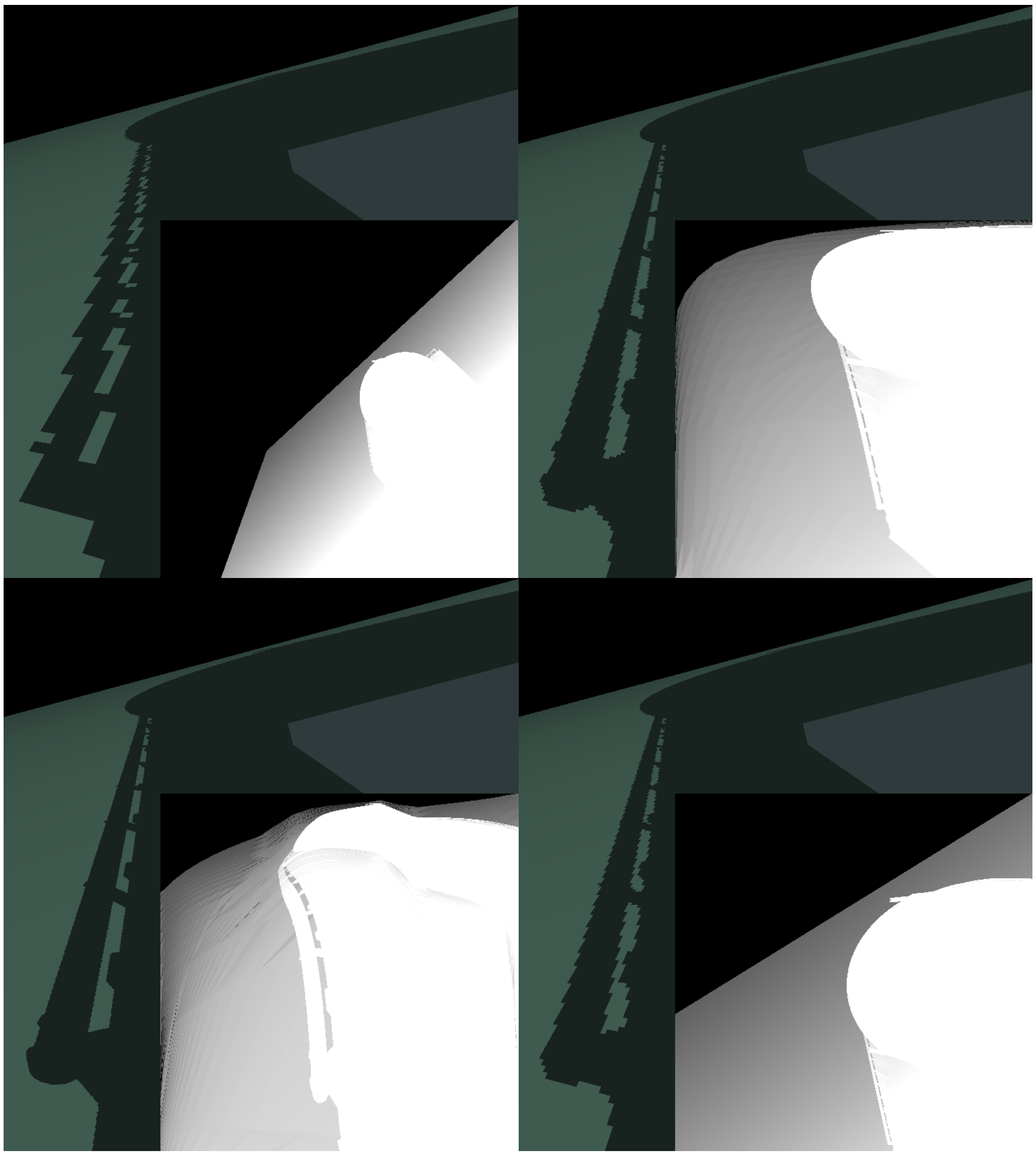


Figure 9.7: Images show shadow maps (grey squared images) for Observatory scene. From left to right: shadow mapping (SM), Rectilinear Texture Warping (RTW), new solution, new solution using only DV.

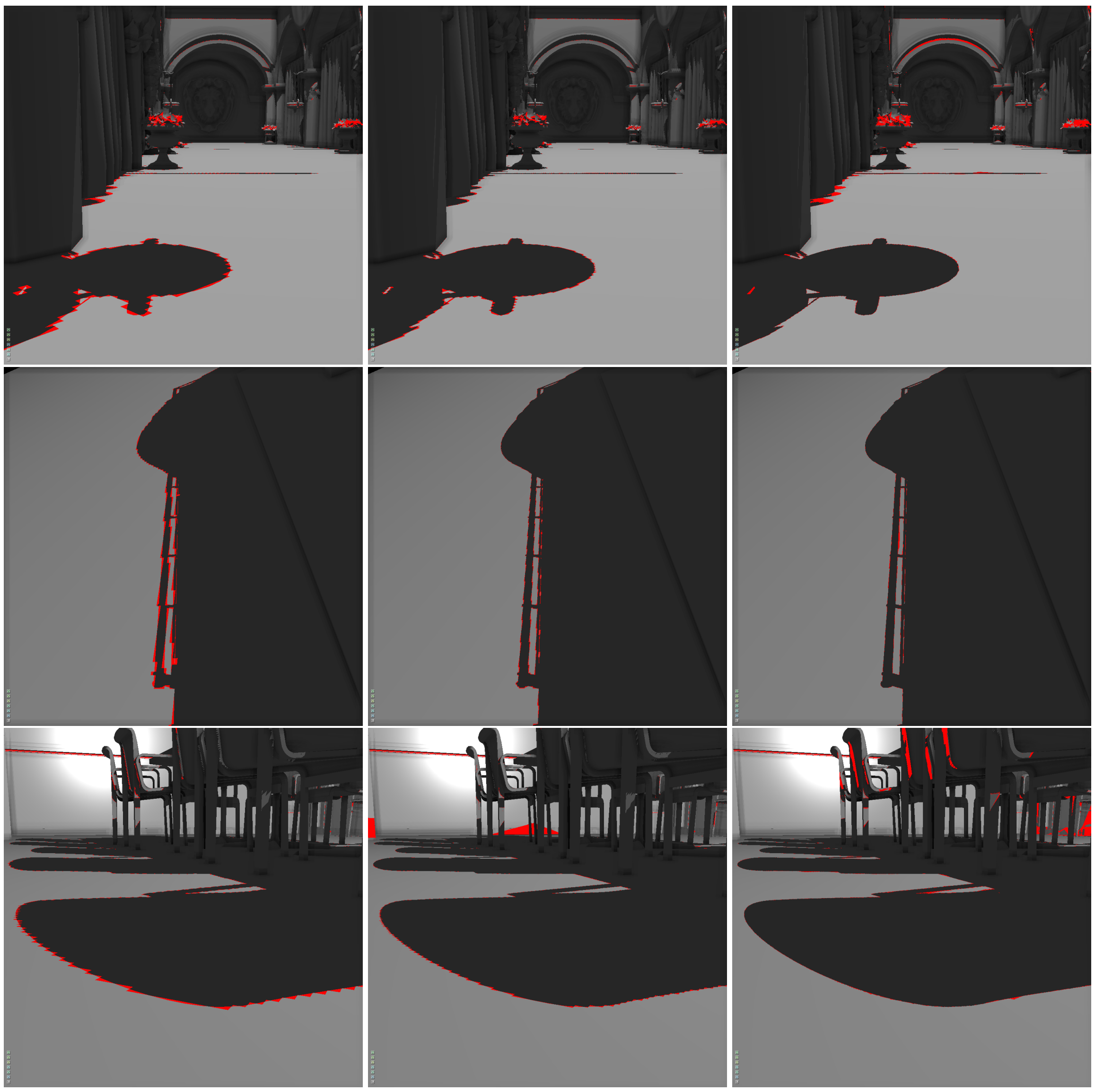


Figure 9.8: Images show difference between shadow mapping techniques and shadow volumes. Images in the first column show basic shadow mapping. The second column shows RTW method and third column shows the new warping method. First scene is Sponza, second scene is Observatory and last scene is Conference room. Times are shown in Table 9.3.

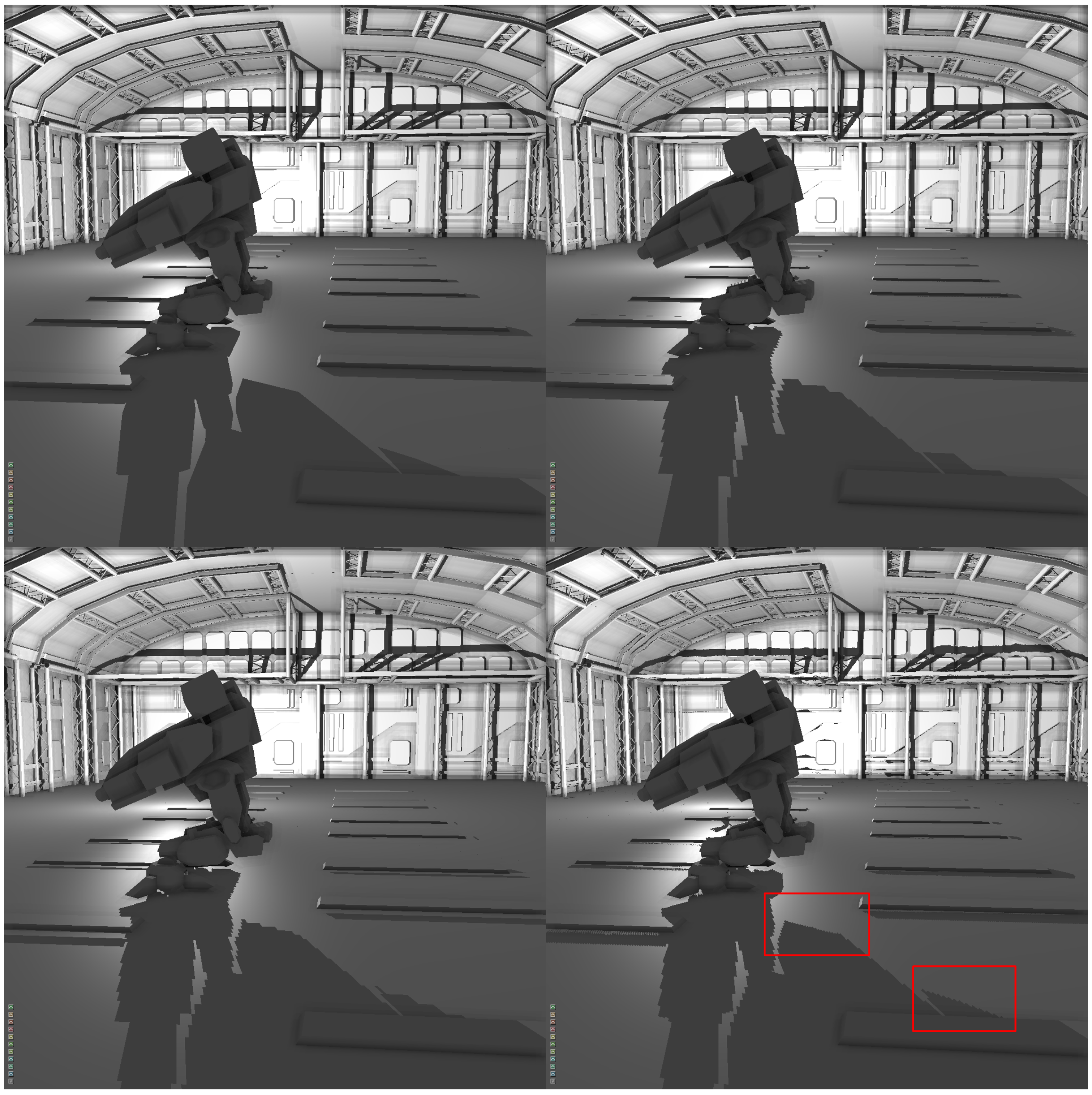


Figure 9.9: The image shows the novel warping process applied to cube shadow mapping. Top left shows shadow volumes, top right shows cube shadow mapping. Bottom left shows cube shadow mapping with desired view. Bottom right shows cube shadow mapping with desired view and warping. The warping process improves the quality in most parts of the scene.

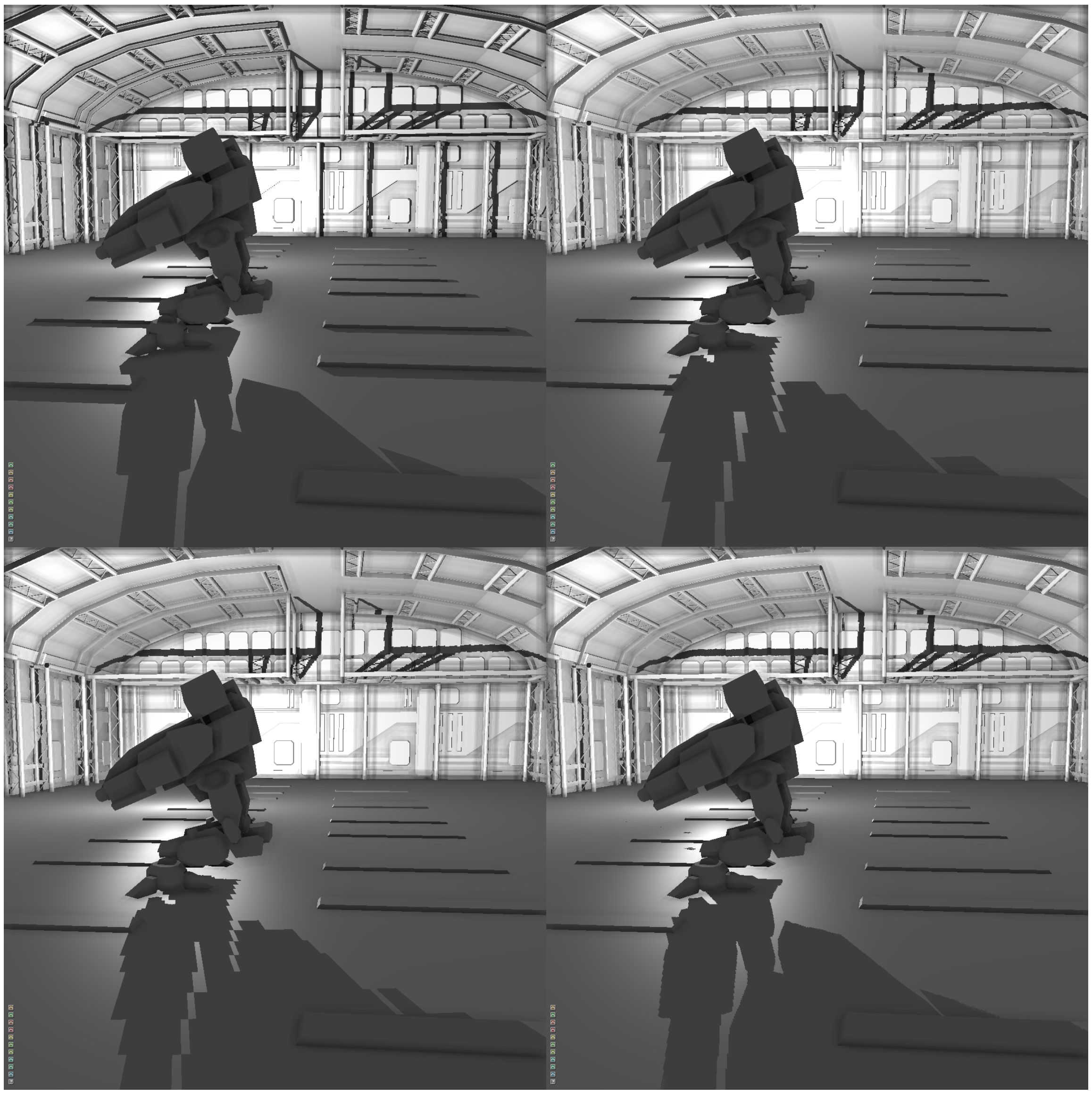


Figure 9.10: The image shows the novel warping process applied to dual paraboloid shadow mapping. Parts show shadow volumes, DPSM, DPSM with desired view, DPSM with desired view and warping. The warping significantly improves the quality.

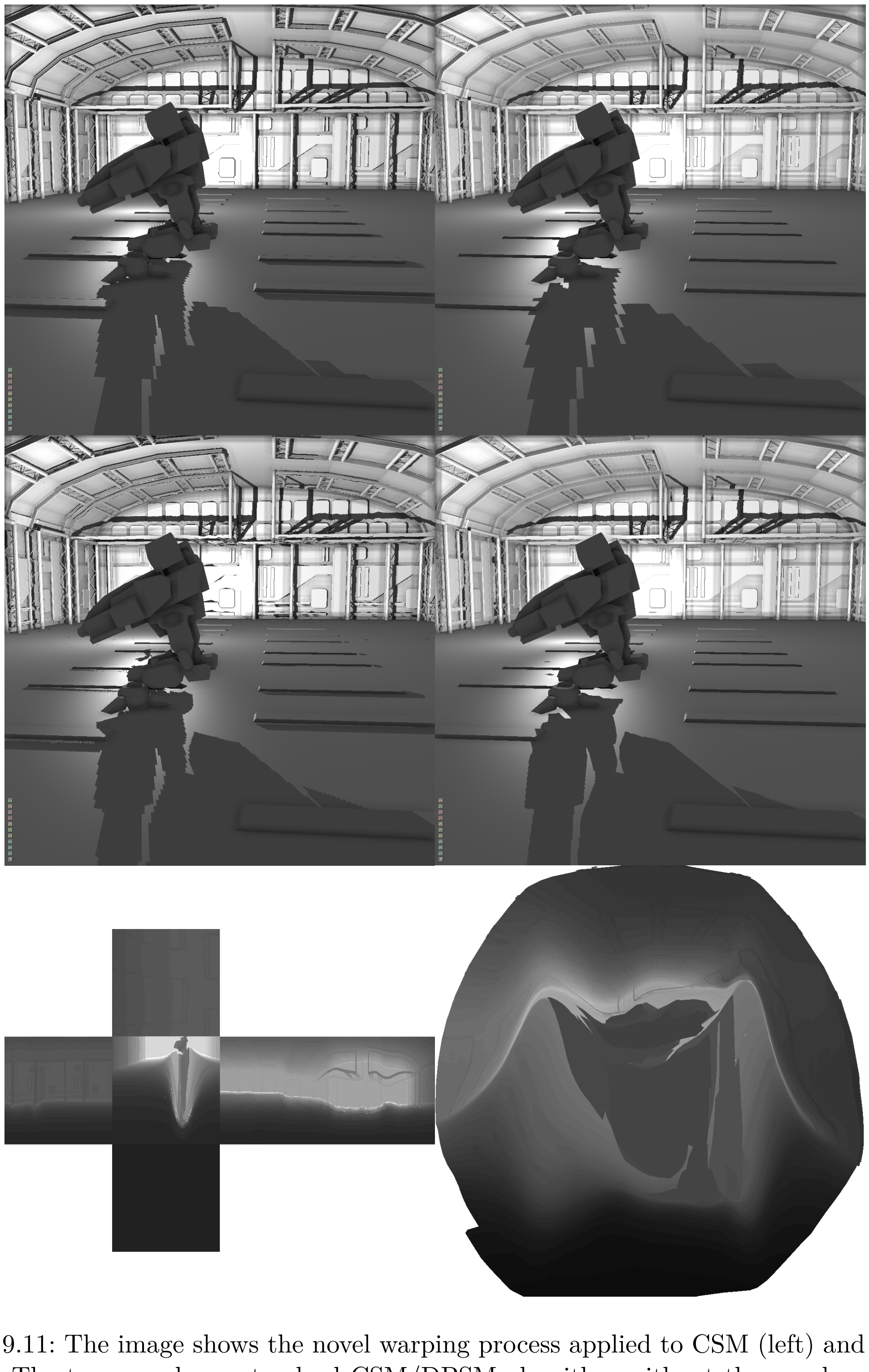


Figure 9.11: The image shows the novel warping process applied to CSM (left) and DPSM (right). The top row shows standard CSM/DPSM algorithm without the novel approach. The middle row shows CSM/DPSM algorithm with the novel approach. The bottom row shows warped shadow map. The warping process works better on DPSM and outperforms CSM. For CSM, the warping is restricted to one cube map face. The warping cannot redistribute view-samples between faces. For DPSM, the warping has more freedom and produces better importance maps. The base DPSM algorithm produces worse results than base CSM algorithm, but the warping improves the quality of DPSM much more.

Chapter 10

Improved Computation of Attenuated Light with Application in Scenes with Many Light Sources

This chapter presents and investigates methods for fast and accurate illumination of scenes containing many light sources that have limited spatial influence, e.g. point light sources. For speeding up the computation, current graphics applications use an assumption that the light sources range can be limited using bounding spheres due to their limited spatial influence and illumination is computed only if a surface lies within the sphere.

This chapter explores the differences in illumination between scenes illuminated with spatially limited light sources and physically more correct computation where the light radius is infinite. Results show that the difference can be decreased if appropriate ambient lighting is added. The contribution is the method for fast estimation of ambient lighting in scenes illuminated by numerous light sources. A method for elimination of color discontinuities at the edges of the bounding spheres is also proposed.

The solution is tested on two different scenes: a procedurally generated city and the Sibenik cathedral, Figure 10.1. The approach allows correct lighting computation in scenes with numerous light sources without any modification of the scene graph or other data structures or rendering procedures. It thus can be applied in various systems without any structural modifications. The full content of this section can be found in paper [74].

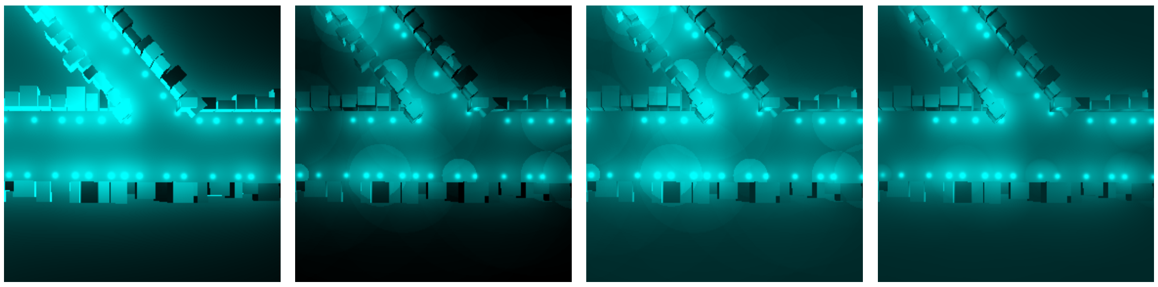


Figure 10.1: Leftmost image: Streets of a city illuminated with 628 light sources with use of attenuation shown in Equation 10.1. Second image: Illumination with use of attenuation shown in Equation 10.2. Mean squared error is $MSE = 0.033$ for $t_{att} = 0.0297$. Third image: The same, except for ambient lighting; $MSE = 0.011$. Rightmost image: Illumination with use of attenuation shown in Equation 10.4 and with use of ambient lighting; $MSE = 0.0195$.

Contemporary graphics hardware is able to render scenes with many light sources in real time [101, 80, 81]. For ease of computation, the point light sources can be bounded by a sphere; then the light range is limited to a given radius of the bounding sphere. The new method deals with rendering of large scenes with complex geometry and numerous point light sources – either real or virtual. Proper evaluation of a per-pixel light source contribution is a costly operation [40]. However, it is not necessary to exactly evaluate lighting in areas where the light contribution is minimal. For this purpose, the point light sources are encapsulated by bounding spheres and the contribution behind the border of the sphere is replaced with estimated ambient light.

To save fill rate, only screen quads or screen-space patches/tiles are rendered for every light source [80] in the deferred shading manner. In its nature, the influence of the light source is unlimited and therefore the light contribution to the whole scene has to be taken into account. The most common lighting model used in computer graphics applications is Phong lighting model [86, 31]. Here, the final color is computed from three parts: diffuse, specular, and ambient colors. The goal is to introduce a fast estimation of ambient lighting that might be applied in every frame without any impact on the overall rendering speed.

Point light sources usually do not significantly illuminate the whole scene due to their limited radius given by light attenuation. They can be divided and clustered into tiles and thus the rendering time can reach real-time rates even for a huge number (e.g. 2k+) of light sources in the scene [80, 81]. In these approaches, each light source is assigned to a screen tile according to the screen-space bounding volume of the light source (typically a sphere). The size of the volume can be large compared to the recognizable contribution of the light source to a given surface. Large bounding volumes, however, decrease the computation performance and thus Olsson et al. advise to set the light radius reasonably small. This can lead to color discontinuity in the computed illumination as can be seen in Figure 10.1.

Dynamic light sources do not present a serious problem in the approaches introduced by Olsson et al. since movement of the light source can smooth out the illumination discontinuities and the discontinuities are hardly recognizable by a human observer.

A similar situation can be observed in techniques based on Virtual Point Light sources (VPLs) [51]. VPLs are employed to approximate the indirect illumination in real-time applications [18, 87]. In these approaches, the properties of the materials or their BRDFs can have a significant impact on the amount of indirect light in the scene. This assumption is not taken into account and the ambient term of the lighting model is set invariably.

For speeding up the computation, the light sources are static and reused between multiple frames [57]. Colors of the light sources may vary due to various materials they belong to. Furthermore, many light sources might be clustered, for instance, according to their position in space [104, 19]. These techniques handle non-diffuse materials and illumination from a wide variety of sources, but they are far from real-time rendering rate.

The ambient term in the material definition ensures the scene to be lit even when no light source illuminates the surface directly. The ambient term should be reasonably small for every material. The amount of light that reaches distant surfaces should vary according to the number of light sources in the scene and their defined radius and spatial distribution. This estimation is not taken into account in any of the previously mentioned approaches.

This chapter describes plausible estimation of ambient lighting that minimizes the effect of inappropriately chosen light truncation caused by the limited spatial influence of light sources. This approach does not need any structural changes in the existing rendering frameworks. The global ambient term is estimated from the properties of all light sources. The estimation process is described in Section 10.2.

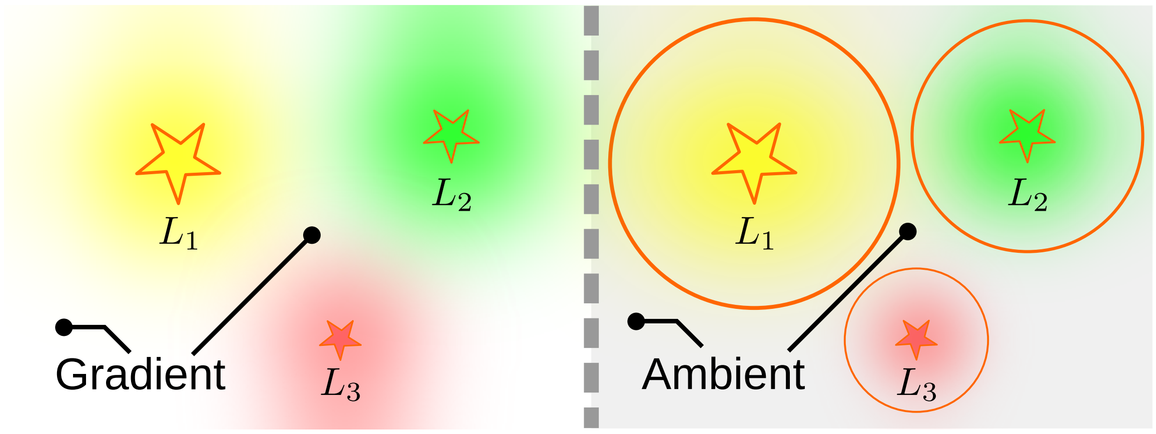


Figure 10.2: The assumption of limited spatial influence of point light sources. **left:** Range-unlimited illumination. **right:** Bounding spheres of light sources represented as orange circles. The algorithm estimates ambient lighting outside of bounding spheres and eliminates discontinuities on the borders of the bounding spheres.

Furthermore, a simple method for color discontinuities elimination on the edges of the bounding spheres is proposed. For this purpose, a simple windowing function is applied directly on the GPU to every pixel. The derivation of the function is explained in Section 10.1.1.

Proposed methods are described and experimental results show the achieved improvement. A method for fast estimation of ambient lighting in scenes illuminated by many point light sources and a method for reducing discontinuities and artifacts in the final renders with use of a range-limited illumination are proposed. The measurements show that the added ambient lighting reduces the mean squared error and the windowing function provides better visual outcomes.

10.1 Lighting Attenuation

This section describes the lighting attenuation and its modifications. Tiled shading [80] and Clustered deferred and forward shading [81] use omnidirectional point light sources represented with spheres. Both methods are based on the assumption (illustrated by Figure 10.2) that the light sources have a limited spatial influence. The scene outside the light's bounding sphere is lit with a constant ambient term.

10.1.1 Attenuated Phong Lighting

The intensity of light naturally decreases with the distance according to the inverse square law [61]. Attenuated Phong Lighting commonly uses an attenuation coefficient:

$$att_0(d) = \frac{1}{k_c + k_l d + k_q d^2} \tag{10.1}$$

Constants k_c, k_l and k_q are the constant, linear and quadratic attenuation constants. Variable d is the distance from the light source. Diffuse and specular terms of the Phong lighting model are multiplied by this attenuation coefficient. This illumination using the attenuation coefficient Equation 10.1 is referred as range-unlimited illumination (RUI).

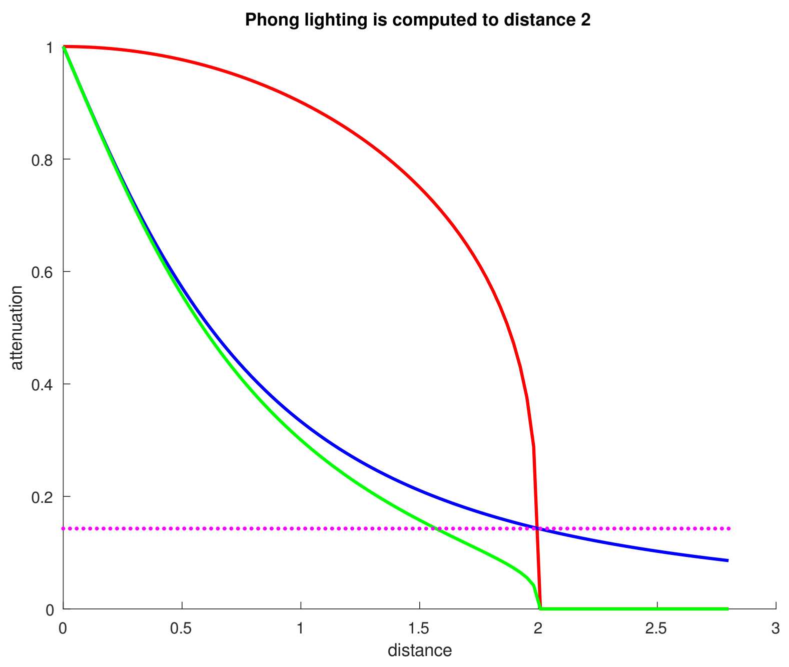


Figure 10.3: The image shows Attenuation coefficient. Constants $k_c = k_l = k_d = 1$, $\mu = .3$. Threshold $t_d = 2$. **Blue** line represents attenuation coefficient given by equation Equation 10.1. **Dotted magenta** line represents the threshold $t_{att} = att_0(t_d)$. **Red** line represents the windowing function. **Green** line represents the final attenuation coefficient given by Equation 10.4.

The attenuation coefficient can be seen in Figure 10.3. If the distance from the light source to the surface point is large, the attenuation coefficient is small and thus the influence of the light on the surface point is close to negligible. A threshold t_{att} can be introduced that modifies Equation 10.1:

$$att_1(d) = \begin{cases} att(d) & \text{if } att(d) \geq t_{att} \\ 0 & \text{otherwise} \end{cases} . \tag{10.2}$$

This modification cuts off the lighting beyond a certain distance and leads to sharp edges which can be visually unpleasing. The threshold t_{att} defines the difference of intensity on the edge. A better result can be obtained by multiplying the attenuation coefficient with a windowing function:

$$w(d) = \cos\left(\frac{\pi}{2} \cdot \frac{d}{t_d}\right)^{\mu}, \tag{10.3}$$

where threshold t_d represents the distance from the light source to the surface and exponent μ modifies/defines the windowing function and its value should be $0 < \mu < 1$.

The final equation of the attenuation is composed from Equation 10.2 and Equation 10.3:

$$att_2(d) = \begin{cases} att(d) \cdot \cos\left(\frac{\pi}{2} \cdot \frac{d}{t_d}\right)^{\mu} & \text{if } d \leq t_d \\ 0 & \text{otherwise} \end{cases} , \tag{10.4}$$

which is continuous but not smooth; the point with the missing derivative is at $att_2(d) = 0$ which does not cause any visible artifacts.

The threshold t_d corresponds to the threshold t_{att} and it can be computed from equation $t_{att} = att_0(t_d)$:

$$t_d = \frac{\sqrt{(k_l^2 - 4k_c k_q)t_{att}^2 + 4k_q t_{att}} - k_l t_{att}}{2k_q t_{att}} \tag{10.5}$$

Threshold t_d represents the radius of the light source's bounding sphere. The illumination using the attenuation coefficient from Equation 10.2 and Equation 10.4 is referred as range-limited illumination (RLI). The attenuation coefficient is illustrated in Figure 10.3.

10.2 Ambient Estimation

This section describes the attenuation coefficient and computation of the bounding sphere radius of the light source given an attenuation coefficient threshold in Section Section 10.1. This section explores the ambient estimation – i.e. how to determine the appropriate ambient term to compensate for the attenuated/truncated light influence. It is certain that if the lighting is computed only on the surface points which lie within the bounding sphere of the light source, the total illumination of the scene is decreased. This can be avoided when a bounding sphere with a larger radius is used and when the number of lights in the scene is low.

However, tiled shading and clustered deferred and forward shading (which are of interest in the targeted applications) are suitable for scenes with large numbers (several thousands) of light sources. Also, the performance of these methods is dependent on the number of lights per pixel. The number of lights per pixel is positively correlated with the radius of the bounding sphere of each light source. Therefore, if the total illumination in the scene is to be increased, the rendering performance will be reduced.

There is an alternative method for increasing the total illumination of the scene: an appropriate amount of ambient lighting can be added in order to reduce the illumination difference between the scene illuminated by RUI and RLI. Estimation of the proper ambient lighting is the goal of this section.

10.2.1 Ambient Estimation Methodology

In order to correctly compute the ambient lighting, the whole scene needs to be illuminated with RUI lights. Then the ambient lighting can be computed as a difference from total illumination of the scene and RLI lighting. This approach would be quite accurate but very computationally expensive. Therefore, the ambient light is estimated with a simple approximation formula.

The total illumination of the scene is given by Equation 10.6 and it is illustrated for a simple virtual scene of a line segment in Figure 10.4. The virtual scene contains only one line segment $[-5, 5]$. Every point on the line segment has a different normal vector. The point light source is located in the origin. The total illumination of this virtual scene is the integral over the illumination of every point located in the scene (line segment). This simple illustration of the total illumination computation of this virtual scene can be extended to any complex scene with many light sources.

$$I_0 = \sum_{l \in Lights} \int_S att_0(|\mathbf{P}_l - \mathbf{x}|) \cdot g_l(\mathbf{x}) \mathrm{d}\mathbf{x} \tag{10.6}$$

Component $\mathbf{P}_l$ is the position of the light source l, $g_l(\mathbf{x})$ is the sum of the diffuse and specular terms of the Phong lighting model at surface point $\mathbf{x}$, S represents every surface point in the scene.

The total illumination of the scene with use of attenuation att_1 is $I_1 \leq I_0$. The goal is to add an ambient lighting term to minimize the difference $I_0 - I_1$.

The method requires an assumption that the sum of diffuse and specular terms of Phong ligthing $g(\mathbf{x})$ has the character of white noise with mean value m. This crude assumption means that there is no coherence of normal vectors and surface color between two surface points. This is not true for two surface points which lie on the same surface and are relatively close to each other. However, in the large scale, this assumption is reasonable. The Equation 10.6 can be approximated as:

$$I_0 = \sum_{l \in Lights} m_l \int_S att_0(|\mathbf{P_l} - \mathbf{x}|)\mathrm{d}\mathbf{x}. \tag{10.7}$$

If a scene is illuminated only with constant ambient lighting a, the total illumination in the scene I_0 is:

$$= \int_S a\mathrm{d}\mathbf{x} = a|S| = I_0 \tag{10.8}$$

If all light source are replaced with constant ambient lighting a, the ambient factor a can be computed as:

$$a = \sum_{l \in Lights} \frac{m_l}{|S|} \int_S att_0(|\mathbf{P_l} - \mathbf{x}|)\mathrm{d}\mathbf{x}, \tag{10.9}$$

where $|S|$ represents the total surface area of the scene.

The computation of the integral in Equation 10.9 is not feasible for complex scenes. Therefore, another crude assumption is needed. The assumption is that all surface points are uniformly distributed in the distance around every light source within a certain maximal distance R (distance from point light source to the farthest point in the scene). This is an approximation of the object distribution in the scene.

Let $A(r)$ be a surface area of geometry inside a sphere with radius r. The assumption means that the ratio of $A(r_1)/A(r_2)$ is close to the ratio of sphere surfaces $(4\pi r_1^2)/(4\pi r_2^2)$. The radii of spheres are within interval $0 < r \leq R$. This assumption may be reasonable for a scene with uniform complexity of geometry.

The equation Equation 10.9 can be simplified using the additional assumption to:

$$a = \sum_{l \in Lights} \frac{m_l}{\frac{4}{3}\pi R^3} \int_0^R 4\pi r^2 \cdot att_0(r)\mathrm{d}r. \tag{10.10}$$

The integral in Equation 10.10 has an analytical solution. The analytical solution of the integral (shown in Equation 10.11) can be seen in Equation 10.12.

$$v(R) = \int_0^R r^2 \cdot att_0(r)\mathrm{d}r \tag{10.11}$$

$$v(R) = \frac{B \arctan\left(\frac{C}{A}\right)}{k_q^2 A} + \frac{k_l \log(att_0(R))}{2k_q^2} + \frac{R}{k_q} \tag{10.12}$$

$$A = \sqrt{4k_c k_q - k_l^2} \tag{10.13}$$

$$B = k_l^2 - 2k_c k_q \tag{10.14}$$

$$C = l_l + 2k_q R \tag{10.15}$$

The ambient lighting can be computed from:

$$a = \sum_{l \in Lights} \frac{3m_l}{R^3} \left(v(R) - v(0) \right). \tag{10.16}$$

Now, Equation 10.16 can be used to compute the ambient lighting when every light source is removed (by setting their radius to 0). The final form of the equation for computation of the ambient lighting is:

$$a = \sum_{l \in Lights} \frac{3m_l}{R^3} \left(v(R) - v(R_l) \right), \tag{10.17}$$

where R_l is the radius of the bounding sphere of light source l.

In order to estimate the ambient lighting in the scene, the maximum distance from every light source R and mean value m of sum of diffuse and specular term for every light has to be chosen. If the value of R is infinite, the ambient lighting will be zero. If the value of R is too small, the ambient lighting may eventually exceed the diffuse and specular terms. The mean m represents degree of influence of particular light source to the scene and should be in interval $0 \leq m \leq 1$.

10.3 Experimental Results

The first goal of experimental testing was to determine the difference between the scene illuminated with range-unlimited illumination and range-limited illumination. The range-limited illumination is the basic assumption of Tiled shading [80]. The second goal was to minimize the difference between the scene illuminated with RUI and RLI using appropriate ambient lighting.

Tiled shading was implemented to compare results. Each scene was illuminated with range-unlimited illumination (RUI) to obtain ground truth. Then, scenes were illuminated using tiled shading with range-limited illumination (RLI). Both attenuation coefficient (Equation 10.2, Equation 10.4) were used. Images with discontinuities were rendered using attenuation coefficient in Equation 10.2. After that, the ambient lighting was estimated using Equation 10.17 from Section 10.2 The estimated ambient lighting was used with both attenuation coefficients and mean squared error (MSE) was computed for result comparison.

The ambient lighting was estimated for attenuation $att_1(d)$. Attenuation $att_2(d)$ returns smaller values than $att_1(d)$ for $\mu > 0$. Therefore, the estimated ambient lighting will be lesser with use of this attenuation and mean squared error potentially higher. However, since attenuation $att_2(d)$ eliminates discontinuities, it provides visually more acceptable outcomes.

10.3.1 Testing Scenes

Testing scenes contain a procedurally generated city with 922 buildings (represented as boxes) and the cathedral of Sibenik. Each scene is illuminated with 628 and 400 of light sources respectively. The city is illuminated with cyan light sources located around the streets. The cathedral of Sibenik is illuminated with VPLs transformed to usual point light sources. These two different scenes were selected to compare the results for inside and outside environments. The focus of the test was to measure the squared error and mean squared error of scene illuminated with RUI and RLI and to evaluate the visual difference between the scenes illuminated with use of the windowing function and without the use of the windowing function.

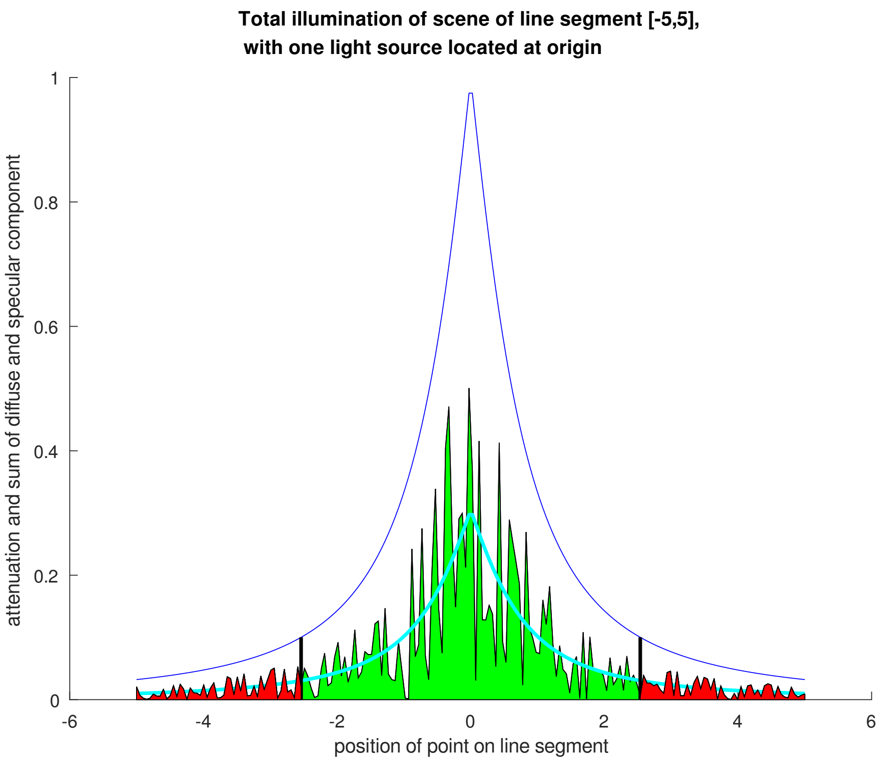


Figure 10.4: The image shows a model example of total illumination of simple 1D scene. The scene contains a single line segment [-5,5] with varying normal vector. The total illumination is represented as the sum of **red and green areas**. Peaks on the **black line** represent the fact that a given surface point is facing towards the light source and has a higher illumination. **Blue line** represents the attenuation. When RLI (range-limited illumination) is used, only the **green area** is computed and the **red area** is omitted decreasing the total illumination. The ambient lighting substitutes **red areas**. The **cyan line** represents the first assumption that replaces illumination at a given point by a mean value.

10.3.2 Comparison to Baseline Solutions

Scenes illuminated with RUI can be seen in the first image of Figure 10.1 and Figure 10.6 and in the first image in the bottom line of Figure 10.7. These images show the correct illumination (every light source influences every surface point within any distance from it).

The methods proposed by Olsson et al. [80, 81] use range-limited illumination (RLI). The interest was in the difference between RUI and RLI. The squared error for the scene of Sibenik cathedral can be seen in Figure 10.5. The left or right image shows the squared error for scene illuminated with or without ambient lighting, respectively. The mean squared error is decreased from value $MSE = 0.013$ to value $MSE = 0.008$.

The squared error for the scene of the procedural city is shown in Figure 10.7. If the ambient lighting is used, the mean squared error is decreased from value $MSE = 0.0553$ to value $MSE = 0.0326$ or $MSE = 0.0426$ with respect to the type of the attenuation (Equation 10.2 or Equation 10.4). The windowing function slightly increases the squared error, but it produces visually more acceptable outcomes. This can be seen in Figure 10.1 and Figure 10.7.

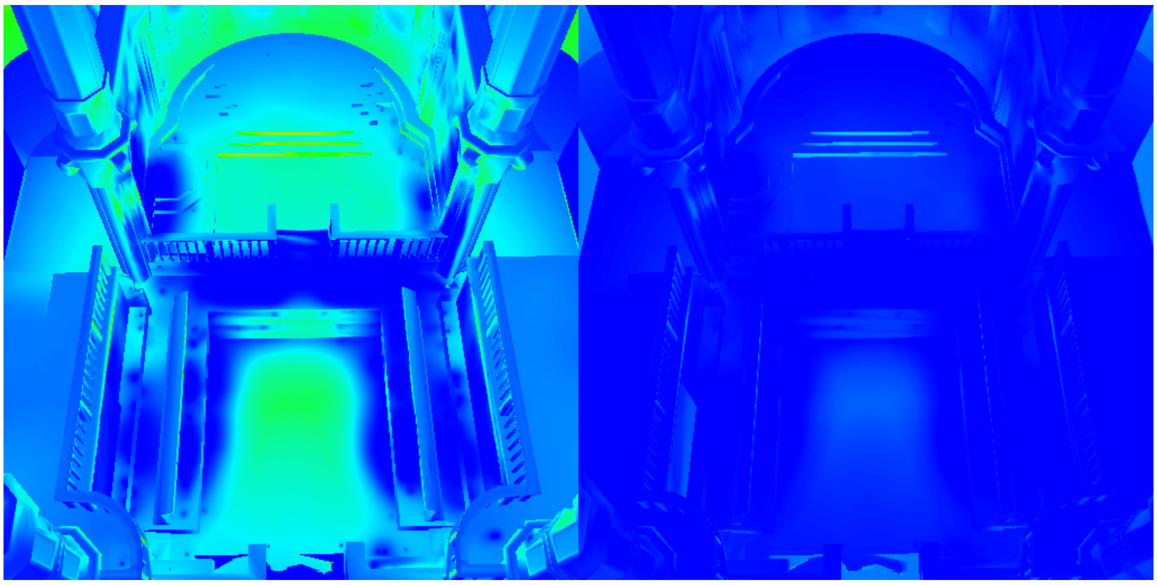


Figure 10.5: The image shows squared error in the cathedral scene illuminated with 400 VPLs. Left image is illuminated without ambient lighting; $MSE = 0.162$. Right image is illuminated with ambient lighting; $MSE = 0.018$. t_{att} was set to value $t_{att} = 0.0096$.

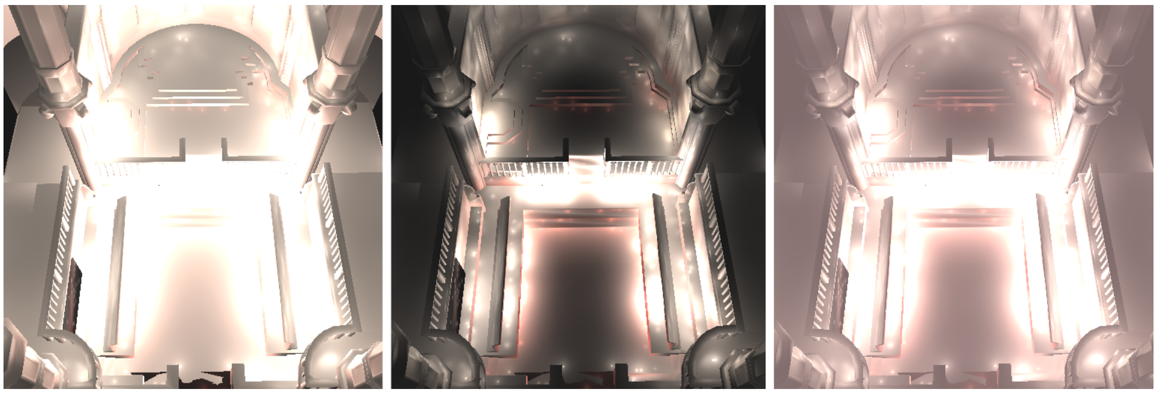


Figure 10.6: The image shows the cathedral scene illuminated with 400 VPL. The left image is illuminated with RUI. The middle image is illuminated with RLI without ambient lighting. The right image is illuminated with RLI with ambient lighting. Some areas in the middle image are darker that in the left image. It is caused by missing illumination. Light sources have limited influence and surface points which are located outside of their bounding spheres are not illuminated. This fact is illustrated in the left image of the Figure 10.5 which shows the squared error.

10.4 Conclusions

This chapter explores the Phong attenuation used with scenes illuminated with multiple light sources using tiled shading. First, it discusses the attenuation in Section 10.1 where it defines three variants of attenuation – range-unlimited illumination with use of attenuation from Equation 10.1 and two range-limited illumination approaches with use of attenua-

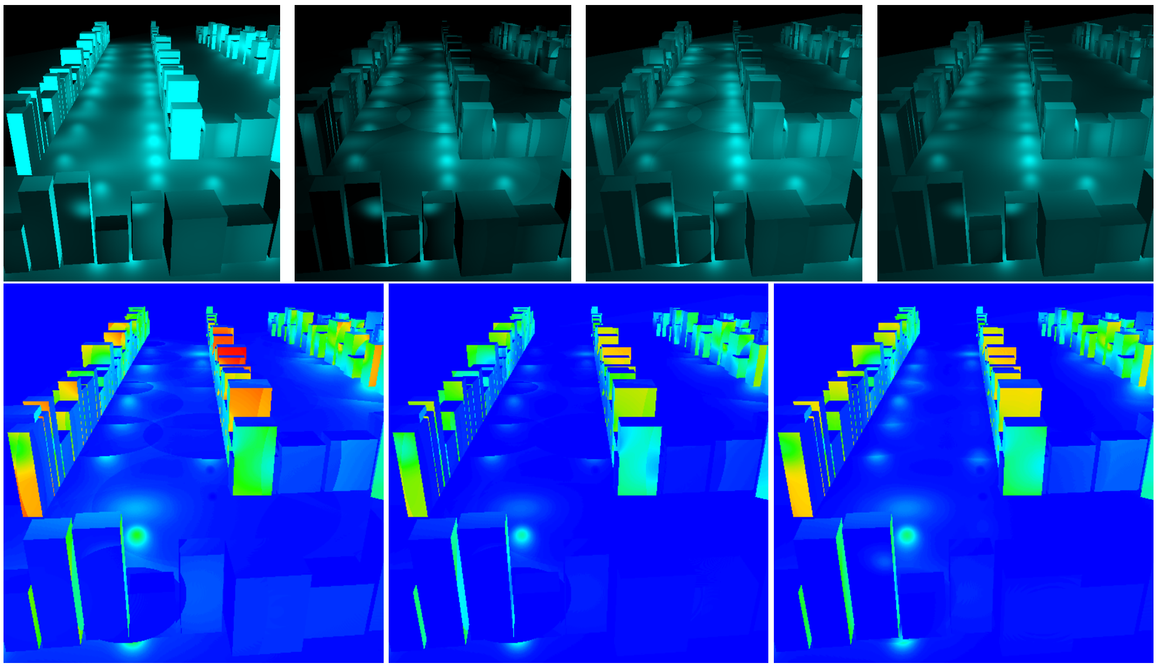


Figure 10.7: The first image in the top row shows the city scene illuminated with RUI. The second image in the top row shows the illumination with RLI with use of attenuation from Equation 10.2. The third image in the top row shows the illumination which differs from the second image by ambient lighting. The forth image in the top row shows the illumination with RLI with use of attenuation from Equation 10.4. Bright spots in the images are specular reflections. The bottom row shows the squared errors of the second, third and forth images. Mean squared errors are $MSE = 0.0553$, $MSE = 0.0326$ and $MSE = 0.0426$.

tion models from Equation 10.2 and Equation 10.4. A windowing function to eliminate discontinuities at the border of the light sources' bounding spheres is proposed.

The second part of this chapter deals with estimation of ambient lighting. A method for estimation of ambient lighting for scenes illuminated with use of the previously defined attenuation models is proposed. The method is based on two crude assumptions. First, it assumes that the sum of diffuse and specular lighting has the character of white noise in space. Second, it assumes that the surface points in the scene are uniformly distributed around every light source within certain maximal distance. These two assumptions were necessary for feasible computation of the integral in Equation 10.6. Both assumptions may not be suitable for every possible scene. Therefore, the estimated ambient lighting differs from the exact value. The estimation process requires the mean value of sum of diffuse and specular lighting of every light source. The mean value was set to $1/2$ for every light source during tests. Also, the estimation requires choice of the maximal distance R for every light source in the scene. It was set to the radius of the bounding sphere of the whole scene.

The ambient estimation process can be quite inaccurate. Both crude assumptions negatively affect the precision. The process can underestimate or overestimate the ambient lighting and only works reasonably well in synthetic scenes.

Nevertheless, results show that the mean squared error is lower if the scene is illuminated by RLI with use of ambient lighting. Added ambient lighting reduced the squared error on surface points which are within the bounding spheres of the most light sources and are

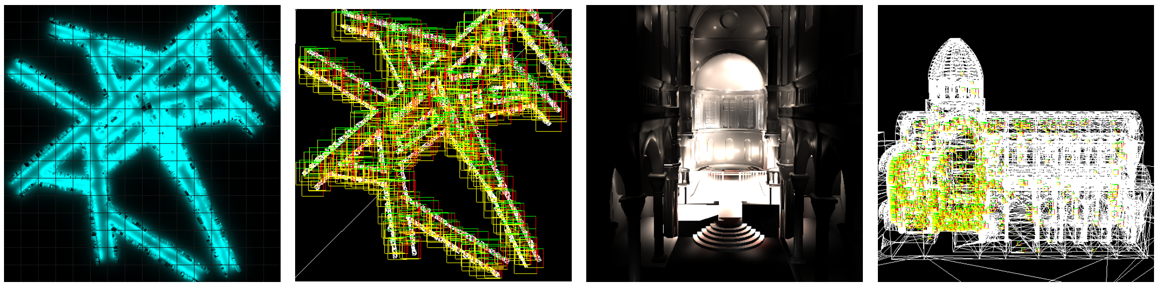


Figure 10.8: The figure shows testing scenes. The first pair of images shows the procedurally generated city. The second pair of images shows the cathedral of Sibenik. The second image of each pair shows the positions of point light sources represented as screen-space bounding quads.

facing them. It can also increase the squared error elsewhere. The surface points on the pillars in the third part of Figure 10.8 are not facing most of the light sources. Therefore, the additional ambient lighting would increase their squared error. On the other hand, the additional ambient lighting would decrease the squared error on the illuminated surface points on the wall behind the altar.

The windowing function reduced the discontinuities and provided visually more acceptable outcomes. This can be seen in Figure 10.1 and Figure 10.7. The windowing function also reduced the total illumination of the scene and therefore it increased the mean squared error.

Future work may contain better estimation of the ambient lighting avoiding the above mentioned two assumptions. The estimation of the ambient lighting could be done per frame with use of information in the G-buffer to minimize the MSE. Another possibility is to use a different windowing function for elimination of the discontinuities.

Chapter 11

Future Goals, Extensions

This chapter describes new extensions to shadow algorithms. These extensions are still in progress during publication of this thesis and are not yet finished.

11.1 Ray-traced Silhouette Shadow Volumes

Ray-traced silhouette shadow volumes (RSSV) combines silhouette algorithms and view-sample cluster hierarchy. The view-sample cluster hierarchy algorithm (CPTSV), proposed by Sintorn et al. [92], computes shadows using per triangle shadow volumes. The method does not use hardware rasterization, but it uses CUDA in order to traverse sample hierarchy. The performance of the method decreases with a scene tessellation. The shadow volumes of each triangle is narrow and the algorithm has to traverse the hierarchy to the leaf level. On the other hand, silhouette shadow volume algorithms have much better performance, because the silhouette contains much less geometry. However, silhouette shadow volumes rasterized on GPUs are fill rate intensive. With increasing screen resolution, the performance quickly degrades, Figure 11.1.

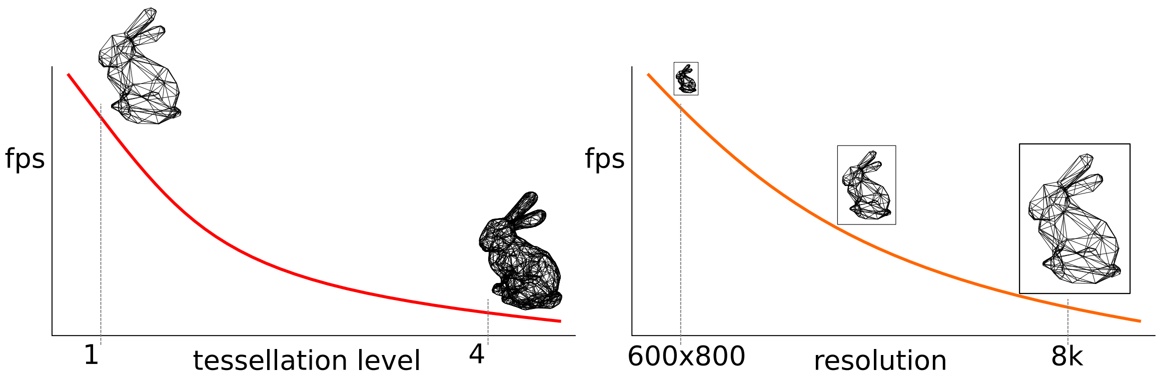


Figure 11.1: The image shows weaknesses of per triangle approaches with sample cluster hierarchy (left) and silhouette shadow volumes (right). The performance of per triangle method quickly degrades with tessellation. Silhouette algorithms are less sensitive to tessellation. On the other hand, stencil silhouette algorithms are more sensitive to screen resolution.

The idea of ray-traced silhouette shadow volumes is to combine both approaches exploiting their strengths and mitigating their weaknesses.

11.1.1 Related Work

Silhouette algorithms and shadow volumes are closely described in previous chapters. The rest of this section describes CPTSV algorithm in more details. CPTSV algorithm can be summed as:

1. Render view-samples.
2. Construct view-sample cluster hierarchy.
3. Construct per triangle shadow volumes.
4. Traverse the hierarchy with every shadow volume and remove clusters and samples that lie inside a shadow volume.
5. Compute shadow mask from remaining clusters and view-samples.

Building a Hierarchy

View-sample cluster hierarchy is constructed using morton codes [77]. A morton code is computed of every view-sample. A view-sample's coordinates (x, y, z) are quantized into 3 short integers $(\mathbf{k})$ using equation Equation 11.1:

$$\mathbf{k} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \lfloor x/8 \rfloor \\ \lfloor y/8 \rfloor \\ \left\lfloor \frac{\log(-z/n)}{\log(1+2\cdot\tan(\phi/2)/S_y)} \right\rfloor \end{pmatrix} \tag{11.1}$$

n is the near plane position, S_y is the number of clusters in height, ϕ is the field-of-view, (x, y) is view-sample screen coordinate, and z is view space depth computed using Equation 11.2, where f is the far plane position and $d \in [-1, 1]$ is a view-sample depth:

$$z = \frac{2 \cdot n \cdot f}{d \cdot (f - n) - f - n} \tag{11.2}$$

The view frustum is divided exponentially in depth creating voxels, Figure 11.2. Samples within the same voxel have the same $\mathbf{k}$ vector and morton code. The algorithm creates 8x8 screen tiles. A tile is further divided into clusters according to samples depths.

Explicit bounds are computed around every view-sample cluster, Figure 11.3. Bounding boxes increase traversal performance, but require additional memory.

A morton code is constructed using a $\mathbf{k}$ vector by interleaving bits, Figure 11.4.

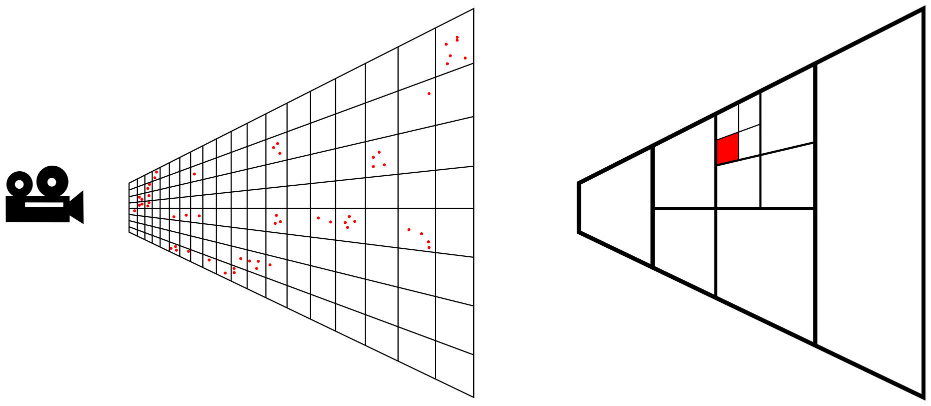


Figure 11.2: The image shows view-samples (red) and view frustum voxels, the left side. The branching factor is 4. View-samples located within the same voxel have the same morton code prefix. The algorithm constructs the tree from the bottom to the top and computes explicit bounding box around view-samples, Figure 11.3.

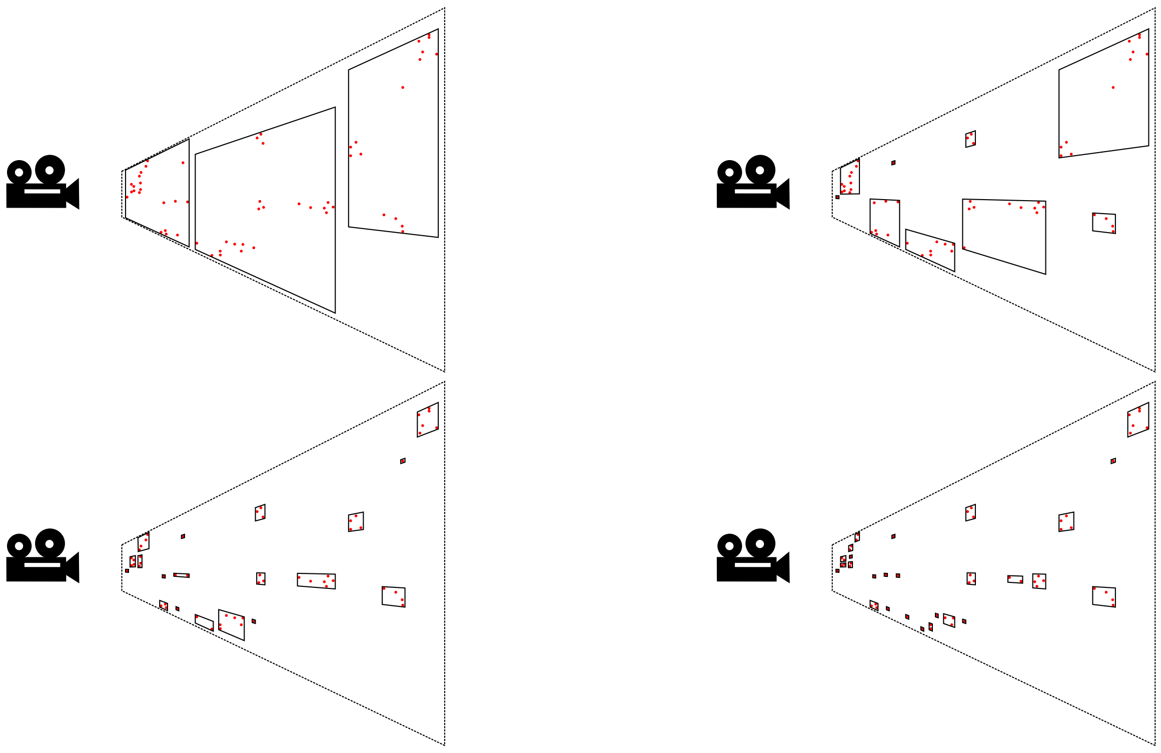


Figure 11.3: The image shows view-sample cluster hierarchy with four levels. Each level shows explicit bounding boxes.

$$\mathbf{k} = \begin{bmatrix} & & x'_6 & x'_5 & x'_4 & x'_3 & x'_2 & x'_1 & x'_0 \\ & & & y'_5 & y'_4 & y'_3 & y'_2 & y'_1 & y'_0 \\ z'_8 & z'_7 & z'_6 & z'_5 & z'_4 & z'_3 & z'_2 & z'_1 & z'_0 \end{bmatrix}$$

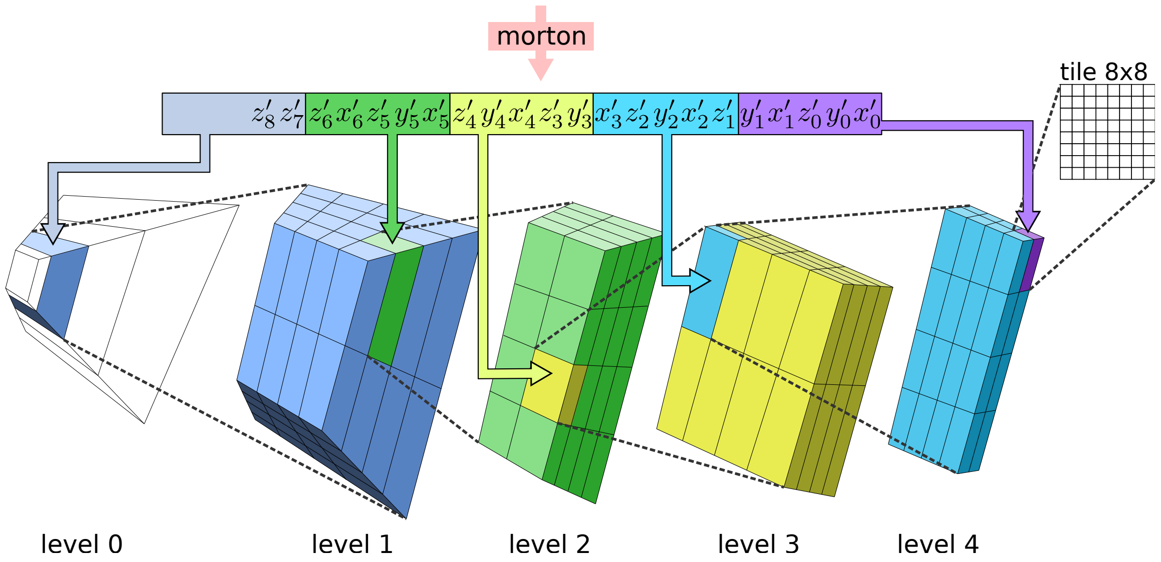

morton
$z'_8 z'_7$
$z'_6 x'_6 z'_5 y'_5 x'_5$
$z'_4 y'_4 x'_4 z'_3 y'_3$
$x'_3 z'_2 y'_2 x'_2 z'_1$
$y'_1 x'_1 z'_0 y'_0 x'_0$
tile 8x8
level 0
level 1
level 2
level 3
level 4

Figure 11.4: The image shows a example of a morton code (the resolution is $1024 \times 512 = 8 \cdot 2^7 \times 8 \cdot 2^6$). The morton code can be divided into 5 parts. Each part addresses a voxel in corresponding level. The view-sample cluster hierarchy, with the branching factor 32 (5 bits), has five levels. The bottom row shows voxels at each hierarchy level. The leaf level nodes represent 8x8 screen tiles. Every cluster is enclosed by tight axis aligned bounding box.

Traversal

When the hierarchy is built, scene's triangles are processed and per triangle shadow volumes (represented as 4 or 7 planes) are constructed. Three additional planes (each for one corner of a triangle) improve testing and remove large portion of false positives. The traversal step of the algorithm processes every shadow volume. If a cluster or view-sample lies within a shadow volume, it is removed from hierarchy, Figure 11.5.

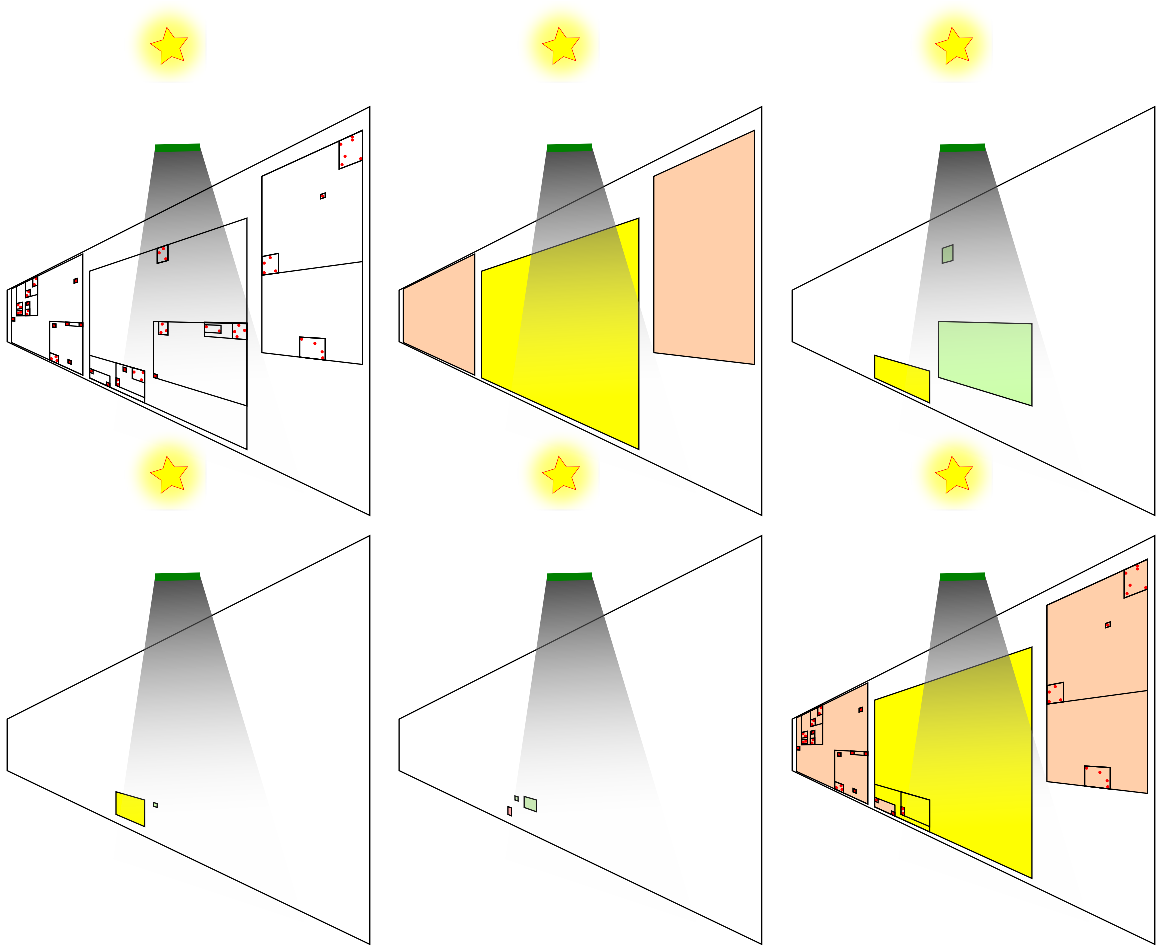


Figure 11.5: The image shows the traversal step of the algorithm. The top left part shows the view-sample cluster hierarchy and one triangle shadow volume. The hierarchy is traversed and trivially accepted clusters (green) are removed. Trivially accepted clusters lie completely inside the shadow volume. Trivially rejected clusters (orange) are not processed deeper. The traversal process descends deeper for intersecting clusters (yellow). The bottom right part shows the resulting hierarchy.

11.1.2 RSSV Algorithm

This section proposes new algorithm. The goal of the algorithm is to reduce the sensitivity to the number of triangles as well as to the screen resolution, Figure 11.6.

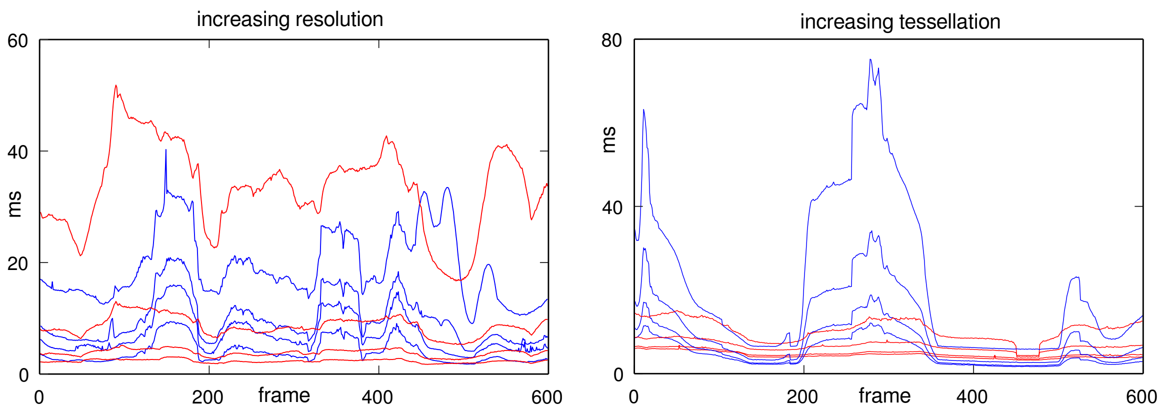


Figure 11.6: The image shows two flythroughs. The x axis represents animation frames. The y axis represents frame time. Red lines represents stencil silhouette method. Blue lines represents CPTSV method. Each line represents one resolution or one tessellation level. As can be seen, CPTSV method is less sensitive to increasing resolution and silhouette method is less sensitive to increasing tessellation.

CPTSV method hierarchically rasterizes simple per triangle shadow volumes. The rasterization is based on the collision test of two types of convex hulls: bounding volumes around view-sample clusters and triangle shadow volumes. The test is simple, because both types of hulls have simple geometry. If a cluster lies inside a shadow volume, it is shadowed. If a cluster does not lie inside any shadow volume, it is lit. If a cluster intersects a shadow volume, cluster's children are tested against the shadow volume.

The main idea of RSSV method is to hierarchically rasterize silhouette shadow volumes using GPGPU. The problem with that is the complex silhouette shadow volume geometry. Generally, silhouette shadow volume geometry is not a convex hull. A view-sample cluster cannot be easily tested.

Instead, RSSV algorithm tests changes to a stencil value using **sample ray** concept, Figure 11.7.

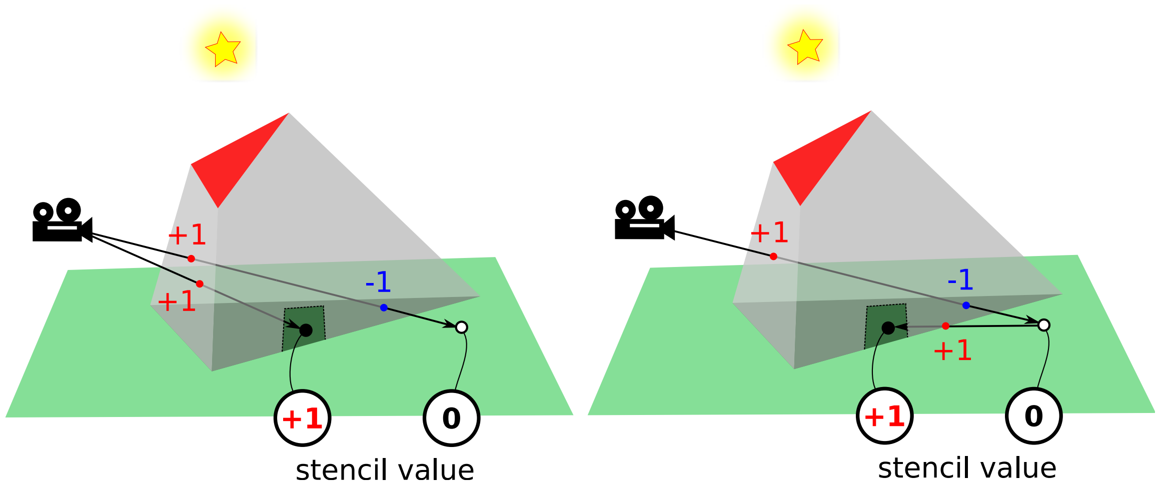


Figure 11.7: The left side of the image shows standard Z-pass shadow volume algorithm. Rays from the camera to the white and black view-samples compute stencil values. The right side shows another way. The stencil value of the black sample is computed using the stencil values of the white sample. The ray from the white sample crosses a front facing shadow volume side. This **sample ray** connecting two view samples computes the difference between stencil values. A sample ray is an oriented line segment connecting two view-samples.

In fact, the stencil value of just one single view-sample has to be computed. The rest of view-sample stencil values can be derived using stencil value differences, Figure 11.8. The concept is related to the Jordan theorem [49] extended to three dimensions.

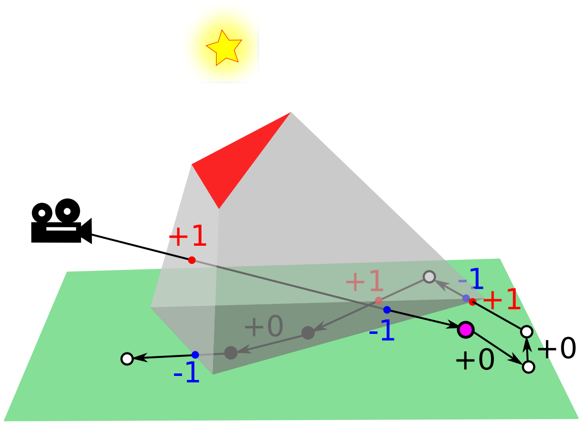


Figure 11.8: The image shows the idea of stencil value differences. Z-pass algorithm computes the stencil value of only one view-sample (magenta). Stencil values of remaining view-samples can be computed using stencil value differences. Sample rays connect view-samples forming a string. If a sample rays crosses a shadow volume geometry, the stencil value of the ray is modified. Since all view-samples form the string, stencil values of view-samples can be computed using parallel prefix sum of rays' stencil values. These sample rays give the RSSV method its name.

All view-samples can be connected using sample rays forming one long view-sample string. Once the view-sample string is constructed, all silhouette shadow volume sides and near caps are tested against it. The process computes stencil values of sample rays. After that, the algorithm computes parallel prefix sum of sample rays' stencil values.

In order to accelerate the algorithm, sample rays can be clustered into tiles. One of the most obvious way of connecting view-samples is to use z-curve [77] in the screen space. The problem view the z-curve is with its length. The better space filling curve is Hilbert curve [42], Figure 11.9. These curves support clustering of sample rays.

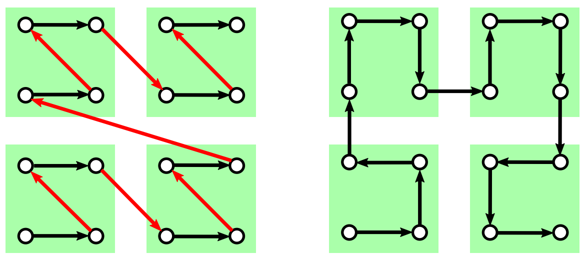


Figure 11.9: The image shows z-curve (left) and Hilbert curve. White points represent view-samples. Arrows represent sample rays. Red arrows are longer than necessary which increase the probability of intersection with a shadow volume. The right side shows Hilbert curve that minimizes lengths of sample rays. Green squares represents clusters of sample rays. If a cluster of sample rays does not intersect shadow volume polygons, its children do not have to be tested.

2D Hilbert curve provides the shortest path through all view-samples in the screen space. However, view-samples depths are not taken into account. Sample rays in 3D could be much longer due to the depth discontinuities. Also, if the algorithm computes stencil value of any sample ray incorrectly, the following view-samples in the string would have wrong stencil value. This can happen due to floating-point arithmetics. The better and more viable solution is to create sample ray tree, Figure 11.10. The sample ray tree, implemented in the testing application can be seen in Figure 11.11. The RSSV algorithm can be summed up as:

1. Render view-samples
2. Create sample ray tree
3. Extract silhouettes
4. Traverse the hierarchy using silhouette shadow volume sides
5. Traverse the hierarchy using near caps
6. Compute stencil mask

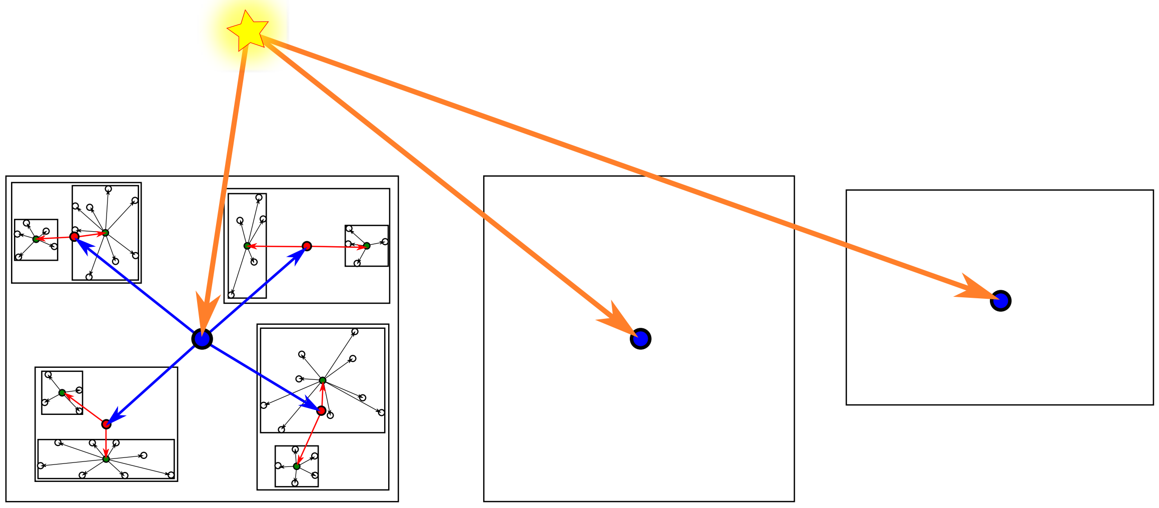


Figure 11.10: The image shows the concept of sample ray tree. View-samples are clustered in the similar fashion as in the CPTSV method. In addition, virtual view samples (blue, red and green) are created. Virtual samples are located at the center of clusters' bounding boxes. The stencil value of a view-sample can be computed by summing up differences along the way from the view-sample to the light source. The sample ray tree allows hierarchical silhouette shadow volume rasterization.

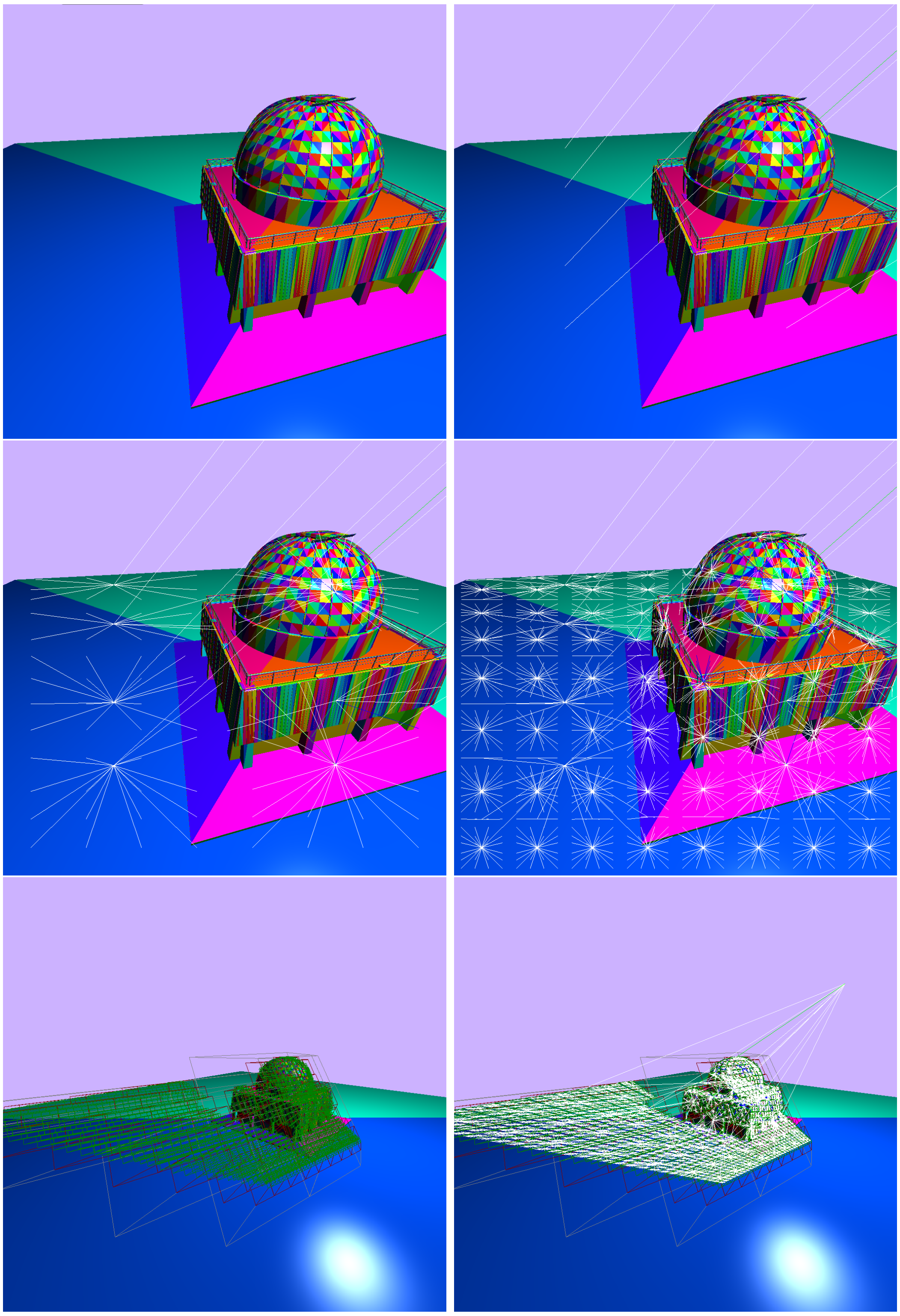


Figure 11.11: The image shows the sample ray tree implemented in the testing application. The top left image shows the scene from the camera point of view. The top right image shows sample rays from the light source to the zeroth level of the hierarchy. The next images show sample rays in lower levels of the hierarchy. The bottom left image shows view-sample clusters. The bottom right image shows view-sample clusters with sample ray tree.

Conclusion

The algorithm have been developed using OpenGL and is fully accelerated using GPU. The algorithm is already able to render shadows, Figure 11.12. Initial measurements show that the new method performs better than the CPTSV algorithm (around 30% higher fps). The traversal step is less costly than in the CPTSV algorithm. The new method requires complex approaches that mitigate problems with floating-point arithmetics, but they are quite involved and are out of scope of this text. Furthermore, the whole algorithm can be implemented using just one mega kernel using persistent threads. Currently, the implementation uses four mega kernels which are not yet merged. Further development is needed in order to fix few remaining robustness issues and to fully measure attributes of the method.

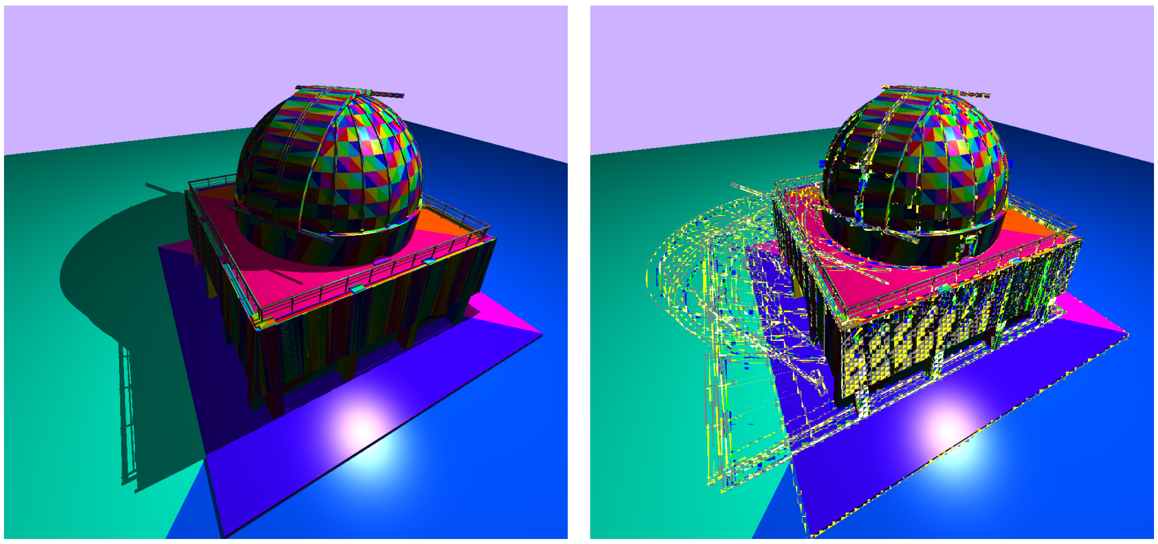


Figure 11.12: The left image shows shadows rendered using the new algorithm. The right side shows stencil values of sample rays at the leaf level. Rays with non-zero stencil values are visualized. As can be seen, sample rays that intersect silhouette shadow volumes sides are highlighted.

Chapter 12

Conclusion

While shadow algorithms have a long history and researchers have focused on many different approaches, this field is still not fully explored. The larger the field gets, the more concepts arise. Current methods and algorithms stimulate new ideas for further research. Even the smallest subsection of the field is large and contains many interesting scientific publications.

This thesis explores state of the art methods for real time precise shadow computation. It describes shadow volume algorithms and their variants in detail with many illustrations. Other precise methods based on shadow mapping or hybrid approaches are also covered.

It also summarizes contributions to silhouette computation on different hardware platforms. Algorithms for robust silhouette computation are designed for different GPU pipeline stages - vertex shaders, geometry shaders, tessellation shaders and compute shaders.

Some presented algorithms are used today by Cadwork informatik AG company in their web applications and rendering software. The thesis also presents the method for silhouette extraction based on potential visibility sets.

The next sections describe state of the art algorithms that are focused on improvements of visual quality of shadow maps. Warping based methods and other approaches are covered. Improved warping scheme which reduces the amount of visual artifacts is presented.

Following sections briefly visit lighting and many light algorithms and present improvement in visual quality for limited range light sources.

The last section presents new ideas of precise shadow rendering that have not yet been published.

Bibliography

[1] 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: ACM. 2004. ISBN 3-905673-15-0. pp. 15–23. doi:10.1145/1058129.1058132.
Retrieved from: http://doi.acm.org/10.1145/1058129.1058132

[2] Airey, J.; Rohlf, J.; Brooks, F., Jr: Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments". *ACM SIGGRAPH Computer Graphics*. vol. 24. 03 1990: pp. 41–50. doi:10.1145/91385.91416.

[3] 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. doi:10.1145/988834.988877.
Retrieved from: http://doi.acm.org/10.1145/988834.988877

[4] Alex Vlachos, C. D.: *Computing Optimized Shadow Volumes*. New York, NY, USA: Charles River Media. 2003. ISBN 9781584502333. 367–371 pp.

[5] Archer, J.; Leach, G.; Knowles, P.; et al.: Hybrid Lighting for faster rendering of scenes with many lights. *The Visual Computer*. vol. 34. 05 2018. doi:10.1007/s00371-018-1535-5.

[6] Batagelo, H.: Real-time shadow generation using BSP trees and stencil buffers. 02 1999. ISBN 0-7695-0481-7. pp. 93–102. doi:10.1109/SIBGRA.1999.805714.

[7] Benichou, F.; Eiber, G.: Output sensitive extraction of silhouettes from polygonal geometry. In *Proceedings. Seventh Pacific Conference on Computer Graphics and Applications (Cat. No.PR00293)*. 1999. pp. 60–69.

[8] Bergeron, P.: A General Version of Crow's Shadow Volumes. *IEEE Computer Graphics and Applications 6*. vol. 6, no. 9. 1986: pp. 17–28. ISSN 0272-1716.

[9] Bilodeau, B.; Songy: Real time shadows. *Creativity 1999*. 05 1999.

[10] Bittner, J.; Mattausch, O.; Silvennoinen, A.; et al.: Shadow Caster Culling for Efficient Shadow Mapping. In *Proceedings of the Symposium on Interactive 3D Graphics*. 01 2011. pp. 81–88. doi:10.1145/1944745.1944759.

[11] Brabec, S.; Annen, T.; Seidel, H.-P.: Practical Shadow Mapping. *J. Graph. Tools*. vol. 7, no. 4. December 2002: page 9–18. ISSN 1086-7651.

doi:10.1080/10867651.2002.10487567.
Retrieved from: https://doi.org/10.1080/10867651.2002.10487567

[12] Brabec, S.; Seidel, H.-P.: Shadow Volumes on Programmable Graphics Hardware. *Computer Graphics Forum (Eurographics)*. vol. 2003. 2003: pp. 433–440.

[13] Brennan, C.: Shadow Volume Extrusion using a Vertex Shader. *ShaderX: Vertex and Pixel Shader Tips and Tricks*. 01 2002: pp. 188–194.

[14] Carmack, J.; Kilgard, M.: John Carmack on Shadow Volumes. *NVIDIA Developer Zone*. 2000.
Retrieved from: http://web.archive.org/web/20090127020935/http://developer.nvidia.com/attach/6832

[15] Chan, E.; Durand, F.: An Efficient Hybrid Shadow Rendering Algorithm. 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. 185–195. doi:10.2312/EGWR/EGSR04/185-195.
Retrieved from: http://dx.doi.org/10.2312/EGWR/EGSR04/185-195

[16] Clark, J. H.: Hierarchical Geometric Models for Visible Surface Algorithms. *Commun. ACM*. vol. 19, no. 10. October 1976: page 547–554. ISSN 0001-0782. doi:10.1145/360349.360354.
Retrieved from: https://doi.org/10.1145/360349.360354

[17] Crow, F. C.: Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '77. New York, NY, USA: ACM. 1977. pp. 242–248. doi:10.1145/563858.563901.
Retrieved from: http://doi.acm.org/10.1145/563858.563901

[18] Dachsbacher, C.; Stamminger, M.: Reflective shadow maps. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*. ACM. 2005. pp. 203–231.

[19] Davidovič, T.; Georgiev, I.; Slusallek, P.: Progressive Lightcuts for GPU. In *ACM SIGGRAPH 2012 Talks*. SIGGRAPH '12. New York, NY, USA: ACM. 2012. ISBN 978-1-4503-1683-5. pp. 1:1–1:1. doi:10.1145/2343045.2343047.
Retrieved from: http://doi.acm.org/10.1145/2343045.2343047

[20] Deves, F.; Mora, F.; Aveneau, L.; et al.: Scalable Real-Time Shadows using Clustering and Metric Trees. In *Eurographics Symposium on Rendering - Experimental Ideas and Implementations*, edited by W. Jakob; T. Hachisuka. The Eurographics Association. 2018.

[21] Dietrich, S.: Using the stencil buffer. *Game Developers Conference*. march 1999.
Retrieved from: https://www.nvidia.pl/object/using_the_stencil_buffer.html

[22] Dimitrov, R.: Cascaded Shadow Maps. Technical report. NVIDIA Corporation. August 2007.
Retrieved from: http://developer.download.nvidia.com

[23] Donnelly, W.; Lauritzen, A.: Variance Shadow Maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. ACM. 2006. ISBN 1-59593-295-X. pp. 161–165.
Retrieved from: http://doi.acm.org/10.1145/1111411.1111440

[24] Eisemann, E.; Assarsson, U.; Schwarz, M.; et al.: Casting Shadows in Real Time. In *ACM SIGGRAPH ASIA 2009 Courses*. SIGGRAPH ASIA '09. New York, NY, USA: ACM. 2009. pp. 0–100. doi:10.1145/1665817.1722963.
Retrieved from: http://doi.acm.org/10.1145/1665817.1722963

[25] Eisemann, E.; Assarsson, U.; Schwarz, M.; et al.: *Real-time Shadows*. A K Peters/CRC Press. 2011. ISBN 9781568814384.

[26] Eisemann, E.; Assarsson, U.; Schwarz, M.; et al.: Efficient Real-time Shadows. In *ACM SIGGRAPH 2013 Courses*. SIGGRAPH '13. New York, NY, USA: ACM. 2013. ISBN 978-1-4503-2339-0. pp. 18:1–18:54. doi:10.1145/2504435.2504453.
Retrieved from: http://doi.acm.org/10.1145/2504435.2504453

[27] Engel, W.: Cascaded shadow maps. Charles River Media. 2006.

[28] Everitt, C. W.; Kilgard, M. J.: Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. *CoRR*. vol. cs.GR/0301002. 2003.
Retrieved from: http://arxiv.org/abs/cs/0301002

[29] Fan Zhang, A. B., Alexander Zaprjagaev: *Practical Cascaded Shadow Maps*. Charles River Media; 2009. 2009. ISBN 978-1-58450-598-3. 305–330 pp.

[30] Fernando, R.; Fernandez, S.; Bala, K.; et al.: Adaptive Shadow Maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM. 2001. ISBN 1-58113-374-X. pp. 387–390. doi:10.1145/383259.383302.
Retrieved from: http://doi.acm.org/10.1145/383259.383302

[31] Foley, J. D.: *Computer graphics: principles and practice*. Addison-Wesley Professional. 1996.

[32] Fuchs, H.; Goldfeather, J.; Hultquist, J. P.; et al.: Fast Spheres, Shadows, Textures, Transparencies, and Imgage Enhancements in Pixel-Planes. *SIGGRAPH Comput. Graph.*. vol. 19, no. 3. July 1985: page 111–120. ISSN 0097-8930. doi:10.1145/325165.325205.
Retrieved from: https://doi.org/10.1145/325165.325205

[33] Gerhards, J.; Mora, F.; Aveneau, L.; et al.: Partitioned Shadow Volumes. In *Computer Graphics Forum, Proceedings of Eurographics 2015*, vol. 34. 05 2015. pp. 0–100. doi:10.1111/cgf.12583.

[34] Goldberg, D.: What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Comput. Surv.*. vol. 23, no. 1. March 1991: page 5–48. ISSN 0360-0300. doi:10.1145/103162.103163.
Retrieved from: https://doi.org/10.1145/103162.103163

[35] Gooch, B.; pike J. Sloan, P.; Gooch, A.; et al.: Interactive Technical Illustration. 1999.

[36] Govindaraju, N.; Lloyd, B.; Yoon, S.-e.; et al.: Interactive Shadow Generation in Complex Environments. *ACM Transactions on Graphics*. vol. 22. 04 2003. doi:10.1145/882262.882299.

[37] Graphics, I. C.; Staff, A.: A Trip Down the Graphics Pipeline: The Homogeneous Perspective Transform. *IEEE Comput. Graph. Appl.*. vol. 13, no. 3. May 1993: page 75–80. ISSN 0272-1716. doi:10.1109/38.210494.
Retrieved from: https://doi.org/10.1109/38.210494

[38] Greene, N.; Kass, M.; Miller, G.: Hierarchical Z-Buffer Visibility. *ACM Computer Graphics*. vol. 27. 09 1993. doi:10.1145/166117.166147.

[39] Harada, T.: A 2.5D Culling for Forward+. In *SIGGRAPH Asia 2012 Technical Briefs*. SA '12. New York, NY, USA: Association for Computing Machinery. 2012. ISBN 9781450319157. doi:10.1145/2407746.2407764.
Retrieved from: https://doi.org/10.1145/2407746.2407764

[40] Hargreaves, S.: Deferred Shading. GDC. 2004.

[41] Heidmann, T.: Real Shadows, Real Time. *Iris Universe, No. 18*. 1991: pp. 23–31.

[42] Hilbert, D.: Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*. 1891: pp. 459 – 460.

[43] Hornus, S.; Hoberock, J.; Lefebvre, S.; et al.: ZP+: Correct Z-pass stencil shadows. *Symposium on Interactive 3D Graphics and Games*. 01 2005. doi:10.1145/1053427.1053459.

[44] Immel, D. S.; Cohen, M. F.; Greenberg, D. P.: A Radiosity Method for Non-Diffuse Environments. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery. 1986. ISBN 0897911962. page 133–142. doi:10.1145/15922.15901.
Retrieved from: https://doi.org/10.1145/15922.15901

[45] Isenberg, T.; Freudenberg, B.; Halper, N.; et al.: A Developer's Guide to Silhouette Algorithms for Polygonal Models. *Computer Graphics and Applications, IEEE*. vol. 23. 08 2003: pp. 28 – 37. doi:10.1109/MCG.2003.1210862.

[46] Jia, N.; Luo, D.; Zhang, Y.: Distorted Shadow Mapping. In *Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology*. VRST '13. New York, NY, USA: ACM. 2013. ISBN 978-1-4503-2379-6. pp. 209–214. doi:10.1145/2503713.2503746.
Retrieved from: http://doi.acm.org/10.1145/2503713.2503746

[47] Johnson, D.; Cohen, E.: Spatialized normal cone hierarchies. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*. 01 2001. pp. 129–134. doi:10.1145/364338.364380.

[48] Johnson, G.; Lee, J.; Burns, C.; et al.: The irregular Z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph.*. vol. 24. 01 2005: pp. 1462–1482.

[49] Jordan, C.: Cours d'analyse de l'Ecole Polytechnique. 1887: pp. 587–594.
Retrieved from: https://www.maths.ed.ac.uk/~v1ranick/jordan/jordan.pdf

[50] 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: Association for Computing Machinery. 1986. ISBN 0897911962. page 143–150. doi:10.1145/15922.15902.
Retrieved from: https://doi.org/10.1145/15922.15902

[51] Keller, A.: Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.. 1997. ISBN 0-89791-896-7. pp. 49–56. doi:10.1145/258734.258769.
Retrieved from: http://dx.doi.org/10.1145/258734.258769

[52] Kim, B.; Kim, K.; Turk, G.: A Shadow Volume Algorithm for Opaque and Transparent Non-Manifold Casters. 2008.

[53] 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. doi:10.24132/JWSCG.2019.27.1.2.
Retrieved from: https://www.fit.vut.cz/research/publication/11975

[54] Kolivand, H.; Sunar, M. S.: A Survey of Shadow Volume Algorithms in Computer Graphics. *IETE Tech Rev 2013*. vol. 30. 2013: pp. 38–46.

[55] Kwoon, H. Y.: *The Theory of Stencil Shadow Volumes*. Wordware Publishing, Inc.. 2003. ISBN 1-55622-902-X. 197–278 pp.

[56] Laine, S.: Split-plane shadow volumes. 01 2005. pp. 23–32. doi:10.1145/1071866.1071870.

[57] Laine, S.; Saransaari, H.; Kontkanen, J.; et al.: Incremental instant radiosity for real-time indirect illumination. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*. EGSR'07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. 2007. ISBN 978-3-905673-52-4. pp. 277–286. doi:10.2312/EGWR/EGSR07/277-286.
Retrieved from: http://dx.doi.org/10.2312/EGWR/EGSR07/277-286

[58] 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. doi:10.1145/1944745.1944761.
Retrieved from: http://doi.acm.org/10.1145/1944745.1944761

[59] Lefohn, A.; Sengupta, S.; Kniss, J. M.; et al.: Dynamic Adaptive Shadow Maps on Graphics Hardware. In *ACM SIGGRAPH 2005 Conference Abstracts and Applications*. August 2005.
Retrieved from: http://graphics.cs.ucdavis.edu/~lefohn/work/glift/

[60] Lengyel, E.: The mechanics of robust stencil shadows. 2002.
Retrieved from: http://www.gamasutra.com/view/feature/2942/the_mechanics_of_robust_stencil_.php

[61] Lengyel, E.: *Mathematics for 3D Game Programming and Computer Graphics, Second Edition*. Charles River Media, Inc.. 2003. ISBN 1584502770.

[62] Lloyd, B.; Govindaraju, N.; Quammen, C.; et al.: Logarithmic Perspective Shadow Maps. *ACM Trans. Graph.*. vol. 27. 10 2008. doi:10.1145/1409625.1409628.

[63] Lloyd, B.; Tuft, D.; Yoon, S.-e.; et al.: Warping and Partitioning for Low Error Shadow Maps. 01 2006. pp. 215–226. doi:10.2312/EGWR/EGSR06/215-226.

[64] Lloyd, B.; Wend, J.; Govindaraju, N.; et al.: CC shadow volumes. 01 2004. pp. 197–206. doi:10.1145/1186223.1186406.

[65] Lloyd, D. B.; Govindaraju, N. K.; Molnar, S. E.; et al.: Practical Logarithmic Rasterization for Low-error Shadow Maps. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. GH '07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. 2007. ISBN 978-1-59593-625-7. pp. 17–24.
Retrieved from: http://dl.acm.org/citation.cfm?id=1280094.1280097

[66] Martin, T.; Tan, T.-S.: Anti-aliasing and Continuity with Trapezoidal Shadow Maps. 01 2004. pp. 153–160. doi:10.2312/EGWR/EGSR04/153-160.

[67] Mattausch, O.; Scherzer, D.; Wimmer, M.; et al.: Tessellation-Independent Smooth Shadow Boundaries. *Computer Graphics Forum*. vol. 4, no. 31. June 2012: pp. 1465–1470. ISSN 1467-8659.
Retrieved from: https://www.cg.tuwien.ac.at/research/publications/2012/MATTAUSCH-2012-TIS/

[68] Mccool, M.: Shadow Volume Reconstruction from Depth Maps. *ACM Trans. Graph.*. vol. 19. 01 2000: pp. 1–26. doi:10.1145/343002.343006.

[69] McGuire, M.: Efficient Shadow Volume Rendering. In *GPU Gems*, edited by R. Fernando. Addison-Wesley. 2004. pp. 137–166.

[70] 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

[71] Michael Wimmer, D. S.: *Robust shadow mapping with light-space perspective shadow maps*. Charles River Media; 1 edition (January 12, 2006). 2006. ISBN 1584504250. 313–330 pp.

[72] Milet, T.; Kobrtek, J.; Zemčík, P.; et al.: Fast and Robust Tessellation-Based Silhouette Shadows. In *22nd International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, WSCG 2014, Poster Papers Proceedings - in co-operation with EUROGRAPHICS Association*. 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

[73] 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

[74] Milet, T.; Navrátil, J.; Herout, A.; et al.: Improved Computation of Attenuated Light with Application in Scenes with Multiple Light Sources. In *Proceedings of SCCG 2013*. Comenius University in Bratislava. 2013. ISBN 978-80-223-3377-1. pp. 155–160.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=10278

[75] Milet, T.; Tóth, M.; Pečiva, 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*. Eurographics Association. 2015. ISBN 978-3-905674-84-2. pp. 85–92. doi:10.2312/egve.20151314.
Retrieved from: https://www.fit.vut.cz/research/publication/10946

[76] Mora, F.; Gerhards, J.; Aveneau, L.; et al.: Deep Partitioned Shadow Volumes using Stackless and Hybrid Traversals. 06 2016.

[77] Morton, G.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. 01 1966.

[78] Navrátil, J.; Zemčík, P.; Juránek, R.; et al.: A Skewed Paraboloid Cut for Better Shadow Rendering. In *Proceedings of Computer Graphics International 2012*. Springer Verlag. 2012. ISBN 978-1-85899-283-9. page 4.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=9880

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

[80] Olsson, O.; Assarsson, U.: Tiled shading. *Journal of Graphics, GPU, and Game Tools 15, 4 (2011)*. 2011.

[81] Olsson, O.; Billeter, M.; Assarsson, U.: Clustered Deferred and Forward Shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. 2012. ISBN 978-3-905674-41-5. pp. 87–96. doi:10.2312/EGGH/HPG12/087-096.
Retrieved from: http://dx.doi.org/10.2312/EGGH/HPG12/087-096

[82] Olsson, O.; Sintorn, E.; Kämpe, V.; et al.: Efficient Virtual Shadow Maps for Many Lights. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '14. New York, NY, USA: Association for Computing Machinery. 2014. ISBN 9781450327176. page 87–96. doi:10.1145/2556700.2556701.
Retrieved from: https://doi.org/10.1145/2556700.2556701

[83] Olsson, O.; Sintorn, E.; Kämpe, V.; et al.: More Efficient Virtual Shadow Maps for Many Lights. 03 2014. pp. 87–96. doi:10.1145/2556700.2556701.

[84] O'Donnell, Y.; Chajdas, M. G.: Tiled Light Trees. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '17. New

York, NY, USA: Association for Computing Machinery. 2017. ISBN 9781450348867. doi:10.1145/3023368.3023376.
Retrieved from: https://doi.org/10.1145/3023368.3023376

[85] 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: http://www.fit.vutbr.cz/research/view_pub.php?id=10408

[86] Phong, B. T.: Illumination for computer generated pictures. *Commun. ACM*. vol. 18, no. 6. June 1975: pp. 311–317. ISSN 0001-0782. doi:10.1145/360825.360839.
Retrieved from: http://doi.acm.org/10.1145/360825.360839

[87] Ritschel, T.; Grosch, T.; Kim, M. H.; et al.: Imperfect shadow maps for efficient computation of indirect illumination. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*. ACM. 2008. pp. 1–8.

[88] Rosen, P.: Rectilinear Texture Warping for Fast Adaptive Shadow Mapping. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '12. New York, NY, USA: ACM. 2012. ISBN 978-1-4503-1194-6. pp. 151–158. doi:10.1145/2159616.2159641.
Retrieved from: http://doi.acm.org/10.1145/2159616.2159641

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

[90] Scherzer, D.; Jeschke, S.; Wimmer, M.: Pixel-correct Shadow Maps with Temporal Reprojection and Shadow Test Confidence. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR'07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. 2007. ISBN 978-3-905673-52-4. pp. 45–50. doi:10.2312/EGWR/EGSR07/045-050.
Retrieved from: http://dx.doi.org/10.2312/EGWR/EGSR07/045-050

[91] 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. doi:10.1111/j.1467-8659.2008.01267.x.
Retrieved from: http://dx.doi.org/10.1111/j.1467-8659.2008.01267.x

[92] Sintorn, E.; Kämpe, V.; Olsson, O.; et al.: Per-triangle Shadow Volumes using a view-sample cluster hierarchy. 03 2014. pp. 111–118. doi:10.1145/2556700.2556716.

[93] 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: pp. 153:1–153:10. ISSN 0730-0301. doi:10.1145/2070781.2024187.
Retrieved from: http://doi.acm.org/10.1145/2070781.2024187

[94] Society, I. C.: IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*. 2008: pp. 1–70. doi:10.1109/IEEESTD.2008.4610935.
Retrieved from: https://ieeexplore.ieee.org/servlet/opac?punumber=4610933

[95] Society, I. C.: IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*. 2019: pp. 1–84. doi:10.1109/IEEESTD.2019.8766229. Retrieved from: https://ieeexplore.ieee.org/servlet/opac?punumber=8766227

[96] Stamminger, M.; Drettakis, G.: Perspective Shadow Maps. *ACM Trans. Graph.*. vol. 21, no. 3. July 2002: pp. 557–562. ISSN 0730-0301. doi:10.1145/566654.566616. Retrieved from: http://doi.acm.org/10.1145/566654.566616

[97] Stich, 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. Addison Wesley Professional. 2007. ISBN 0-321-51526-9. pp. 239–256.

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

[99] Tokuyoshi, Y.; Harada, T.: Stochastic Light Culling for VPLs on GGX Microsurfaces. *Computer Graphics Forum*. vol. 36, no. 4. 2017: pp. 55–63. doi:10.1111/cgf.13224. https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13224. Retrieved from: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13224

[100] Tomas Akenine-Moller, N. H., Eric Haines: *Real-time Rendering. 3rd edition*. A K Peters/CRC Press. 2008. ISBN 1568814240.

[101] Trebilco, D.: Light Indexed Deferred Rendering. In *Shader X7: Advanced Rendering Techniques*, edited by W. Engel. 2009.

[102] Valient, M.: *Stable rendering of cascaded shadow maps*. Charles River Media; 2008. 2008. ISBN 978-1-58450-544-0. 231–238 pp.

[103] Vanek, J.; Navrátil, J.; Herout, A.; et al.: High-quality Shadows with Improved Paraboloid Mapping. In *Advances in Visual Computing*. Lecture Notes in Computer Science 6938. Faculty of Information Technology BUT. 2011. ISBN 978-3-642-24027-0. pp. 421–430. Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=9695

[104] Walter, B.; Fernandez, S.; Arbree, A.; et al.: Lightcuts: A Scalable Approach to Illumination. *ACM Trans. Graph.*. vol. 24, no. 3. July 2005: pp. 1098–1107. ISSN 0730-0301. doi:10.1145/1073204.1073318. Retrieved from: http://doi.acm.org/10.1145/1073204.1073318

[105] van Waveren, J.: Shadow Volume Construction. 2005. Retrieved from: http://fabiensanglard.net/doom3_documentation/37730-293752.pdf

[106] Williams, L.: Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*. vol. 12, no. 3. August 1978: pp. 270–274. ISSN 0097-8930. doi:10.1145/965139.807402. Retrieved from: http://doi.acm.org/10.1145/965139.807402

[107] Wimmer, M.; Scherzer, D.; Purgathofer, W.: Light Space Perspective Shadow Maps. In *Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering)*, edited by A. Keller; H. W. Jensen. Eurographics. Eurographics Association. June

2004. ISBN 3-905673-12-6. pp. 143–151.
Retrieved from:
http://www.cg.tuwien.ac.at/research/publications/2004/Wimmer-2004-LSPM/

[108] Woo, A.; Poulin, P.: *Shadow Algorithms Data Miner*. CRC Press. 2012. ISBN 978-1-4398-8023-4.

[109] 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: Association for Computing Machinery. 2015. ISBN 9781450333924. page 15–23. doi:10.1145/2699276.2699280. Retrieved from: https://doi.org/10.1145/2699276.2699280

[110] 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

[111] Zioma, R.: *Reverse extruded shadow volumes*. New York, NY, USA: Wirdware Publishing, Plano, TX. 2003. ISBN 1556229887. 587–593 pp.