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 SUMMARY
TEZE DISERTAČNÍ PRÁCE – AUTOREFERÁT

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 summary. 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.

Contents

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.

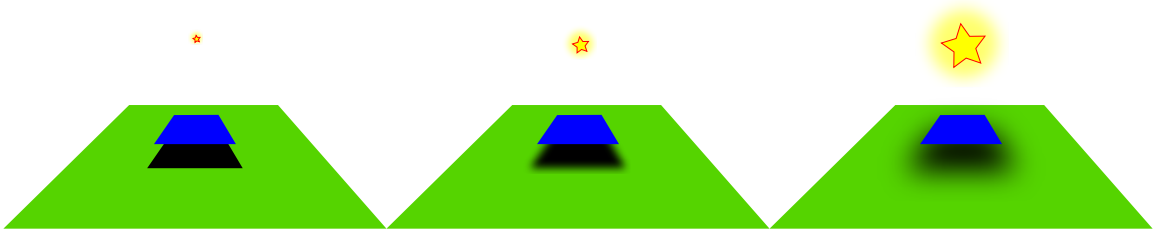

Figure 2.1: Point light sources create hard shadows, the left side. Large light sources (or area lights) create soft shadows, the right side. The shadow border smoothness depends on the light source size and the distance of the light and the shadow caster to the shadow receiver. Soft shadows are usually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

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

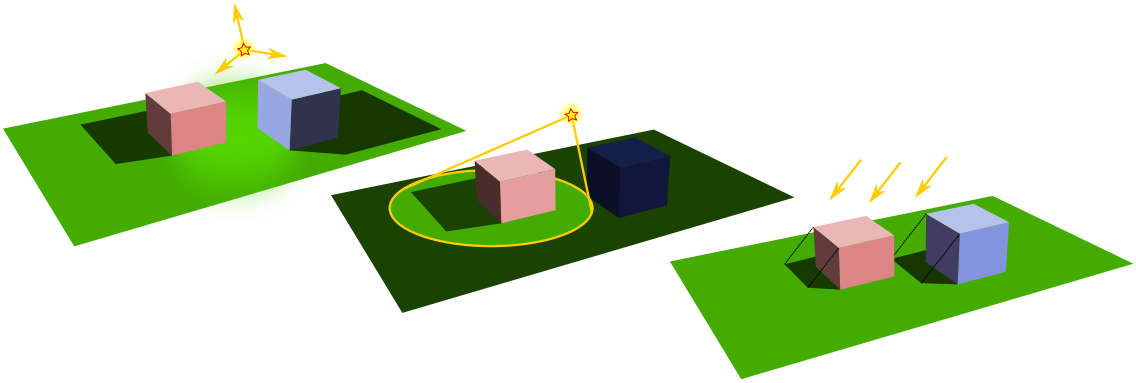

Figure 2.2: The left side of the image shows omnidirectional point light. The middle part shows spot light source. The right side shows directional light source. All light types are able to create hard shadows. Omnidirectional light sources are usually the most computationally expensive point light sources.

2.2 Shadows

Shadows are an important part of 3D rendering. They improve depth perception and visual quality, see Figure 2.3 and Figure 2.4.

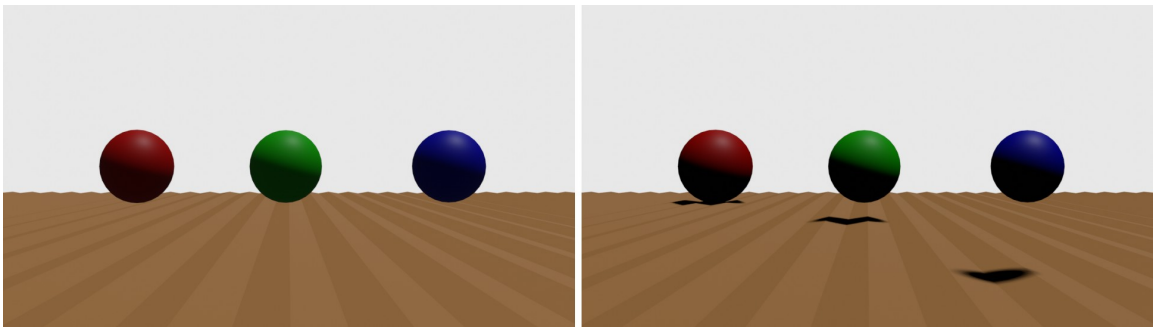

Figure 2.3: The left side of the figure shows a scene with three spheres without shadows. All spheres seem to be located at the same height above the ground. Also, spheres appear to have the same radius. The right side of the figure shows the same scene but with shadows. The shadows improved the depth 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.

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 [8]) can be seen in Figure 2.5.

Figure 2.4: 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.

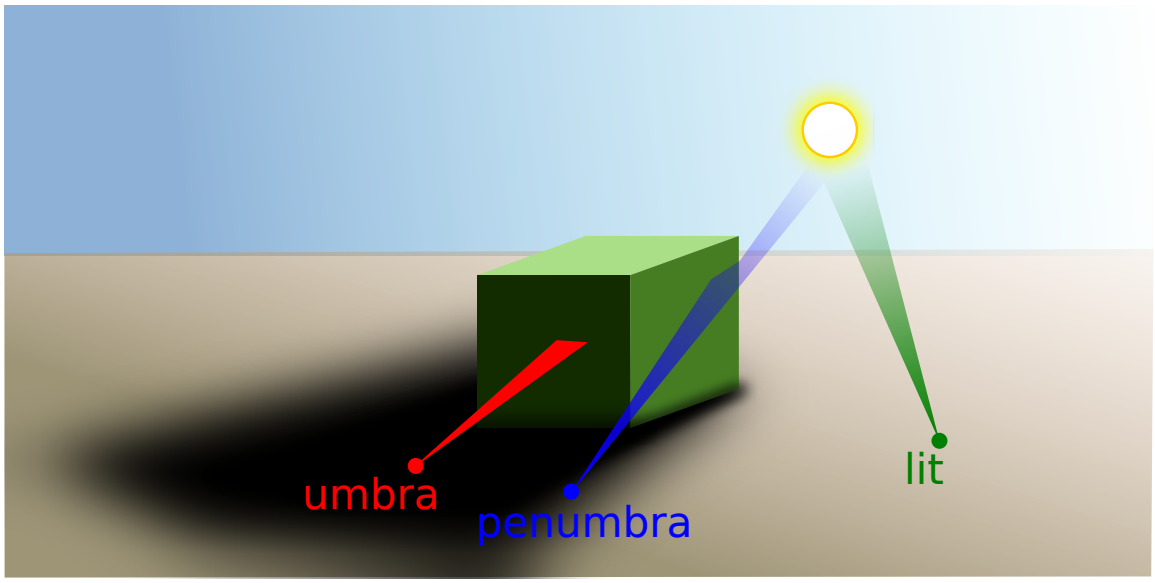

Figure 2.5: This Figure is inspired by Figure 1.4 from the book Real-Time Shadows by Eisemann [7]. The omnidirectional area light source $\mathfrak{L}$ is a set of points that emits a light. The surface point P is considered shadowed if there is an occluding geometry between P and $\mathfrak{L}$ and 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 2.6.

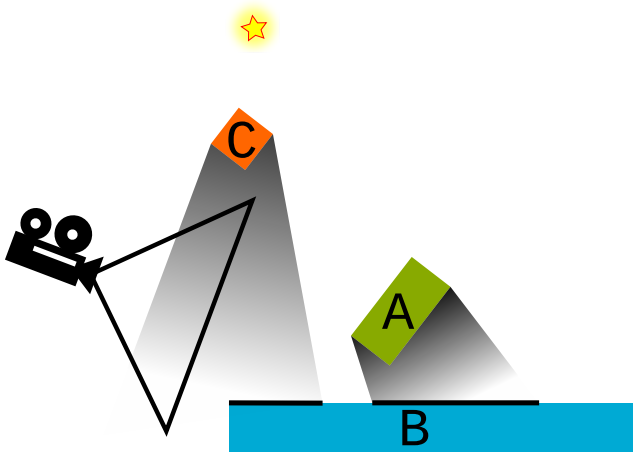

Figure 2.6: Objects A and C are shadow casters. Object B is shadow receiver. In reality, all object are shadow casters and receivers. However, It is useful to distinguish between objects that can receive shadows (shadows can be seen by camera on their surfaces) and shadow casters that influence observed shadows. Shadow casters are sometimes called blockers or occluders.

There are different kind of shadows. Hard shadows and soft shadows are just two categories.

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 2.7.

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 2.8.

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 2.9. The algorithm produces per sample precise shadows. Shadow volumes are supported by hardware with

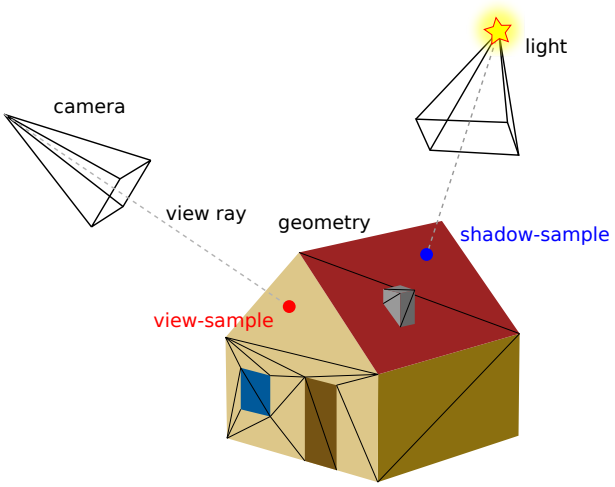

Figure 2.7: A scene geometry is composed of triangles. A view-sample (red point) is the closest intersection point of a view ray from camera with the scene geometry. A view-sample is associated with a screen space pixel. There is usually one view-sample per pixel. If multi-sampling is enabled, there 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).

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.

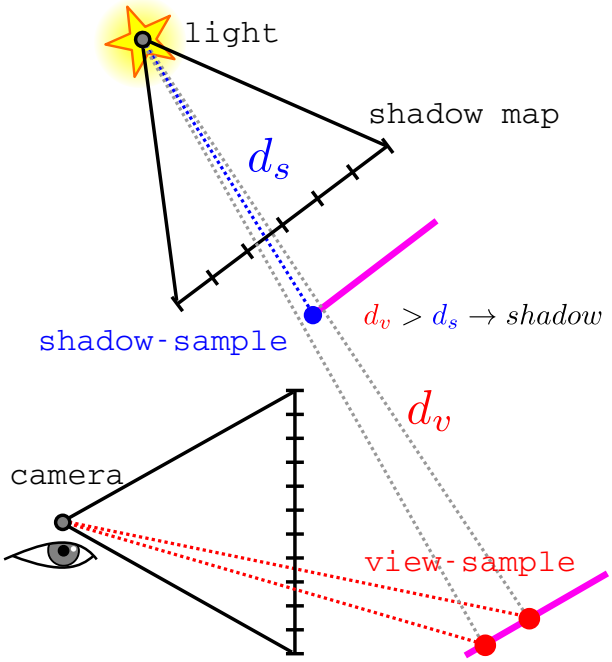

Figure 2.8: The image shows shadow mapping algorithm. The first render pass rasterizes a scene from light point of view. The rasterization creates shadow-samples that are stored in a 2D texture called shadow map. Shadow map stores depth values for each shadow-sample d_s. A shadow sample could be viewed 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.

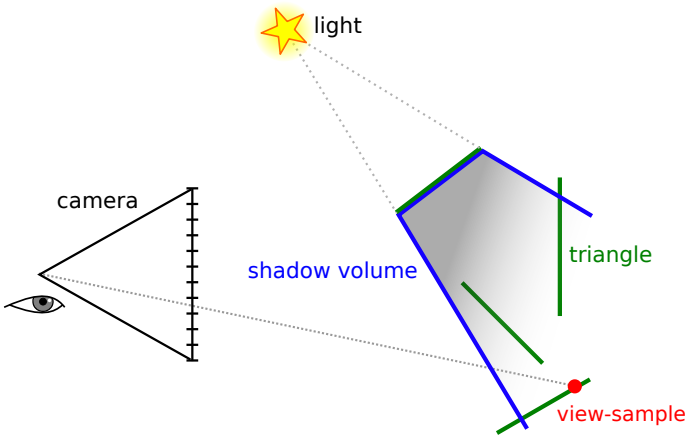

Figure 2.9: The image shows shadow volume algorithm. The algorithm creates shadow volume geometry that encloses shadows. Shadow volumes are created using a scene geometry. If a view-sample lies within any shadow volume, it is shadowed.

2.3.3 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 2.10.

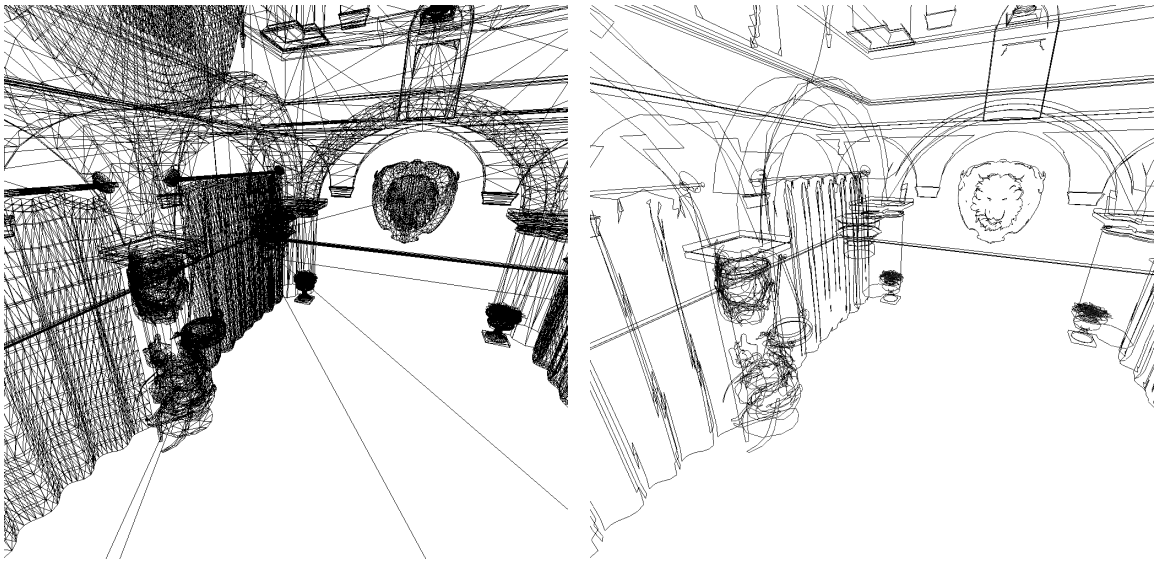

Figure 2.10: The left side shows edges in Sponza scene and the right side shows silhouette edges. The number of silhouette edges is much smaller which improves shadow volume performance.

Shadow volume rasterization is commonly the slowest part. Per-triangle shadow volumes contain unnecessary polygon which have to be rasterized, see Figure 2.11.

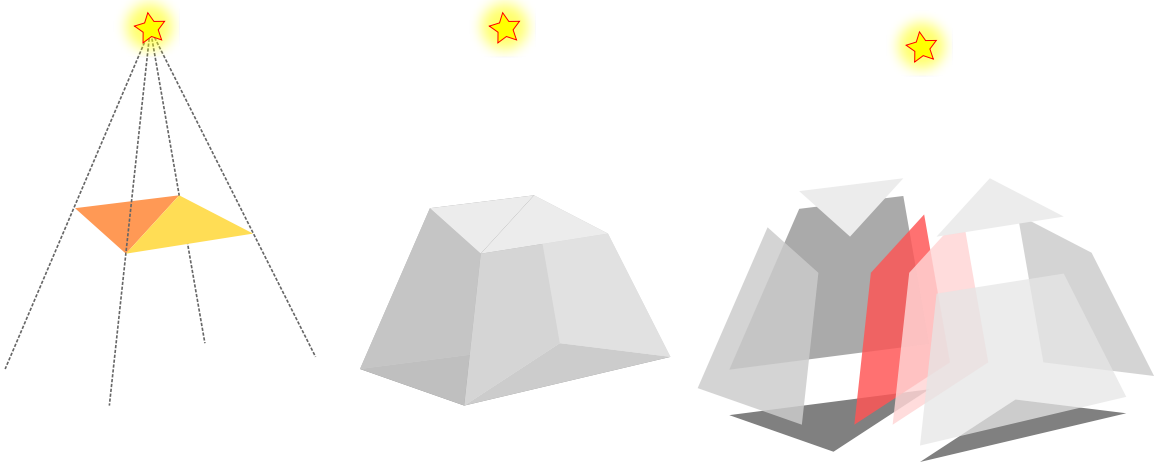

Figure 2.11: The figure shows two triangles and their corresponding shadow volumes. Red polygons are identical, but they have different orientation. A ray from camera has to intersect both or none. Therefore, the stencil value is decremented and incremented at the same time leading to no change. Silhouette based algorithms remove such polygons and reduce rasterization cost, see Figure 2.12.

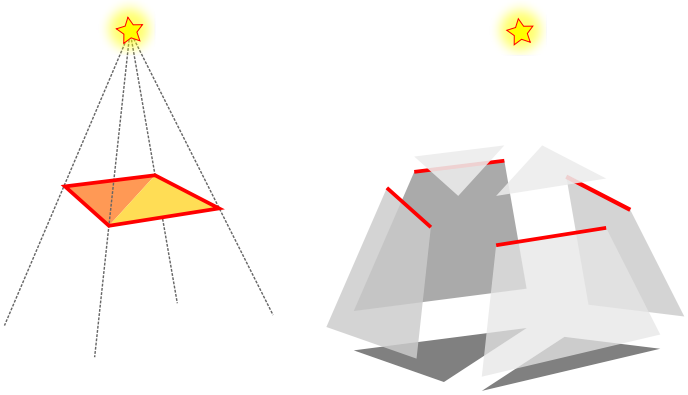

Figure 2.12: The image shows two triangles and the silhouette shadow volume. The red edges are silhouettes and only these silhouette edges are used during sides construction. By removing unnecessary edges, the shadow volume becomes simpler and the rasterization cost is reduced.

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 2.13.

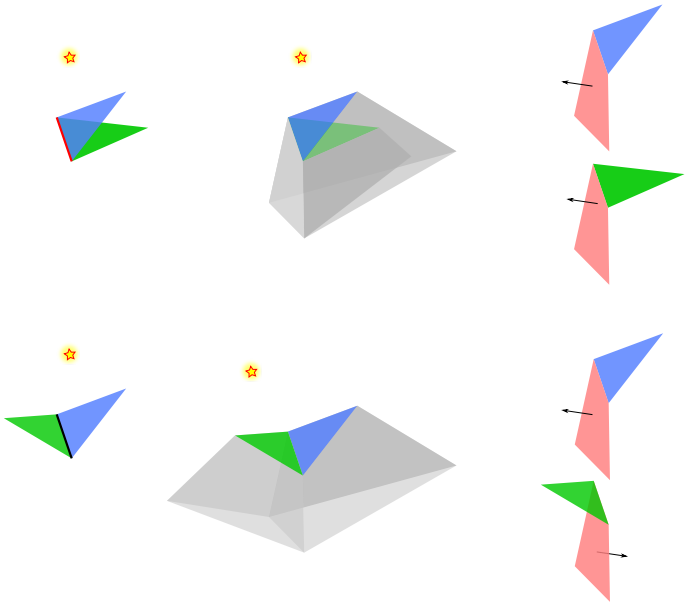

Figure 2.13: The top row shows two triangles that share an edge. The edge is a silhouette edge, because shadow volume sides have the same orientation. The bottom row shows a non-silhouette edge.

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 [5] and Brabec [4]. McGuire [15] implemented the whole algorithm in vertex shader. Van Waveren in 2005 [26] extracted silhouettes using SSE instructions on CPU for DOOM 3. Geometry shader based solution was proposed by [25]. Many of the solutions requires 2-manifold models. Bergeron [3] focused on manifold objects with boundary edge cases. Aldridge [2] further removed constraints and proposed algorithm for non-manifold oriented meshes. Finally, Kim [11] presented algorithm for any non-manifold object. Overview of silhouette algorithms is provided by Kolivand [13].

The most general algorithm by Kim Byungmoon in 2008 [11] is able to extract silhouettes from arbitrary triangle models. The principle of the algorithm is visualized in Figure 2.14 and in Figure 2.16.

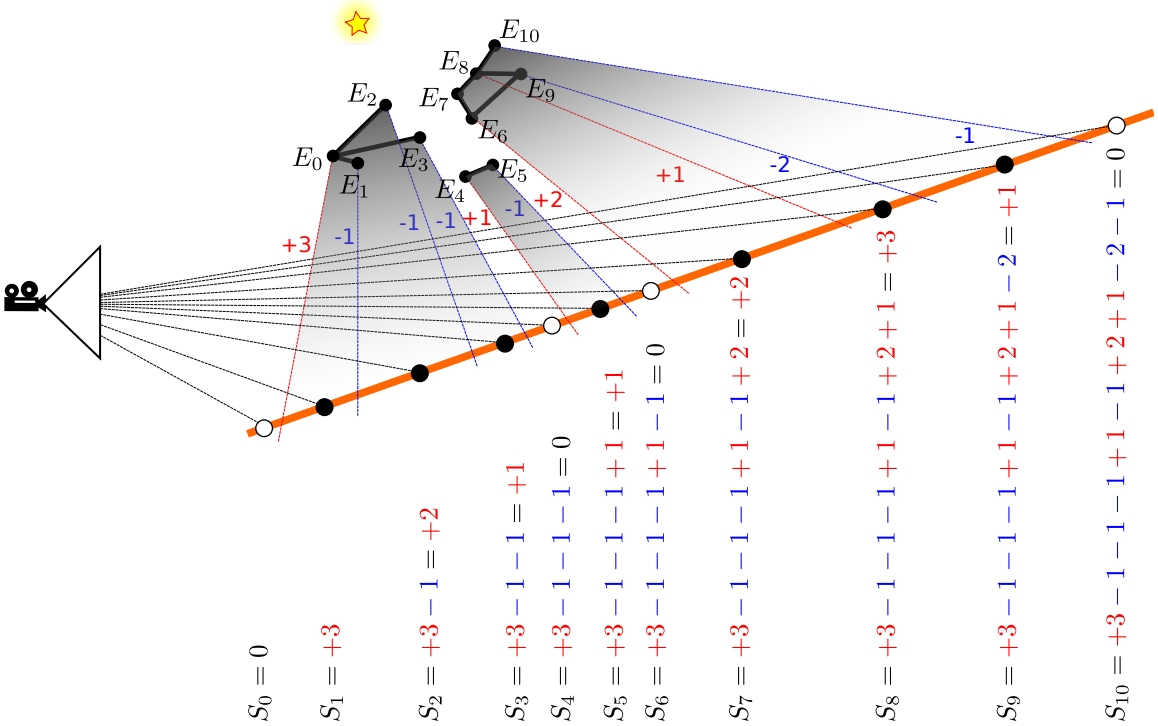

Figure 2.14: The image shows principle of Kim's algorithm [11]. Points E_0 - E_{10} represent edges and black line segments represent triangles. The only non-silhouette edge is edge E_7. Red and blue dashed lines represent shadow volume sides. During rasterization, shadow volume sides do not simply increment 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 2.15. 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.

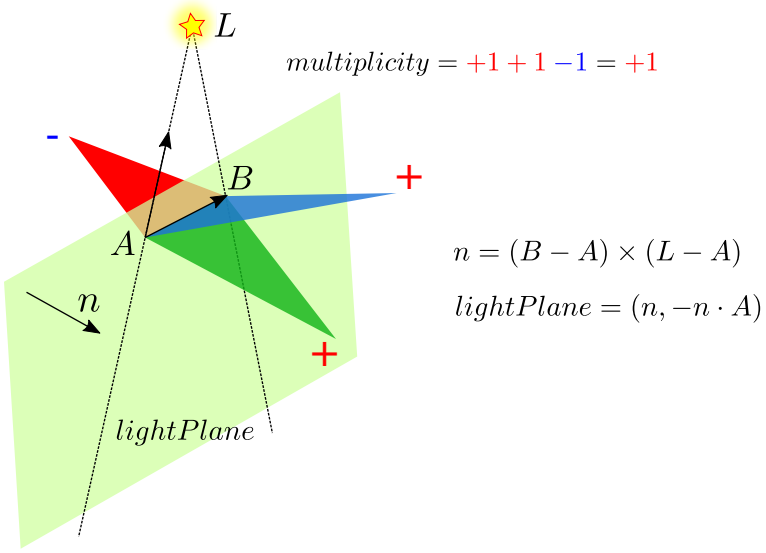

Figure 2.15: The image shows light-plane and multiplicity computation. The edge is shared by 3 triangles, two are in front of the light-plane and one is behind. The multiplicity value is +1.

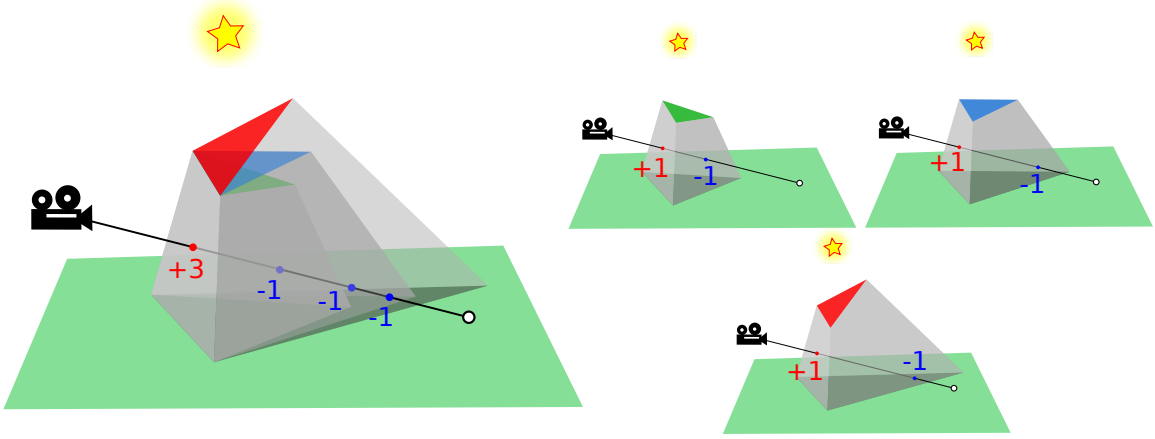

Figure 2.16: The image shows the idea of Kim's algorithm. Three triangles share an edge with multiplicity +3. A ray from camera colides with silhouette shadow volume and computes stencil value, left side. This can be imagined as three separate triangles with per triangle shadow volumes, right side.

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 2.17.

Eisemann et al. [7] 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. [21] proposed similar edge list structure. However, the solution does not require precomputed triangle

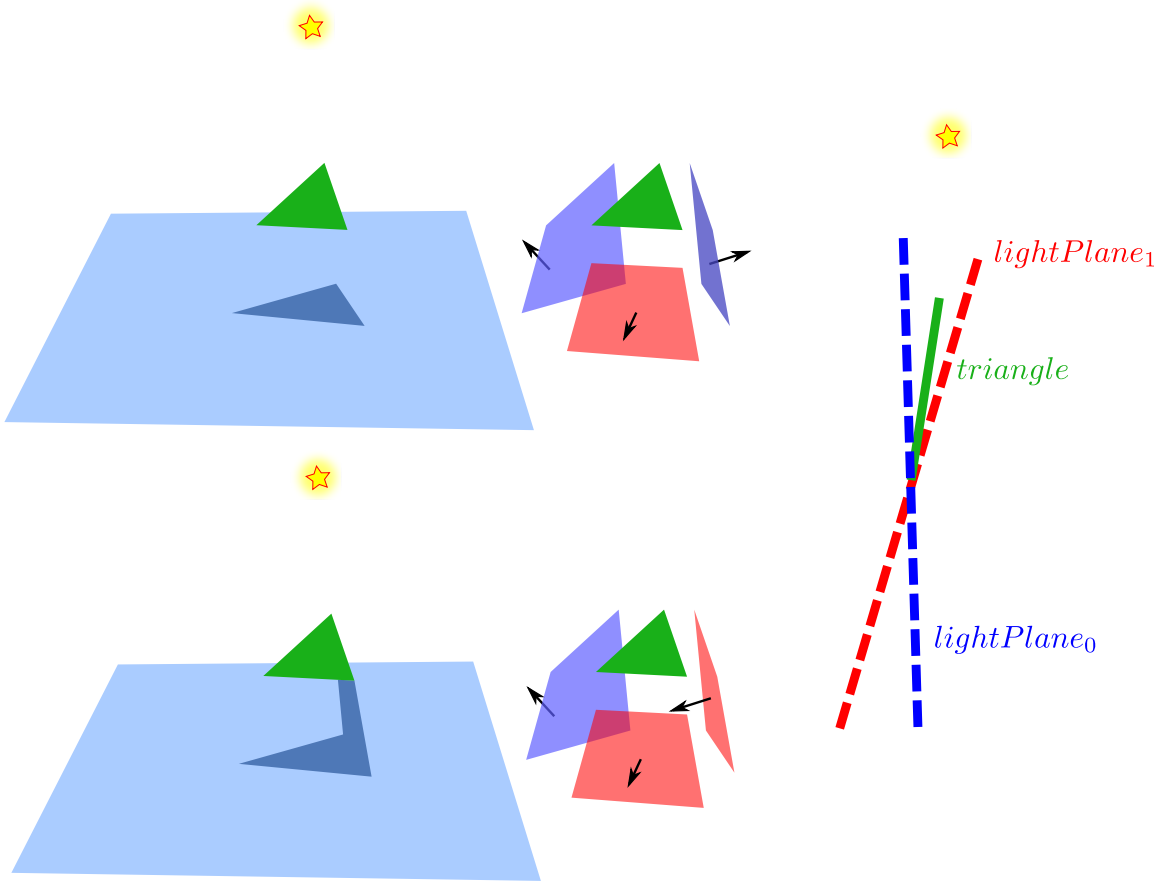

Figure 2.17: The left column shows a scene with a single triangle shadow caster. The middle column shows corresponding shadow volumes. Red polygons face the viewer while blue polygons back face the viewer. The bottom part shows visual artifacts caused by inconsistent multiplicity computation. The inconsistency 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. [16] 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. [16] 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 2.18.

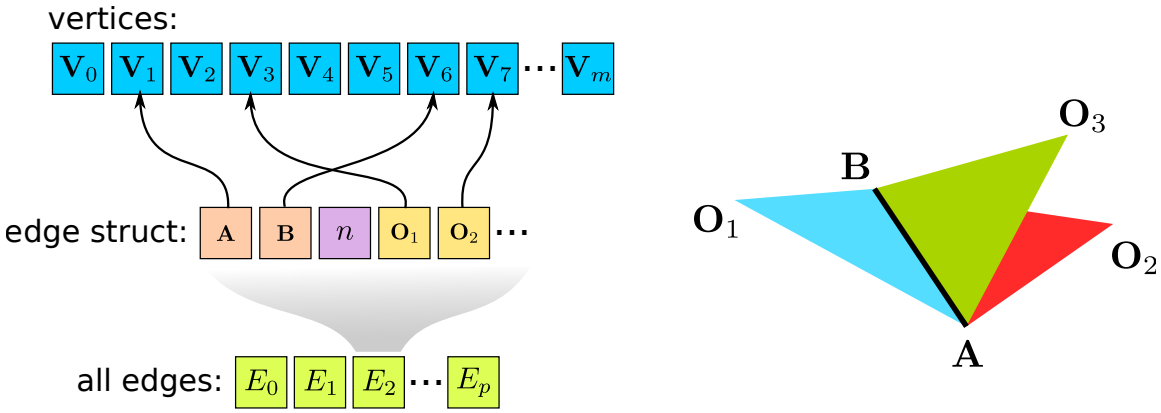

Figure 2.18: The right side shows a shared edge A, B with three adjacent triangles. The edge with opposite vertices is stored in edge structure. The left side shows model vertices V_i and the edge structure. The edge structure contains number of opposite vertices, indices to edge vertices and opposite 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 1.

Algorithm 1: 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 2. The function drawSide is visualized in Figure 2.19.

```
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 ≤ |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;
```

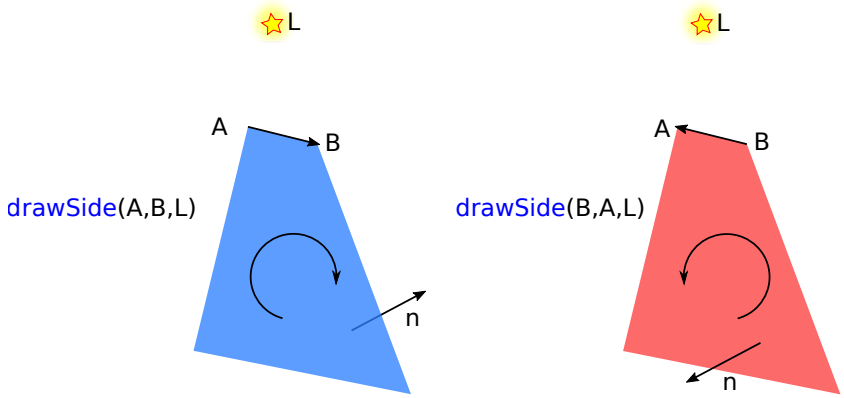

Figure 2.19: The function drawSide draws one side with correct orientation given by the order of input variables. The vertex winding order matches the order of edge vertices.

The algorithm computes multiplicity of reference edge (the largest oriented edge of adjacent triangle) and transforms it to correct edge multiplicity, see Algorithm 2.

Algorithm 2: 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 → R_max is the reference edge
2 n = (R_max − R_min) × (light − R_min);      // light plane normal
3 referenceMultiplicity = sign(n · (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 2.20.

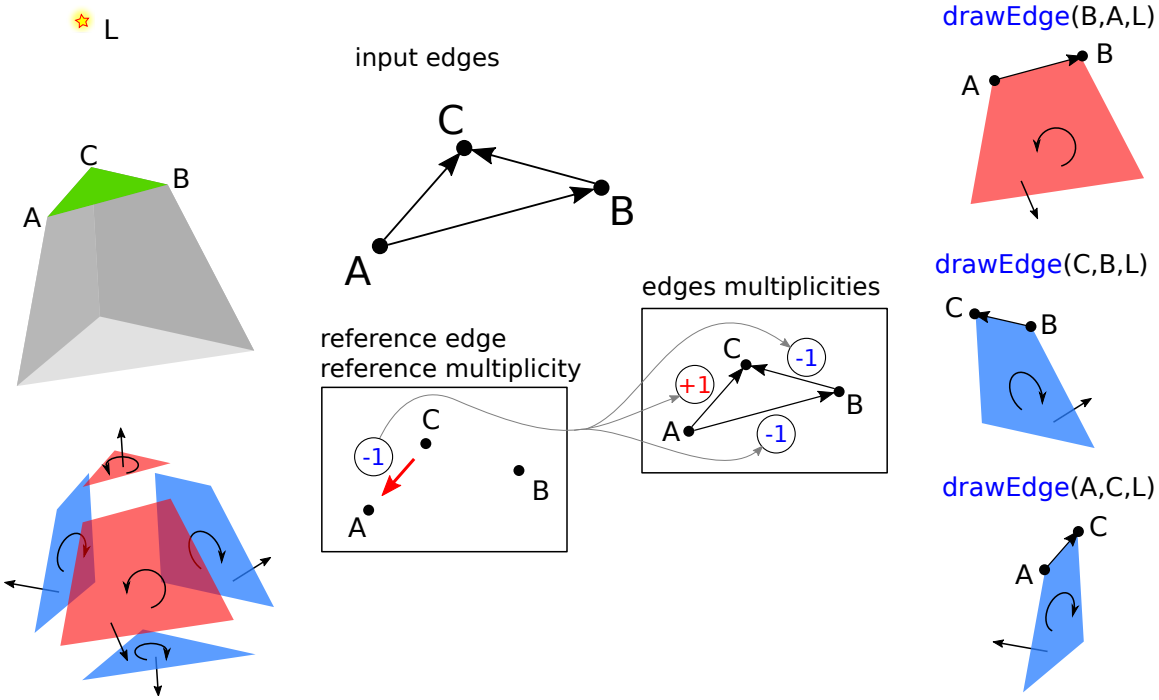

Figure 2.20: The image shows an example of computation of reference edge algorithm. The left side shows one triangle with shadow volume. The exploded shadow volume shows vertex winding orders and orientations. All normals point outside. Counter clockwise sides front face the viewer. The middle part shows 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.

Chapter 3

Proposed Methods

This chapter contains abstracts and algorithms of proposed and developed methods. Whole content of each method can be found in the full version of the thesis.

3.1 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 3.1), overview of the implementation and result of the optimizations on individual platforms. The full content of this section can be found in paper [21].

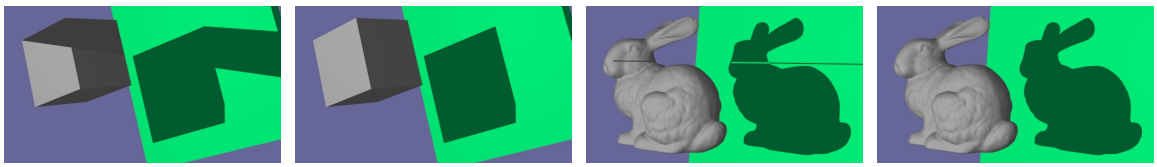

Figure 3.1: The Figure shows the difference between the original algorithm and the new robust algorithm. The right image of each couple shows result of new algorithm. The first couple of images shows very simple model, where artefacts are most visible. The second couple shows artefacts on more complex model, which could appear in real applications.

3.1.1 Algorithm Description and Robustness improvement

The algorithm, described in [11], generates the output shadow silhouette based on the triangular mesh representing the model and the position of light source.

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 3.2.

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 3.2.

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

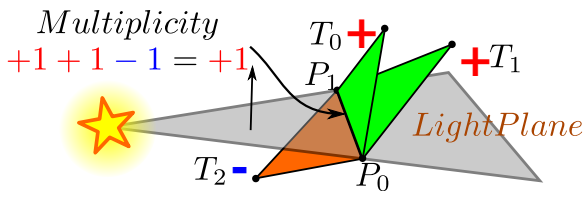

Figure 3.2: Multiplicity of Edge P_0-P_1 for the Opposite Vertices (OVs) T_0-T_2.

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 "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. 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 3.1).

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 3.2.

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

The LP itself is defined as:

$$\begin{aligned} \mathbf{V} &= (\mathbf{L}_x - \mathbf{A}_x\mathbf{L}_w, \mathbf{L}_y - \mathbf{A}_y\mathbf{L}_w, \mathbf{L}_z - \mathbf{A}_z\mathbf{L}_w) \\ \mathbf{N} &= \text{normalize}((\mathbf{A} - \mathbf{B}) \times \mathbf{V}) \\ \mathbf{LP} &= (\mathbf{N}_x, \mathbf{N}_y, \mathbf{N}_z, -\mathbf{N} \cdot \mathbf{A}) \end{aligned} \tag{3.1}$$

The multiplicity of the edge is:

$$m = \sum_{\mathbf{o} \in \mathfrak{O}} \text{sign}(\mathbf{LP} \cdot \mathbf{o}) \tag{3.2}$$

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.

3.2 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 3.3), providing better performance than current state-of-the art algorithms. The full content of this section can be found in paper [16].

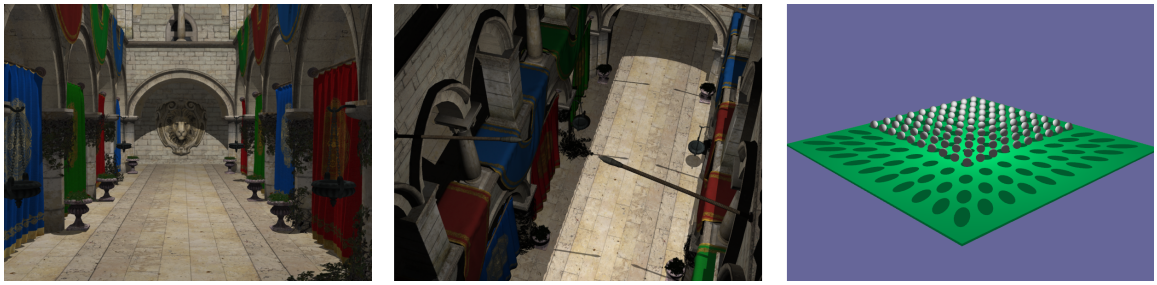

Figure 3.3: Test scenes - Crytek Sponza and Spheres

3.2.1 Method Description

The full version of 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 [11]. 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.

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 3.4.

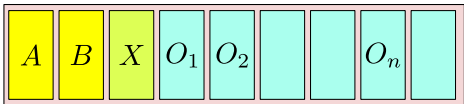

Figure 3.4: The image shows an input patch for tessellation control shader. It is composed of two vertices that describe an edge, one integer that contains number of opposite vertices and n opposite vertices. Because the patch size must be constant, some positions might not be used.

A 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 3.5.

Kim's algorithm [11], 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. [21], 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*.

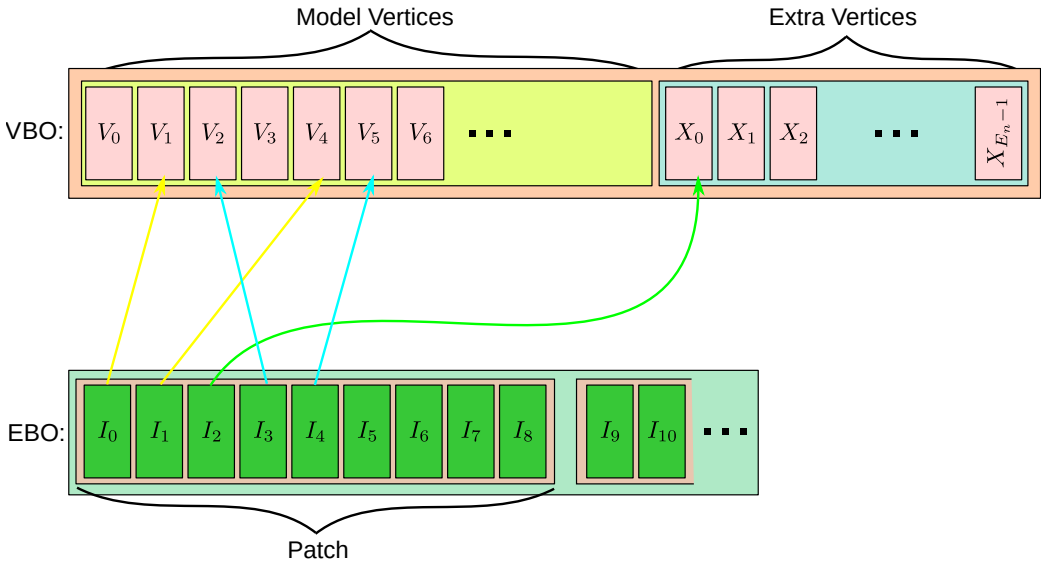

Figure 3.5: Combining per-edge patch data using Element Buffer Object. Vertex Buffer Object needs to be extended by E_n (number of edges) vertices, where X_i is number of adjacent triangles to edge E_i. Because input vertices are 4-component vectors and X_i is scalar value, only a single value per vector is used.

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 3.3 and Algorithm 3.

$$\begin{aligned} \mathbf{A} < \mathbf{B} &\Leftrightarrow \text{greater}(\mathbf{A}, \mathbf{B}) < 0 \\ \mathbf{A} = \mathbf{B} &\Leftrightarrow \text{greater}(\mathbf{A}, \mathbf{B}) = 0 \\ \mathbf{A} > \mathbf{B} &\Leftrightarrow \text{greater}(\mathbf{A}, \mathbf{B}) > 0 \end{aligned} \tag{3.3}$$

Algorithm 3: 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 $r = \mathbf{S} \cdot \mathbf{K}$

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 3. More options for such method are available, but evaluation per each triangle occurrence must be consistent in order to get correct results.

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 3.6, which demonstrates edge A-B having multiplicity of 3. The procedure of multiplicity calculation is described in the Algorithm 4 and in the Algorithm 5.

Algorithm 4: 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

```
m = 0;
for O_i ∈ 𝔒 do
    if A > O_i then
        m = m + compMultiplicity(O_i, A, B, L);
    else
        if B > O_i then
            m = m − compMultiplicity(A, O_i, B, L);
        else
            m = m + compMultiplicity(A, B, O_i, L);
```

Algorithm 5: compMultiplicity$(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{L})$ function used in Algorithm 4

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

```
X = C − A;
Y = (l_x − a_x l_w, l_y − a_y l_w, l_z − a_z l_w);
N = X × Y;
m = sign(N · (B − A));
```

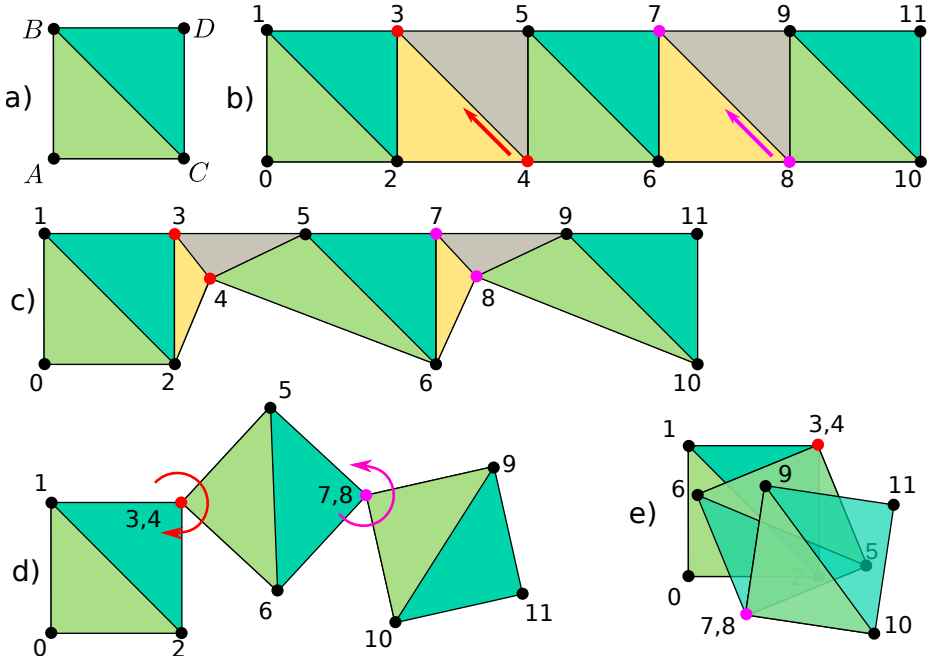

Figure 3.6: The image shows the transformation of a quad into three overlapping shadow volume sides. The transition from part a) to part b) is tessellation of quad with Multiplicity = 3. Only green and blue triangles will be drawn. Yellow and gray triangles will be degenerated. The transition from part 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 3.4.

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 6 and in the Equation 3.4.

$$\begin{aligned}
\mathbf{A} &= (a_x, a_y, a_z, 1)^T \\
\mathbf{B} &= (b_x, b_y, b_z, 1)^T \\
\mathbf{C} &= (a_x - l_x, a_y - l_y, a_z - l_z, 0)^T \\
\mathbf{D} &= (b_x - l_x, b_y - l_y, b_z - l_z, 0)^T
\end{aligned} \tag{3.4}$$

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

3.3 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

Algorithm 6: 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 3.4.

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

$\mathbf{P}_0 = \mathbf{A}$;
$\mathbf{P}_1 = \mathbf{B}$;
$\mathbf{P}_2 = \mathbf{C}$;
$\mathbf{P}_3 = \mathbf{D}$;
$a = \text{round}(x \cdot m)$;
$b = \text{round}(y)$;
$id = a \cdot 2 + b$;
$t = (id \mod 2) \hat{} (\lfloor id/4 \rfloor \mod 2)$;
$l = \lfloor (id + 2)/4 \rfloor \mod 2$;
$n = t + l \cdot 2$;
$\mathbf{V} = \mathbf{P}_n$;

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 [19].

Figure 3.7: Three testing scenes - Sponza, Conference Room and Sibenik rendered using WebGL on NVIDIA Shield.

3.3.1 Algorithm

The algorithm is based on the robust silhouette algorithm developed by [21] and enhanced algorithm by [16]. 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 [14] 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. [14] designed solution just for 2-manifold meshes so this new method merges McGuire's approach with the algorithm presented in [21]. Furthermore, following sections present a number of data-related optimizations for memory footprint reduction.

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 [6]. 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.

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 3.11).

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 3.3.1 and Section 7).

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.

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 3.8. 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.

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 3.9).

This section closely describes shader for sides and format of sides-data. Every edge of the scene can extrude n sides of SV (see [11]), where n is the number of attached triangles to this edge, see Figure 3.8. 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 [16].

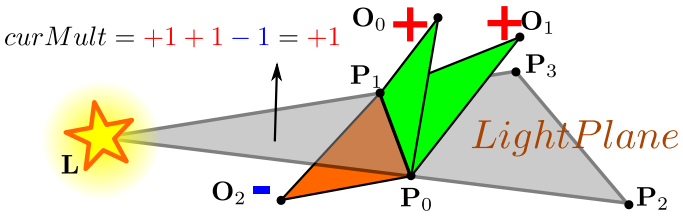

Figure 3.8: Edge $\mathbf{P}_0 \rightarrow \mathbf{P}_1$ has $n = 3$ opposite vertices: $\mathfrak{O} = \{\mathbf{O}_0, \mathbf{O}_1, \mathbf{O}_2\}$. Its maximum multiplicity is $n = 3$. Current multiplicity of this edge, given by light source and transformation, is $+1$. One quad $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ 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 3.8):

$$toInf(\mathbf{P}, \mathbf{L}) = (p_x l_w - l_x, p_y l_w - l_y, p_z l_w - l_z, 0) \tag{3.5}$$

$$\begin{aligned} \mathbf{P}_0 &= (a_x, a_y, a_z, 1) \\ \mathbf{P}_1 &= (b_x, b_y, b_z, 1) \\ \mathbf{P}_2 &= toInf(\mathbf{P}_0, \mathbf{L}) \\ \mathbf{P}_3 &= toInf(\mathbf{P}_1, \mathbf{L}) \end{aligned} \tag{3.6}$$

In Equation 3.6 $\mathbf{P}_0 \rightarrow \mathbf{P}_1$ 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 3.7, Equation 3.8 and Equation 3.9:

$$id = \begin{cases} idCCW & \text{if } curMult > 0 \\ idCW & \text{otherwise} \end{cases} \tag{3.7}$$

$$idCW = \begin{cases} 6 - v & \text{if } v > 2 \\ v & \text{otherwise} \end{cases} \tag{3.8}$$

$$idCCW = \begin{cases} v - 2 & \text{if } v > 2 \\ 2 - v & \text{otherwise} \end{cases} \tag{3.9}$$

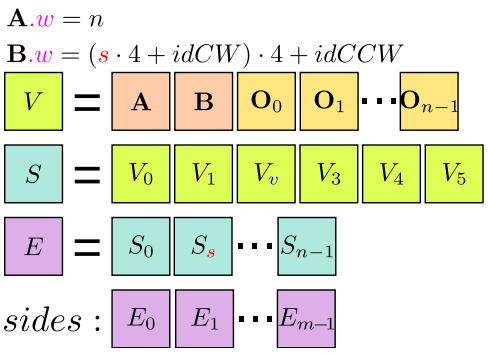

Figure 3.9: The image shows data structure for sides of SV. Every vertex (V) contains $n+2$ 4D vectors. **A** and **B** are vertices of an edge and $\mathbf{O}_j$ are opposite vertices. Forth components of **A** and **B** are used for special purposes - storing number of opposite vertices, index of side 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 3.9) is in Algorithm 7.

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 3.10).

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.

Algorithm 7: 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: $gl\_Position$

```
compute vertices P_i according to Equation 3.6;
curMult = computeMultiplicity(A, B, O, L);
if s < |curMult| then
    compute id according to Equation 3.7;
    gl_Position = M · P_id;
else
    gl_Position = (0, 0, 0, 0);
```

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 [16]. Compared to sides, there is only one opposite vertex.

First, VS finds reference edge using method described in [16]. Then it computes multiplicity using third vertex and reference edge. The multiplicity determines orientation of particular triangle.

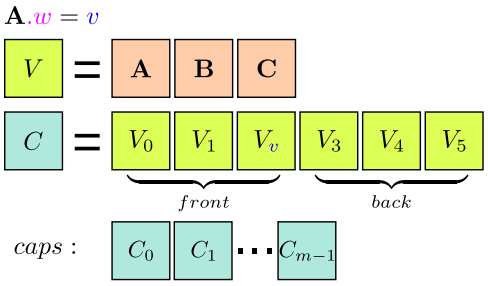

Figure 3.10: The image shows data structure for caps of SV. Every vertex (V) contains 3 4D vectors. $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are vertices of scene triangles. Forth component of $\mathbf{A}$ is used for special purposes - storing index of vertex v. Six vertices form couple of front and back 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 3.11. 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 [15]. 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 3.10:

$$\mathbf{B} = (b_x, b_y, b_z, b_w) = (a_x, a_y, a_{\mathbf{w}}, a_{\mathbf{w}}) \tag{3.10}$$

Pseudo code implemented in VS for caps-data (Figure 3.10) can be seen in Algorithm 8.

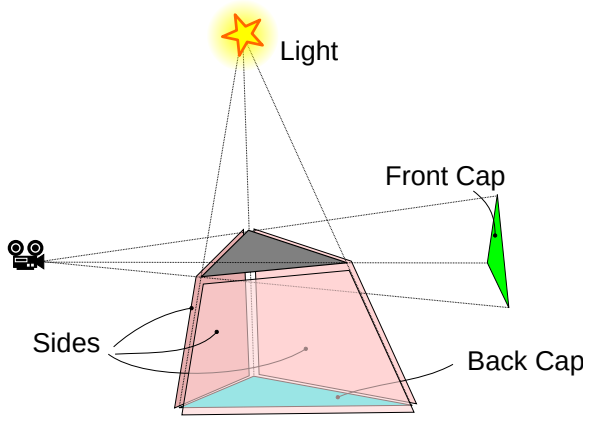

Figure 3.11: The image shows shadow volume that is constructed from a triangle. The front cap is shifted to infinity using homogeneous coordinates in order to prevent self-shadowing artifacts.

Algorithm 8: 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: $gl\_Position$

```
curMult = computeMultiplicity(P0, P1, P2, L);
if curMult = 0 then
    gl_Position = (0, 0, 0, 0);
    return;
if curMult < 0 then
    swap(P0, P1);
if v < 3 then
    V = M · Pv;
    gl_Position = (Vx, Vy, Vw, Vw);
else
    gl_Position = M · toInf(P5−v, L);
```

3.4 Silhouette Extraction for Shadow Volumes Using Potentially Visible Sets

The full content of this section can be found in paper [12]. 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 3.12. 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.

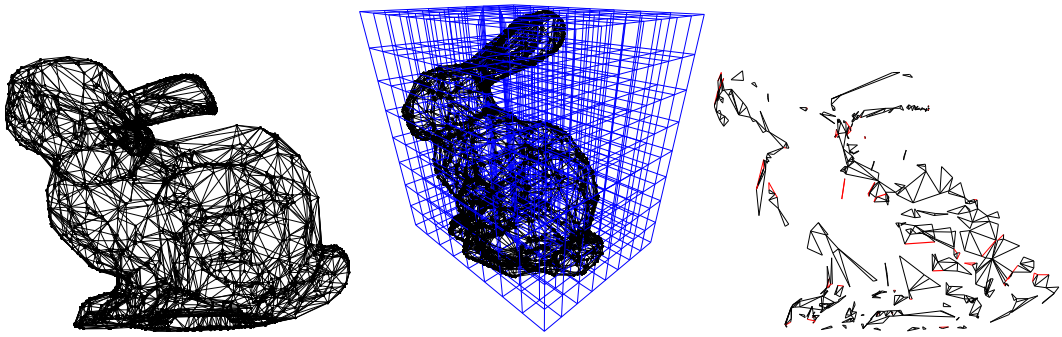

Figure 3.12: **left:** Wireframe representation of a given model. **middle:** Voxelized space around the model. One voxel on the lowest octree level is selected, based on the light position, and all potentially-silhouette (need to be tested) and silhouette edges (guaranteed to be silhouette) can be collected 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.

3.4.1 Algorithm

The algorithm is based on the concept of the *potentially visible set* (PVS) introduced by Airey et al. [1]. 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 3.12. 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**.

Silhouette Extraction

The brute force silhouette extraction process is described in previous sections and it is based on Kim's algorithm [11] with concept of reference edge [16].

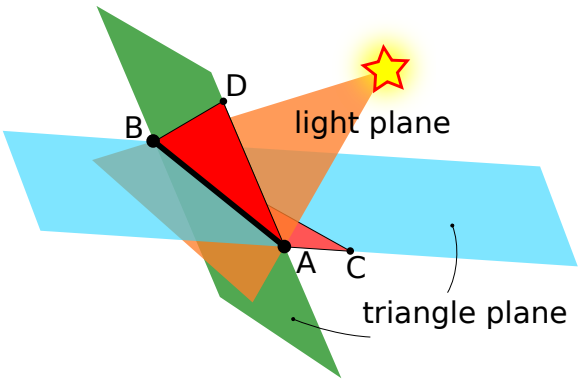

Figure 3.13: Edge AB is not a silhouette edge because triangles ABC and ABD do not lie on the same side of the light plane. Two triangles partition the world space into four subspaces.

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. [11] proposed a technique that computes the difference in the number of triangles on the left and the right side of the light plane (Figure 3.13) 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.

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 3.14. 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).

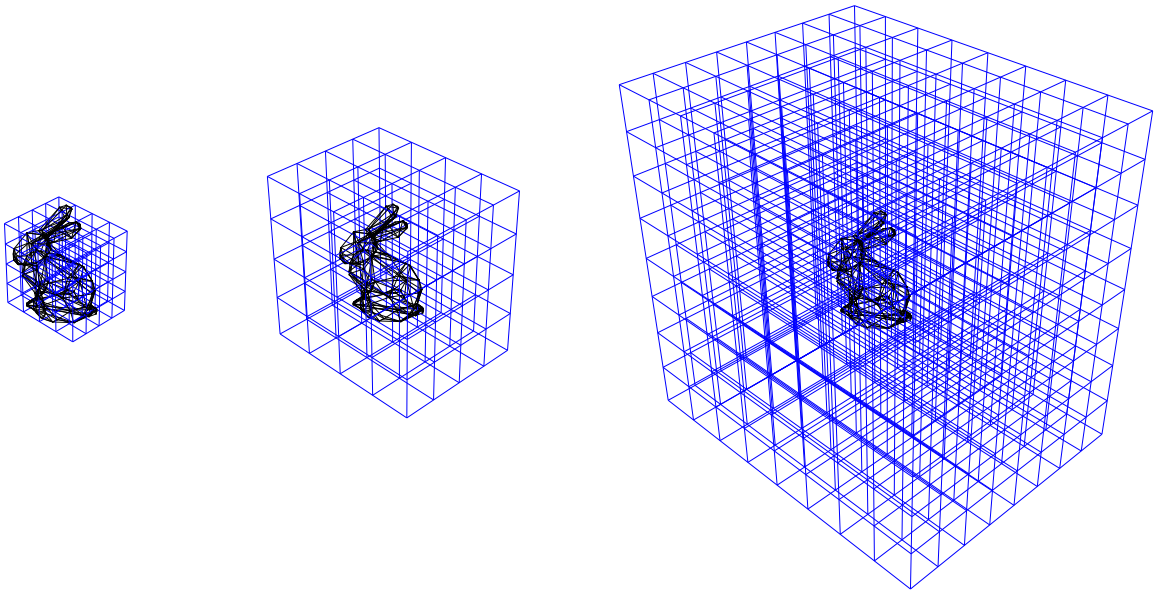

Figure 3.14: The algorithm supports custom scales of the scene bounding box. If a larger scale is selected, the light can be moved farther from the model. The image shows three different scales with the same voxelization level (in this case 2 levels of octree, $4 \times 4 \times 4$ voxels).

This scaled bounding volume circumscribes all the possible light positions. The user can then choose the maximal level of the octree hierarchy, see Figure 3.15. AABB scaling and maximum octree depth define the octree granularity and voxel size on the deepest level.

The depth level of 3–5 is suitable in most scenarios. Larger scales tend to consume too much memory (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 3.16. 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

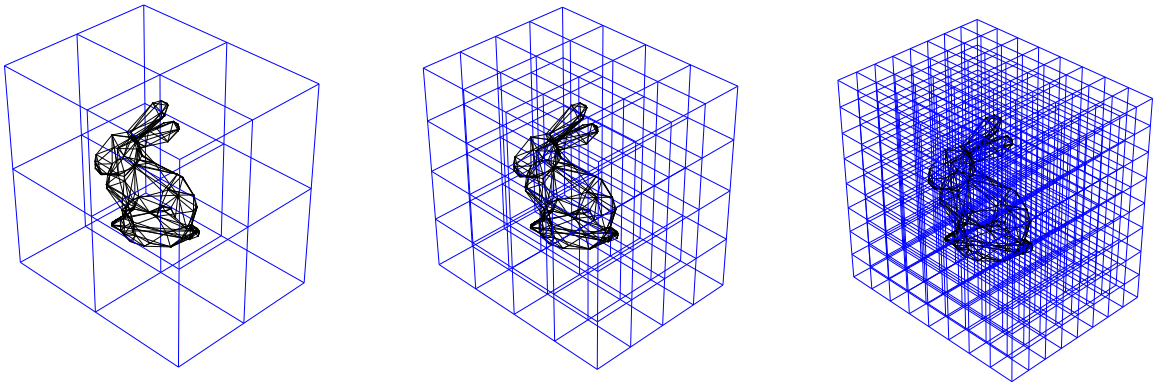

Figure 3.15: The algorithm supports a custom level of voxelization. The image shows three levels (1,2,3) of depth of the octree for the same scale of the scene bounding box.

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 3.13.

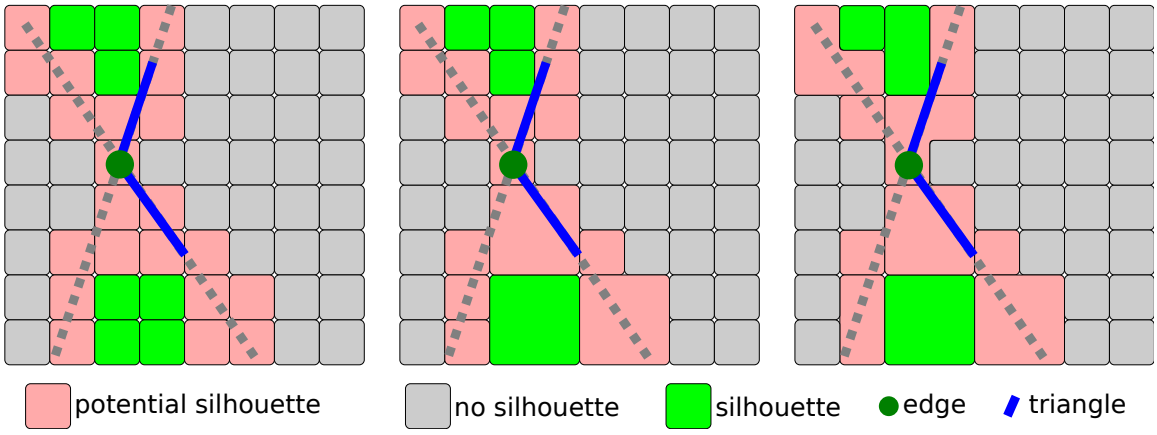

Figure 3.16: Overview of the proposed approach in 2D. The image shows voxels for one edge. The left side of the image shows the first step of the voxel building algorithm. Voxels are classified into 3 categories – no silhouette, silhouette and potentially silhouette. The next step is to propagate this 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 3.17. Every subvoxel in these voxel shapes contains the same set of edges, see Figure 3.18. 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

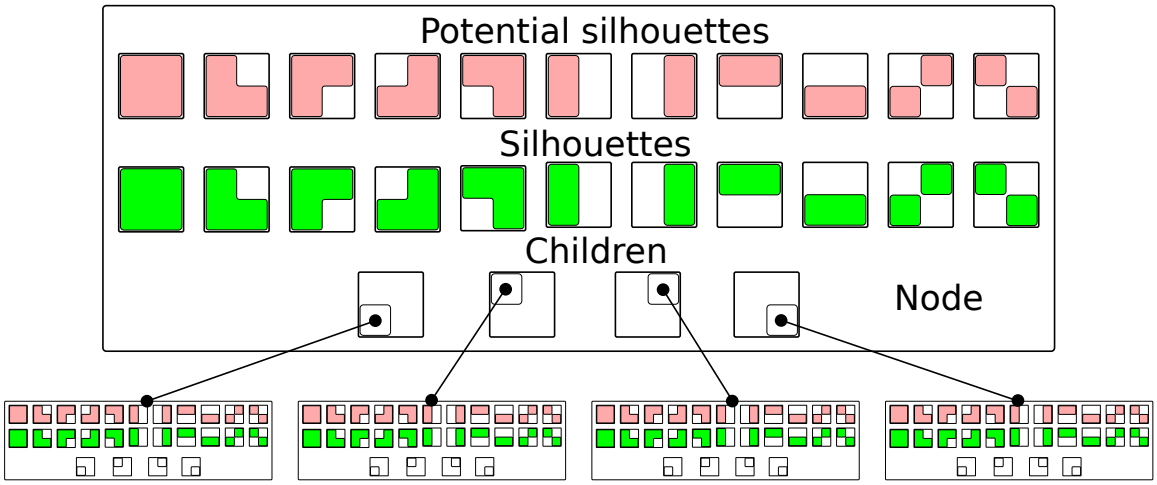

Figure 3.17: Node data in 2D space for the 8-bit compression. One node contains sets of silhouette and potentially silhouette edges, each addressed by its bitmask value. If a set shape does not intersect the triangle planes of an edge, the edge is stored into the set. The largest set shape is chosen if multiple set shapes do not intersect the triangle planes. A node also contains pointers to child nodes.

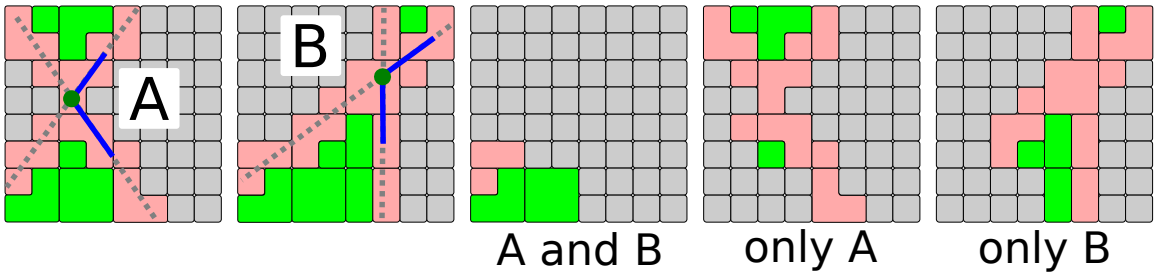

Figure 3.18: The first two images show two edges – A and B. Each edge partitions all voxels into voxel shapes for silhouette case and of potential silhouette case. If some voxel shapes are the same for both edges, the edge subsets of those voxel shapes contain both edges (middle image). Otherwise, voxel shapes contain only one edge.

(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 3.17.

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 3.19.

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.

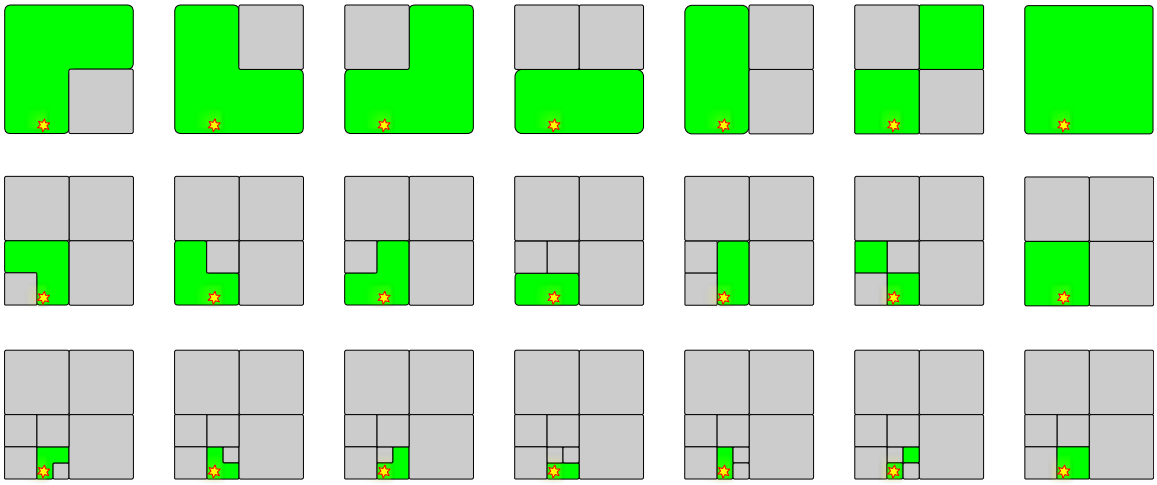

Figure 3.19: 2D illustration of all edge subsets that contain silhouette edges for a given light position. The hierarchy level is 3. The union of all subsets forms the set of all precomputed silhouette edges. Similar subsets are selected for all potentially silhouette edges. Note that some subsets could be empty. A single edge is contained only in one of the subsets (the largest possible).

3.5 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 3.20. The full content of this section can be found in paper [17].

3.5.1 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

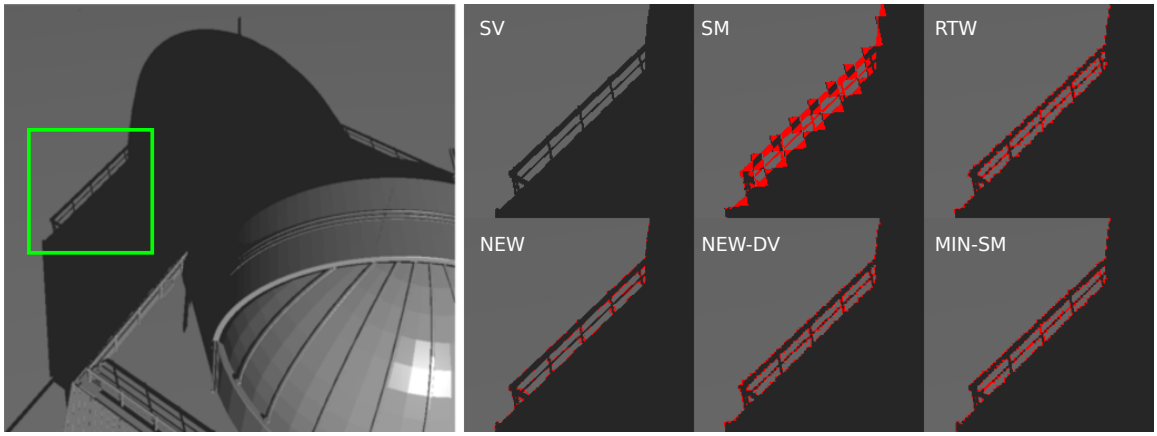

Figure 3.20: The figure shows the difference in quality of shadows cast from Observatory scene using different methods. Red pixels are wrongly evaluated. From left to right: Shadow Volumes (SV), Shadow Mapping (SM), Rectilinear Texture Warping (RTW), the new solution, the new solution using only desired view (DV), SM + minimal shadow frustum.

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 3.14. 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.

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 [22]. 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 3.21. The steps are described in detail in the following section.

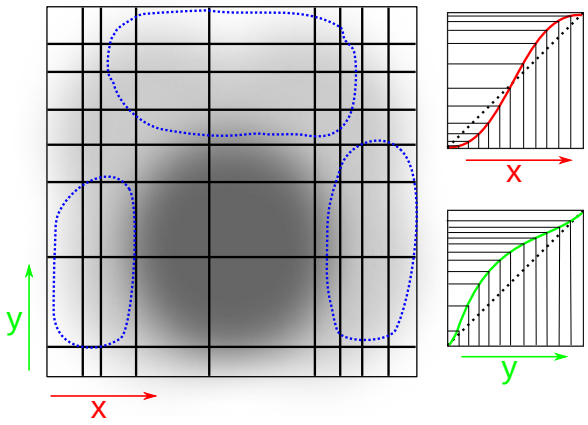

Figure 3.21: Importance map for RTW, Left: Combination of two 1D warping functions, Right: two 1D warping functions. It can be seen that blue parts are oversampled. The larger cells cover more important areas of the shadow map.

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:

$$n = f(x) \quad x \in \langle 0, 1 \rangle \tag{3.11}$$

$$s = g(x) = \int_0^x f(x)dx \tag{3.12}$$

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:

$$\frac{g(x)}{g(1)} = \frac{x}{1} \tag{3.13}$$

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:

$$o(x) = \frac{g(x)}{N} - x \tag{3.14}$$

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:

$$x' = x + o(x) \tag{3.15}$$

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 3.22). When 1D warping functions for columns are derived, all view samples are distributed almost uniformly.

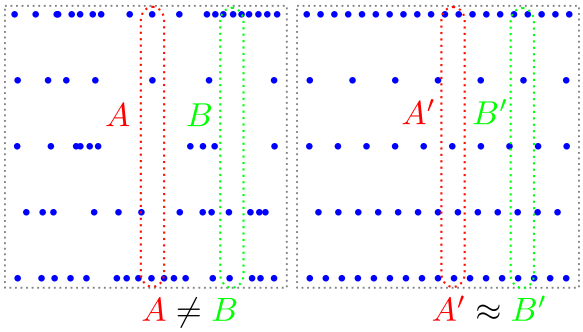

Figure 3.22: Left: Five rows of the importance map. Blue dots indicate view samples. Right: the importance map constructed using the set of row warping functions. Columns in the left do not contain the same number of view samples. Columns in the right contain approximately the same number of view samples.

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 3.23).

The complete warping function can be expressed as:

$$\begin{aligned} warp(x,y) &= (x + o_x^{(i)}(x), y + o_y^{(j)}(y)) \\ i &= \lfloor y \cdot w \rfloor \\ j &= \lfloor (x + o_x^{(i)}(x)) \cdot w \rfloor \end{aligned} \tag{3.16}$$

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 3.24.

Once the algorithm constructs both sets of the warping functions, the shadow map can be generated (see Step 8 of the proposed algorithm).

Minimal Shadow Frustum Extension

The algorithm was improved by finding a minimal shadow frustum (MSF) [24] 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.

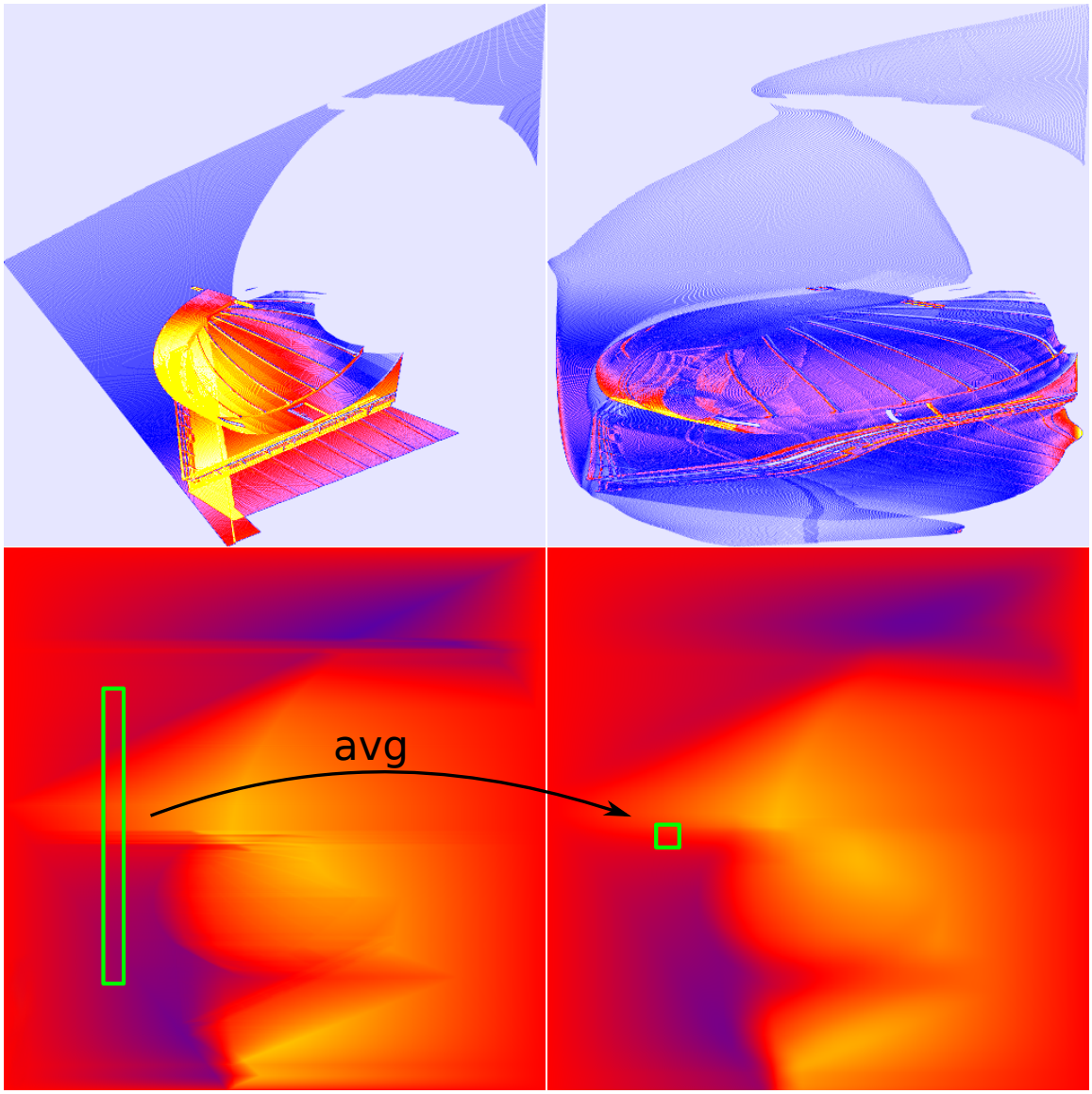

Figure 3.23: Top Left: Importance map. Top Right: The importance map after application of row warping functions - importance map for columns. Bottom left: a set of warping functions for every row of the importance map. Bottom right: warping functions smoothed using an averaging window shown in green. Yellow color in warping functions means positive offset to the right for certain position in the row. Blue color means offset to the left.

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.

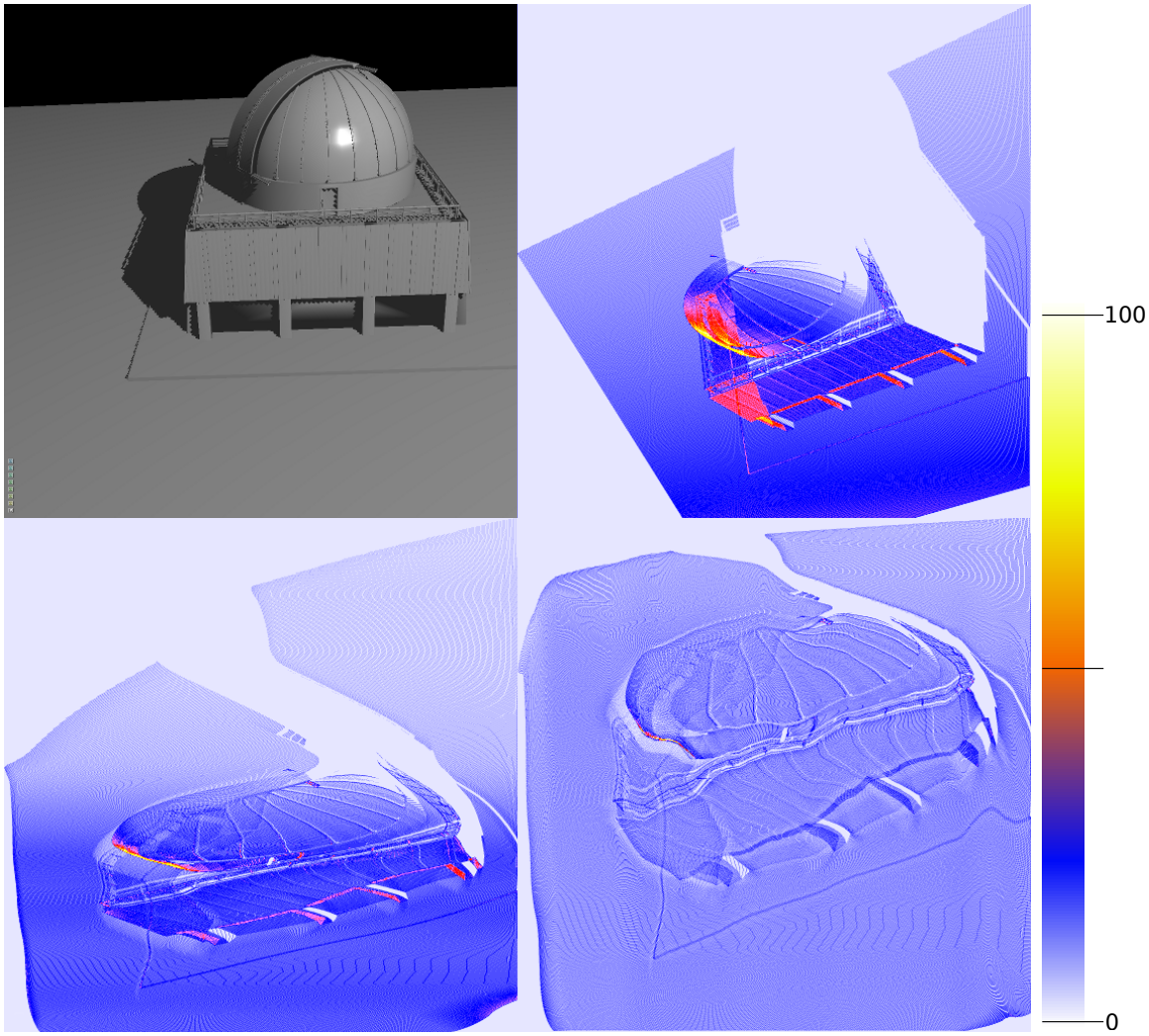

Figure 3.24: Top Left: Scene rendered from a camera point of view. Top Right: the importance map created from view samples. Bottom Left: reprojected view samples using only row warping functions. Bottom right: reprojected view samples using both sets for warping functions. It can be seen that view samples 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.

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.

3.6 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 3.25. 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 [18].

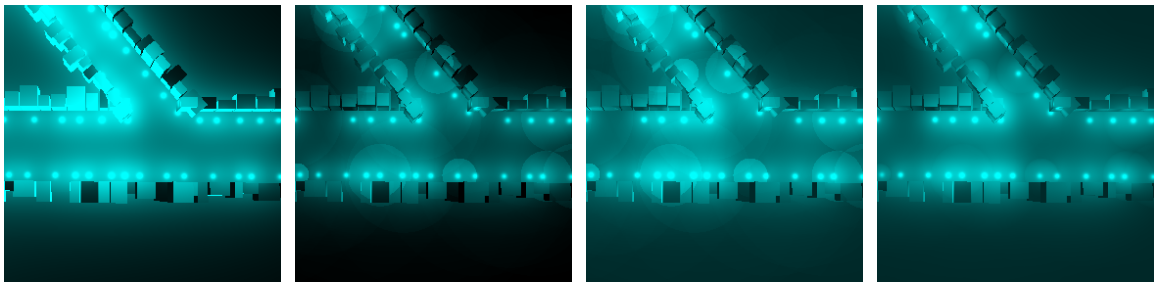

Figure 3.25: Leftmost image: Streets of a city illuminated with 628 light sources. Second image: Illumination with use of simple proposed attenuation. Mean squared error is $MSE = 0.033$ for $t_{att} = 0.0297$. Third image: The same, except for ambient lighting; $MSE = 0.011$. Rightmost image: Illumination with use of proposed attenuation; $MSE = 0.0195$.

Chapter 4

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.

4.1 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. [23], 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 4.1.

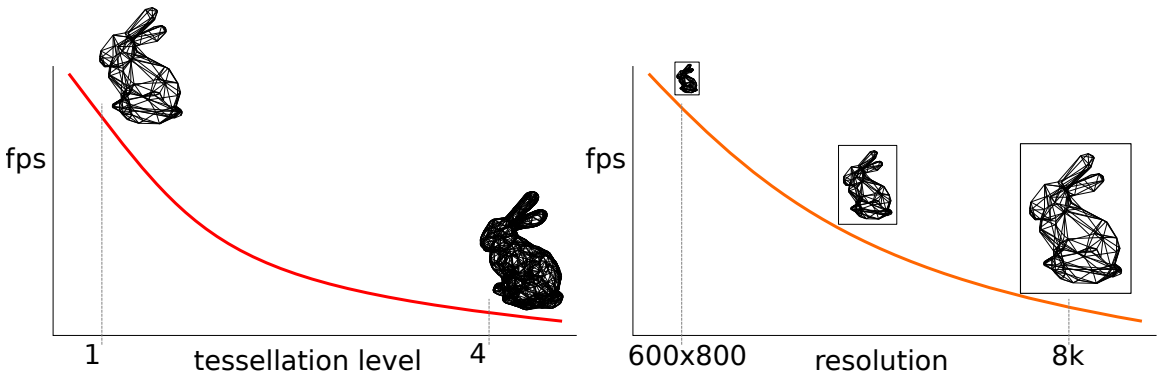

Figure 4.1: The image shows weaknesses of per triangle approaches with sample cluster hierarchy (left) and silhouette shadow volumes (right). The performance of per triangle method quickly degrades with tessellation. Silhouette algorithms are less sensitive to tessellation. On the other hand, stencil 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.

4.1.1 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.

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 4.2.

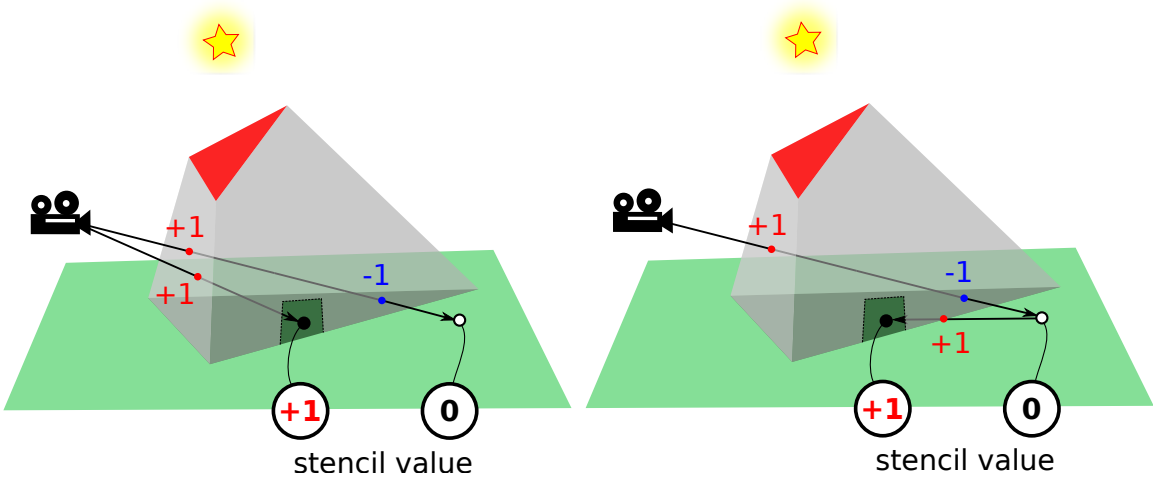

Figure 4.2: The left side of the image shows standard Z-pass shadow volume algorithm. Rays from the camera to the white and black view-samples compute stencil values. The right side shows another way. The stencil value of the black sample is computed using the stencil values of the white sample. The 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. The concept is related to the Jordan theorem [10] extended to three dimensions.

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 [20] in the screen space. The

problem view the z-curve is with its length. The better space filling curve is Hilbert curve [9]. These curves support clustering of sample rays.

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 4.3. The sample ray tree, implemented in the testing application can be seen in Figure 4.4. 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

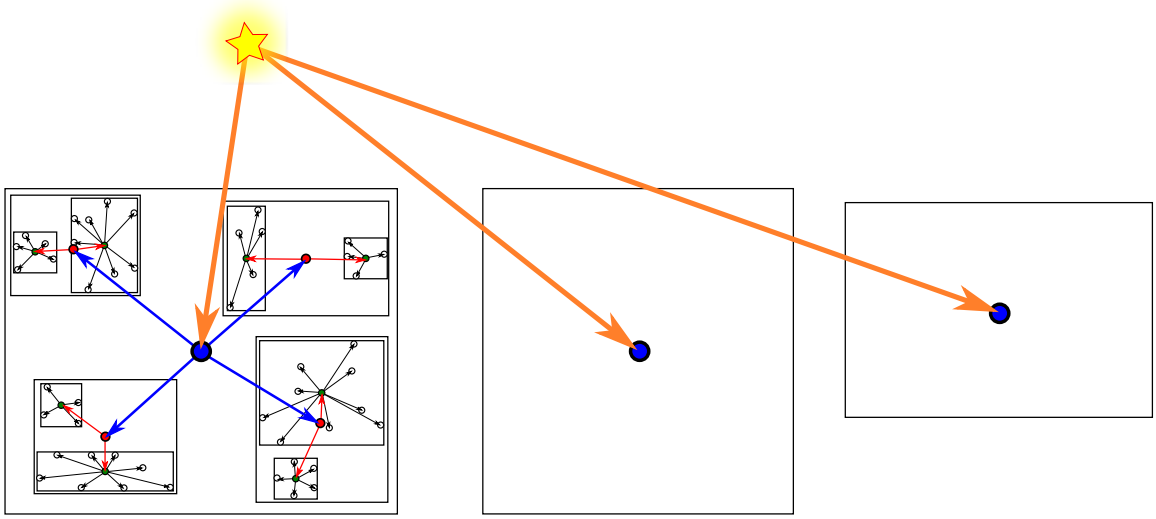

Figure 4.3: The image shows the concept of sample ray tree. View-samples are clustered in the similar fashion as in the CPTSV method. In addition, virtual view samples (blue, red and green) are created. Virtual samples are located at the center of clusters' bounding boxes. The stencil value of a view-sample 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.

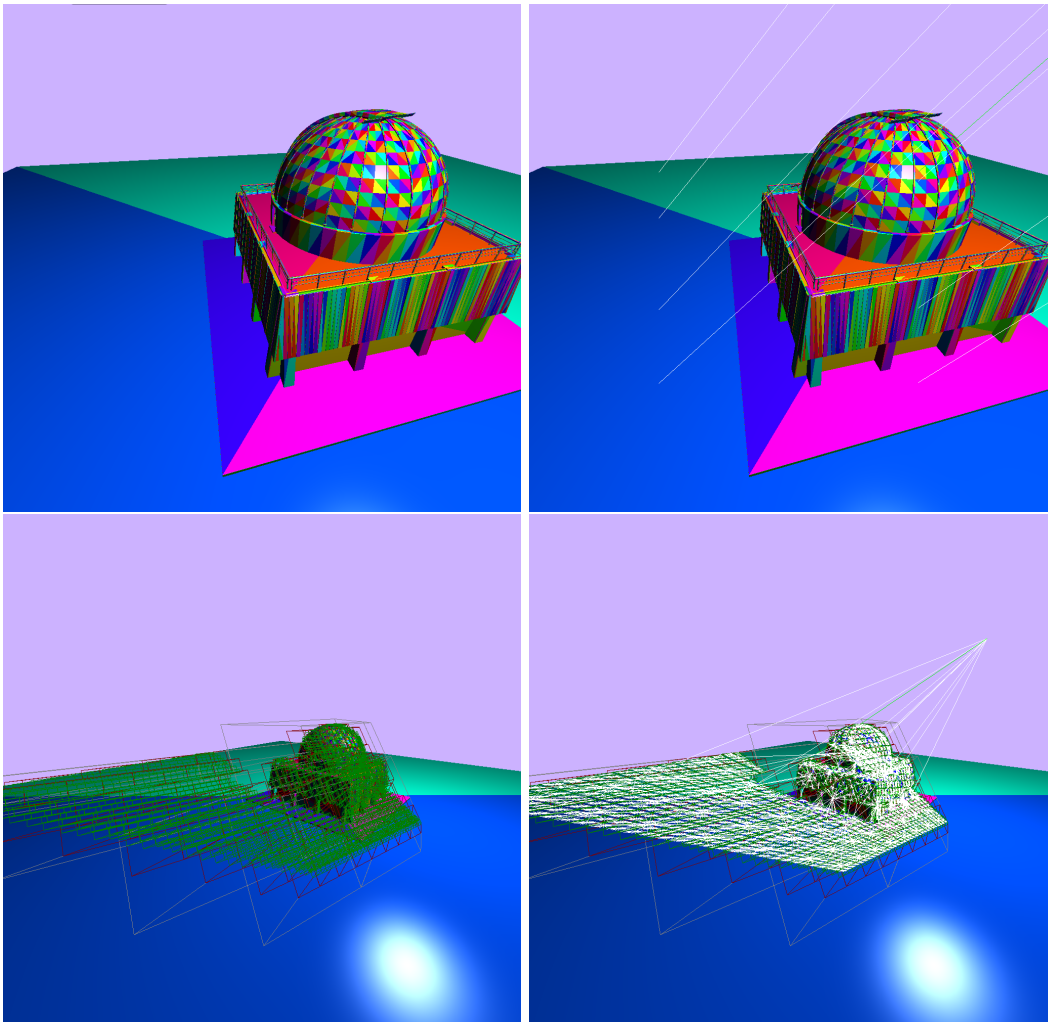

Figure 4.4: The image shows the sample ray tree implemented in the testing application. The top left image shows the scene from the camera point of view. The top right image shows sample rays from the light source to the zeroth level of the hierarchy. The bottom left image shows view-sample clusters. The bottom right image shows view-sample clusters with sample ray tree.

Chapter 5

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.

Curriculum Vitae

Name: Tomáš Milet
Email: imilet@fit.vutbr.cz

Education

- The highest level of education: The Master Degree, 2007 – 2012, Brno University of Technology, Faculty of Information Technology, Computer Graphics and Multimedia programme.
- In progress: The Doctoral Degree, 2012 - present, Brno University of Technology, Faculty of Information Technology.

Work Experience

- Researcher in computer graphics, acceleration and visualization, 2012 - now
- Researcher and developer for Cadwork informatik AG 2012 - 2015
- Developer and designer for NXP company 2017-2018
- Researcher for Honeywell company 2019
- Researcher and developer for Smarter Instrument 2019 - now
- Open-source developer

Teaching

- Computer Graphics Principles (bachelor course)
- Computer Graphics (master course)
- Advanced Computer Graphics (master course)
- Graphic and Multimedia Processors (master course)
- Multimedia (master course)
- Visualization and CAD (master course)
- Seminar of mathematics, programming, clean code, design patterns and computer graphics
- Supervisor of bachelor (52) and master (14) theses

Research

- KOBRTEK Jozef, MILET Tomáš and HEROUT Adam. Silhouette Extraction for Shadow Volumes Using Potentially Visible Sets. In: International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG). Plzeň: Union Agency, 2019, pp. 9-16. ISBN 978-80-86943-37-4.
- MILET Tomáš, NAVRÁTIL Jan and ZEMČÍK Pavel. An Improved Non-Orthogonal Texture Warping for Better Shadow Rendering. In: WSCG 2015 - Full Papers Proceedings. Plzeň: Union Agency, 2015, pp. 99-107. ISBN 978-80-86943-65-7.
- MILET Tomáš, TÓTH Michal, PEČIVA Jan, STARKA Tomáš, KOBRTEK Jozef and ZEMČÍK Pavel. Fast robust and precise shadow algorithm for WebGL 1.0 platform. In: ICAT-EGVE 2015 - International Conference on Artificial Reality and Telexistence and Eurographics Symposium on Virtual Environments. Kyoto: Eurographics Association, 2015, pp. 85-92. ISBN 978-3-905674-84-2.
- MILET Tomáš, KOBRTEK Jozef, ZEMČÍK Pavel and PEČIVA Jan. 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. Plzeň: University of West Bohemia in Pilsen, 2014, pp. 33-38. ISBN 978-80-86943-72-5.
- MILET Tomáš, NAVRÁTIL Jan, HEROUT Adam and ZEMČÍK Pavel. Improved Computation of Attenuated Light with Application in Scenes with Many Light Sources. In: Proceedings of SCCG 2013. Bratislava: Comenius University in Bratislava, 2013, pp. 155-160. ISBN 978-80-223-3377-1.
- PEČIVA Jan, STARKA Tomáš, MILET Tomáš, KOBRTEK Jozef and ZEMČÍK Pavel. Robust Silhouette Shadow Volumes on Contemporary Hardware. In: 23rd International Conference on Computer Graphics and Vision, GraphiCon 2013 - Conference Proceedings. Vladivostok: GraphiCon Scientific Society, 2013, pp. 56-59. ISBN 978-5-8044-1402-4.
- KOBRTEK Jozef, MILET Tomáš, TÓTH Michal and HEROUT Adam. Comparison of Modern Omni-Directional Precise ShadowingTechniques Versus Ray Tracing. Accepted in: Computer Graphics Forum after revisions.
- CHLUBNA Tomáš, MILET Tomáš and ZEMČÍK Pavel. Real-time per-pixel focusing method for light field rendering. In peer-review process of Computer Graphics Forum

Skills

- English - C1 level, C++, C, Python, CMake, Git, OpenGL, OpenCL
- Computer graphics, GPGPU, Mathematics, Procedural generation, Design patterns, Clean code, Unit testing, Compiler construction, Physics simulation, Linux

Bibliography

[1] 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.

[2] 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

[3] 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.

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

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

[6] 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

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

[8] 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

[9] Hilbert, D.: Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*. 1891: pp. 459 – 460.

[10] 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

[11] Kim, B.; Kim, K.; Turk, G.: A Shadow Volume Algorithm for Opaque and Transparent Non-Manifold Casters. 2008.

[12] 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

[13] Kolivand, H.; Sunar, M. S.: A Survey of Shadow Volume Algorithms in Computer Graphics. *IETE Tech Rev 2013*. vol. 30. 2013: pp. 38–46.

[14] McGuire, M.: Efficient Shadow Volume Rendering. In *GPU Gems*, edited by R. Fernando. Addison-Wesley. 2004. pp. 137–166.

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] Morton, G.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. 01 1966.

[21] 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

[22] 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

[23] 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.

[24] 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

[25] 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.

[26] van Waveren, J.: Shadow Volume Construction. 2005.
Retrieved from:
http://fabiensanglard.net/doom3_documentation/37730-293752.pdf